James E. Gentle

Elements of Computational and Data Sciences
This book covers many of the traditional areas of numerical analysis as well as topics in software development and methods of Monte Carlo simulations. The book is designed to serve both as a textbook for a course in numerical methods for students in the natural sciences, and as a reference for scientists whose research involves numerical computations and simulations.

The natural sciences have come to rely heavily on computations. Scientific computing often involves the evaluation of a mathematical expression and the computations are part of a larger computational program. Examples of computations for the evaluation of a mathematical expressions include solution of a system of equations, evaluation of an integral, and solution of a system of differential equations.

An important type of computation in the natural sciences is one that simulates natural phenomena. These computations are often large programs that take as input observed or assumed physical conditions, and produce output that describes or predicts behavior of physical systems.

Computer simulation has become, alongside experimentation and abstract reasoning, a third major tool of science. Simulation is itself a form of experimentation; the experimental units are entities built from computer-generated random numbers. Since Fermi, Ulam, and Pasta discovered energy solitons in metallic lattices in the 1950’s using a Maniac I computer, the use of simulation in the physical sciences has increased to form important subdisciplines. These are the “computational sciences”. They include “computational physics”, “computational fluid dynamics”, “computational biology”, and so on.

There are two main distinguishing characteristics of the computational sciences. One is the computational intensity of the methods. Datasets are often huge, but even for datasets of medium size, high performance computers may be required because the computations are applied repetitively to different subsets of the data. The other characteristic of the computational sciences is the attitude that computation is an instrument of discovery; that is, the
role of the computer is not just to store data, to perform computations, and to produce graphs and tables, but additionally its role is to suggest to the scientist alternative models and theories. Graphical displays and visualization methods are usually integral features of the computational sciences.

The student in the computational sciences needs a background in numerical analysis, and it is one of the objectives of this text to provide that background material. Implementation of the numerical methods, as well as general principles of software development, receives attention throughout the text. Two other aspects distinguish this text from others on numerical methods are an emphasis on simulation and Monte Carlo methods, and an emphasis on applying the numerical methods to the analysis of observational data.
## Contents

Preface .................................................................................................................. v

### Part I Simulation and the Computational and Data Sciences

1 The Computational and Data Sciences .............................................. 5  
1.1 The New Paradigm of Scientific Research ................................. 5  
1.2 Data-Driven Sciences ................................................................. 5  

2 Simulation: Introduction and Overview ........................................... 7  
2.1 Monte Carlo Methods .............................................................. 7  
  2.1.1 Evaluating an Integral ...................................................... 7  
  2.1.2 Experimental Error in Monte Carlo Methods .................. 9  
  2.1.3 Variance of Monte Carlo Estimators ................................. 11  
  2.1.4 Variance Reduction ....................................................... 12  
  2.1.5 Computer Experiments ................................................... 17  
  2.1.6 Computational Physics ................................................... 18  
2.2 Building Mathematical Models .................................................. 21  
  2.2.1 Models Based on Physical Laws ..................................... 21  
  2.2.2 Models Based on Observational Data ............................ 21  
2.3 Discrete Numerical Approximations ......................................... 21  
  2.3.1 Particles ................................................................. 21  
  2.3.2 Trajectories ............................................................. 21  
2.4 Numerical Algorithms and Codes ............................................ 21  
2.5 Numerical Experiments .......................................................... 21  
  2.5.1 Verification ................................................................. 21  
  2.5.2 Validation ............................................................... 21  
Exercises ................................................................. 21
5 Numerical Linear Algebra ........................................... 227
  5.1 Basic Vector/Matrix Computations .......................... 227
    5.1.1 Notation, Definitions, and Basic Properties ....... 228
    5.1.2 Computer Representations and Basic Operations .... 274
  5.2 Solution of Linear Systems .................................. 279
    5.2.1 Gaussian Elimination ................................... 279
    5.2.2 Matrix Factorizations .................................. 283
    5.2.3 Iterative Methods ...................................... 295
    5.2.4 Numerical Accuracy ..................................... 299
    5.2.5 Iterative Refinement ................................... 301
    5.2.6 Updating a Solution .................................... 302
    5.2.7 Overdetermined Systems; Least Squares .............. 303
    5.2.8 Other Computations for Linear Systems .............. 307
  5.3 Computation of Eigenvectors and Eigenvalues and the
    Singular Value Decomposition ................................. 310
    5.3.1 Power Method .......................................... 311
    5.3.2 Jacobi Method .......................................... 313
    5.3.3 QR Method for Eigenanalysis .......................... 316
    5.3.4 Singular Value Decomposition .......................... 318
  5.4 Software for Numerical Linear Algebra ........................ 321
    5.4.1 Fortran and C ........................................... 321
    5.4.2 Interactive Systems for Array Manipulation ......... 331
    5.4.3 Software for High-Performance Computers and Other
        Specialized Software ...................................... 338
    5.4.4 Test Data ................................................ 339
  Exercises .......................................................... 342

6 Nonlinear Systems .................................................. 349
  6.1 Basic Definitions and Properties ........................... 349
    6.1.1 Shapes of Functions .................................... 350
    6.1.2 Minima and Maxima of Functions ....................... 352
  6.2 Solution of Nonlinear Systems of Equations ................. 358
    6.2.1 Basic Methods for a Single Equation ................ 359
    6.2.2 Accuracy of the Solution .............................. 370
    6.2.3 Systems of Equations .................................. 371
  Exercises .......................................................... 374

7 Optimization and Nonlinear Systems ............................... 379
  7.1 Unconstrained Descent Methods in Dense Domains .......... 380
    7.1.1 Direction of Search .................................... 380
    7.1.2 Line Searches .......................................... 381
    7.1.3 Steepest Descent ...................................... 383
    7.1.4 Newton’s Method ....................................... 384
    7.1.5 Accuracy of Optimization Using Gradient Methods ... 387
    7.1.6 Quasi-Newton Methods ................................. 387
7.1.7 Fitting Models to Data Using Least Squares;
    Gauss-Newton Methods 391
7.1.8 Iteratively Reweighted Least Squares 395
7.1.9 Conjugate Gradient Methods 396
7.1.10 Derivative-Free Methods 397
7.1.11 Stochastic Search Methods 403
7.1.12 Summary of Continuous Descent Methods 405
7.2 Unconstrained Combinatorial Optimization; Other Direct
    Search Methods 406
7.2.1 Simulated Annealing 407
7.2.2 Evolutionary Algorithms 412
7.2.3 Other Combinatorial Search Methods 416
7.3 Optimization under Constraints 417
    7.3.1 Constrained Optimization in Dense Domains 418
    7.3.2 Constrained Combinatorial Optimization 423
7.4 Multiple Extrema and Multiple Objectives 425
    7.4.1 Multiple Extrema and Global Optimization 426
    7.4.2 Optimization with Multiple Criteria 427
    7.4.3 Optimization under Soft Constraints 428
7.5 Software for Optimization 429
    7.5.1 Fortran and C Libraries 430
    7.5.2 Optimization in General-Purpose Interactive Systems 435
    7.5.3 Software for General Classes of Optimization Problems 436
    7.5.4 Modeling Languages and Data Formats 438
    7.5.5 Testbeds for Optimization Software 439
Exercises 441
8 Quadrature and Evaluation of Special Functions 447
8.1 Numerical Quadrature 448
    8.1.1 Newton-Cotes Quadrature+ 448
    8.1.2 Adaptive Quadrature Rules+ 451
    8.1.3 Gaussian Quadrature+ 451
    8.1.4 Monte Carlo Methods+ 452
    8.1.5 Quasi Monte Carlo Methods+ 457
    8.1.6 Other Methods+ 458
8.2 Function Approximation 458
    8.2.1 Linear Operators 459
    8.2.2 Polynomial Approximation+ 461
    8.2.3 Multivariate Approximation 462
    8.2.4 Edgeworth Expansions+ 462
    8.2.5 Recursive Schemes for Accumulating Series+ 462
    8.2.6 Support Vector Methods+ 463
8.3 Orthogonal Systems 463
    8.3.1 Basis Functions 463
    8.3.2 Orthogonal Polynomials+ 465
8.4 Evaluation of Special Functions+ ........................................ 474
  8.4.1 General Methods ...................................................... 475
  8.4.2 Evaluation of Distribution Functions and Their Inverses 475
Exercises ............................................................................. 476

9 Solution of Differential Equations ...................................... 479
  9.1 Ordinary Differential Equations+ .................................. 480
    9.1.1 Initial Value Problems in ODEs .............................. 484
    9.1.2 Boundary Value Problems in ODEs ......................... 489
    9.1.3 Phase Planes in ODEs .............................................. 490
  9.2 Partial Differential Equations+ ....................................... 491
    9.2.1 Classification of PDEs+ ........................................ 491
    9.2.2 Phase Planes in PDEs+ .......................................... 493
    9.2.3 Methods of Solution of PDEs+ .............................. 493
    9.2.4 Higher Dimensions+ ............................................. 495
  9.3 Stochastic Differential Equations+ .................................. 495
  9.4 Software for Differential Equations .................................. 497
    9.4.1 Software for ODEs ................................................ 497
    9.4.2 Software for PDEs ................................................ 505
    9.4.3 Testbeds for Software for Differential Equations ....... 506
  9.5 Differential Equation Models ......................................... 506
    9.5.1 Simulations Using Differential Equation Models+ .... 507
    9.5.2 Qualitative Analysis of Differential Equation Models+ 509
    9.5.3 Fitting Differential Equation Models with Observed
          Data+ .............................................................. 509
Exercises ............................................................................. 510

Part III Data Science

10 Regression Models .......................................................... 527
  10.1 Linear Models ............................................................ 527
  10.2 Least Squares Fitting .................................................. 527
  10.3 Assessing the Fit ......................................................... 527
  10.4 Variable Selection ....................................................... 527
Exercises ............................................................................. 527

11 Smoothing Data .............................................................. 529
  11.1 General Considerations in Smoothing Data with Models 529
    11.1.1 Origins and Nature of the Model+ ....................... 529
    11.1.2 Fitting a Model to Observed Data ....................... 530
    11.1.3 Building Models+ ............................................... 535
    11.1.4 Stochastics ......................................................... 535
  11.2 Smoothing Functions .................................................. 536
    11.2.1 General Properties of Splines+ ........................ 536
## Contents

11.2.2 Smoothing Splines+ .............................................. 536  
11.2.3 Adaptive Splines+ ............................................. 536  
11.2.4 Bézier Curves .................................................. 537  
11.2.5 Multivariate Smoothing+ ...................................... 537  

11.3 Smoothing and Nonparametric Regression .......................... 538  
11.3.1 Bin Smoothers ................................................... 540  
11.3.2 Nearest Neighbor and Moving Average Smoothers .......... 541  
11.3.3 Smoothing by Averaging of Functions ...................... 541  
11.3.4 Local Weighting ................................................ 542  
11.3.5 Spline Regression .............................................. 543  
11.3.6 Smoothing over Two Dimensions ............................. 543  
11.3.7 Kriging .......................................................... 544  

Exercises ............................................................................... 544  

12 Density and Function Estimation ......................................... 545  
12.1 Basic Properties and Operations on Vectors and Functions ... 545  
12.1.1 Inner Products and Norms ...................................... 547  
12.1.2 Complete Spaces .................................................. 549  
12.1.3 Moments ........................................................... 549  
12.1.4 Convolutions and Covariances and Correlations .......... 550  
12.1.5 Tensor Products .................................................. 552  
12.2 Linear Transforms ..................................................... 552  
12.2.1 Fourier Transforms .............................................. 555  
12.2.2 Wavelets+ ........................................................ 556  

Exercises ............................................................................... 557  

13 Structure in Multivariate Data ............................................. 561  
13.1 Data Structures for Multivariate Data ............................... 561  
13.1.1 Variables and Observations in a Multivariate Dataset ... 562  
13.1.2 Grouped Data ...................................................... 563  
13.1.3 Graphs and Trees ................................................ 564  
13.1.4 Other Structures and Transformations of the Data ..... 566  
13.2 Measures of Similarity and Dissimilarity ......................... 567  
13.2.1 Similarities: Covariances and Correlations ............... 568  
13.2.2 Dissimilarities: Distances ..................................... 570  
13.2.3 Effects of Transformations of the Data .................... 574  
13.2.4 Outlying Observations and Collinear Variables .......... 575  
13.3 Ordering and Ranking Multivariate Data ......................... 576  
13.3.1 Minimal Spanning Trees ....................................... 576  
13.3.2 Nearest Neighbors and $k$-$d$-Trees ....................... 583  
13.3.3 Space-Filling Curves ........................................... 584  
13.4 Linear Structures in Data ........................................... 586  
13.4.1 Flats ................................................................. 586  
13.4.2 Rotations and Projections .................................... 587  
13.4.3 Linear Principal Components ................................. 589
### 13.4.4 Factor Analysis

600

### 13.4.5 Projection Pursuit

604

### 13.4.6 Linear Discriminant Analysis+

611

### 13.5 More General Structure+

611

### Exercises

612

### 14 Statistical Learning

615

#### 14.1 Clustering and Classification

615

#### 14.2 Methods for Clustering

616

##### 14.2.1 K-Means Clustering

617

##### 14.2.2 Hierarchical Clustering

618

##### 14.2.3 Agglomerative Hierarchical Clustering

618

##### 14.2.4 Divisive Hierarchical Clustering

622

##### 14.2.5 Self-Organizing Maps+

622

##### 14.2.6 Other Divisive Clustering Schemes

623

##### 14.2.7 Fuzzy Clustering

624

##### 14.2.8 Clustering and Transformations of the Data

624

##### 14.2.9 Comparing Clusterings

626

##### 14.2.10 Computational Complexity of Clustering

627

#### 14.3 Classification by Space Tessellations

628

#### 14.4 Heuristic Approaches to Clustering and Classification+

631

#### Exercises

632

### A Notation and Definitions

633

### B Solutions and Hints for Selected Exercises

647

### Bibliography

649

### Index

699
Part I

Simulation and the Computational and Data Sciences
***** what computational science is all about
***** also data sciences
1

The Computational and Data Sciences

1.1 The New Paradigm of Scientific Research

1.2 Data-Driven Sciences
2

Simulation: Introduction and Overview

2.1 Monte Carlo Methods

Monte Carlo simulation is the use of experiments with random numbers to evaluate mathematical expressions. These expressions may be definite integrals, systems of equations, or more complicated mathematical models. In most cases the standard approximations from numerical analysis are to be preferred, but Monte Carlo methods provide an alternative that is sometimes the only tractable approach. Monte Carlo is often the preferred method for evaluating integrals over high-dimensional domains, for example. Very large and sparse systems of equations can sometimes be solved effectively by Monte Carlo methods. Evaluating models of large numbers of particles is a common application in statistical physics.

2.1.1 Evaluating an Integral

In its simplest form (and all instances of Monte Carlo simulation can be reduced to this form), Monte Carlo simulation is the evaluation of a definite integral

$$\theta = \int_{D} f(x) \, dx \tag{2.1}$$

by identifying a random variable $Y$ with support on $D$ and density $p(y)$, and a function $g$ such that the expected value of $g(Y)$ is $\theta$:

$$E(g(Y)) = \int_{D} g(y)p(y) \, dy$$

$$= \int_{D} f(y) \, dy$$

$$= \theta.$$

Let us first consider the case in which $Y$ is taken to be a random variable with a uniform density over the interval $[a, b]$ and $g$ is taken to be $f$. In this
The problem of evaluating the integral becomes the familiar statistical problem of estimating a mean, $E(f(Y))$.

The statistician quite naturally takes a random sample and uses the sample mean. For a sample of size $m$, an estimate of $\theta$ is

$$\hat{\theta} = (b - a) \frac{\sum_{i=1}^{m} f(y_i)}{m},$$

(2.2)

where the $y_i$ are values of a random sample from a uniform distribution over $(a, b)$. The estimate is unbiased:

$$E(\hat{\theta}) = (b - a) \frac{\sum E(f(Y_i))}{m} = (b - a) E(f(Y)) = \int_{a}^{b} f(x) \, dx.$$ 

The variance is

$$V(\hat{\theta}) = (b - a)^2 \frac{\sum V(f(Y_i))}{m^2} = \frac{(b - a)^2}{m} V(f(Y)) = \frac{(b - a)}{m} \int_{a}^{b} \left( f(x) - \int_{a}^{b} f(t) \, dt \right)^2 \, dx.$$ 

(2.3)

The integral in (2.3) is a measure of the roughness of the function. (There are various ways of defining roughness. Most definitions involve derivatives. The more derivatives that exist, the less rough the function. Other definitions, such as the one here, are based on a norm of a function. The $L_2$ norm of the difference of the function from its integrated value is a very natural measure of roughness of the function. Another measure is just the $L_2$ norm of the function itself, which, of course, is not translation invariant.)

The method of estimating an integral just described is sometimes called “crude Monte Carlo”. In Exercise 2.1, page 21, we describe another method, called “hit-or-miss”, and ask the reader to show that the crude method is superior to hit-or-miss.

Suppose the original integral can be written as

$$\theta = \int_{D} f(x) \, dx = \int_{D} g(x) w(x) \, dx,$$

(2.4)
where $w(x)$ is a probability density over $D$. As with the uniform example considered earlier, it may require some scaling to get the density to be over the interval $D$. (In the uniform case, $D = (a, b)$, and both $a$ and $b$ must be finite.)

Now, suppose we can generate $m$ random variates $y_i$ from the distribution with density $w$. Then our estimate of $\theta$ is just

$$\hat{\theta} = \frac{\sum g(y_i)}{m}.$$  \hfill (2.5)

Compare this estimator with the estimator in (2.2). The variance of the estimator in (2.5) is likely to be smaller than that of the estimator in (2.2) (see Exercise 2.3 on page 22 and see Section 2.1.4).

The use of a probability density as a weighting function allows us to apply the Monte Carlo method to improper integrals, that is, integrals with infinite ranges of integration (see Exercise ??).

The Monte Carlo quadrature methods extend directly to multivariate integrals, although, obviously, it takes larger samples to fill the space. It is, in fact, only for multivariate integrals that Monte Carlo quadrature should ordinarily be used. The preference for Monte Carlo in multivariate quadrature results from the independence of the pseudoprobabilistic error bounds and the dimensionality mentioned above. In Section 8.1, page 448, we consider deterministic methods of numerical quadrature that are generally better than the Monte Carlo methods, especially for lower-dimensional integrals.

### 2.1.2 Experimental Error in Monte Carlo Methods

Monte Carlo methods are sampling methods; therefore the estimates that result from Monte Carlo procedures have associated sampling errors. The fact that the estimate is not equal to its expected value (assuming the estimator is unbiased) is not an “error” or a “mistake”; it is just a result of the variance of the random (or pseudorandom) data. Monte Carlos methods are experiments using random data. The variability of the random data results in experimental error, just as other scientific experiments in which randomness is a recognized component.

As in any statistical estimation problem, an estimate should be accompanied by an estimate of its variance. The estimate of the variance of the estimator of interest is usually just the sample variance of computed values of the estimator of interest.

In Chapter 3, page 55, we discuss bounds on error in numerical computations. Following standard practice, we could use the square root of the variance (that is, the standard deviation) of the Monte Carlo estimator to form an approximate confidence interval for the integral being estimated. Of course, the confidence limits would include the unknown terms in the variance. We could, however, estimate the variance of the estimator using the same sample that we use to estimate the integral.
The standard deviation in the approximate confidence limits is sometimes called a “probabilistic error bound”. The word “bound” is misused here, of course, but in any event, the standard deviation does provide some measure of a sampling “error”. The important thing to note from equation (2.3) is the order of error; it is $O(m^{-\frac{1}{2}})$.

An important property of the standard deviation of a Monte Carlo estimate of a definite integral is that the order in terms of the number of function evaluations is independent of the dimensionality of the integral. On the other hand, the usual error bounds for numerical quadrature are $O(m^{-\frac{2}{d}})$, where $d$ is the dimensionality.

We should be aware of a very important aspect of this discussion of error bounds for the Monte Carlo estimator. It applies to random numbers. The pseudorandom numbers we actually use only simulate the random numbers, so “unbiasedness” and “variance” must be interpreted carefully.

In Monte Carlo applications, a major reason for being interested in the variance of the estimator is to determine whether to increase the Monte Carlo sample size. The sample size is often determined so that the length of a confidence interval for the estimator meets a given maximum length requirement. For example, we may wish that the length of a 95% confidence interval for the parameter of interest, $\theta$, be no more than $d$. The confidence interval is of the form

$$
(\hat{\theta} - I_1, \hat{\theta} + I_2).
$$

Without knowing the distribution of $\hat{\theta}$, of course, we cannot determine $I_1$ and $I_2$. The usual approach is to approximate the confidence interval using the normal distribution. This leads to $I_1$ and $I_2$ having the forms $t_{\nu_1}\nu$ and $t_{\nu_2}\nu$, where $t_{\nu_1}$ and $t_{\nu_2}$ are quantiles of a Student’s $t$ distribution with $\nu$ degrees of freedom, and $\nu$ is the square root of an estimator of the variance of $\hat{\theta}$, based on a sample size related to $\nu$. Because $\nu$ decreases approximately as $1/\sqrt{\nu}$, increasing the sample size ultimately results in the restriction on the length being satisfied:

$$(t_{\nu_1} + t_{\nu_2})\nu \leq d,$$

(assuming the variance is finite).

The experimental error of Monte Carlo experiments should be treated just as carefully as the experimental error or measurement error in other scientific experimentation. The error determines a bound on the number of significant digits in numerical results. The error is propagated through any subsequent computations, and thus bounds on the number of significant digits are propagated.

In reporting numerical results from Monte Carlo simulations, it is mandatory to give some statement of the level of the experimental error. An effective way of doing this is by giving the sample standard deviation. When a number of results are reported, and the standard deviations vary from one to the other, a good way of presenting the results is to write the standard deviation in parentheses beside the result itself, for example:
2.1 Monte Carlo Methods

3.147 (0.0051)  

Notice that if the standard deviation is of order $10^{-3}$, the precision of the main result is not greater than $10^{-3}$. Just because the computations are done at a higher precision is no reason to write the number as if it had more significant digits.

2.1.3 Variance of Monte Carlo Estimators

The variance of an Monte Carlo estimator has important uses in assessing the quality of the estimate of the integral. The expression for the variance, as in equation (2.3), is likely to be very complicated and to contain terms that are unknown. We therefore need methods for estimating the variance of the Monte Carlo estimator.

**Estimating the Variance**

A Monte Carlo estimate usually has the form of the estimator of $\theta$ in equation (2.2):

$$\hat{\theta} = c \frac{\sum f_i}{m},$$

for which a variance estimator is

$$\hat{V}(\hat{\theta}) = c^2 \frac{\sum (f_i - \bar{f})^2}{m - 1}. \tag{2.6}$$

This is because the elements of the set of random variables $\{F_i\}$, on which we have observations $\{f_i\}$, are (assumed to be) independent, and thus to have zero correlations.

**Estimating the Variance Using Batch Means**

If the $F_i$ do not have zero correlations, the estimator (2.6) has an expected value that includes the correlations; that is, it is biased for estimating $V(\hat{\theta})$. This situation arises often in simulation. In many processes of interest, however, observations are “more independent” of observations farther removed within the sequence than they are of observations closer to them in the sequence. A common method for estimating the variance in a sequence of non-independent observations, therefore, is to use the means of successive subsequences that are long enough that the observations in one subsequence are almost independent of the observations in another subsequence. The means of the subsequences are called “batch means”.

If $F_1, \ldots, F_b, F_{b+1}, \ldots, F_{2b}, F_{2b+1}, \ldots, F_{kb}$ is a sequence of random variables such that the correlation of $F_i$ and $F_{i+b}$ is approximately zero, an estimate of the variance of the mean, $\bar{F}$, of the $m = kb$ random variables can be developed by observing that
variance reduction

swindle, Monte Carlo

\[ V(\bar{F}) = V \left( \frac{1}{m} \sum F_i \right) \]

\[ = V \left( \frac{1}{k} \sum_{j=1}^{k} \left( \frac{1}{b} \sum_{i=(j-1)b+1}^{jb} F_i \right) \right) \]

\[ \approx \frac{1}{k^2} \sum_{j=1}^{k} V \left( \frac{1}{b} \sum_{i=(j-1)b+1}^{jb} F_i \right) \]

\[ \approx \frac{1}{k} V(\bar{F}_b), \]

where \( \bar{F}_b \) is the mean of a batch of length \( b \). If the batches are long enough, it may be reasonable to assume the means have a common variance. An estimator of the variance of \( \bar{F}_b \) is the standard sample variance from \( k \) observations, \( \bar{f}_1, \ldots, \bar{f}_k \):

\[ \frac{\sum (\bar{f}_j - \bar{f})^2}{k-1}. \]

Hence, the batch-means estimator of the variance of \( \bar{F} \) is

\[ \hat{V}(\bar{F}) = \frac{\sum (\bar{f}_j - \bar{f})^2}{k(k-1)}. \] (2.7)

This batch-means variance estimator should be used if the Monte Carlo study yields a stream of nonindependent observations, such as in a time series or when the simulation uses a Markov chain. The size of the subsamples should be as small as possible and still have means that are independent. A test of the independence of the \( \bar{F}_b \) may be appropriate to help in choosing the size of the batches.

2.1.4 Variance Reduction

An objective in sampling is to reduce the variance of the estimators while preserving other good qualities, such as unbiasedness. In this section we briefly discuss variance reduction in Monte Carlo applications. The emphasis on efficient Monte Carlo sampling goes back to the early days of digital computing, but the issues are just as important today (or tomorrow), because, presumably, we are solving bigger problems.

Except for straightforward analytic reduction, discussed in the next section, techniques for reducing the variance of a Monte Carlo estimator are called “swindles”.

Analytic Reduction

The first principle in estimation is to use any known quantity to improve the estimate. For example, suppose the problem is to evaluate the integral
2.1 Monte Carlo Methods

\[ \theta = \int_D f(x) \, dx \]

by Monte Carlo. Now suppose \( D_1 \) and \( D_2 \) are such that \( D_1 \cup D_2 = D \) and \( D_1 \cap D_2 = \emptyset \), and consider the representation of the integral:

\[ \theta = \int_{D_1} f(x) \, dx + \int_{D_2} f(x) \, dx = \theta_1 + \theta_2. \]

Now suppose a part of this decomposition of the original problem is known, that is, suppose we know \( \theta_1 \). It is very likely that it would be better to use Monte Carlo only to estimate \( \theta_2 \), and take as our estimate of \( \theta \) the sum of the known \( \theta_1 \) and the estimated value of \( \theta_2 \). This seems intuitively obvious, and it is generally true unless there is some relationship between \( f(x_1) \) and \( f(x_2) \), where \( x_1 \) is in \( D_1 \) and \( x_2 \) is in \( D_2 \). If there is some known relationship, however, it may be possible to improve the estimate \( \hat{\theta}_2 \) of \( \theta_2 \) by using a transformation of the same random numbers used for \( \hat{\theta}_1 \) to estimate \( \theta_1 \). For example, if \( \hat{\theta}_1 \) is larger than the known value of \( \theta_1 \), the proportionality of the overestimate, \( (\hat{\theta}_1 - \theta_1)/\theta_1 \), may be used to adjust \( \hat{\theta}_2 \). This is the same principle as ratio or regression estimation in ordinary sampling theory (see, for example, Särndal, Swensson, and Wretman, 1992).

Now consider a different representation of the integral, in which \( f \) is expressed as \( g + h \), where \( g \) and \( h \) have the same signs. We have

\[ \theta = \int_D (g(x) + h(x)) \, dx = \int_D g(x) \, dx + \int_D h(x) \, dx = \theta_3 + \theta_4, \]

and suppose a part of this decomposition, say \( \theta_3 \), is known. In this case, the use of the known value of \( \int_D g(x) \, dx \) is likely to help only if \( g(x) \) tends to vary similarly with \( f(x) \). In this case it would be better to use Monte Carlo only to estimate \( \theta_4 \), and take as our estimate of \( \theta \) the sum of the known \( \theta_3 \) and the estimated value of \( \theta_4 \). This is because \( |h(x)| \) is less rough than \( |f(x)| \). Also, as in the case above, if there is some known relationship between \( g(x) \) and \( h(x) \), such as one tends to decrease as the other increases, it may be possible to use the negative correlation of the individual estimates to reduce the variance of the overall estimate.

**Antithetic Variates**

Again consider the problem of estimating the integral

\[ \theta = \int_a^b f(x) \, dx \]
The standard crude Monte Carlo estimator, equation (2.2), is \( (b - a) \frac{1}{n} \sum f(x_i) \), where \( x_i \) is uniform over \( (a, b) \). It would seem intuitively plausible that our estimate would be subject to less sampling variability if, for each \( x_i \), we used its “mirror”

\[ \tilde{x}_i = a + (b - x_i). \]

This mirror value is called an antithetic variate, and use of antithetic variates can be effective in reducing the variance of the Monte Carlo estimate, especially if the integral is nearly uniform. For a sample of size \( n \), the estimator is

\[ \frac{b - a}{n} \sum_{i=1}^{n} (f(x_i) + f(\tilde{x}_i)) \]

The variance of the sum is the sum of the variances plus twice the covariance. Antithetic variates have negative covariances, thus reducing the variance of the sum.

Antithetic variates from distributions other than the uniform can also be formed. The linear transformation that works for uniform antithetic variates cannot be used, however. A simple way of obtaining negatively correlated variates from other distributions is just to use antithetic uniforms in the inverse CDF. If the variates are generated using acceptance/rejection, for example, antithetic variates can be used in the majorizing distribution.

**Importance and Stratified Sampling**

In *importance sampling*, regions corresponding to large values of the integrand are sampled more heavily. This is accomplished by careful choice of \( w \) in the decomposition implied by equation (2.4) on page 8. We have

\[ \theta = \int_D f(x) \, dx = \int_D \frac{f(x)}{w(x)} w(x) \, dx. \]

where \( w(x) \) is a probability density over \( D \). The density \( w(x) \) is called the *importance function*.

From a sample of size \( m \) from the distribution with density \( w \), we have the estimator,

\[ \hat{\theta} = \frac{1}{m} \sum \frac{f(x_i)}{w(x_i)}. \quad (2.8) \]

The variance of this estimator is

\[ V(\hat{\theta}) = \frac{1}{m} V \left( \frac{f(X)}{w(X)} \right), \]
where the variance is taken with respect to the distribution of the random variable $X$ with density $w(x)$. Now,

$$V\left(\frac{f(X)}{w(X)}\right) = E\left(\frac{f^2(X)}{w^2(X)}\right) - \left(E\left(\frac{f(X)}{w(X)}\right)\right)^2.$$  

The objective in importance sampling is to choose $w$ so this variance is minimized. Because

$$\left(E\left(\frac{f(X)}{w(X)}\right)\right)^2 = \left(\int_D f(x) \, dx\right)^2,$$

the choice involves only the first term in the expression for the variance. By Jensen’s inequality (see Exercise 6.8, page 375), we have a lower bound on that term:

$$E\left(\frac{f^2(X)}{w^2(X)}\right) \geq \left(E\left(\frac{|f(X)|}{w(X)}\right)\right)^2 = \left(\int_D |f(x)| \, dx\right)^2.$$  

That bound is obviously achieved when

$$w(x) = \frac{|f(x)|}{\int_D |f(x)| \, dx}.$$  

Of course if we knew $\int_D |f(x)| \, dx$, we would probably know $\int_D f(x) \, dx$, and would not even be considering the Monte Carlo procedure to estimate the integral. In practice, for importance sampling we would seek a probability density $w$ that is nearly proportional to $|f|$; that is, such that $|f(x)|/w(x)$ is nearly constant. The problem of choosing an importance function is very similar to the problem of choosing a majorizing function for the acceptance/rejection method, as we discussed in Sections 4.2.3 and 4.2.4. Selection of an importance function involves the principals of function approximation, such as we discuss in Section 8.2, page 458, with the added constraint that the approximating function be a probability density from which it is easy to generate random variates.

If the integrand $f$ is multimodal, it is very likely that the best importance function would be a mixture of densities.

Importance sampling is similar to hit-or-miss Monte Carlo (see Exercise 2.1, page 21), and the relationship is particularly apparent when the weighting function is sampled by acceptance/rejection. In some of the literature on Monte Carlo methods, importance sampling is called use of “weight windows”.

In stratified sampling, the rule is to sample more heavily where the function is rough, that is, where the values $f(x_i)$ are likely to exhibit a lot of variability. This can be thought of as importance sampling in which the importance hit-or-miss Monte Carlo weight window
function is composed of a mixture of densities. Stratified sampling is usually performed by forming distinct subregions with different importance functions in each.

Stratified sampling is based on exactly the same principle that is applied in sampling methods in which the allocation is proportional to the variance (see Särndal, Swensson, and Wretman, 1992). In some of the literature on Monte Carlo methods, stratified sampling is called “geometric splitting”.

**Common Variates**

Often in Monte Carlo simulation the objective is to estimate the differences in parameters of two random processes. The two parameters are likely to be positively correlated. If that is the case, then the variance in the individual differences is likely to be smaller than the variance of the difference of the overall estimates.

Suppose, for example, that we have two statistics, $T$ and $S$, that are unbiased estimators of some parameter of a given distribution. We would like to know the difference in the variances of these estimators

$$V(T) - V(S)$$

(because the one with the smaller variance is better). We assume each statistic is a function of a random sample: $\{x_1, \ldots, x_n\}$. A Monte Carlo estimate of the variance of the statistic $T$ for a sample of size $n$ is obtained by generating $m$ samples of size $n$ from the given distribution, computing $T_i$ for the $i^{th}$ sample, and then computing

$$\hat{V}(T) = \frac{\sum_{i=1}^{m} (T_i - \overline{T})^2}{m - 1}.$$ 

Rather than doing this for $T$ and $S$ separately, using the unbiasedness, we could first observe

$$V(T) - V(U) = \text{E}(T^2) - \text{E}(S^2)$$

$$= \text{E}(T^2 - S^2),$$

and hence estimate the latter quantity. Because the estimators are likely to be positively correlated, the variance of the Monte Carlo estimator $\text{E}(T^2 - S^2)$ is likely to be smaller than the variance of $\hat{V}(T) - \hat{V}(S)$. If we compute $T^2 - S^2$ from each sample, that is, if we use common variates, we are likely to have a more precise estimate of the difference in the variances of the two estimators, $T$ and $S$.

**Constrained Sampling**

Sometimes in Monte Carlo methods it is desirable that certain sample statistics match the population parameters exactly; for example, the sample mean
may be adjusted by transforming each observation in the Monte Carlo sample by

$$\tilde{x}_i = x_i + \mu - \bar{x},$$

where $\mu$ is the mean of the target population and $\bar{x}$ is the mean of the original Monte Carlo sample.

This idea can be extended to more than one sample statistic, but requires more algebra to match up several statistics with the corresponding parameters.

Variance estimators that result from constrained samples must be used with care. This is because the constraints change the sampling variability, usually by reducing it.

2.1.5 Computer Experiments

Some of the most important questions in science and industry involve the relationship of an entity of interest to other entities that can either be controlled or more easily measured than the quantity of interest. We envision a relationship expressed by a model

$$y \approx f(x).$$

The quantity of interest $y$, usually called a “response” (although it may not be a response to any of the other entities), may be the growth of a crystal, the growth of a tumor, the growth of corn, the price of a stock one month hence, etc. The other variables $x$, called “factors”, “regressors”, or just “input variables”, may be temperature, pressure, amount of a drug, amount of a type of fertilizer, interest rates, etc. Both $y$ and $x$ may be vectors. An objective is to determine a suitable form of $f$ and the nature of the approximation. The simplest type of approximation is one in which an additive deviation can be identified with a random variable:

$$Y = f(x) + E.$$

The most important objective, whatever the nature of the approximation, usually is to determine values of $x$ that are associated with optimal realizations of $Y$. The association may or may not be one of causation.

One of the major contributions of the science of statistics to the scientific method is the experimental methods that efficiently help to determine $f$, the nature of an unexplainable deviation $E$, and the values of $x$ that yield optimal values of $y$. Design and analysis of experiments is a fairly mature subdiscipline of statistics.

In computer experiments the function $f$ is a computer program, $x$ is the input, and $y$ is the output. The program implements known or supposed relationships among the phenomena of interest. In cases of practical interest, the function is very complicated, the number of input variables may be in the
hundreds, and the output may consist of many elements. The objective is to find a tractable function, \( \hat{f} \), that approximates the true behavior, at least over ranges of interest, and to find the values of the input, say \( \hat{x}_0 \), such that \( \hat{f}(\hat{x}_0) \) is optimal. How useful \( \hat{x}_0 \) is depends on how close \( \hat{f}(\hat{x}_0) \) is to \( f(x_0) \), where \( x_0 \) yields the optimal value of \( f \).

What makes this an unusual statistical problem is that the relationships are deterministic. The statistical approach to computer experiments introduces randomness into the problem. The estimate \( \hat{f}(\hat{x}_0) \) can then be described in terms of probabilities or variances.

In a Bayesian approach, randomness is introduced by considering the function \( f \) to be a realization of a random function, \( F \). The prior on \( F \) may be specified only at certain points, say \( F(x_0) \). A set of input vectors \( x_1, \ldots, x_n \) is chosen, and the output \( y_i = f(x_i) \) is used to estimate a posterior distribution for \( F(x) \), or at least for \( F(x_0) \). The Bayesian approach generally involves extensive computations.

In a frequentist approach, randomness is introduced by taking random values of the input, \( x_1, \ldots, x_n \). This randomness in the input yields randomness in the output \( y_i = f(x_i) \), which is used to obtain the estimates \( \hat{x}_0 \) and \( \hat{f}(\hat{x}_0) \) and estimates of the variances of the estimators. See Koehler and Owen (1996) for further discussion of this approach.

2.1.6 Computational Physics

Many models of physics that describe the behavior of an ensemble of particles are stochastic. Generally, the individual particles obey simple laws that govern their motion or state.

One of the most widely studied models is the Ising model, introduced by Ernst Ising in the 1920’s. The model can be used effectively to study phase transitions in ferromagnetism, which was the original use by Ising; to study state transitions; and to model binary amalgamations. In the Ising model a lattice is used to locate positions of the entities of interest. In applications in physics, of course, the lattice is generally of three dimensions, but often a two dimensional lattice is useful.

In the applications in magnetism, we think of the lattice as representing locations of atoms that have binary magnetic moments, either + or −. If a linear ordering is imposed on the lattice, say by a systematic traversal of rows, then columns, then planes, and so on, the system can be described by a configuration vector \( \sigma = (\sigma_1, \ldots, \sigma_n) \). In a two-dimensional lattice the state of the system shown in Figure 2.1, for example, can be represented by \( \sigma = (+1, -1, +1, +1, \ldots, -1, -1) \), in which the ordering is row-wise beginning with the top row.

Each particle interacts with all the others in the system. The total energy in the system is given by the Hamiltonian

\[
H(\sigma) = -\sum_{i<j} J_{ij} \sigma_i \sigma_j - h \sum_i \sigma_i,
\]  

(2.10)
where $J_{ij}$ represents the strength of attraction between particles $i$ and $j$, and $h$ is the strength of an external magnetic field. In the Ising model, only interactions between particles at adjacent points on the lattice are considered, and the strength of attraction is assumed to be equal for all adjacent pairs. Hence, $J_{ij} = 0$ for nonadjacent pairs, and $J_{ij} = J$ for adjacent pairs.

The system is subject to “thermal agitation”, that is random changes in state. The probability model for states is a Boltzmann distribution or a Gibbs distribution. In this model, the probability of a state $\sigma$ is given by

$$\Pr(\sigma) \propto e^{-\beta H(\sigma)},$$

where $\beta$ includes units; often $\beta = 1/(kT)$, where $k$ is Boltzmann’s constant and $T$ is temperature in absolute degrees.

The normalizing constant for this probability distribution is

$$Z(\beta, J, h, n) = \sum_{\text{all configurations}} e^{-\beta H(\sigma)},$$

which in physics is called the partition function. The probability function then becomes

$$p(\sigma) = \frac{1}{Z} e^{-\beta H(\sigma)}. \quad (2.11)$$

In a continuous medium, the free energy density is

$$F(\beta, J, h) = \lim_{n \to \infty} \frac{1}{n} Z(\beta, J, h, n).$$

The system changes randomly, but generally in such a way that the energy decreases. The changes can be modeled as a Markov process, with transition probabilities favoring an energy decrease. It may be possible to determine a steady-state distribution analytically in lower dimensions, but in higher dimensions (currently three or greater), it is necessary to resort to simulation.

We can simulate the system changes by randomly generating a change and then using the Metropolis acceptance criterion (which we discuss on page 181.
The Metropolis algorithm is also sometimes called “M(RT)^2” after the names of all five authors of the original 1953 paper (Metropolis et al., 1953). This method is usually applied to one site at a time; that is, to a single spin. More efficient algorithms consider clusters of points. One of the earliest cluster algorithms is Swendsen-Wang algorithm, which can be viewed from a statistical perspective as a data augmentation algorithm.

Currently the most widely-used method for simulating the Ising model is the Wolff algorithm.

Algorithm 2.1 Wolff’s Algorithm for the Ising Model, for Fixed β

1. Choose a lattice point \( x \) at random uniformly.
2. Form a cluster about the chosen point:
   a) check each neighbor point, and if the point has the same polarity as \( x \), add it to the cluster with probability \( 1 - e^{-2\beta J} \);
   b) for each point added to the cluster, check each of its uncheck neighbors, and if the point has the same polarity as \( x \), add it to the cluster with probability \( 1 - e^{-2\beta J} \);
   c) repeat step 2b until there are no points in the cluster with any uncheck neighbors.
3. Flip the cluster.

Figure 2.2. One Step of Wolff’s Algorithm

Figure 2.2 shows one step in Wolff’s algorithm on a portion of the system represented by the lattice in Figure 2.1. The lattice on the left hand side represents the original configuration with the randomly chosen point shown by a circle. In the lattice in the middle, the point shown with three concentric circles is the one initially chosen, and the points with a single small circle represent ones that were inspected during the process. Those with a second larger concentric circle were accepted into the cluster. The lattice on the right hand side represents the state of the system following that step of Wolff’s algorithm.

In the Ising model there are only two possible states at each lattice point. The Potts model is a generalization to allow any finite number of discrete

2.2 Building Mathematical Models

2.2.1 Models Based on Physical Laws

2.2.2 Models Based on Observational Data

2.3 Discrete Numerical Approximations

2.3.1 Particles

2.3.2 Trajectories

2.4 Numerical Algorithms and Codes

2.5 Numerical Experiments

2.5.1 Verification

2.5.2 Validation

Exercises

2.1. The “hit-or-miss” method is another Monte Carlo method to evaluate an integral. To simplify the exposition, let us assume that $f(x) \geq 0$ on $[a,b]$, and we wish to evaluate the integral, $I = \int_{a}^{b} f(x) \, dx$. First, determine $c$ such that $c \geq f(x)$ on $[a,b]$. Generate a random sample of $m$ pairs of uniform deviates $(x_i, y_i)$, in which $x_i$ is from a uniform distribution over $[a,b]$, $y_i$ is from a uniform distribution over $[0,c]$, and $x_i$ and $y_i$ are independent. Let $m_1$ be the number of pairs such that $y_i \leq f(x_i)$. Estimate $I$ by $c(b-a)m_1/m$. (Sketch a picture, and you can see why it is called hit-or-miss. Notice also the similarity of the hit-or-miss method to acceptance/rejection. It can be generalized by allowing the $y$s to arise from a more general distribution (that is, any distribution with a majorizing density). Another way to think of the hit-or-miss method is as importance sampling in which the sampling of the weighting function is accomplished by acceptance/rejection.)

a) Is this a better method than the crude method described in Section ??? (What does this question mean? Think “bias” and “variance”.

To simplify the computations for answering this question, consider the special case in which $[a,b]$ is $[0,1]$ and $c = 1$. For a further discussion, see Hammersley and Handscomb, 1964.)
b) Suppose that $f$ is a probability density and the hit-or-miss method is used. Consider the set of the $m$ $x_i$s for which $y_i \leq f(x_i)$. What can you say about this set with respect to the probability density $f$? Because a hit-or-miss estimate is a rational fraction, the methods are subject to granularity. See the discussion following the Buffon’s needle problem in Exercise ?? below.

2.2. Describe a Monte Carlo method for evaluating each of the integrals

a) \[ \int_{-\infty}^{\infty} \int_{0}^{2} \int_{0}^{\infty} y \cos(\pi(x + y + z)) e^{-x^2} e^{-z^2} \, dz \, dy \, dx. \]

b) \[ \int_{-\infty}^{\infty} \int_{0}^{2} \int_{0}^{\infty} y \cos(\pi y) e^{-x^2} e^{-yz} \, dz \, dy \, dx. \]

2.3. Obtain a simplified expression for the variance of the Monte Carlo estimator (??) on page ??.
Part II

Computational Science
what computational science is all about

also data sciences
3.1 Computer Storage and Manipulation of Data

The computer is a tool for storage, manipulation, and presentation of data. The data may be numbers, text, or images. For each type of data, there are several ways of coding that can be used to store the data, and specific ways the data may be manipulated.

How much a computer user needs to know about the way the computer works depends on the complexity of the use and the extent to which the necessary operations of the computer have been encapsulated in software that is oriented toward the specific application. This chapter covers many of the basics of how digital computers represent data and perform operations on the data. Although some of the specific details we discuss will not be important for the computational scientist or for someone doing statistical computing, the consequences of those details are important, and the serious computer user must be at least vaguely aware of the consequences. The fact that multiplying two positive numbers on the computer can yield a negative number should cause anyone who programs a computer to take care.

Bits, Bytes, Codes, ...

Data of whatever form is represented by groups of 0’s and 1’s, called bits from the words “binary” and “digits”. (The word was coined by John Tukey.) For representing simple text, that is, strings of characters with no special representation, the bits are usually taken in groups of eight, called bytes, and associated with a specific character according to a fixed coding rule. Because of the common association of a byte with a character, those two words are often used synonymously.

The most widely used code for representing characters in bytes is “ASCII” (pronounced “askey”, from American Standard Code for Information Interchange). Because the code is so widely used, the phrase “ASCII data” is
sometimes used as a synonym for text or character data. The ASCII code for the character “A”, for example, is 01000001; for “a” is 01100001; and for “5” is 00110101. Strings of bits are read by humans more easily if grouped into strings of fours; a four-bit string is equivalent to a hexadecimal digit, 1, 2, ..., 9, A, B, ..., or F. Thus, the ASCII codes just shown could be written in hexadecimal notation as 41 (“A”), 61 (“a”), and 35 (“5”).

Because the common character sets differ from one language to another (both natural languages and computer languages), there are several modifications of the basic ASCII code set. Also, when there is a need for more different characters than can be represented in a byte (2^8), codes to associate characters with larger groups of bits are necessary. For compatibility with the commonly used ASCII codes using groups of 8 bits, these codes usually are for groups of 16 bits. These codes for “16-bit characters” are useful for representing characters in some Oriental languages, for example. The Unicode Consortium (1990, 1992) has developed a 16-bit standard, called Unicode, that is widely used for representing characters from a variety of languages. For any ASCII character, the Unicode representation uses eight leading 0’s and then the same eight bits as the ASCII representation.

A standard scheme of representing data is very important when data are moved from one computer system to another, or when researchers at different sites want to share data. Except for some bits that indicate how other bits are to be formed into groups (such as an indicator of the end of a file, or the delimiters of a record within a file), a set of data in ASCII representation is the same on different computer systems. Software systems that process documents either are specific to a given computer system or must have some standard coding to allow portability. The Java system, for example, uses Unicode for representing characters so as to insure that documents can be shared among widely disparate platforms.

In addition to standard schemes for representing the individual data elements, there are some standard formats for organizing and storing sets of data. Although most of these formats are defined by commercial software vendors, two that are open and may become more commonly used are the Common Data Format (CDF), developed by the National Space Science Data Center, and the Hierarchical Data Format (HDF), developed by the National Center for Supercomputing Applications. Both standards allow a variety of types and structures of data; the standardization is in the descriptions that accompany the datasets.

Types of Data

Bytes that correspond to characters are often concatenated to form character string data (or just “strings”). Strings represent text without regard to the appearance of the text if it were to be printed. Thus, a string representing “ABC” does not distinguish between “ABC”, “ABC”, and “ABC”. The ap-
The appearance of characters or of other visual entities such as graphs or pictures is often represented more directly as a “bitmap”. Images on a display medium such as paper or a CRT screen consist of an arrangement of small dots, possibly of various colors. The dots must be coded into a sequence of bits, and there are various coding schemes in use, such as GIF (Graphical Interchange File) or WMF (Windows MetaFile). Image representations of “ABC”, “ABC”, and “ABC” would all be different. The computer internal representation may correspond directly to the dots that are displayed, or it may be a formula to generate the dots, but in each case, the data are represented as a set of dots located with respect to some coordinate system. More dots would be turned on to represent “ABC” than to represent “ABC”. The location of the dots and the distance between the dots depend on the coordinate system; thus the image can be repositioned or rescaled.

Computers initially were used primarily to process numeric data, and numbers are still the most important type of data in statistical computing. There are important differences between the numerical quantities with which the computer works and the numerical quantities of everyday experience. The fact that numbers in the computer must have a finite representation has very important consequences.

### 3.1.1 Digital Representation of Numeric Data

For representing a number in a finite number of digits or bits, the two most relevant things are the magnitude of the number and the precision to which the number is to be represented. Whenever a set of numbers is to be used in the same context, we must find a method of representing the numbers that will accommodate their full range and will carry enough precision for all of the numbers in the set.

Another important aspect in the choice of a method to represent data is the way data are communicated within a computer and between the computer and peripheral components such as data storage units. Data are usually treated as a fixed-length sequence of bits. The basic grouping of bits in a computer is sometimes called a “word”, or a “storage unit”. The lengths of words or storage units commonly used in computers are 32 or 64 bits.

Unlike data represented in ASCII (in which the representation is actually of the characters, which in turn, represent the data themselves), the same numeric data will very often have different representations on different computer systems. It is also necessary to have different kinds of representations for different sets of numbers, even on the same computer. Like the ASCII standard for characters, however, there are some standards for representation of, and operations on, numeric data. The Institute for Electrical and Electronics Engineers (IEEE) has been active in promulgating these standards, and the standards themselves are designated by an IEEE number.
The two mathematical models that are often used for numeric data are the ring of integers, \( \mathbb{Z} \), and the field of reals, \( \mathbb{R} \). We use two computer models, \( \mathbb{I} \) and \( \mathbb{F} \), to simulate these mathematical entities. (Unfortunately, neither \( \mathbb{I} \) nor \( \mathbb{F} \) is a simple mathematical construct such as a ring or field.)

**Representation of Relatively Small Integers:**

**Fixed-Point Representation**

Because an important set of numbers is a finite set of reasonably sized integers, efficient schemes for representing these special numbers are available in most computing systems. The scheme is usually some form of a base 2 representation, and may use one storage unit (this is most common), two storage units, or one half of a storage unit. For example, if a storage unit consists of 32 bits and one storage unit is used to represent an integer, the integer 5 may be represented as in binary notation using the low-order bits, as shown in Figure 3.1.

![Figure 3.1. The Value 5 in a Binary Representation](image)

The sequence of bits in Figure 3.1 represents the value 5; the ASCII code shown previously, 00110101 or 35 in hexadecimal, represents the character “5”.

If the set of integers includes the negative numbers also, some way of indicating the sign must be available. The first bit in the bit sequence (usually one storage unit) representing an integer is usually used to indicate the sign; if it is 0, a positive number is represented; if it is 1, a negative number. In a common method for representing negative integers, called “twos-complement representation”, the sign bit is set to 1, and the remaining bits are set to their opposite values (0 for 1; 1 for 0) and then 1 is added to the result. If the bits for 5 are ...00101, the bits for \(-5\) would be ...11010 \(+\) 1, or ...11011. If there are \( k \) bits in a storage unit (and one storage unit is used to represent a single integer), the integers from 0 through \( 2^{k-1} - 1 \) would be represented in ordinary binary notation using \( k - 1 \) bits. An integer \( i \) in the interval \([-2^{k-1}, -1]\) would be represented by the same bit pattern by which the nonnegative integer \( 2^{k-1} - i \) is represented, except the sign bit would be 1.

The sequence of bits in Figure 3.2 represents the value \(-5\) using twos-complement notation in 32 bits, with the leftmost bit being the sign bit, and the rightmost bit being the least significant bit, that is, the 1’s position. The ASCII code for “\(-5\)” consists of the codes for “\(-\)” and “5”, that is, 00101101 00110101.
The special representations for numeric data are usually chosen so as to facilitate manipulation of data. The twos-complement representation makes arithmetic operations particularly simple.

It is easy to see that the largest integer that can be represented in the twos-complement form is $2^{k-1} - 1$, and the smallest integer is $-2^{k-1}$.

A representation scheme such as that described above is called fixed-point representation or integer representation, and the set of such numbers is denoted by $\mathbb{I}$. The notation $\mathbb{I}$ is also used to denote the system built on this set. This system is similar in some ways to a field instead of a ring, which is what the integers $\mathbb{Z}$ are.

There are several variations of the fixed-point representation. The number of bits used and the method of representing negative numbers are two aspects that generally vary from one computer to another. Even within a single computer system, the number of bits used in fixed-point representation may vary; it is typically one storage unit or a half of a storage unit.

### Representation of Larger Numbers and Nonintegral Numbers: Floating-Point Representation

In a fixed-point representation all bits represent values greater than or equal to 1; the base point or radix point is at the far right, before the first bit. In a fixed-point representation scheme using $k$ bits, the range of representable numbers is of the order of $2^k$, usually from approximately $-2^{k-1}$ to $2^{k-1}$. Numbers outside of this range cannot be represented directly in the fixed-point scheme. Likewise, nonintegral numbers cannot be represented. Large numbers and fractional numbers are generally represented in a scheme similar to what is sometimes called "scientific notation", or in a type of logarithmic notation. Because within a fixed number of digits, the radix point is not fixed, this scheme is called floating-point representation, and the set of such numbers is denoted by $\mathbb{F}$. The notation $\mathbb{F}$ is also used to denote the system built on this set.

In a misplaced analogy to the real numbers, a floating-point number is also called "real". Both computer “integers”, $\mathbb{I}$, and “reals”, $\mathbb{F}$, represent useful subsets of the corresponding mathematical entities, $\mathbb{Z}$ and $\mathbb{R}$; but while the computer numbers called “integers” do constitute a fairly simple subset of the integers, the computer numbers called “real” do not correspond to the real numbers in a natural way. In particular, the floating-point numbers do not occur uniformly over the real number line.

Within the allowable range, a mathematical integer is exactly represented by a computer fixed-point number; but a given real number, even a rational,
of any size may or may not have an exact representation by a floating-point number. This is the familiar situation of fractions such as $\frac{1}{3}$ not having a finite representation in base 10. The simple rule, of course, is that the number must be a rational number whose denominator in reduced form factors into only primes that appear in the factorization of the base. In base 10, for example, only rational numbers whose factored denominators contain only 2’s and 5’s have an exact, finite representation; and in base 2, only rational numbers whose factored denominators contain only 2’s have an exact, finite representation.

For a given real number $x$, we will occasionally use the notation

$$[x]_e$$

to indicate the floating-point number used to approximate $x$, and we will refer to the exact value of a floating-point number as a computer number. We will also use the phrase “computer number” to refer to the value of a computer fixed-point number. It is important to understand that computer numbers are members of proper, finite subsets, $\mathbb{F}$ and $\mathbb{F}$, of the corresponding sets $\mathbb{Z}$ and $\mathbb{R}$.

Our main purpose in using computers, of course, is not to evaluate functions of the set of computer floating-point numbers or of the set of computer integers; the main immediate purpose usually is to perform operations in the field of real (or complex) numbers, or occasionally in the ring of integers. Doing computations on the computer, then, involves use of the sets of computer numbers to simulate the sets of reals or integers.

**The Parameters of the Floating-Point Representation**

The parameters necessary to define a floating-point representation are the base or radix, the range of the mantissa or significand, and the range of the exponent. Because the number is to be represented in a fixed number of bits, such as one storage unit or word, the ranges of the significand and exponent must be chosen judiciously so as to fit within the number of bits available. If the radix is $b$, and the integer digits $d_i$ are such that $0 \leq d_i < b$, and there are enough bits in the significand to represent $p$ digits, then a real number is approximated by

$$\pm 0.d_1d_2\cdots d_p \times b^e,$$

(3.1)

where $e$ is an integer. This is the standard model for the floating-point representation. (The $d_i$ are called “digits” from the common use of base 10.)

The number of bits allocated to the exponent $e$ must be sufficient to represent numbers within a reasonable range of magnitudes; that is, so that the smallest number in magnitude that may be of interest is approximately $b^{e_{\min}}$, and the largest number of interest is approximately $b^{e_{\max}}$, where $e_{\min}$ and $e_{\max}$ are, respectively, the smallest and the largest allowable values of the exponent. Because $e_{\min}$ is likely negative and $e_{\max}$ is positive, the exponent requires a
In practice, most computer systems handle the sign of the exponent by defining a bias, and then subtracting the bias from the value of the exponent evaluated without regard to a sign.

The parameters $b$, $p$, and $e_{\min}$ and $e_{\max}$ are so fundamental to the operations of the computer that on most computers they are fixed, except for a choice of two or three values for $p$, and maybe two choices for the range of $e$.

In order to insure a unique representation for all numbers (except 0), most floating-point systems require that the leading digit in the significand be nonzero, unless the magnitude is less than $b^{-e_{\min}}$. A number with a nonzero leading digit in the significand is said to be normalized.

The most common value of the base $b$ is 2, although 16 and even 10 are sometimes used. If the base is 2, in a normalized representation, the first digit in the significand is always 1; therefore, it is not necessary to fill that bit position, and so we effectively have an extra bit in the significand. The leading bit, which is not represented, is called a “hidden bit”. This requires a special representation for the number 0, however.

In a typical computer using a base of 2 and 64 bits to represent one floating-point number, 1 bit may be designated as the sign bit, 52 bits may be allocated to the significand, and 11 bits allocated to the exponent. The arrangement of these bits is somewhat arbitrary, and of course, the physical arrangement on some kind of storage medium would be different from the “logical” arrangement. A common logical arrangement assigns the first bit as the sign bit, the next 11 bits as the exponent, and the last 52 bits as the significand. (Computer engineers sometimes label these bits as 0, 1, ..., and then get confused as to which is the $i^{\text{th}}$ bit. When we say “first”, we mean “first”, whether an engineer calls it the “0th” or the “1st”. The range of exponents for the base of 2 in this typical computer would be 2,048. If this range is split evenly between positive and negative values, the range of orders of magnitude of representable numbers would be from $-2^{308}$ to $2^{308}$. The bits allocated to the significand would provide roughly 16 decimal places of precision.

Figure 3.3 shows the bit pattern to represent the number 5, using $b = 2$, $p = 24$, $e_{\min} = -126$, and a bias of 127, in a word of 32 bits. The first bit on the left is the sign bit, the next 8 bits represent the exponent, 129, in ordinary base 2 with a bias, and the remaining 23 bits represent the significand beyond the leading bit, known to be 1. (The binary point is to the right of the leading bit that is not represented.) The value is therefore $+1.01 \times 2^2$ in binary notation.

![Figure 3.3. The Value 5 in a Floating-Point Representation](image-url)
of that integer (see Figures 3.1 and 3.2), the only difference in the floating-point representation of a number and of its additive inverse is usually just in one bit. In the example of Figure 3.3, only the first bit would be changed to represent the number $-5$.

As mentioned above, the set of floating-point numbers is not uniformly distributed over the ordered set of the reals. There are the same number of floating-point numbers in the interval $[b^i, b^{i+1}]$ as in the interval $[b^{i+1}, b^{i+2}]$, even though the second interval is $b$ times as long as the first. Figures 3.4 through 3.6 illustrate this. The fixed-point numbers, on the other hand, are uniformly distributed over their range, as illustrated in Figure 3.7.

The density of the floating-point numbers is generally greater closer to zero. Notice that if floating-point numbers are all normalized, the spacing between 0 and $b^{e_{min}}$ is $b^{e_{min}}$ (that is, there is no floating-point number in that open interval), whereas the spacing between $b^{e_{min}}$ and $b^{e_{min}+1}$ is $b^{e_{min}-p+1}$. Most systems do not require floating-point numbers less than $b^{e_{min}}$ in magnitude to be normalized. This means that the spacing between 0 and $b^{e_{min}}$ can be $b^{e_{min}-p}$, which is more consistent with the spacing just above $b^{e_{min}}$. 
When these nonnormalized numbers are the result of arithmetic operations, the result is called “graceful” or “gradual” underflow.

The spacing between floating-point numbers has some interesting (and, for the novice computer user, surprising!) consequences. For example, if 1 is repeatedly added to $x$, by the recursion

$$x^{(k+1)} = x^{(k)} + 1,$$

the resulting quantity does not continue to get larger. Obviously, it could not increase without bound, because of the finite representation. It does not even approach the largest number representable, however! (This is assuming that the parameters of the floating-point representation are reasonable ones.) In fact, if $x$ is initially smaller in absolute value than $b^{e_{\text{max}}-p}$ (approximately), the recursion

$$x^{(k+1)} = x^{(k)} + c$$

will converge to a stationary point for any value of $c$ smaller in absolute value than $b^{e_{\text{max}}-p}$.

The way the arithmetic is performed would determine these values precisely; as we shall see below, arithmetic operations may utilize more bits than are used in the representation of the individual operands.

The spacings of numbers just smaller than 1 and just larger than 1 are particularly interesting. This is because we can determine the relative spacing at any point by knowing the spacing around 1. These spacings at 1 are sometimes called the “machine epsilons”, denoted $\epsilon_{\text{min}}$ and $\epsilon_{\text{max}}$ (not to be confused with $e_{\text{min}}$ and $e_{\text{max}}$). It is easy to see from the model for floating-point numbers on page 32 that

$$\epsilon_{\text{min}} = b^{-p}$$

and

$$\epsilon_{\text{max}} = b^{1-p}$$

The more conservative value, $\epsilon_{\text{max}}$, sometimes called “the machine epsilon”, $\epsilon$ or $\epsilon_{\text{mach}}$, provides an upper bound on the rounding that occurs when a floating-point number is chosen to represent a real number. A floating-point number near 1 can be chosen within $\epsilon_{\text{max}}/2$ of a real number that is near 1. This bound, $\frac{1}{2}b^{1-p}$, is called the unit roundoff.

These machine epsilons are also called the “smallest relative spacing” and the “largest relative spacing” because they can be used to determine the
relative spacing at the point \( x \). If \( x \) is not zero, the relative spacing at \( x \) is approximately

\[
\frac{x - (1 - \epsilon_{\min})x}{x}
\]
or

\[
\frac{(1 + \epsilon_{\max})x - x}{x}.
\]

Notice we say “approximately”. First of all, we do not even know that \( x \) is representable. Although \((1 - \epsilon_{\min})\) and \((1 + \epsilon_{\max})\) are members of the set of floating-point numbers by definition, that does not guarantee that the product of either of these numbers and \([x]\) is also a member of the set of floating-point numbers. However, the quantities \([(1 - \epsilon_{\min})[x]\) and \([(1 + \epsilon_{\max})[x]\) are representable (by definition of \([x]\) as a floating point number approximating the quantity within the brackets); and, in fact, they are respectively the next smallest number than \([x]\) (if \([x]\) is positive, or the next largest number otherwise), and the next largest number than \([x]\) (if \([x]\) is positive). The spacings at \([x]\) therefore are

\[
[x] - [(1 - \epsilon_{\min})[x]
\]

and

\[
(1 + \epsilon_{\max})[x] - [x].
\]

As an aside, note that this implies it is probable that

\[
[(1 - \epsilon_{\min})[x] = [(1 + \epsilon_{\min})[x].
\]

Figure 3.9. Relative Spacings

In practice, to compare two numbers \( x \) and \( y \), we must compare \([x]\) and \([y]\). We consider \( x \) and \( y \) different if

\[
|[y]| < [x] - \epsilon_{\min}[x]c,
\]
or if

\[
|[y]| > [x] + \epsilon_{\max}[x]c.
\]

The relative spacing at any point obviously depends on the value represented by the least significant digit in the significand. This digit (or bit) is called the “unit in the last place”, or “ulp”. The magnitude of an ulp depends of course on the magnitude of the number being represented. Any real number
within the range allowed by the exponent can be approximated within $\frac{1}{2}$ ulp by a floating-point number.

The subsets of numbers that we need in the computer depend on the kinds of numbers that are of interest for the problem at hand. Often, however, the kinds of numbers of interest change dramatically within a given problem. For example, we may begin with integer data in the range from 1 to 50. Most simple operations such as addition, squaring, and so on, with these data would allow a single paradigm for their representation. The fixed-point representation should work very nicely for such manipulations.

Something as simple as a factorial, however, immediately changes the paradigm. It is unlikely that the fixed-point representation would be able to handle the resulting large numbers. When we significantly change the range of numbers that must be accommodated, another change that occurs is the ability to represent the numbers exactly. If the beginning data are integers between 1 and 50, and no divisions or operations leading to irrational numbers are performed, one storage unit would almost surely be sufficient to represent all values exactly. If factorials are evaluated, however, the results cannot be represented exactly in one storage unit and so must be approximated (even though the results are integers). When data are not integers, it is usually obvious that we must use approximations, but it may also be true for integer data.

As we have indicated, different computers represent numeric data in different ways. There has been some attempt to provide standards, at least in the range representable and in the precision for floating-point quantities. There are two IEEE standards that specify characteristics of floating-point numbers (IEEE, 1985). The IEEE Standard 754 (sometimes called the “binary standard”) specifies the exact layout of the bits for two different precisions, “single” and “double”. In both cases, the standard requires that the radix be 2. For single precision, $p$ must be 24, $e_{\text{max}}$ must be 127, and $e_{\text{min}}$ must be $-126$. For double precision, $p$ must be 53, $e_{\text{max}}$ must be 1023, and $e_{\text{min}}$ must be $-1022$.

The IEEE Standard 754 also defines two additional precisions, “single extended” and “double extended”. For each of the extended precisions, the standard sets bounds on the precision and exponent ranges, rather than specifying them exactly. The extended precisions have larger exponent ranges and greater precision than the corresponding precision that is not “extended”.

The IEEE Standard 854 requires that the radix be either 2 or 10 and defines ranges for floating-point representations. Formerly, the most widely used computers (IBM System 360 and derivatives) used base 16 representation; and some computers still use this base. Additional information about the IEEE Standards for floating-point numbers can be found in Cody (1988a), Goldberg (1991), and Overton (2001).

The environmental inquiry program MACHAR by Cody (1988b) can be used to determine the characteristics of a computer’s floating-point representation and its arithmetic. The program, which is available in CALGO from
Special Floating-Point Numbers

It is convenient to be able to represent certain special numeric entities, such as infinity or “indeterminate” \((0/0)\), which do not have ordinary representations in any base-digit system. Although 8 bits are available for the exponent in the single-precision IEEE binary standard, \(e_{\text{max}} = 127\) and \(e_{\text{min}} = -126\). This means there are two unused possible values for the exponent; likewise, for the double-precision standard there are two unused possible values for the exponent. These extra possible values for the exponent allow us to represent certain special floating-point numbers. An exponent of \(e_{\text{min}} - 1\) allows us to handle 0 and the numbers between 0 and \(b^{e_{\text{min}}}\) unambiguously even though there is a hidden bit (see the discussion above about normalization and gradual underflow). The special number 0 is represented with an exponent of \(e_{\text{min}} - 1\) and a significand of \(00\ldots0\).

An exponent of \(e_{\text{max}} + 1\) allows us to represent \(\pm\infty\) or the indeterminate value. A floating-point number with this exponent and a significand of 0 represents \(\pm\infty\) (the sign bit determines the sign, as usual). A floating-point number with this exponent and a nonzero significand represents an indeterminate value such as \(\infty/\infty\). This value is called “not-a-number”, or NaN. In statistical data processing, a NaN is sometimes used to represent a missing value. Because a NaN is indeterminate, if a variable \(x\) has a value of NaN, \(x \neq x\). Also, because a NaN can be represented in different ways, however, a programmer must be careful in testing for NaNs. Some software systems provide explicit functions for testing for a NaN. The IEEE binary standard recommended that a function \(\text{isnan}\) be provided to test for a NaN. Cody and Coonen (1993) provide C programs for \(\text{isnan}\) and other functions useful in working with floating-point numbers.

Language Constructs for Representing Numeric Data

Most general-purpose computer programming languages, such as Fortran and C, provide constructs for the user to specify the type of representation for numeric quantities. These specifications are made in declaration statements that are made at the beginning of some section of the program for which they apply.

The difference between fixed-point and floating-point representations has a conceptual basis that may correspond to the problem being addressed. The
differences between other kinds of representations are often not because of conceptual differences; rather, they are the results of increasingly irrelevant limitations of the computer. The reasons there are “short” and “long”, or “signed” and “unsigned” representations do not arise from the problem the user wishes to solve; the representations are to allow for more efficient use of computer resources. The wise software designer nowadays eschews the space-saving constructs that apply to only a relatively small proportion of the data. In some applications, however, the short representations of numeric data still have a place.

In C the types of all variables must be specified with a basic declarator, which may be qualified further. For variables containing numeric data, the possible types are shown in Table 3.1.

<table>
<thead>
<tr>
<th>Basic type</th>
<th>Basic declarator</th>
<th>Fully qualified declarator</th>
</tr>
</thead>
<tbody>
<tr>
<td>fixed-point</td>
<td>int</td>
<td>signed short int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned short int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signed long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long int</td>
</tr>
<tr>
<td>floating-point</td>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td></td>
<td></td>
<td>double</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long double</td>
</tr>
</tbody>
</table>

Exactly what these types mean is not specified by the language definition, but depends on the specific implementation, which associates each type with some natural type supported by the specific computer. An unsigned quantity of either type specifies that no bit is to be used as a sign bit, which effectively doubles the largest representable number. Of course, this is essentially irrelevant for scientific computations, so unsigned integers are generally just a nuisance. If neither short nor long is specified, there is a default interpretation that is implementation-dependent. The default always favors signed over unsigned. There is a movement toward standardization of the meanings of these types. The American National Standards Institute (ANSI) and its international counterpart, the International Standards Organization (ISO) have specified standard definitions of several programming languages. ANSI (1989) is a specification of the C language. ANSI C requires that short int use at least 16 bits, that long int use at least 32 bits, and that long int is at least as long as int, which in turn is at least as long as short int. The long double type may or may not have more precision and a larger range than the double type.

C does not provide a complex data type. This deficiency can be overcome to some extent by means of a user-defined data type. The user must write
functions for all the simple arithmetic operations on complex numbers, just as is done for the simple exponentiation for floats.

The object-oriented hybrid language built on C, C++ (ANSI, 1998), provides the user the ability also to define operator functions, so that the four simple arithmetic operations can be implemented by the operators, “+”, “−”, “∗”, and “/”. There is no good way of defining an exponentiation operator, however, because the user-defined operators are limited to extended versions of the operators already defined in the language.

In Fortran variables have a default numeric type that depends on the first letter in the name of the variable. The type can be explicitly declared also. The signed and unsigned qualifiers of C, which have very little use in scientific computing, are missing in Fortran. Fortran has a fixed-point type that corresponds to integers, and two floating-point types that correspond to reals and to complex numbers. The possible types for variables containing numeric data are shown in Table 3.2.

<table>
<thead>
<tr>
<th>Basic type</th>
<th>Basic declarator</th>
<th>Default variable name</th>
</tr>
</thead>
<tbody>
<tr>
<td>fixed-point</td>
<td>integer</td>
<td>begin with i-n or i-N</td>
</tr>
<tr>
<td>floating-point</td>
<td>real</td>
<td>begin with a-h or o-z</td>
</tr>
<tr>
<td></td>
<td>double precision</td>
<td>or with A-H or O-Z</td>
</tr>
<tr>
<td>complex</td>
<td>complex</td>
<td>no default, although</td>
</tr>
<tr>
<td></td>
<td></td>
<td>d or D is sometimes used</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c or C is sometimes used</td>
</tr>
</tbody>
</table>

Although the standards organizations have defined these constructs for the Fortran language, just as is the case with C, exactly what these types mean is not specified by the language, but depends on the specific implementation. Some extensions to the language allow the number of bytes to use for a type to be specified (e.g., real*8) and allow the type double complex.

The complex type is not so much a data type as a data structure composed of two floating-point numbers that has associated operations that simulate the operations defined on the field of complex numbers.

The Fortran 95 language and subsequent versions of Fortran support the same types as earlier versions of Fortran, but also provides much more flexibility in selecting the number of bits to use in the representation of any of the basic types. A fundamental concept for the numeric types in Fortran 95 is called “kind”. The kind is a qualifier for the basic type; thus a fixed-point number may be an integer of kind 1 or of kind 2, for example. The actual value of the qualifier kind may differ from one compiler to another, so the user defines a program parameter to be the kind that is appropriate to the
3.1 Computer Storage and Manipulation of Data

range and precision required for a given variable. Fortran 95 provides the functions \texttt{selected_int_kind} and \texttt{selected_real_kind} to do this. Thus, to declare some fixed-point variables that have at least 3 decimal digits and some more fixed-point variables that have at least 8 decimal digits, the user may write the following statements

\begin{verbatim}
integer, parameter :: little = selected_int_kind(3)
integer, parameter :: big = selected_int_kind(8)
integer (little) :: ismall, jsmall
integer (big) :: itotal_accounts, igain
\end{verbatim}

The variables \texttt{little} and \texttt{big} would have integer values, chosen by the compiler designer, that could be used in the program to qualify integer types to ensure that range of numbers could be handled. Thus, \texttt{ismall} and \texttt{jsmall} would be fixed-point numbers that could represent integers between $-999$ and $999$, and \texttt{itotal_accounts} and \texttt{igain} would be fixed-point numbers that could represent integers between $-99,999,999$ and $99,999,999$. Depending on the basic hardware, the compiler may assign two bytes as \texttt{kind = little}, meaning that integers between $-32,768$ and $32,767$ could probably be accommodated by any variable, such as \texttt{ismall}, that is declared as \texttt{integer (little)}. Likewise, it is probable that the range of variables declared as \texttt{integer (big)} could handle numbers in the range $-2,147,483,648$ and $2,147,483,647$. For declaring floating-point numbers, the user can specify a minimum range and precision with the function \texttt{selected_real_kind}, which takes two arguments, the number of decimal digits of precision, and the exponent of 10 for the range. Thus, the statements

\begin{verbatim}
integer, parameter :: real4 = selected_real_kind(6,37)
integer, parameter :: real8 = selected_real_kind(15,307)
\end{verbatim}

would yield designators of floating-point types that would have either 6 decimals of precision and a range up to $10^{37}$ or 15 decimals of precision and a range up to $10^{307}$. The statements

\begin{verbatim}
real (real4) :: x, y
real (real8) :: dx, dy
\end{verbatim}

declare \texttt{x} and \texttt{y} as variables corresponding roughly to \texttt{real} on most systems, and \texttt{dx} and \texttt{dy} as variables corresponding roughly to \texttt{double precision}.

If the system cannot provide types matching the requirements specified in \texttt{selected_int_kind} or \texttt{selected_real_kind}, these functions return $-1$. Because it is not possible to handle such an error situation in the declaration statements, the user should know in advance the available ranges. Fortran 95 and subsequent versions of Fortran provide a number of intrinsic functions, such as \texttt{epsilon}, \texttt{rrspacing}, and \texttt{huge}, to use in obtaining information about the fixed- and floating-point numbers provided by the system.
Fortran 95 and 95 also provide a number of intrinsic functions for dealing with bits. These functions are essentially those specified in the MIL-STD-1753 standard of the U.S. Department of Defense. These bit functions, which have been a part of many Fortran implementations for years, provide for shifting bits within a string, extracting bits, exclusive or inclusive oring of bits, and so on. (See ANSI, 1992; Kerrigan, 1993; or Metcalf and Reid, 1999, for more extensive discussions of the types and intrinsic function provided in Fortran 95 and 95.)

Many higher-level languages and application software packages do not give the user a choice of the way to represent numeric data. The software system may consistently use a type thought to be appropriate for the kinds of applications addressed. For example, many statistical analysis application packages choose to use a floating-point representation with about 64 bits for all numeric data. Making a choice such as this yields more comparable results across a range of computer platforms on which the software system may be implemented.

Whenever the user chooses the type and precision of variables it is a good idea to use some convention to name the variable in such a way as to indicate the type and precision. Books or courses on elementary programming suggest use of mnemonic names, such as “time” for a variable that holds the measure of time. If the variable takes fixed-point values, a better name might be “itime”. It still has the mnemonic value of “time”, but it also helps us to remember that, in the computer, itime/length may not be the same thing as time/xlength. Although the variables are declared in the program to be of a specific type, the programmer can benefit from a reminder of the type. Even as we “humanize” computing, we must remember that there are details about the computer that matter. (The operator “/” is said to be “overloaded”: in a general way, it means “divide”, but it means different things depending on the contexts of the two expressions above.) Whether a quantity is a member of I or F may have major consequences for the computations, and a careful choice of notation can help to remind us of that, even if the notation may look old-fashioned.

Numerical analysts sometimes use the phrase “full precision” to refer to a precision of about 16 decimal digits, and the phrase “half precision” to refer to a precision of about 7 decimal digits. These terms are not defined precisely, but they do allow us to speak of the precision in roughly equivalent ways for different computer systems without specifying the precision exactly. Full precision is roughly equivalent to Fortran double precision on the common 32-bit workstations and to Fortran real on “supercomputer” machines such as Cray computers. Half precision corresponds roughly to Fortran real on the common 32-bit workstations. Full and half precision can be handled in a portable way in Fortran 95 and subsequent versions of Fortran. The following statements declare a variable x to be one with full precision:
3.1 Computer Storage and Manipulation of Data

In a construct of this kind, the user can define “full” or “half” as appropriate.

**Other Variations in the Representation of Data; Portability of Data**

As we have indicated already, computer designers have a great deal of latitude in how they choose to represent data. The ASCII standards of ANSI and ISO have provided a common representation for individual characters. The IEEE standard 754 referred to previously (IEEE, 1985) has brought some standardization to the representation of floating-point data, but does not specify how the available bits are to be allocated among the sign, exponent, and significand.

Because the number of bits used as the basic storage unit has generally increased over time, some computer designers have arranged small groups of bits, such as bytes, together in strange ways to form words. There are two common schemes of organizing bits into bytes and bytes into words. In one scheme, called “big end” or “big endian”, the bits are indexed from the “left”, or most significant end of the byte; and bytes are indexed within words and words are indexed within groups of words in the same direction.

In another scheme, called “little end” or “little endian”, the bytes are indexed within the word in the opposite direction. Figures 3.11 through 3.13 illustrate some of the differences, using the program shown in Figure 3.10.

```fortran
integer, parameter :: full = selected_real_kind(15,307)
real (full) :: x

character a
character*4 b
integer i, j
equivalence (b,i), (a,j)
print '(10x, a7 , a8)', ' Bits ', ' Value'
a = 'a'
print '(1x, a10, z2, 7x, a1)', 'a: ', a, a
print '(1x, a10, z8, 1x, i12)', 'j (=a): ', j, j
b = 'abcd'
print '(1x, a10, z8, 1x, a4)', 'b: ', b, b
print '(1x, a10, z8, 1x, i12)', 'i (=b): ', i, i
end
```

**Figure 3.10.** A Fortran Program Illustrating Bit and Byte Organization

These differences are important only when accessing the individual bits and bytes, when making data type transformations directly, or when moving data from one machine to another without interpreting the data in the process (“binary transfer”). One lesson to be learned from observing such
44 Numerical Computations and Algorithms

portability
XDR (external data representation)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: 61</td>
<td>a</td>
</tr>
<tr>
<td>j (=a):</td>
<td>61       97</td>
</tr>
<tr>
<td>b: 64636261</td>
<td>abcd</td>
</tr>
<tr>
<td>i (=b):</td>
<td>64636261 1684234849</td>
</tr>
</tbody>
</table>

Figure 3.11. Output from a Little Endian System (DEC VAX; Unix, VMS)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: 61</td>
<td>a</td>
</tr>
<tr>
<td>j (=a):</td>
<td>00000061 97</td>
</tr>
<tr>
<td>b: 61626364</td>
<td>abcd</td>
</tr>
<tr>
<td>i (=b):</td>
<td>64636261 1684234849</td>
</tr>
</tbody>
</table>

Figure 3.12. Output from a Little Endian System (Intel x86, Pentium, etc.; DOS, Windows 95/8)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: 61</td>
<td>a</td>
</tr>
<tr>
<td>j (=a):</td>
<td>61000000 1627389952</td>
</tr>
<tr>
<td>b: 61626364</td>
<td>abcd</td>
</tr>
<tr>
<td>i (=b):</td>
<td>61626364 1633837924</td>
</tr>
</tbody>
</table>

Figure 3.13. Output from a Big Endian System (Sun SPARC, Silicon Graphics MIPS, etc.; Unix)

subtle differences in the way the same quantities are treated in different computer systems is that programs should rarely rely on the inner workings of the computer. A program that does not be portable; that is, it will not give the same results on different computer systems. Programs that are not portable may work well on one system, and the developers of the programs may never intend for them to be used anywhere else. As time passes, however, systems change or users change systems. When that happens, the programs that were not portable may cost more than they ever saved by making use of computer-specific features.

The external data representation, or XDR, standard format, developed by Sun Microsystems for use in remote procedure calls, is a widely-used machine independent standard for binary data structures.

3.1.2 Computer Operations on Numeric Data

As we have emphasized above, the numerical quantities represented in the computer are used to simulate or approximate more interesting quantities, namely the real numbers or perhaps the integers. Obviously, because the sets (computer numbers and real numbers) are not the same, we could not de-
fine operations on the computer numbers that would yield the same field as
the familiar field of the reals. In fact, because of the nonuniform spacing of
floating-point numbers, we would suspect that some of the fundamental prop-
erties of a field may not hold. Depending on the magnitudes of the quantities
involved, it is possible, for example, that if we compute \( ab \) and \( ac \) and then
\( ab + ac \), we may not get the same thing as if we compute \( (b + c) \) and then
\( a(b + c) \). Just as we use the computer quantities to simulate real quantities,
we define operations on the computer quantities to simulate the familiar oper-
ations on real quantities. Designers of computers attempt to define computer
operations so as to correspond closely to operations on real numbers, but we
must not lose sight of the fact that the computer uses a different arithmetic
system.

The basic operational objective in numerical computing, of course, is that
a computer operation, when applied to computer numbers, yields computer
numbers that approximate the number that would be yielded by a certain
mathematical operation applied to the numbers approximated by the original
computer numbers. Just as we introduced the notation
\[ [x]_c \]
on page 32 to denote the computer floating-point number approximation to
the real number \( x \), we occasionally use the notation
\[ [\circ]_c \]
to refer to a computer operation that simulates the mathematical operation \( \circ \).
Thus,
\[ [+]_c \]
represents an operation similar to addition, but which yields a result in a set of
computer numbers. (We use this notation only where necessary for emphasis,
however, because it is somewhat awkward to use it consistently.) The failure
of the familiar laws of the field of the reals, such as distributive law cited
above, can be anticipated by noting that
\[ [(a + b)]_c = [a + b]_c, \]
or by considering the simple example in which all numbers are rounded to one
decimal and so \( \frac{1}{3} + \frac{1}{3} \neq \frac{2}{3} \) (that is, \( .3 + .3 \neq .7 \)).

The three familiar laws of the field of the reals (commutativity of addition
and multiplication, associativity of addition and multiplication, and distrib-
ution of multiplication over addition) result in the independence of the order
in which operations are performed; the failure of these laws implies that the
order of the operations may make a difference. When computer operations
are performed sequentially, we can usually define and control the sequence
fairly easily. If the computer performs operations in parallel, the resulting
differences in the orders in which some operations may be performed can occasionally yield unexpected results.

The computer operations for the two different types of computer numbers are different, and we discuss them separately.

Because the operations are not closed, special notice may need to be taken when the operation would yield a number not in the set. Adding two numbers, for example, may yield a number too large to be represented well by a computer number, either fixed-point or floating-point. When an operation yields such an anomalous result, an exception is said to exist.

**Fixed-Point Operations**

The operations of addition, subtraction, and multiplication for fixed-point numbers are performed in an obvious way that corresponds to the similar operations on the ring of integers. Subtraction is addition of the additive inverse. (In the usual two's-complement representation we described earlier, all fixed-point numbers have additive inverses except $-2^{k-1}$.) Because there is no multiplicative inverse, however, division is not multiplication by the inverse. The result of division with fixed-point numbers is the result of division with the corresponding real numbers rounded toward zero. This is not considered an exception.

As we indicated above, the set of fixed-point numbers together with addition and multiplication is not the same as the ring of integers, if for no other reason than the set is finite. Under the ordinary definitions of addition and multiplication, the set is not closed under either operation. The computer operations of addition and multiplication, however, are defined so that the set is closed. These operations occur as if there were additional higher-order bits and the sign bit were interpreted as a regular numeric bit. The result is then whatever would be in the standard number of lower-order bits. If the higher-order bits would be necessary, the operation is said to overflow. If fixed-point overflow occurs, the result is not correct under the usual interpretation of the operation, so an error situation, or an exception, has occurred. Most computer systems allow this error condition to be detected, but most software systems do not take note of the exception. The result, of course, depends on the specific computer architecture. On many systems, aside from the interpretation of the sign bit, the result is essentially the same as would result from a modular reduction. There are some special-purpose algorithms that actually use this modified modular reduction, although such algorithms would not be portable across different computer systems.

**Floating-Point Operations; Errors**

As we have seen, real numbers within the allowable range may or may not have an exact floating-point operation, and the computer operations on the computer numbers may or may not yield numbers that represent exactly the
real number that would result from mathematical operations on the numbers. If the true result is \( r \), the best we could hope for would be \([r]_e\). As we have mentioned, however, the computer operation may not be exactly the same as the mathematical operation being simulated, and further, there may be several operations involved in arriving at the result. Hence, we expect some error in the result. If the computed value is \( \tilde{r} \) (for the true value \( r \)), we speak of the absolute error,

\[ |\tilde{r} - r|, \]

and the relative error,

\[ \frac{|\tilde{r} - r|}{|r|} \]

(so long as \( r \neq 0 \)). An important objective in numerical computation obviously is to insure that the error in the result is small.

Ideally, the result of an operation on two floating-point numbers would be the same as if the operation were performed exactly on the two operands (considering them to be exact also) and then the result were rounded. Attempting to do this would be very expensive in both computational time and complexity of the software. If care is not taken, however, the relative error can be very large. Consider, for example, a floating-point number system with \( b = 2 \) and \( p = 4 \). Suppose we want to add 8 and \(-7.5\). In the floating-point system we would be faced with the problem:

\[
\begin{align*}
8 & : 1.000 \times 2^3 \\
7.5 & : 1.111 \times 2^2
\end{align*}
\]

To make the exponents the same, we have

\[
\begin{align*}
8 & : 1.000 \times 2^3 \\
7.5 & : 0.111 \times 2^3 \text{ or } 7.5 : 1.000 \times 2^3
\end{align*}
\]

The subtraction will yield either 0.000₂ or 1.000₂ \times 2⁻¹, whereas the correct value is 1.000₂ \times 2⁻¹. Either way, the absolute error is 0.5₁₀, and the relative error is 1. Every bit in the significand is wrong. The magnitude of the error is the same as the magnitude of the result. This is not acceptable. (More generally, we could show that the relative error in a similar computation could be as large as \( b - 1 \), for any base \( b \).) The solution to this problem is to use one or more guard digits. A guard digit is an extra digit in the significand that participates in the arithmetic operation. If one guard digit is used (and this is the most common situation), the operands each have \( p + 1 \) digits in the significand. In the example above, we would have

\[
\begin{align*}
8 & : 1.0000 \times 2^3 \\
7.5 & : 0.1111 \times 2^3
\end{align*}
\]

and the result is exact. In general, one guard digit can insure that the relative error is less than \( 2\epsilon_{\text{max}} \). Use of guard digits requires that the operands be
stored in special storage units. Whenever more than one operation is to be performed together, the operands and intermediate results can all be kept in the special registers to take advantage of the guard digits or even longer storage units. This is called chaining of operations.

When several numbers \( x_i \) are to be summed, it is likely that as the operations proceed serially, the magnitudes of the partial sum and the next summand will be quite different. In such a case, the full precision of the next summand is lost. This is especially true if the numbers are of the same sign. As we mentioned earlier, a computer program to implement serially the algorithm implied by \( \sum_{i=1}^{\infty} i \) will converge to some number much smaller than the largest floating-point number.

If the numbers to be summed are not all the same constant (and if they are constant, just use multiplication!), the accuracy of the summation can be increased by first sorting the numbers and summing them in order of increasing magnitude. If the numbers are all of the same sign and have roughly the same magnitude, a pairwise “fan-in” method may yield good accuracy. In the fan-in method the \( n \) numbers to be summed are added two at a time to yield \( \lceil n/2 \rceil \) partial sums. The partial sums are then added two at a time, and so on, until all sums are completed. The name “fan-in” comes from the tree diagram of the separate steps of the computations:

\[
\begin{align*}
s_1^{(1)} &= x_1 + x_2 \quad & s_2^{(1)} &= x_3 + x_4 \quad & \ldots \quad & s_{2m-1}^{(1)} &= x_{4m-3} + x_{4m-2} \quad & s_{2m}^{(1)} &= \ldots \\
s_1^{(2)} &= s_1^{(1)} + s_2^{(1)} \quad & \ldots \quad & \ldots \quad & \ldots \quad & \ldots \\
s_1^{(3)} &= s_1^{(2)} + s_2^{(2)} \\
\end{align*}
\]

It is likely that the numbers to be added will be of roughly the same magnitude at each stage. Remember we are assuming they have the same sign initially; this would be the case, for example, if the summands are squares.

Another way that is even better is due to W. Kahan (see Goldberg, 1991):

\[
s = x_1 \\
a = 0 \\
\text{for } i = 2, \ldots, n \\
\begin{array}{l}
y = x_i - a \\
t = s + y \\
a = (t - s) - y \\
s = t \end{array} \tag{3.2}
\]

Another kind of error that can result because of the finite precision used for floating-point numbers is catastrophic cancellation. This can occur when two rounded values of approximately equal magnitude and opposite signs are added. (If the values are exact, cancellation can also occur, but it is benign.) After catastrophic cancellation, the digits left are just the digits that represented the rounding. Suppose \( x \approx y \), and that \([x]_c = [y]_c\). The computed result will be zero, whereas the correct (rounded) result is \([x - y]_c\). The relative error is 100%. This error is caused by rounding, but
it is different from the “rounding error” discussed above. Although the loss of information arising from the rounding error is the culprit, the rounding would be of little consequence were it not for the cancellation.

To avoid catastrophic cancellation watch for possible additions of quantities of approximately equal magnitude and opposite signs, and consider rearranging the computations. Consider the problem of computing the roots of a quadratic polynomial, $ax^2 + bx + c$ (see Rice, 1993). In the quadratic formula,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, \quad (3.3)$$

the square root of the discriminant, $(b^2 - 4ac)$, may be approximately equal to $b$ in magnitude, meaning that one of the roots is close to zero, and, in fact, may be computed as zero. The solution is to compute only one of the roots, $x_1$, by the formula (the “−” root if $b$ is positive, and the “+” root if $b$ is negative), and then compute the other root, $x_2$ by the relationship $x_1 x_2 = c/a$.

The IEEE Binary Standard 754 (IEEE, 1985) applies not only to the representation of floating-point numbers, but also to certain operations on those numbers. The standard requires correct rounded results for addition, subtraction, multiplication, division, remaindering, and extraction of the square root. It also requires that conversion between fixed-point numbers and floating-point numbers yields correct rounded results.

The standard also defines how exceptions should be handled. The exceptions are divided into five types: overflow, division by zero, underflow, invalid operation, and inexact operation.

If an operation on floating-point numbers would result in a number beyond the range of representable floating-point numbers, the exception, called overflow, is generally very serious. (It is serious in fixed-point operations, also, if it is unplanned. Because we have the alternative of using floating-point numbers if the magnitude of the numbers is likely to exceed what is representable in fixed-point, the user is expected to use this alternative. If the magnitude exceeds what is representable in floating-point, however, the user must resort to some indirect means, such as scaling, to solve the problem.)

Division by zero does not cause overflow; it results in a special number if the dividend is nonzero. The result is either $\infty$ or $-\infty$, and these have special representations, as we have seen.

Underflow occurs whenever the result is too small to be represented as a normalized floating-point number. As we have seen, a nonnormalized representation can be used to allow a gradual underflow.

An invalid operation is one for which the result is not defined because of the value of an operand. The invalid operations are addition of $\infty$ to $-\infty$, multiplication of $\pm \infty$ and 0, 0 divided by 0 or by $\pm \infty$, $\pm \infty$ divided by 0 or by $\pm \infty$, extraction of the square root of a negative number (some systems, such as Fortran, have a special type for complex numbers and deal correctly with them), and remaindering any quantity with 0 or remaindering $\pm \infty$ with any quantity. An invalid operation results in a NaN. Any operation with a NaN also results in a NaN. Some systems distinguish two types of NaN, a “quiet NaN” and a “signaling NaN”.

An inexact operation is one for which the result must be rounded. For example, if all $p$ bits of the significand are required to represent both the multiplier and root of a function zero of a function IEEE standards overflow, in computer operations exception, in computer operations infinity, floating-point representation underflow, in computer operations exception, in computer operations gradual underflow infinity, floating-point representation NaN (“not-a-number”)
error, rounding error, cancellation error, rounding error, exact computations, error-free computations, residue arithmetic, overloading, infix operator, C (programming language)

multiplicand, approximately $2^p$ bits would be required to represent the product. Because only $p$ are available, however, the result must be rounded.

Conformance to the IEEE Binary Standard 754 does not ensure that the results of a floating-point computation will be the same on all computers. The slight differences are usually unimportant, but Blackford et al. (1997) describe some examples of problems that occurred when computations were performed in parallel using a heterogeneous network of computers all of which conformed to the IEEE standard.

**Exact Computations; Rational Fractions**

If the input data can be represented exactly as rational fractions, it may be possible to preserve exact values of the results of computations. Use of rational fractions allows avoidance of reciprocation, which is the operation that most commonly yields a nonrepresentable value from one that is representable. Of course, any addition or multiplication that increases the magnitude of an integer in a rational fraction beyond a value that can be represented exactly (that is, beyond approximately $2^{23}$, $2^{31}$, or $2^{53}$, depending on the computing system), may break the error-free chain of operations. Exact computations with integers can be carried out using *residue arithmetic*, in which each quantity is as a vector of residues, all from a vector of relatively prime moduli. (See the discussion of modular arithmetic beginning on page 120 for some basic definitions; see Szabó and Tanaka, 1967, for discussion of the use of residue arithmetic in numerical computations; and see Stallings and Boullion, 1972, and Keller-McNulty and Kennedy, 1986, for applications of this technology in matrix computations.)

Computations with rational fractions are sometimes performed using a fixed-point representation. Gregory and Krishnamurthy (1984) discuss in detail these and other methods for performing error-free computations.

**Language Constructs for Operations on Numeric Data**

Most general-purpose computer programming languages, such as Fortran and C, provide constructs for operations that correspond to the common operations on scalar numeric data, such as “+”, “−”, “∗” (multiplication), and “/”. These operators simulate the corresponding mathematical operations. As we mentioned on page 45, we will occasionally use a notation such as $[+]_c$

to indicate the computer operator. The operators have slightly different meanings depending on the operand objects; that is, the operations are “overloaded”. Most of these operators are *binary infix* operators, meaning that the operator is written between the two operands.

Some languages provide operations beyond the four basic scalar arithmetic operations. C provides some specialized operations, such as the unary postfix increment “++” and decrement “--” operators, for trivial common operations; but does not provide an operator for exponentiation. (Exponentiation is handled by a function provided in a standard supplemental library in C, `<math.h>`.) C also overloads the basic multiplication operator so that it can indicate a change of the meaning of a
variable, in addition to indicating the multiplication of two scalar numbers. A standard library in C (<signal.h>) allows for easy handling of arithmetic exceptions. With this facility, for example, the user can distinguish a quiet NaN from a signaling NaN.

The C language does not directly provide for operations on special data structures. For operations on complex data, for example, the user must define the type and its operations in a header file (or else, of course, just do the operations as if they were operations on an array of length 2).

Fortran provides the four basic scalar numeric operators, plus an exponentiation operator ("**"). (Exactly what this operator means may be slightly different in different versions of Fortran. Some versions interpret the operator always to mean

1. take log
2. multiply by power
3. exponentiate

if the base and the power are both floating-point types. This, of course, would not work if the base is negative, even if the power is an integer. Most versions of Fortran will determine at run time if the power is an integer, and use repeated multiplication if it is.)

Fortran also provides the usual five operators for complex data (the basic four, plus exponentiation). Fortran 95 and subsequent versions of Fortran provide the same set of scalar numeric operators, plus a basic set of array and vector/matrix operators. The usual vector/matrix operators are implemented as functions, or prefix operators, in Fortran 95.

In addition to the basic arithmetic operators, both Fortran and C, as well as other general programming languages, provide several other types of operators, including relational operators and operators for manipulating structures of data.

Software packages have been built on Fortran and C to extend their accuracy. Two ways in which this is done are by use of multiple precision (see Brent, 1978; Smith, 1991; Bailey, 1993 and 1995; and Doman, Pursglove, and Coen, for example) and by use of interval arithmetic (see Yohe, 1979; Kulisch, 1983; Kulisch and Miranker, 1981 and 1983; and Jaulin et al., 2001, for example). Multiple precision operations are performed in the software by combining more than one computer storage unit to represent a single number. For example, to operate on $x$ and $y$, we may represent $x$ as $a \cdot 10^p + b$ and $y$ as $c \cdot 10^p + d$. The product $xy$ then is formed as $ac \cdot 10^{2p} + (ad + bc) \cdot 10^p + bd$. The representation is chosen so that any of the coefficients of the scaling factors (in this case powers of 10) can be represented to within the desired accuracy.

Multiple precision is different from “extended precision” discussed earlier; extended precision is implemented at the hardware level or at the microcode level. A multiple precision package may allow the user to specify the number of digits to use in representing data and performing computations. The software packages for symbolic computations, such as Maple, generally provide multiple precision capabilities.

Interval arithmetic maintains intervals in which the exact data and solution are known to lie. Instead of working with single-point approximations, for which we used notation such as

$[x]_c$

on page 32 for the value of floating-point approximation to the real number $x$, and

$[0]_c$
interval arithmetic

on page 45 for the simulated operation \( \circ \), we can approach the problem by identifying a closed interval in which \( x \) lies and a closed interval in which the result of the operation \( \circ \) lies. We denote the interval operation as 

\[
\lbrack \circ \rbrack_I.
\]

For the real number \( x \), we identify two floating-point numbers, \( x_l \) and \( x_u \), such that

\[ x_l \leq x \leq x_u. \]

(This relationship also implies \( x_l \leq [x]_c \leq x_u. \) The real number \( x \) is then considered to be the interval \( [x_l, x_u] \). For this approach to be useful, of course, we seek tight bounds. If \( x = [x]_c \), the best interval is degenerate. In other cases either \( x_l \) or \( x_u \) is \( [x]_c \) and the length of the interval is the floating-point spacing from \( [x]_c \) in the appropriate direction.

Addition and multiplication in interval arithmetic yields intervals:

\[
x + I y = [x_l + y_l, x_u + y_u]
\]

and

\[
x I y = [\min(x_l y_l, x_l y_u, x_u y_l, x_u y_u), \max(x_l y_l, x_l y_u, x_u y_l, x_u y_u)].
\]

Change of sign results in \( [-x_u, -x_l] \) and if \( 0 \not\in [x_l, x_u] \), reciprocation results in \( [1/x_u, 1/x_l] \). See Moore (1979) or Alefeld and Herzberger (1983) for an extensive treatment of interval arithmetic. The journal Reliable Computing is devoted to interval computations.

The ACRITH package of IBM (see Jansen and Weidner, 1986) is a library of Fortran subroutines that perform computations in interval arithmetic and also in extended precision. Kearfott et al. (1994) have produced a portable Fortran library of basic arithmetic operations and elementary functions in interval arithmetic, and Kearfott (1996) gives a Fortran 90 module defining an interval data type. Jaulin et al. (2001) give additional source of software.

### 3.2 Algorithms

We will use the term “algorithm” rather loosely, but always in the general sense of a method or a set of instructions for doing something. Algorithms are sometimes distinguished as “numerical”, “semi-numerical”, and “non-numerical”, depending on the extent to which operations on real numbers are simulated. We discuss numerical algorithms in Section 3.2.1 and semi- and non-numerical algorithms in Section 3.2.2. In Section 3.3 we discuss various types of computer architectures and how these may affect the performance of algorithms In Section 3.3.3 we address some specific issues that arise in parallel processing.

**Algorithms and Programs**

Algorithms are expressed by means of a flowchart, a series of steps, or in a computer language or pseudolanguage. The expression in a computer language is a source program or module; hence, we sometimes use the words “algorithm” and “program” synonymously.

The program is the set of computer instructions that implement the algorithm. A poor implementation can render a good algorithm useless. A good implementation
will preserve the algorithm’s accuracy and efficiency, and will detect data that are inappropriate for the algorithm. Robustness is more a property of the program than of the algorithm.

The exact way an algorithm is implemented in a program depends of course on the programming language, but it also may depend on the computer and associated system software. A program that will run on most systems without modification is said to be portable.

3.2.1 Numerical Algorithms and Analysis

We will use the term “algorithm” rather loosely, but always in the general sense of a method or a set of instructions for doing something. Algorithms are sometimes distinguished as “numerical”, “semi-numerical”, and “non-numerical”, depending on the extent to which operations on real numbers are simulated.

Algorithms and Programs

Algorithms are expressed by means of a flowchart, a series of steps, or in a computer language or pseudolanguage. The expression in a computer language is a source program or module; hence, we sometimes use the words “algorithm” and “program” synonymously.

The program is the set of computer instructions that implement the algorithm. A poor implementation can render a good algorithm useless. A good implementation will preserve the algorithm’s accuracy and efficiency, and will detect data that are inappropriate for the algorithm. Robustness is more a property of the program than of the algorithm.

The exact way an algorithm is implemented in a program depends of course on the programming language, but it also may depend on the computer and associated system software. A program that will run on most systems without modification is said to be portable.

The two most important aspects of a computer algorithm are its accuracy and its efficiency. Although each of these concepts appears rather simple on the surface, each is actually fairly complicated, as we shall see.

Error in Numerical Computations

An “accurate” algorithm is one that gets the “right” answer. Knowing that the right answer may not be representable, and rounding within a set of operations may result in variations in the answer, we often must settle for an answer that is “close”. (An alternative to settling for a “close” answer is to express the question slightly differently. See Exercise 8.5 in Chapter 8, page 478.) As we have discussed previously, we measure error, or closeness, either as the absolute error or the relative error of a computation.

Another way of considering the concept of “closeness” is by looking backward from the computed answer, and asking what perturbation of the original problem would yield the computed answer exactly. This approach, developed by Wilkinson (1963), is called backward error analysis. The backward analysis is followed by
an assessment of the effect of the perturbation on the solution. Although backward error analysis may not seem as natural as the “forward” analysis (in which we assess the difference in the computed and true solutions), it is easier to perform because all operations in the backward analysis are performed in $\mathbb{F}$ instead of in $\mathbb{R}$. Each step in the backward analysis involves numbers in the set $\mathbb{F}$, that is, numbers that could actually have participated in the computations that were performed. Because the properties of the arithmetic operations in $\mathbb{R}$ do not hold, and at any step in the sequence of computations, the result in $\mathbb{F}$ may not exist in $\mathbb{R}$, it is very difficult to carry out a forward error analysis.

There are other complications in assessing errors. Suppose the answer is a vector, such as a solution to a linear system. What norm do we use to compare closeness of vectors? Another, more complicated situation for which assessing correctness may be difficult is random number generation. It would be difficult to assign a meaning to “accuracy” for such a problem.

The basic source of error in numerical computations is the inability to work with the reals. The field of reals is simulated with a finite set. This has several consequences. A real number is rounded to a floating-point number; the result of an operation on two floating-point numbers is rounded to another floating-point number; and passage to the limit, which is a fundamental concept in the field of reals, is not possible in the computer.

Rounding errors that occur just because the result of an operation is not representable in the computer’s set of floating-point numbers are usually not too bad. Of course, if they accumulate through the course of many operations, the final result may have an unacceptably large accumulated rounding error.

A natural approach to studying errors in floating-point computations is to define random variables for the rounding at all stages, from the initial representation of the operands through any intermediate computations to the final result. Given a probability model for the rounding error in representation of the input data, a statistical analysis of rounding errors can be performed. Wilkinson (1963) introduced a uniform probability model for rounding of input, and derived distributions for computed results based on that model. Linna¨maa (1975) discusses the effects of accumulated error in floating-point computations based on a more general model of the rounding for the input. This approach leads to a forward error analysis that provides a probability distribution for the error in the final result. Analysis of errors in fixed-point computations present altogether different problems. For a discussion, see Bareiss and Barlow (1980).

The obvious probability model for floating-point representations is that the reals within an interval between any two floating-point numbers have a uniform distribution (see Figure 3.4, page 34, and Calvetti, 1991). A probability model for the real line can be built up as a mixture of the uniform distributions (see Exercise 3.9, page 111). The density is obviously 0 in the tails. While a model based on simple distributions may be appropriate for the rounding error due to finite-precision representation of real numbers, probability models for rounding errors in floating point computations are not so simple. This is because the rounding errors in computations are not random. See Chaitin-Chatelin and Frayssé (1996) for further discussion of probability models for rounding errors. Dempster and Rubin (1983) discuss the application of statistical methods for dealing with grouped data to the data resulting from rounding in floating-point computations.
Another, more pernicious effect of rounding can occur in a single operation, resulting in catastrophic cancellation, as we have discussed previously.

### Measures of Error and Bounds for Errors

For the simple case of representing the real number $r$ by an approximation $\tilde{r}$, we define absolute error, $|\tilde{r} - r|$, and relative error, $|\tilde{r} - r|/|r|$ (so long as $r \neq 0$). These same types of measures are used to express the errors in numerical computations. As we indicated above, however, the result may not be a simple real number; it may consist of several real numbers. For example, in statistical data analysis, the numerical result, $\tilde{r}$, may consist of estimates of several regression coefficients, various sums of squares and their ratio, and several other quantities. We may then be interested in some more general measure of the difference of $\tilde{r}$ and $r$,

$$\Delta(\tilde{r}, r),$$

where $\Delta(\cdot, \cdot)$ is a nonnegative, real-valued function. This is the absolute error, and the relative error is the ratio of the absolute error to $\Delta(r, r_0)$, where $r_0$ is a baseline value, such as 0. When $r$, instead of just being a single number, consists of several components, we must measure error differently. If $r$ is a vector, the measure may be some norm, such as we will discuss in Chapter 5, and in that case, $\Delta(\tilde{r}, r)$ may be denoted by $\|\tilde{r} - r\|$. A norm tends to become larger as the number of elements increases, so instead of using a raw norm, it may be appropriate to scale the norm to reflect the number of elements being computed.

However the error is measured, for a given algorithm we would like to have some knowledge of the amount of error to expect or at least some bound on the error. Unfortunately, almost any measure contains terms that depend on the quantity being evaluated. Given this limitation, however, often we can develop an upper bound on the error. In other cases, we can develop an estimate of an “average error”, based on some assumed probability distribution of the data comprising the problem. Another measure we will encounter in Monte Carlo methods in Section 2.1.1, page 7, is based on the “variance” of a “random” quantity. (The quotation marks are used because the quantities are not really random, or at least we do not contemplate a probability model for them.) In a Monte Carlo method we estimate the solution based on a “random” sample, so just as in ordinary statistical estimation, we are concerned about the variance of the estimate. We can usually derive expressions for the variance of the estimator in terms of the quantity being evaluated, and of course we can estimate the variance of the estimator using the realized random sample. The standard deviation of the estimator provides an indication of the distance around the computed quantity within which we may have some confidence that the true value lies. The standard deviation is sometimes called the “standard error”, and nonstatisticians speak of it as a “probabilistic error bound”.

It is often useful to identify the “order of the error”, whether we are concerned about error bounds, average expected error, or the standard deviation of an estimator. In general, we speak of the order of one function in terms of another function, as the argument of the functions approach a given value. A function $f(t)$ is said to be of order $g(t)$ at $t_0$, written $O(g(t))$ (“big $O$ of $g(t)$”), if there exists a positive constant $M$ such that

$$|f(t)| \leq M|g(t)| \quad \text{as } t \to t_0.$$
This is the order of convergence of one function to another function at a given point.

If our objective is to compute \( f(t) \) and we use an approximation \( \hat{f}(t) \), the order of the error due to the approximation is the order of the convergence. In this case, the argument of the order of the error may be some variable that defines the approximation. For example, if \( \hat{f}(t) \) is a finite series approximation to \( f(t) \) using, say, \( k \) terms, we may express the error as \( O(h(k)) \), for some function \( h(k) \). Typical orders of errors due to the approximation may be \( O(1/k) \), \( O(1/k^2) \), or \( O(1/k!) \). An approximation with order of error \( O(1/k!) \) is to be preferred over one order of error \( O(1/k) \) because the error is decreasing more rapidly. The order of error due to the approximation is only one aspect to consider; roundoff error in the representation of any intermediate quantities must also be considered.

We will discuss the order of error in iterative algorithms further in the section beginning on page 64. We will discuss order also in measuring the speed of an algorithm in the section beginning on page 60.

The special case of convergence to the constant zero is often of interest. A function \( f(t) \) is said to be “little o of \( g(t) \)” at \( t_0 \), written \( o(g(t)) \), if

\[
\frac{f(t)}{g(t)} \to 0 \quad \text{as} \quad t \to t_0.
\]

If the function \( f(t) \) approaches 0 at \( t_0 \), \( g(t) \) can be taken as a constant and \( f(t) \) is said to be \( o(1) \).

Big O and little o convergence are defined in terms of dominating functions. In the analysis of algorithms it is often useful to consider analogous types of convergence in which the function of interest dominates another function. This type of relationship is similar to a lower bound. A function \( f(t) \) is said to be \( \Omega(g(t)) \) (“big omega of \( g(t) \)”), if there exists a positive constant \( m \) such that

\[
|f(t)| \geq m|g(t)| \quad \text{as} \quad t \to t_0.
\]

Likewise, a function \( f(t) \) is said to be “little omega of \( g(t) \)” at \( t_0 \), written \( \omega(g(t)) \), if

\[
\frac{g(t)}{f(t)} \to 0 \quad \text{as} \quad t \to t_0.
\]

Usually the limit on \( t \) in order expressions is either 0 or \( \infty \), and because it is obvious from the context, mention of it is omitted. The order of the error in numerical computations usually provides a measure in terms of something that can be controlled in the algorithm, such as the point at which an infinite series is truncated in the computations. The measure of the error usually also contains expressions that depend on the quantity being evaluated, however.

**Sources of Error in Numerical Computations**

Some algorithms are exact, such as an algorithm to multiply two matrices that just uses the definition of matrix multiplication. Other algorithms are approximate because the result to be computed does not have a finite closed-form expression. An example is the evaluation of the normal cumulative distribution function. One way of evaluating this is by use of a rational polynomial approximation to the distribution function. Such an expression may be evaluated with very little rounding error, but the expression has an error of approximation.
We need to have some knowledge of the magnitude of the error. For algorithms that use approximations, it is often useful to express the order of the error in terms of some quantity used in the algorithm or in terms of some aspect of the problem itself.

When solving a differential equation on the computer, the differential equation is often approximated by a difference equation. Even though the differences used may not be constant, they are finite and the passage to the limit can never be effected. This kind of approximation leads to a discretization error. The amount of the discretization error has nothing to do with rounding error. If the last differences used in the algorithm are $\delta t$, then the error is usually of order $O(\delta t)$, even if the computations are performed exactly.

Another type of error occurs when the algorithm uses a series expansion. The infinite series may be exact, and in principle the evaluation of all terms would yield an exact result. The algorithm uses only a finite number of terms, and the resulting error is truncation error. When a truncated Taylor’s series is used to evaluate a function at a given point $x_0$, the order of the truncation error is the derivative of the function that would appear in the first unused term of the series, evaluated at $x_0$.

**Algorithms and Data**

The performance of an algorithm may depend on the data. We have seen that even the simple problem of computing the roots of a quadratic polynomial, $ax^2 + bx + c$, using the quadratic formula, equation (3.3), can lead to severe cancellation. For many values of $a$, $b$, and $c$, the quadratic formula works perfectly well. Data that are likely to cause computational problems are referred to as ill-conditioned data, and, more generally, we speak of the “condition” of data. The concept of condition is understood in the context of a particular set of operations. Heuristically, data for a given problem are ill-conditioned if small changes in the data may yield large changes in the solution.

Consider the problem of finding the roots of a high-degree polynomial, for example. Wilkinson (1959) gave an example of a polynomial that is very simple on the surface, yet whose solution is very sensitive to small changes of the values of the coefficients:

$$f(x) = (x - 1)(x - 2) \cdots (x - 20) = x^{20} - 210x^{19} + \cdots + 20!$$

While the solution is easy to see from the factored form, the solution is very sensitive to perturbations of the coefficients. For example changing the coefficient $210$ to $210 + 2^{-23}$ changes the roots drastically; in fact, 10 of them are now complex. Of course the extreme variation in the magnitudes of the coefficients should give us some indication that the problem may be ill-conditioned.

We attempt to quantify the condition of a set of data for a particular set of operations by means of a condition number. Condition numbers are defined to be positive and so that large values of the numbers means that the data or problems are ill-conditioned. A useful condition number for the problem of finding roots of a function can be defined in terms of the derivative of the function in the vicinity of a discretization error, discretization error, truncation error, truncation error, truncation condition (problem or data), ill-conditioned (problem or data), ill-conditioned data, condition number.
root. We will also see that condition numbers must be used with some care. For example, according to the condition number for finding roots, Wilkinson’s polynomial is well-conditioned.

In the solution of a linear system of equations, the coefficient matrix determines the condition of this problem. In Sections 5.1.1 and 5.2.4 we will consider a condition number for a matrix with respect to the problem of solving a linear system of equations.

The ability of an algorithm to handle a wide range of data, and either to solve the problem as requested or to determine that the condition of the data does not allow the algorithm to be used is called the robustness of the algorithm. Another concept that is quite different from robustness is stability. An algorithm is said to be stable if it always yields a solution that is an exact solution to a perturbed problem; that is, for the problem of computing \( f(x) \) using the input data \( x \), an algorithm is stable if the result it yields, \( \tilde{f}(x) \), is \( f(x + \delta x) \) for some (bounded) perturbation \( \delta x \) of \( x \). Stated another way, an algorithm is stable if small perturbations in the input or in intermediate computations do not result in large differences in the results. The concept of stability, for an algorithm, should be contrasted with the concept of condition, for a problem or a dataset. If a problem is ill-conditioned, a stable algorithm (a “good algorithm”) will produce results with large differences for small differences in the specification of the problem. This is because the exact results have large differences. An algorithm that is not stable, however, may produce large differences for small differences in the computer description of the problem, which may involve rounding, truncation, or discretization, or for small differences in the intermediate computations performed by the algorithm.

The concept of stability arises from backward error analysis. The stability of an algorithm may depend on how continuous quantities are discretized, as when a range is gridded for solving a differential equation. See Higham (2002) for an extensive discussion of stability.

### Reducing the Error in Numerical Computations

An objective in designing an algorithm to evaluate some quantity is to avoid accumulated rounding error and to avoid catastrophic cancellation. In the discussion of floating-point operations above, we have seen two examples of how an algorithm can be constructed to mitigate the effect of accumulated rounding error (using equations (3.2), page 48, for computing a sum) and to avoid possible catastrophic cancellation in the evaluation of the expression (3.3) for the roots of a quadratic equation.

Another example familiar to statisticians is the computation of the sample sum of squares:

\[
\sum_{i=1}^{n} (x_i - \bar{x})^2 = \sum_{i=1}^{n} x_i^2 - n\bar{x}^2 \tag{3.4}
\]

This quantity is \((n - 1)s^2\), where \(s^2\) is the sample variance.

Either expression in equation (3.4) can be thought of as describing an algorithm. The expression on the left implies the “two-pass” algorithm:
3.2 Algorithms

\[ a = x_1 \]
\[
\text{for } i = 2, \ldots, n
\]
\[
\begin{align*}
\{ & \\
& a = x_i + a \\
& b = (x_1 - a)^2 \\
& \text{for } i = 2, \ldots, n
\end{align*}
\]
\[
\begin{align*}
& b = (x_i - a)^2 + b \\
& a = a/n \\
& b = (x_1 - a)^2 = b
\end{align*}
\]

(3.5)

This algorithm yields \( \bar{x} = a \) and \((n - 1)s^2 = b\). Each of the sums computed in this algorithm may be improved by use of equations (3.2). A problem with this algorithm is the fact that it requires two passes through the data. Because the quantities in the second summation are squares of residuals, they are likely to be of relatively equal magnitude. They are of the same sign, so there will be no catastrophic cancellation in the early stages when the terms being accumulated are close in size to the current value of \( b \). There will be some accuracy loss as the sum \( b \) grows, but the addends \((x_i - a)^2\) remain roughly the same size. The accumulated rounding error, however, may not be too bad.

The expression on the right of equation (3.4) implies the “one-pass” algorithm:

\[ a = x_1 \]
\[
\text{for } i = 2, \ldots, n
\]
\[
\begin{align*}
\{ & \\
& a = x_i + a \\
& b = x_i^2 + b \\
& a = a/n \\
& b = b - na^2
\end{align*}
\]

(3.6)

This algorithm requires only one pass through the data, but if the \( x_i \)'s have magnitudes larger than 1, the algorithm has built up two relatively large quantities, \( b \) and \( na^2 \). These quantities may be of roughly equal magnitude; subtracting one from the other may lead to catastrophic cancellation. See Exercise 3.16, page 112.

Another algorithm is shown in (3.7). It requires just one pass through the data, and the individual terms are generally accumulated fairly accurately.

\[ a = x_1 \]
\[
\text{for } i = 2, \ldots, n
\]
\[
\begin{align*}
\{ & \\
& d = (x_i - a)/i \\
& a = d + a \\
& b = i(i - 1)a^2 + b
\end{align*}
\]

(3.7)

Chan and Lewis (1979) propose a condition number to quantify the sensitivity in \( s \), the sample standard deviation, to the data, the \( x_i \)'s. Their condition number

condition number with respect to computing a sample standard deviation

condition number
It is clear that if the mean is large relative to the variance, this condition number will be large. (Recall that large condition numbers imply ill-conditioning; and also recall that condition numbers must be interpreted with some care.) Notice that this condition number achieves its minimum value of 1 for the data $x_i - \bar{x}$, so if the computations for $\hat{x}$ and $x_i - \hat{x}$ were exact, the data in the last part of the algorithm in (3.5) would be perfectly conditioned. A dataset with a large mean relative to the variance is said to be stiff.

Often when a finite series is to be evaluated, it is necessary to accumulate a set of terms of the series that have similar magnitude, and then combine this with similar partial sums. It may also be necessary to scale the individual terms by some very large or very small multiplicative constant while the terms are being accumulated, and then remove the scale after some computations have been performed.

Chan, Golub, and LeVeque (1982) propose a modification of the algorithm in (3.7) to use pairwise accumulations (as in the fan-in method discussed previously). Chan, Golub, and LeVeque (1983) make extensive comparisons of the methods, and give error bounds based on the condition number.

**Efficiency**

The efficiency of an algorithm refers to its usage of computer resources. The two most important resources are the processing units and memory. The amount of time the processing units are in use and the amount of memory required are the key measures of efficiency. A limiting factor for the time the processing units are in use is the number and type of operations required. Some operations take longer than others; for example, the operation of adding floating-point numbers may take more time than the operation of adding fixed-point numbers. This, of course, depends on the computer system and on what kinds of floating-point or fixed-point numbers we are dealing with. If we have a measure of the size of the problem, we can characterize the performance of a given algorithm by specifying the number of operations of each type, or just the number of operations of the slowest type.

If more than one processing unit is available, it may be possible to perform operations simultaneously. In this case the amount of time required may be drastically smaller for an efficient parallel algorithm than it would for the most efficient serial algorithm that utilizes only one processor at a time. An analysis of the efficiency must take into consideration how many processors are available, how many computations can be performed in parallel, and how often they can be performed in parallel.

Often instead of the exact number of operations, we use the order of the number of operations in terms of the measure of problem size. If $n$ is some measure of the size of the problem, an algorithm has order $O(f(n))$ if, as $n \to \infty$, the number of computations $\to cf(n)$, where $c$ is some constant. For example, to multiply two $n \times n$ matrices in the obvious way requires $O(n^3)$ multiplications and additions; to multiply an $n \times m$ matrix and an $m \times p$ matrix requires $O(nmp)$ multiplications and additions. In the latter case, $n$, $m$, and $p$ are all measures of the size of the problem.

Notice that in the definition of order there is a constant $c$. Two algorithms that have the same order may have different constants, and in that case are said to
“differ only in the constant”. The order of an algorithm is a measure of how well the algorithm “scales”; that is, the extent to which the algorithm can deal with truly large problems.

Let \( n \) be a measure of the problem size, and let \( b \) and \( q \) be constants. An algorithm of order \( O(b^n) \) has exponential order, one of order \( O(n^q) \) has polynomial order, and one of order \( O(\log n) \) has log order. Notice that for log order, it does not matter what the base is. Also, notice that \( O(\log n^q) = O(\log n) \). For a given task with an obvious algorithm that has polynomial order, it is often possible to modify the algorithm to address parts of the problem so that in the order of the resulting algorithm one \( n \) factor is replaced by a factor of \( \log n \).

Although it is often relatively easy to determine the order of an algorithm, an interesting question in algorithm design involves the order of the problem, that is, the order of the most efficient algorithm possible. A problem of polynomial order is usually considered tractable, whereas one of exponential order may require a prohibitively excessive amount of time for its solution. An interesting class of problems are those for which a solution can be verified in polynomial time, yet for which no polynomial algorithm is known to exist. Such a problem is called a nondeterministic polynomial, or NP, problem. “Nondeterministic” does not imply any randomness; it refers to the fact that no polynomial algorithm for determining the solution is known. Most interesting NP problems can be shown to be equivalent to each other in order by reductions that require polynomial time. Any problem in this subclass of NP problems is equivalent in some sense to all other problems in the subclass and so such a problem is said to be NP-complete. (See Garey and Johnson, 1979, for a complete discussion of NP-completeness.)

For many problems it is useful to measure the size of a problem in some standard way and then to identify the order of an algorithm for the problem with separate components. A common measure of the size of a problem is \( L \), the length of the stream of data elements. An \( n \times n \) matrix would have length proportional to \( L = n^2 \), for example. To multiply two \( n \times n \) matrices in the obvious way requires \( O(L^{3/2}) \) multiplications and additions, as we mentioned above.

In analyzing algorithms for more complicated problems, we may wish to determine the order in the form

\[
O(f(n)g(L)),
\]

because \( L \) is an essential measure of the problem size, and \( n \) may depend on how the computations are performed. For example, in the linear programming problem, with \( n \) variables and \( m \) constraints with a dense coefficient matrix, there are order \( nm \) data elements. Algorithms for solving this problem generally depend in the limit on \( n \), so we may speak of a linear programming algorithm as being \( O(n^3L) \), for example, or of some other algorithm as being \( O(\sqrt{n}L) \). (In defining \( L \), it is common to consider the magnitudes of the data elements or the precision with which the data are represented, so that \( L \) is the order of the total number of bits required to represent the data. This level of detail can usually be ignored, however, because the limits involved in the order are generally not taken on the magnitude of the data, only on the number of data elements.)

The order of an algorithm (or, more precisely, the “order of operations of an algorithm”) is an asymptotic measure of the operation count as the size of the problem goes to infinity. The order of an algorithm is important, but in practice the actual count of the operations is also important. In practice, an algorithm whose operation count is approximately \( n^2 \) may be more useful than one whose count is
1000(n \log n + n), although the latter would have order \( O(n \log n) \), which is much better than that of the former, \( O(n^2) \). When an algorithm is given a fixed-size task many times, the finite efficiency of the algorithm becomes very important.

The number of computations required to perform some tasks depends not only on the size of the problem, but also on the data. For example, for most sorting algorithms, it takes fewer computations (comparisons) to sort data that are already almost sorted than it does to sort data that are completely unsorted. We sometimes speak of the \textit{average} time and the \textit{worst-case} time of an algorithm. For some algorithms these may be very different, whereas for other algorithms or for some problems these two may be essentially the same.

Our main interest is usually not in how many computations occur, but rather in how long it takes to perform the computations. Because some computations can take place simultaneously, even if all kinds of computations required the same amount of time, the \textit{order of time} may be different from the order of the number of computations.

The actual number of floating-point operations divided by the time required to perform the operations is called the FLOPS (floating-point operations per second) rate. Confusingly, “FLOP” also means “floating-point operation”, and “FLOPs” is the plural of “FLOP”. Of course, as we tend to use lowercase more often, we must use the context to distinguish “flops” as a rate from “flops”, the plural of “flop”.

In addition to the actual processing, the data may need to be copied from one storage position to another. Data movement slows the algorithm, and may cause it not to use the processing units to their fullest capacity. When groups of data are being used together, blocks of data may be moved from ordinary storage locations to an area from which they can be accessed more rapidly. The efficiency of a program is enhanced if all operations that are to be performed on a given block of data are performed one right after the other. Sometimes a higher-level language prevents this from happening. For example, to add two arrays (matrices) in Fortran 95, a single statement is sufficient:

\[
A = B + C
\]

Now, if also we want to add \( B \) to the array \( E \) we may write:

\[
A = B + C \\
D = B + E
\]

These two Fortran 95 statements together may be less efficient than writing a traditional loop in Fortran or in C, because the array \( B \) may be accessed a second time needlessly. (Of course, this is relevant only if these arrays are very large.)

**Improving Efficiency**

There are many ways to attempt to improve the efficiency of an algorithm. Often the best way is just to look at the task from a higher level of detail, and attempt to construct a new algorithm. Many obvious algorithms are serial methods that would be used for hand computations, and so are not the best for use on the computer.

An effective general method of developing an efficient algorithm is called \textit{divide and conquer}. In this method, the problem is broken into subproblems, each of which is solved, and then the subproblem solutions are combined into a solution for the
original problem. In some cases, this can result in a net savings either in the number of computations, resulting in improved order of computations, or in the number of computations that must be performed serially, resulting in improved order of time.

Let the time required to solve a problem of size $n$ be $t(n)$, and consider the recurrence relation

$$t(n) = pt(n/p) + cn,$$

for $p$ positive and $c$ nonnegative. Then $t(n) = O(n \log n)$ (see Exercise 3.18, page 113). Divide and conquer strategies can sometimes be used together with a simple method that would be $O(n^2)$ if applied directly to the full problem to reduce the order to $O(n \log n)$. The effect of such a reduction is dramatically apparent from Table 3.3 on page 74.

The “fan-in algorithm” is an example of a divide and conquer strategy that allows $O(n)$ operations to be performed in $O(\log n)$ time if the operations can be performed simultaneously. The number of operations does not change materially; the improvement is in the time.

Although there have been orders of magnitude improvements in the speed of computers because the hardware is better, the order of time required to solve a problem is dependent almost entirely on the algorithm. The improvement in efficiency resulting from hardware improvements are generally differences only in the constant. The practical meaning of the order of the time must be considered, however, and so the constant may be important. In the fan-in algorithm, for example, the improvement in order is dependent on the unrealistic assumption that as the problem size increases without bound the number of processors also increases without bound. Divide and conquer strategies do not require multiple processors for their implementation, of course.

Some algorithms are designed so that each step is as efficient as possible, without regard to what future steps may be part of the algorithm. An algorithm that follows this principle is called a greedy algorithm. A greedy algorithm is often useful in the early stages of computation for a problem, or when a problem lacks an understandable structure.

**Bottlenecks and Limits**

There is maximum FLOPS rate possible for a given computer system. This rate depends on how fast the individual processing units are, how many processing units there are, and how fast data can be moved around in the system. The more efficient an algorithm is, the closer its achieved FLOPS rate is to the maximum FLOPS rate.

For a given computer system, there is also a maximum FLOPS rate possible for a given problem. This has to do with the nature of the tasks within the given problem. Some kinds of tasks can utilize various system resources more easily than other tasks. If a problem can be broken into two tasks, $T_1$ and $T_2$, such that $T_1$ must be brought to completion before $T_2$ can be performed, the total time required for the problem depends more on the task that takes longer. This tautology has important implications for the limits of efficiency of algorithms. It is the basis of “Amdahl’s law” or “Ware’s law” (Amdahl, 1967) that puts limits on the speedup of problems that consist of both tasks that must be performed sequentially and tasks that can be performed in parallel. It is also the basis of the childhood riddle:
You are to make a round trip to a city 100 miles away. You want to average 50 miles per hour. Going, you travel at a constant rate of 25 miles per hour. How fast must you travel coming back?

The efficiency of an algorithm may depend on the organization of the computer, on the implementation of the algorithm in a programming language, and on the way the program is compiled.

**Iterations and Convergence**

Many numerical algorithms are iterative; that is, groups of computations form successive approximations to the desired solution. In a program, this usually means a loop through a common set of instructions in which each pass through the loop changes the initial values of operands in the instructions.

We will generally use the notation $x^{(k)}$ to refer to the computed value of $x$ at the $k$th iteration.

An iterative algorithm terminates when some convergence criterion or stopping criterion is satisfied. An example is to declare that an algorithm has converged when

$$
\Delta(x^{(k)}, x^{(k-1)}) \leq \epsilon,
$$

where $\Delta(x^{(k)}, x^{(k-1)})$ is some measure of the difference of $x^{(k)}$ and $x^{(k-1)}$ and $\epsilon$ is a small positive number. Because $x$ may not be a single number, we must consider general measures of the difference of $x^{(k)}$ and $x^{(k-1)}$. For example, if $x$ is a vector, the measure may be some Chapter 5. In that case, $\Delta(x^{(k)}, x^{(k-1)})$ may be denoted by $\|x^{(k)} - x^{(k-1)}\|$.

An iterative algorithm may have more than one stopping criterion. Often, a maximum number of iterations is set, so that the algorithm will be sure to terminate whether it converges or not. (Some people define the term “algorithm” to refer only to methods that converge. Under this definition, whether or not a method is an “algorithm” may depend on the input data, unless a stopping rule based on something independent of the data, such as number of iterations, is applied. In any event, it is always a good idea, in addition to stopping criteria based on convergence of the solution, to have a stopping criterion that is independent of convergence and that limits the number of operations.)

The convergence ratio of the sequence $x^{(k)}$ to a constant $x_0$ is

$$
\lim_{k \to \infty} \frac{\Delta(x^{(k+1)}, x_0)}{\Delta(x^{(k)}, x_0)},
$$

if this limit exists. If the convergence ratio is greater than 0 and less than 1, the sequence is said to converge linearly. If the convergence ratio is 0, the sequence is said to converge superlinearly.

Other measures of the rate of convergence are based on

$$
\lim_{k \to \infty} \frac{\Delta(x^{(k+1)}, x_0)}{(\Delta(x^{(k)}, x_0))^r} = c,
$$

(again, assuming the limit exists, i.e., $c < \infty$.) In (3.9), the exponent $r$ is called the rate of convergence, and the limit $c$ is called the rate constant. If $r = 2$ (and $c$ is
finite), the sequence is said to converge \textit{quadratically}. It is clear that for any \( r > 1 \) (and finite \( c \)), the convergence is superlinear.

Convergence defined in terms of equation (3.9) is sometimes referred to as \textit{Q-convergence}, because the criterion is a quotient. Types of convergence may then be referred to as \textit{Q-linear"}, \textit{Q-quadratic"}, and so on.

The convergence rate is often a function of \( k \), say \( h(k) \). The convergence is then expressed as an order in \( k \), \( O(h(k)) \).

\textbf{Extrapolation}

As we have noted, many numerical computations are performed on a discrete set that approximates the reals or \( \mathbb{R}^d \), resulting in \textit{discretization errors}. By “discretization error” we do not mean a rounding error resulting from the computer’s finite representation of numbers. The discrete set used in computing some quantity such as an integral is often a grid. If \( h \) is the interval width of the grid, the computations may have errors that can be expressed as a function of \( h \). For example, if the true value is \( x \), and because of the discretization, the \textit{exact value} that would be computed is \( x_h \), then we can write
\[
x = x_h + e(h).
\]

For a given algorithm, suppose the error \( e(h) \) is proportional to some power of \( h \), say \( h^n \), and so we can write
\[
x = x_h + ch^n,
\] (3.10)
for some constant \( c \). Now, suppose we use a different discretization, with interval length \( rh \), with \( 0 < r < h \). We have
\[
x = x_{rh} + c(rh)^n,
\]
and, after subtracting from equation (3.10),
\[
0 = x_h - x_{rh} + c(h^n - (rh)^n),
\]
or
\[
ch^n = \frac{(x_h - x_{rh})}{r^n - 1}.
\] (3.11)
This analysis relies on the assumption that the error in the discrete algorithm is proportional to \( h^n \). Under this assumption, \( ch^n \) in (3.11) is the discretization error in computing \( x \), using exact computations, and is an estimate of the error due to discretization in actual computations. A more realistic regularity assumption is that the error is \( O(h^n) \) as \( h \to 0 \); that is, instead of (3.10), we have
\[
x = x_h + ch^n + O(h^{n+\alpha}),
\]
for \( \alpha > 0 \).

Whenever this regularity assumption is satisfied, equation (3.11) provides us with an inexpensive improved estimate of \( x \):
\[
x_R = \frac{x_{rh} - r^n x_h}{1 - r^n}.
\] (3.12)
Richardson extrapolation
curse of dimensionality
splitting extrapolation
recursion

It is easy to see that $|x - x_R|$ is less than the absolute error using an interval size of either $h$ or $rh$.

This process described above is called Richardson extrapolation and the value in (3.12) is called the Richardson extrapolation estimate. Richardson extrapolation is also called “Richardson’s deferred approach to the limit”. It has general applications in numerical analysis, but is most widely used in numerical quadrature. Bickel and Yahav (1988) use Richardson extrapolation to reduce the computations in a bootstrap.

Extrapolation can be extended beyond just one step, as in the presentation above.

Reducing the computational burden by use of extrapolation is very important in higher dimensions. In many cases, for example in direct extensions of quadrature rules, the computational burden grows exponentially in the number of dimensions. This is sometimes called “the curse of dimensionality”, and can render a fairly straightforward problem in one or two dimensions unsolvable in higher dimensions.

A direct extension of Richardson extrapolation in higher dimensions would involve extrapolation in each direction, with an exponential increase in the amount of computation. An approach that is particularly appealing in higher dimensions is splitting extrapolation, which avoids independent extrapolations in all directions. See Liem, Lü, and Shih (1995) for an extensive discussion of splitting extrapolation, with numerous applications.

Recursion

The algorithms for many computations perform some operation, update the operands, and perform the operation again.

1. perform operation
2. test for exit
3. update operands
4. go to 1

If we give this algorithm the name doit, and represent its operands by $x$, we could write the algorithm as

Algorithm doit($x$)
1. operate on $x$
2. test for exit
3. update $x$: $x'$
4. doit($x'$)

The algorithm for computing the mean and the sum of squares (3.7), page 59, can be derived as a recursion. Suppose we have the mean $a_k$ and the sum of squares, $s_k$, for $k$ elements $x_1, x_2, \ldots, x_k$, and we have a new value $x_{k+1}$ and wish to compute $a_{k+1}$ and $s_{k+1}$. The obvious solution is

$$a_{k+1} = a_k + \frac{x_{k+1} - a_k}{k+1}$$

and

$$s_{k+1} = s_k + \frac{k(x_{k+1} - a_k)^2}{k+1}.$$
These are the same computations as in equations (3.7) on page 59.

Another example of how viewing the problem as an update problem can result in an efficient algorithm is in the evaluation of a polynomial of degree $d$,

$$p_d(x) = c_d x^d + c_{d-1} x^{d-1} + \cdots + c_1 x + c_0.$$ 

Doing this in a naive way would require $d - 1$ multiplications to get the powers of $x$, $d$ additional multiplications for the coefficients, and $d$ additions. If we write the polynomial as

$$p_d(x) = x(c_d x^{d-1} + c_{d-1} x^{d-2} + \cdots + c_1) + c_0,$$

we see a polynomial of degree $d - 1$ from which our polynomial of degree $d$ can be obtained with but one multiplication and one addition; that is, the number of multiplications is equal to the increase in the degree — not two times the increase in the degree. Generalizing, we have

$$p_d(x) = x(\cdots x(c_d x + c_{d-1}) + \cdots) + c_1) + c_0,$$

which has a total of $d$ multiplications and $d$ additions. The method for evaluating polynomials in (3.13) is called Horner’s method.

A computer subprogram that implements recursion invokes itself. Not only must the programmer be careful in writing the recursive subprogram, the programming system must maintain call tables and other data properly to allow for recursion. Once a programmer begins to understand recursion, there may be a tendency to overuse it. To compute a factorial, for example, the inexperienced C programmer may write

```c
float Factorial(int n)
{
    if(n==0)
        return 1;
    else
        return n*Factorial(n-1);
}
```

The problem is that this is implemented by storing a stack of statements. Because $n$ may be relatively large, the stack may become quite large and inefficient. It is just as easy to write the function as a simple loop, and it would be a much better piece of code.

Both C and Fortran 95 allow for recursion. Many versions of Fortran have supported recursion for years, but it was not part of the earlier Fortran standards.

### Computations without Storing Data

For computations involving large sets of data, it is desirable to have algorithms that sequentially use a single data record, update some cumulative data, and then discard the data record. Such an algorithm is called a real-time algorithm, and operation of such an algorithm is called online processing. An algorithm that has all of the data available throughout the computations is called a batch algorithm.

An algorithm that generally processes data sequentially in a similar manner as a real-time algorithm, but which may have subsequent access to the same data, is
called an online algorithm or an “out-of-core” algorithm. (This latter name derives from the erstwhile use of “core” to refer to computer memory.) Any real-time algorithm is an online or out-of-core algorithm, but an online or out-of-core algorithm may make more than one pass through the data. (Some people restrict “online” to mean “real-time” as we have defined it above.)

If the quantity \( t \) is to be computed from the data \( x_1, x_2, \ldots, x_n \), a real-time algorithm begins with a quantity \( t^{(0)} \), and from \( t^{(0)} \) and \( x_1 \) computes \( t^{(1)} \). The algorithm proceeds to compute \( t^{(2)} \) using \( x_2 \), and so on, never retaining more than just the current value, \( t^{(k)} \). The quantities \( t^{(k)} \) may of course consist of multiple elements. The point is that the number of elements in \( t^{(k)} \) is independent of \( n \).

Many summary statistics can be computed in online processes. For example, the algorithms discussed beginning on page 59 for computing the sample sum of squares are real-time algorithms. The algorithm in (3.5) requires two passes through the data, so is not an real-time algorithm, although it is out-of-core. There are stable online algorithms for other similar statistics, such as the sample variance-covariance matrix. The least squares linear regression estimates can also be computed by a stable one-pass algorithm, that, incidentally, does not involve computation of the variance-covariance matrix (or the sum-of-squares-and-crossproducts matrix). There is no real-time algorithm for finding the median. The number of data records that must be retained and reexamined depends on \( n \).

In addition to the reduced storage burden, an real-time algorithm allows a statistic computed from one sample to be updated using data from a new sample. An real-time algorithm is \( O(n) \).

### 3.2.2 Methods for Ordering and Partitioning Data

Much information can often be gained from data just by organizing the data in such a way that emphasizes relationship among the data elements. Reordering the dataset can often reveal simple structures in the data. These structures, which may be rankings in the data or patterns of data density, often reveal information about the underlying phenomena that generated the data. The algorithms for organizing and grouping data are often non-numerical, or the numerical computations may be a relatively minor part of the overall algorithms.

#### Sorting

For data whose elements have an ordinal ranking, one of the most important computer operations is sorting. Sorting is also something we often do by hand, as filing folders or arranging cards in order. A good hand algorithm, such as starting with \( j \) cards that are sorted and inserting the \((j+1)\)th card in its proper place, is of order \( n^2 \) for either the average or the worst case, where \( n \) is the total number of cards. This is called an “insertion sort”.

How quickly sorting can be accomplished depends on the arrangement of the data prior to sorting. Study of the behavior of sorting algorithms generally considers the data to be in a “random” arrangement, and so the behavior is characterized in terms of the probability distribution of the number of comparisons. The parameter of the distribution that is of most interest is usually the mean number of comparisons.
3.2 Algorithms

Mahmoud (2000) develops probability distributions associated with various sorting algorithms.

There are several sorting methods whose average performance is of order $O(n \log n)$. These methods employ a divide and conquer strategy in which the sorting problem is recursively broken into two problems of roughly equal size. The performance of some of these methods is worse than that of an insertion sort for small $n$. Therefore, whatever sorting method is implemented in a general sorting program, a simple insertion sort is usually used if the number of items to be sorted is small.

One of the best of the $O(n \log n)$ methods, and the one that is most often available in software, is the "quick sort", or "quicksort", due originally to C. A. R. Hoare in the early 1960’s (see Knuth, 1973). Eddy and Schervish (1995) study the probability distribution of the number of comparisons needed by quicksort, when the items are in random order prior to being sorted.

The quicksort works by repeatedly partitioning the data based on their relationship to chosen splitting elements. Each partition results in the splitting element being in its final sorted position, all elements below it being smaller than it is, and all elements above it being larger than it is. There are various ways this can be done; all of them require some additional storage space to keep up with what is being done.

One way to do the quicksort is to choose the first element, $x_1$, as the splitting element, that is, the one to move to its proper place. First, rearrange the remaining elements into two partitions (one of which is possibly null), such that all in the “left” partition are smaller than $x_1$ and all in the “right” partition are larger than $x_1$; then we put $x_1$ between the two partitions. Then do the same thing in each partition. It is clear that at the end, the array will be sorted.

To perform the first partitioning, consider the two elements on the ends of the subarray (of length $n - 1$). The element on the “left” is $x_2$ and the element on the “right” is $x_n$. These are the candidate elements for moving. If a candidate element is on the proper side of $x_1$ (the left candidate is smaller, or the right candidate is larger), proceed inward from the left or right of the subarray until an element is encountered that is on the “wrong” side of $x_1$. If two candidates are encountered that are both on the wrong side, exchange them, and proceed. If two candidates are not found that are both on the wrong side of the splitting element, then move the splitting element to its proper place. This completes a pass; there are now possibly two partitions. Continue on each partition separately.

As an illustration, consider the array

\[(66, 99, 33, 77, 45, 22, 55, 88, 11).\]

In the first pass, we will move 66 to its proper place. The candidate elements are shown with asterisks:

\[
\begin{align*}
66 & *99 33 77 45 22 55 88 *11 \\
66 & 11 33 *77 45 22 *55 88 99 \\
66 & 11 33 55 *45 *22 77 88 99 \\
11 & 33 55 45 22 66 77 88 99
\end{align*}
\]

After the first pass shown above, 66 is in its proper place and the two partitions are now (11, 33, 55, 45, 22) and (77, 88, 99).
Sorting in Parallel

Using a machine with a single processor, the best sorting algorithm is $O(n \log n)$. Sorting can be done more quickly in parallel. The optimal order depends on the arrangement of the parallel processors. Because $O(n \log n)$ is the lower bound for a single processor, a lower bound for sorting in parallel must be at least $O(\log n)$. No such algorithm is known, but there are various ways of achieving speeds better than $O(n \log n)$.

For a simple linear array of processors (such as shown in Figure 3.20, page 81), the optimal order is $O(n)$, and a sorting algorithm that achieves this is the so-called “odd-even” method. In this method, the $n$ elements are distributed across $n$ processors. In the first step, the first and second processors are paired, the third and fourth are paired, and so on. Within each pair the elements in the array to be sorted are interchanged if necessary so they are in the proper order. In the second step, the second and third processors are paired, the fourth and fifth are paired, and so on; and within each pair the elements in the array to be sorted are interchanged if necessary so they are in the proper order. In the third step, again the first and second processors are paired, the third and fourth are paired, and so on; and the comparisons are performed within each pair. This method continues until after $n$ steps the array is sorted. (It is easy to see that the method works on a simple example. See Libeskind-Hadas, 1998, for a simple proof that this works in general.)

Order Statistics

Sorted arrays, that is to say, order statistics, are very important in statistical analyses, especially nonparametric analyses. In such analyses, we may use central order statistics, such as the median, as measures of the center of the distribution, or we may use distance between order statistics as measures of spread. We may also use points that are close together to smooth the data or to make local inferences.

It is not necessary to sort a set of data to find a single order statistic. It is obvious that the operation of finding the minimum, the maximum, or any other fixed-rank order statistic is $O(n)$. Although it is not so obvious, the operation of finding the median is also $O(n)$ (see Floyd and Rivest, 1975a, 1975b, and Schönhage, Paterson, and Pippenger, 1976). For moderately sized arrays, however, it is almost as efficient just to sort the array.

Another statistic that can require a lot of sorting is the Hodges-Lehmann estimator, which is the median of the averages of all pairs,

$$\frac{x_i + x_j}{2}, \quad 1 \leq i \leq j \leq n.$$

Robinson and Sheather (1988) gave an algorithm to compute the Hodges-Lehmann estimator in $O(n \log n)$ time. They first sort the data, and then identify the $O(n^2)$ sums with a double index. Next they proceed to partition the set of sums by proceeding from the $(1, n)$ element to the $(2, n)$ element, then to the $(2, n - 1)$ element and so on. (See also Choudhury, 1989.)

Ordering Multivariate Data

More interesting problems arise in multivariate data, that is, data in which each observation consists of a vector. In such cases, the definition of an ordering or of a
median is not straightforward. One simple way of defining an ordering is by defining some norm and then sorting the norms of the observations. Sorting of multivariate data, however, is often defined simply as sorting of the observations based on only one element of the observations, called a key, and then, if there are ties with respect to that key, sorting the tied observations with respect to a second key, and so on. If there is some particular element in the vectors of the dataset that is so much more important than all of the others, this kind of sorting makes sense. On the other hand, we may want to arrange the data in some particular order, without giving preference to any specific elements in the observations. For example, we may want to arrange the data based on the distance of the observations from a specified point. These methods of sorting are trivial. We will describe some more interesting methods beginning in Section 13.3.

An important step in understanding data is to group the data into subsets that are similar in some way. Classification and clustering of data is one of the most fundamental activities in science. In order to define, and then to find, nearest neighbors or clusters in multivariate data, we use data structures called graphs and trees, which we discuss beginning on page 564.

**Structures for Multivariate Data**

We use the term “observation” to refer to data that relates to a single item, and the term “variable” to refer to individual elements that are measured or observed. For example, an observation in a set of meteorological records may relate to a specific place and time; and the observation may contain values of variables that represent location, date and time, temperature, pressure, and so on.

We will often denote the observations by \(x_1, x_2, \ldots, x_n\). We will speak of structure in the data and of data structures. Structure in the data, which may be clusters or other patterns, is what is of interest to the scientist. Data structures are devices to help organize the data to provide better access to the individual data elements.

A multivariate dataset consists of \(n\) observations, each of which consists of \(m\) elements (or observations on “variables”). Although we think of the observations as column vectors, we may often write them in a horizontal notation, as

\[
x_i = (x_{i1}, x_{i2}, \ldots, x_{im}),
\]

for example. We will represent the dataset as an \(n \times m\) matrix \(X\), where

\[
X = \begin{bmatrix}
x_1^T \\
x_2^T \\
\vdots \\
x_n^T
\end{bmatrix} = \begin{bmatrix}
x_{11} & x_{12} & \ldots & x_{1m} \\
x_{21} & x_{22} & \ldots & x_{2m} \\
\vdots & \vdots & \ddots & \vdots \\
x_{n1} & x_{n2} & \ldots & x_{nm}
\end{bmatrix}.
\] (3.14)

This organization is an array data structure. An array data structure, which corresponds to a “flat file”, shows the relationships among the variables and the observations in a very simple, intuitive manner.
The kinds of operations we may perform on a dataset depend on the nature of the elements it contains. If the data are numeric, appropriate numerical operations, such as summing, may be performed. In most of the following discussion we will assume that the data are numeric, and, usually, we assume they have the properties of the reals.

Statistical summary operations, such as finding the mean and variance, are generally applied to the variables, that is, to the columns of $X$; and we use notation that corresponds to that structure. For example, $\bar{x}_1$ denotes the mean of the first variable, or the first column of $X$. Notice the slight inconsistency in notation here: a single subscript on the raw data, such as $x_i$, refers to an observation (a row of $X$), but a single subscript on a summary statistic, such as $\bar{x}_j$, refers to a variable (a column of $X$). The meaning of the notation is usually clear from the context.

If the data have some structure, we may wish to rearrange the dataset so as to show the structure better. If there are groups or clusters in the data, we may put observations in the same group together. Clustering and classifying data can provide a method of sorting or ranking multivariate data.

There are many ways of clustering and classifying multivariate data. We discuss some of these in Chapter 13.

Other data structures are sometimes useful. If the dataset contains medical records of families, for example, a hierarchical structure in which a single observation may contain data on several generations of persons. The observations generally have unequal numbers of elements, and relations among variables are not as obvious as in an array structure. A graph or tree data structure (see Section 13.1.3, page 564) is a useful way of organizing the relationships in multivariate data.

Often the raw data have an array structure, but we form other data structures in an attempt to extract information from the data. Many approaches to extract information from multivariate data are based on some method of dimension reduction. Statistics of lower dimensionality, computed from the raw data, are used to summarize salient aspects of the data. Dimension reduction generally carries a loss of some information. Whether or not the lost information is important is the major concern in dimension reduction.

Understanding of data is greatly enhanced by visual displays of the data. Many of the methods of dimension reduction have as their primary purpose the production of a dataset that can be viewed graphically.

An understanding of the geometry of higher dimensions can lead to projections and other transformations that are useful in discovering important structure in the data. Computer graphics allows easy exploration of these transformations. Banchoff (1996) discusses geometric properties of higher dimensions and their application in computer graphics.

### 3.2.3 Massive Datasets

One of the most significant developments in recent years, along with the general growth of computing power, has been the growth of data. An impressive example of the growth of data is the set of data generated by the various NASA satellites, whose sensors are constantly measuring both ground-based and extraterrestrial variables.

An important activity currently is the coordination of disparate datasets that contain at least some fields that relate to common variables. The objective is to
allow a consistent and integrated access to the various datasets. This coordination, which may involve some cleaning of the data as well as development of common indexes across the datasets, is called “data warehousing”. The major commercial database management software packages provide facilities to “warehouse” the data from disparate datasets.

It is now common to search through datasets and compute summary statistics from various items that may indicate relationships that were not previously recognized. The individual items or the relationships among them may not have been of primary interest when the data were originally collected. This process of prowling through the data is sometimes called data mining or knowledge discovery in databases (KDD). The objective is to discover characteristics of the data that may not be expected based on the existing theory. In the language of the database literature, the specific goals of data mining are:

- predictive modeling
- database partitioning
- linkage analysis
- deviation detection

Of course, these objectives are the ordinary ones of any exploratory statistical data analysis. Data mining is exploratory data analysis or “EDA” applied to large datasets, usually with the primary objective of discovering relationships among variables. The explorations in massive datasets must be performed without much human intervention. Searching algorithms need to have some means of learning and adaptively improving. This will be a major area of research for some time.

Predictive modeling uses inductive reasoning, rather than the more common deductive reasoning that is much easier to automate. Neural nets (see Chapter 7) are often used for the induction or “learning”.

Database partitioning is essentially statistical classification. Partitioning is done recursively. The partitioning results in a classification tree, which is a decision tree each node of which represents a partition of the dataset. The decision at each node is generally based on the values of a single variable at a time, as in the two most commonly used procedures, CART (see Breiman et al., 1984) and C4.5 or its successors C5.0 and See5 (see Quinlan, 1993). CART can also build nodes based on linear combinations of the variables. This is sometimes called “oblique partitioning”, because the partitions are not parallel to the axes representing the individual variables. Seeking good linear combinations of variables on which to build oblique partitions is a much more computationally intensive procedure than just using single variables. Heath, Kasif, and Salzberg (1993) and Murthy, Kasif, and Salzberg (1994) describe methods that use randomization in selecting oblique partitions.

Linkage analysis is often the most important activity of data mining. In linkage analysis, relationships among different variables are discovered and analyzed. This step follows partitioning and is the interpretation of the partitions that were formed.

It is also important to identify data that do not fit the patterns that are discovered. The deviation of some subsets of the data often makes it difficult to develop models for the remainder of the data.

An overview of some of the issues in mining large dataset is given in Fayyad, Piatetsky-Shapiro, and Smyth (1996) in the book edited by Fayyad et al. (1996), which contains several articles addressing specific aspects and approaches to the problem. Glymour et al. (1996) discuss some possible roles for statistical inference.
There are several commercially available software packages that implement data mining, usually of datasets in some standard format.

**Computational Feasibility**

Data must be stored; it must be transported; it must be sorted, searched, and otherwise rearranged; and computations must be performed on it. The size of the dataset largely determines whether these actions are feasible. Huber (1994, 1996) proposed a classification of datasets by the number of bytes required to store them. He described as “tiny” those requiring on order $10^2$ bytes; “small” those requiring on order $10^4$ bytes; “medium” those requiring on order $10^6$ bytes (one megabyte); “large”, $10^8$ bytes; “huge”, $10^{10}$ bytes (10 gigabytes); “massive”, $10^{12}$ bytes (one terabyte). (“Tera” in Greek means “monster”.) This log scale of two orders of magnitude is useful to give a perspective on what can be done with data. Online or out-of-core algorithms are generally necessary for processing massive datasets.

For processing massive datasets, the order of computations is a key measure of feasibility. We can quickly determine that a process whose computations are $O(n^2)$ cannot be reasonably contemplated for massive data sets. If computations can be performed at a rate of $10^{12}$ per second (teraflop), it would take over three years to complete the computations. Table 3.3 shows some representative times required for computations of various orders for various sizes of problems. (A rough order of magnitude for quick “year” computations is $\pi \times 10^7$ seconds equals approximately one year.)

<table>
<thead>
<tr>
<th>Size $n$</th>
<th>$O(n)$</th>
<th>$O(n \log n)$</th>
<th>$O(n^2)$</th>
<th>$O(2^n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10$</td>
<td>0.01 nanosecond</td>
<td>0.023 nanosecond</td>
<td>0.1 nanosecond</td>
<td>1 nanosecond</td>
</tr>
<tr>
<td>$10^2$</td>
<td>0.1 nanosecond</td>
<td>0.46 nanosecond</td>
<td>10 nanoseconds</td>
<td>4 centuries</td>
</tr>
<tr>
<td>$10^4$</td>
<td>10 nanoseconds</td>
<td>92 nanoseconds</td>
<td>100 microseconds</td>
<td>a long time</td>
</tr>
<tr>
<td>$10^6$</td>
<td>1 microsecond</td>
<td>14 microseconds</td>
<td>1 second</td>
<td>a long time</td>
</tr>
<tr>
<td>$10^8$</td>
<td>100 microseconds</td>
<td>1.8 milliseconds</td>
<td>17 minutes</td>
<td>a long time</td>
</tr>
<tr>
<td>$10^{10}$</td>
<td>10 milliseconds</td>
<td>230 milliseconds</td>
<td>3.2 years</td>
<td>a long time</td>
</tr>
<tr>
<td>$10^{12}$</td>
<td>1 second</td>
<td>28 seconds</td>
<td>320 centuries</td>
<td>a long time</td>
</tr>
</tbody>
</table>

Sometimes it is appropriate to reduce the size of the dataset by forming groups of data. “Bins” can be defined, usually as nonoverlapping intervals covering $\mathbb{R}^d$, and the number of observations falling into each bin determined. This process is linear in the number of observations. The amount of information loss, of course, depends on the sizes of the bins. Binning of data has long been used for reducing the size of a dataset, and earlier books on statistical analysis usually had major sections dealing with “grouped data”.

Another way of reducing the size of a dataset is by sampling. This must be done with some care, and often, in fact, sampling is not a good idea. Sampling is likely to miss the unusual observations, and it is precisely these outlying observations that are most likely to yield new information.
Advances in computer hardware continue to expand what is computationally feasible. It is interesting to note, however, that the order of computations is determined by the problem to be solved and the algorithm to be used, and not by the hardware. Advances in algorithm design have reduced the order of computations for many standard problems, while advances in hardware have not changed the order of the computations. Hardware advances change the constant in the order of time.

**Parallel Processing of Datasets**

Searching through datasets or retrieving items from datasets are operations that lend themselves to parallel processing. So long as the data are not being modified, the operation is embarrassingly parallel.

Simple, but massive, parallel processors can be put to the task of data mining in huge datasets or in data warehouses. Summary statistics from various subsets of the data may yield valuable information that was not even recognized as being of interest when the data were originally collected.

Parallel processors work well in searching datasets, but in the more general operations on datasets, independence of the operations must be maintained. This is an issue in any system that allows simultaneous access to datasets by different processors. The larger commercial database management software packages generally provide for distributed processing and provide controlled replication of the database updates.

### 3.3 Computer Architecture and System Software

The design of the computer can have major effects on the efficiency and accuracy of algorithms and programs. A good program to implement a given algorithm on one computer may not be a very good program for the same algorithm on a different computer. This fact also implies that the best algorithm for a given problem on one computer may not be the best algorithm for the same problem on a different computer. Dowd and Severance (1998) describe several modern computer architectures and discuss some of the considerations for developing efficient software.

#### 3.3.1 Overview of the Organization of a Computer

One of the most important considerations in designing a computer is to maximize the overall operational efficiency. This means, among other things, to allow independent tasks to be performed by independently operating units of the computer system. The simplest example of this (which involves a peripheral unit) is the parallel execution of a printing job and some other task.

**Components of a Computer**

The basic organizational units of a computer are the processing unit, the memory, communication mechanisms, and the control unit. The processing unit is where most of the computations or other data manipulations take place. There may be a parallel processing...
single or main processing unit that is called the central processing unit (CPU). Additional processing units within the same computer are sometimes called peripheral processing units (PPU).

The operands on which the processing unit performs computations are stored in registers. The number and arrangement of registers vary widely among the processing units of different computers. Some registers may be for fixed-point numbers and some for floating-point numbers. The registers may provide more bits for the significand than the standard for floating-point representation in that computer. The number of bits may provide an extended precision even beyond what the guard digits would yield. (See page 48 for a discussion of guard digits.)

There is a wide variety of processing units. They are distinguished by how they accept data and the set of instructions they can execute. The complexity of data and of the instruction set usually reduces the speed of the processor. On the other hand, if the processor cannot perform many different operations, it may have to perform the same operations more often. (Suppose, for example, the processor could only add and could not multiply. The processor may have to do many additions to simulate one multiplication.) Determining the optimal level of complexity of processor instructions is an important task for a computer designer. The optimum, of course, depends on the type of computations the computer will be called on to perform. For scientific computations, the instruction set can be reduced to those most common and useful in numeric computations. The processors in most scientific workstations implement a reduced set, so the computers (processors) are called reduced instruction set computers, or RISCs.

The speed of a processor is sometimes measured in millions of instructions per second (MIPS) that it can perform. Of course, because the complexity of the instructions varies, one cannot rely too much on the MIPS (“meaningless indicator of processor speed” — that is not original, but I do not know whom to credit). In addition to the issue of the varying complexity of the instructions, the ability of the processor to achieve and sustain a given rate may be dependent on availability of data or other resources. To emphasize this dependence, we sometimes use the phrase millions of theoretical operations per second (MTOPS). Another measure of speed that may be more important in scientific computation is the number (or the theoretical number) of floating-point operations per second (FLOPS). The maximum FLOPS rate possible and the rate achievable for a given problem may be quite different. The efficiency of an algorithm is often measured by the FLOPS rate achieved with the algorithm, in the context of the maximum achievable.

The memory is where data are stored. There are usually several types of memory, distinguished primarily by its capacity and speed of access. A relatively small but fast memory where are stored data that are expected to be used soon is called cache memory. A computer may even have more than one type of cache memory with different access speeds. The fastest cache is sometimes referred to as “L1 cache”, and other levels as “L2 cache”, and so on.

The efficiency of an algorithm depends not only on how well the algorithm utilizes the processor(s), but also on how efficient it is in moving data from the various levels of memory and in and out of the processor storage units.

The speed of the computer (the number of computations performed in a given time) can be greatly enhanced by having individual computations and even larger tasks performed simultaneously.
Pipelining in Vector and Superscalar Computational Units

Because even simple tasks such as addition of floating-point numbers require sub-tasks that are somewhat different from each other, an obvious organization of the processing units is by a division of labor among processing units (or just the components of what is logically a single processing unit) that allows various subtasks to be performed simultaneously. This idea has been incorporated in the design of computers since the late 1950’s.

Consider the problem of computing \( c = a + b \), using floating-point representation. The steps are

1. Input operands to the processing unit
2. Subtract exponents
3. Align the radix or base point
4. Add the significands
5. Normalize
6. Output sum to memory

A single processing unit may be designed with subunits to perform these individual operations, as shown in Figure 3.14.

When more than one addition is to be performed, each operand pair \((a_i, b_i)\) can be inserted into the pipeline, as shown in Figure 3.15.

Pipelining is particularly useful for computations on vectors. An entire segment of each vector operand can be brought into a vector processing unit to go through the computational pipeline. The speed of such a system will depend to a large extent on the time taken in the first and last steps, that is, the time to load the vector registers and the time to store the results in memory. If the computer views the vector data as being in contiguous locations in memory, the data transfer can be expedited. The computer may view the data as contiguous whether or not they are physically contiguous; in fact, the memory may be arranged into banks, and “contiguous” data may be stored alternately in the different banks. The electronics of the data transfer may be such that alternating between different banks of memory...
is faster than transferring data from a single bank. If the data are not contiguous, but are at a fixed distance from each other, as the data in either rows or columns of a matrix may be, the fixed “stride” allows for fairly rapid data transfer. If, on the other hand, the data in the vectors are neither contiguous nor at a fixed stride, an index vector may be used to perform a “gather” and subsequently a “scatter”.

### 3.3.2 Multiple Processing Units

If there are limits to how fast individual computations can be performed, the only way to increase the speed is to increase the number of operations that can be performed in parallel and to provide multiple processing units to perform these computations. Parallel processing is currently a very active area of research.

The individual processors may either all perform the same operation at the same time (on different data, of course), or they may perform operations independently. Even though the operations are independent, they must be coordinated. These two methods of control of the processors are respectively called SIMD, “single instruction, multiple data”, and MIMD, “multiple instruction, multiple data”. In a SIMD computer all processors are controlled by a master processor. In a variation of this arrangement, the processors execute the same program, but with allowable variations. This is sometimes called SPMD. The nature of the computations in a SIMD or SPMD computer is called “data parallel”, because the data are going through a sequence of the same instructions in parallel.

MIMD computers are more general. Individual processors have their own programs, so the computations are more flexible. The problem of synchronization becomes more important, however.

A very cost-effective way of constructing a computer with multiple processing units is to begin with individual computers and network them into a cluster. There are various ways of doing this, and the characteristics of the individual computers and of the network determine the suitability of the cluster for specific types of computations. Pfister (1998) gives an interesting discussion of various types of clusters.

The main problem when two or more processing units are working on the same problem is the currency of the data. In serial processing, operations should not be repeated unnecessarily, but in parallel processing exactly the same computations may be performed multiple times. The communication costs to transfer results from one processor to another or for one processor to check to see whether a computation has already been done may exceed the cost of the computation itself.

### Organization of Memory and Processing Units

In order for more than one processing unit to work together on a task, the processors must have access to the necessary data for the task, but they must also lock access from other processors to data that they are currently updating. There are two extremes of memory organization to address these objectives.

- Shared memory.
- Distributed memory.
3.3 Computer Architecture and System Software

In a shared memory organization, the processors share all of memory more-or-less equally. In this arrangement, the processors usually perform symmetric multiprocessing (SMP). An SMP computer is basically a shared memory MIMD machine, usually with a small number of processors (two to eight). There must be a data bus or a connection network (if memory is divided into chunks) connecting the processors to the memory. A simple shared memory arrangement is illustrated in Figure 3.16.

![Figure 3.16. Simple Shared Memory with Data Bus](image)

The data bus in this shared memory arrangement is relatively simple; it is just a transfer path to memory from each processor. The processors must be controlled so as to insure that the computations are performed on current data.

A modification of the simple shared memory organization is shown in Figure 3.17. In this arrangement, the memory is broken into chunks that are associated with a given processor or set of processors for a certain number of computations and then later associated with different processors. This helps to alleviate the problem of data currency. The data transfer mechanism must be more complicated in this arrangement. Instead of a simple bus, a "connection network" is required.

In a distributed memory organization each processor has its own local memory, but there must be a communication network to allow each processor's memory to be updated. A simple arrangement of distributed memory is shown in Figure 3.18. The distributed memory arrangement allows for massively parallel processing.

Another obvious generalization of the memory and processor organizations is a shared memory arrangement with local memory modules. In such an organization, as shown in Figure 3.19, there is a relatively large part of memory that is shared by all processing units, with appropriate controls, and each processor has its own local memory. The connection network in this arrangement must be fairly complicated. This kind of arrangement allows for scalable parallel processing, in which the benefits of parallel processing are less dependent on the size of the problem.
Organization of Distributed Memory Processors

Distributed memory processors, called nodes of the “multicomputer”, can be connected in several ways. The connection networks may be completely connected, switched with crossbars, or connected in some other arrangement such as a mesh or a hypercube.

The organization of the connection is called its topology. The basic topologies that are or have been used in distributed memory processors:

- Line or ring topology (Figures 3.20 and 3.21)
- Mesh topology, in which the nodes are arranged in a \( k \)-dimensional lattice, with connections between each adjacent lattice point (Figure 3.22)
• Torus topology, which is a mesh with boundary nodes connected as in the ring topology (Figure 3.23)
• Hypercube topology, which is organized as a $k$-dimensional cube, with $2^k$ vertices, each connected to $k$ adjacent vertices (Figure 3.24)

The important characteristics of the topology are
• Connectivity, the number lines at each node, which affects the cost of the hardware
message passing
MPI (message passing interface)

![Mesh Topology (2D)](image1)

Figure 3.22. Mesh Topology (2D)

![Torus Topology (2D)](image2)

Figure 3.23. Torus Topology (2D)

- Diameter, the maximum length of a communication line, which affects performance, especially when communication is important

These characteristics are shown in Table 3.4 for the topologies we have discussed.

The maximum communication time, or “broadcast” time, depends on the diameter of the topology. Thus, for a line topology, the broadcast time is $O(k)$, for a 3D torus it is $O(k^{1/3})$, and for a hypercube it is $O(\log k)$.

Distributed memory processors must coordinate their work by message passing. The “messages” may be data that have been updated or they may just be notification that certain operations have been performed. A standard interface for message passing, MPI, has been developed and it is becoming widely supported.
Network Computing

It is very common for a research department or a laboratory to have a number of computers that are networked. It may be possible to make the computers work as one multicomputer, or a single “virtual machine”. Even if the individual computers are of very different types, it may be possible to form a heterogeneous networked computer. The virtual machine is a cluster of machines with a software system that make them appear as a single parallel processing computer. The machine obviously has a distributed memory. One or more machines on a network can function primarily as computing servers, while other machines function as communication clients.

The software itself is an integrated framework that is layered over the native operating systems of various computers. Some software packages to do this are:
PVM (Parallel Virtual Machine)
MPI (message passing interface)
Condor (software system)
Beowulf
Linux (operating system)
MPI (message passing interface)
grid, computational

- Express, which is a commercial toolkit developed by ParaSoft Corporation to aid in conversion of sequential algorithms to parallel. VTOL is a graphical tool to help visualize data structures. Other tools actually accept Fortran or C code and translate it to parallel segments, and analyze the extent of parallelization. It has parallel debugger similar to dbx.

- Linda, developed at Yale University. Linda uses the concept of “tuple-space” by which cooperating processes communicate. The tuples are pieces of memory that must be maintained together. Linda provides a shared memory abstraction without requiring physical sharing of memory.

- P4 and Parmacs, which are sets of language extensions. P4 is a library of macros and subroutines developed at Argonne National Laboratories. It can work in either a shared memory or distributed memory environment. The user must supply a configuration file. P4 has been implemented both in heterogeneous environments as well as on single computers. Parmacs is a similar system.

- PVM, Parallel Virtual Machine, which was developed at Oak Ridge National Lab, University of Tennessee, and Emory University. It provides a set of C functions or Fortran subroutines that allow a heterogeneous collection of Unix computers to operate smoothly as a multicomputer. PVM is freely available and has excellent documentation (Geist et al., 1994).

- MPI, the Message Passing Interface standard, which was developed by a consortium of computer vendors, software developers, and users, and various library implementations of the standard. The standard specifies a library of approximately 200 functions and Fortran and C bindings for them.

Condor is a system developed at the University of Wisconsin for managing distributed resources (see Epema et al., 1996). Condor runs on a cluster of heterogeneous workstations. It seeks out idle or underutilized resources and matches them with requests from users. Ferris, Mesnier, and Moreé (2000) describe the use of Condor together with networked optimization software for solving optimization problems over the internet. More information about Condor is available at http://www.cs.wisc.edu/condor/

Even if all of the computers in a heterogeneous network conform to the IEEE standards for floating-point computations, slight differences in the way the computations are done may yield erroneous results. Blackford et al. (1997) describe some problems they encountered when using a heterogeneous network.

A type of homogeneous network computer originally built at NASA Goddard is called Beowulf. While there are some variations in Beowulf computers, the usual architecture consists of a single server node connected with several client nodes. All components, both hardware and software, are commercial off-the-shelf (“COTS”) items, such as Intel processors, the Linux operating system, and MPI. Each client node consists of a processor and memory. Usually the client node does not have a disk, keyboard, or monitor. In this way a Beowulf computer is different from other network computers such as a cluster of workstations. See Sterling et al. (1999).

A computational infrastructure similar to the energy infrastructure provided by national power grids can provide computational power to researchers at sites in distributed locations. In addition to providing the computing power, the grid can allow access to files and to software at multiple computing centers. The proceedings volume edited by Engquist et al. (2000), contains articles on grid technologies as well as examples of the use of a grid in various applications.
### 3.3.3 Methods for Parallel Processing

A problem (or a program to solve it) may have three types of inherent parallelism:

- **Data parallelism** – different data items can be processed at the same time in the same manner.
- **Functional or control parallelism** – different tasks can be performed independently, and hence simultaneously.
- **Overlapped tasks** – a sequence of tasks that can be executed in an overlapped manner. Pipelining, as illustrated in Figures 3.14 and 3.15, is an example of this.

The main concern when computations are performed in parallel is keeping the data current for all processors, but yet keeping all of the processors busy. The major issues are:

- Memory contention
- Process creation time
- Communication delay
- Synchronization delay
- Load imbalance

As we have already seen, slower subprocesses will dominate faster subprocesses in the overall performance of an algorithm (“Amdahl’s law”). Most jobs will consist of some subprocesses that must be performed sequentially, and so whenever a parallel processor is available, the speedup relative to a sequential processor is reduced by the inherently sequential subprocesses. If \( p \) processors are available, \( T \) is time for one processor to complete task, \( \alpha \) is the fraction of operations that are not parallelizable (and so \( 1 - \alpha \) is the fraction of operations completely distributed over the \( p \) processors), and \( t_d \) is the total delay time due to setup or communications, then the speedup is given by

\[
\frac{T}{(\alpha + (1 - \alpha)/p)T + t_d}
\]

A fortunate characteristic of many computational problems in science is that as the problem grows in size and complexity, the size of the parallelizable components grows more rapidly than the strictly sequential components.

### Organization of Parallel Computations

There are several ways to go about organizing computations to be done in parallel. The books by Angus et al. (1990), Goedecker and Hoisie (2001), Quinn (1994), and Roosta (2000), and the collection of articles in Sabot (1995a) covers many of the issues, and gives examples of approaches for specific problems.

Some of the methods include:

- Data partitioning
- Data parallelism
- Synchronous iteration (systolic, SIMD)
  - “systolic” – pipeline of processors thru which data are pulsed
- Replicated processors (task pool)
- Pipelined computation
Divide and Conquer in Parallel

The objective of a divide and conquer method in a parallel processing platform is often directed toward the time, rather than the number of computations. Consider the problem of adding a large number of elements:

\[ x_1 + x_2 + \cdots + x_n \]

Dividing the problem into \( \lceil n/2 \rceil \) subproblems, and recursively applying a divide and conquer strategy, we have a “fan-in” algorithm we discussed on page 48. At each successive stage in the fan-in, the number of processors doing the additions goes down by approximately one-half. There is no difference in the number of computations.

It is clear that a divide and conquer algorithm could be applied to any problem that consists of a series of \( n - 1 \) associative binary operations applied to \( n \) operands, such as \( \sum x_i \) or \( \prod x_i \), to yield a method with time of order \( O(\log n) \).

Control Parallelism and Data Parallelism

In a shared memory multiprocessor, there is a control parallelism, while in a distributed memory message-passing (DMMP) multiprocessor, there is data parallelism. A control parallel machine has the following characteristics:

- The machine consists of homogeneous processors with small local caches.
- The processors operate asynchronously.
- All processors can access a large shared memory.
- Any exchange of data between processors can take place only via the shared memory.

The programming of a control parallel machine generally is characterized by the following:

- Parallelism is specified using the forall construct.
- Parallel execution follows the fork-join model.
- Synchronization is achieved using semaphores, that is flags to indication completion of certain tasks.
- Access of the parallel processes to particular variables can be serialized using critical sections.

The characteristics of a data parallel machine are:

- The machine consists of multiple homogeneous processors, each with a fairly large local memory.
- The processors operate asynchronously.
- The processors can exchange data only via message passing.

The programming of a data parallel machine generally is characterized by the following:

- The fundamental message-passing mechanisms are send and recv. The recv instruction causes the processor to block until the specified message is received.
- All processors execute the same node program, but on different disjoint partitions of the data.
- The data domain is statically partitioned across the processors.
3.4 Scientific Software

• Synchronization and serialization of the processors can only be done using message passing.

With cheaper memory, the DMMP computers have become more common. An active area of work is to provide automatic translation of control parallel programs to data parallel programs.

Parallel Programming

Experiences with different parallel programming systems are discussed in Droegemeier et al. (1995) and Sabot (1995b).

An important consideration is that the programs be scalable; that is, that the organization of the program and the use of parallel constructs not be too dependent on the size of the problem (see Wholey, 1995, and Droegemeier et al., 1995, for discussion and examples).

The objective is to use the available resources as efficiently as possible. The Numerical Aerodynamic Simulation (NAS) Program at NASA Ames Research Center has developed a set of benchmarks, the NAS Parallel Benchmarks, to evaluate the extent to which parallelism is achieved by a program running on a given parallel machine.

3.4 Scientific Software

Software includes libraries, systems, and environments for computing. The software depends ultimately on a language for expressing the actions to be performed. In many computing applications, an efficient computational environment allows the users to express actions “nonverbally” by use of pointing devices.

Most software is developed to address a specific problem. It is very likely, whatever the problem, that someone else has addressed that problem before and someone will face it again. Tremendous gains in efficiency of software development could be realized if everyone could use the same program for a similar problem, and not have to write a similar program. If the program were developed by a competent numerical analyst and programmer, it is likely that the program will be faster and more accurate than programs developed by persons not expert in numerical analysis and programming. A common problem, however, is that a lot of the available software was not developed by competent analysts and programmers.

Despite the problems with using software developed by someone else, over the past several years a large body of software for scientific computing has evolved, and reviews and other references on the software have helped the user to identify the more reliable software. Many scientists and mathematicians have become aware of the importance of software reuse and have learned some of the general principles that help in developing reusable software.

Understanding the causes of poor software can be of help in formulating guidelines for the development of high-quality software. Gentle (1982) lists some of the reasons for low-quality software:

• Software production is often a cottage industry. Adequate resources are not available to address all of the aspects of software development. The entry cost is low; almost anyone can be a software producer.
• The magnitude of the development effort is often underestimated. This results in shortcuts, especially in the latter stages of the development process.
• Software that is produced ad hoc often falls into general purpose usage. The software does not perform well in areas that it was not designed to address. A very common situation is one in which an amateur software developer writes a program for a specific task, and then other people use the software for tasks that are not quite the same.
• The attitude in the software market is “caveat usitorem”; “let the user beware”. The user receives no guarantees.
• No entirely adequate testing and validation methodology is in common use.

An active field of study and one that, happily, has had some effect on practice is software engineering, which, among other things examines the characteristics of good software and the programming discipline that helps to achieve them. This is a field that is ripe for the application of statistical methods for assessing and improving reliability. Singpurwalla and Wilson (1999) develop a framework for addressing uncertainties in software engineering and for using quantitative measurements for monitoring and improving the process.

Standards
In order to facilitate exchange of tools and information, it is important to have standards that define common characteristics of software and data. At the lowest level of data storage, there are standards for the representation of individual characters, such as the ASCII code and the Unicode standards, and for the representation of numeric values, such as the IEEE standards. At a higher level of data storage, there are the Common Data Format (CDF) and the Hierarchical Data Format (HDF).

For computer software, there are standard definitions for languages, such as Fortran, C, and Java. There are also standards for software libraries, such as the OpenGL graphics library and the message passing library for communication among distributed processors. Standards for software libraries define the function of individual modules and how the modules are to be invoked from a standard language like Fortran. The definitions of a library in terms of a given language is called a binding, or an applications programming interface or API.

Various scientific organizations promote standards that are of particular relevance to their own areas of interest. The International Standards Organization (ISO) and the American National Standards Institute (ANSI), as well as other national standards bodies, promulgate standards for a wide range of products and processes.

Software systems can be classified as open, meaning generally that the definition of the system is controlled by a standards organization, or proprietary, meaning that a single, private entity controls the definition of the system.

Source
Whether or not a software system is open, the source of the program may or may not be open. Open source software is software whose source code is freely accessible. Two major advantages of open source software are the ability to inspect the code to determine what computations are actually performed (they may not be the same
3.4 Scientific Software

The value added by the commercial software houses is often in the form of support and help for the users. The most important aspects of support are responsibility and responsiveness. Whenever the responsible support entity (person, group, company) lacks incentive, the support will be less than satisfactory. A commercial software house has an incentive of profit. Noncommercial software developers often are motivated by a desire to see their work have impact.

Language Constructs for Parallel Processing

Large problems in scientific computing often require parallel processing. As we discuss in the section on computer architecture (Section 3.3), the variations in the design of parallel processors may require ad hoc software development. Great efficiencies in development of application software could be achieved by computer language constructs with similar interfaces for various computers. Although this may not be possible in general, a certain degree of commonality may be possible separately within two groups architectures, the symmetric multiprocessors that share memory, and the distributed memory clusters.

Computer languages to support parallel processing must provide constructs to create processes, to communicate among processes, and to control processes. These constructs have names that are fairly standard, but the details may vary from one system to another.
HPF
Fortran D
Fortran 90
Fortran 95
Fortran 2000
OpenMP

- Process creation
  - forall
  - fork (and join)
- Communication
  - send
  - receive
- Process control
  - lock
  - unlock

An example of the forall construct is the following:

```c
/* Matrix multiplication C = A*B */
forall i := 1 to n do
  forall j := 1 to n do
    compute C[i,j] as vector product of
    row i of A with column j of B
```

Such constructs can be added to programming languages such as Fortran and C as language extensions or as program units (subroutines or functions). Several packages or special languages have been developed to support parallel processing, such as for C:

- C* (based on ANSI C, arrays),
- MultiC, and
- DataParallel C;

and for Fortran:

- HPF (High Performance Fortran),
- CM Fortran,
- Fortran D,
- Fortran 90, Fortran 95, and Fortran 2000.

**Symmetric Multiprocessors**

OpenMP is a standard of shared memory parallel programming. The OpenMP application program interface, which was jointly defined by a group of major computer hardware and software vendors, supports shared-memory parallel programming in C/C++ and Fortran on a variety of architectures, including Unix platforms and Windows 2000 platforms. OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for different platforms. More information on the OpenMP standard is available at

www.openmp.org/
Distributed Memory Processors

Network computing on clusters is emerging as one of the most powerful technologies for parallel computing. Central to network computing is message passing and the message passing interface standard, MPI, which is becoming widely used, is significantly contributing to the success of network computing. MPI was developed by a consortium of computer vendors, software developers, and, most importantly, users. The standard specifies a library of functions and Fortran and C bindings for them. Although there are over 200 functions in the MPI library, a small subset of these functions can allow users to develop parallel programs relatively easily. A list of the functions and more information about them can be obtained at www-unix.mcs.anl.gov/mpi/www/

Expert Systems

Computers not only perform numerical computations, provide support for development of text, produce graphics, and manage and search databases, they can also provide decision support at a much deeper level. By evaluating the same decision rules, they can, up to a point, simulate the thinking of an expert in some field of knowledge. The objective of software known as expert systems is to do just that. While the hyperbole accompanying discussions of computer software that simulates “thinking” or that demonstrates “artificial intelligence” often renders the discussions uninteresting, clearly the role of “expert” is appropriate for computers. The same hyperbole that voids reason from the discussions has led to unmet expectations and consequent disillusion with purported expert system software.

Research into artificial intelligence has yielded deep insights into human reasoning, and has resulted in some software for relatively simple decision problems. On the other hand, the more useful software expert systems have generally been more modest “advisors”. A statistical analysis package, for example, when called on to perform computations for a particular analysis, can quietly perform many additional computations that may or may not be important for the analyst to know about. In a regression analysis, it is now common for software packages to provide options for computing various measures of outliers; but an expert system may do this without being asked, and report only results of interest. The reports could easily be accompanied by a discussion and references to the literature for what the analyst should consider. DuMouchel (1990) describes a system called Mulreg that provides advice, in addition to just performing the computations for a regression analysis. This system is not billed as an “expert system”; rather it is a “less stupid” system, meaning it does more than just perform the simple computations implied by the user’s request for a regression fit. It is likely that the useful developments in expert systems in the near future will be evolutionary from software that merely provides advice in the form of additional computations to test assumptions and context-sensitive information.

Two other systems for statistical analysis, SPRINGEX and the STATISTICAL NAVIGATOR are described by Raes (1993). While Mulreg performs the computations for a regression analysis, as well as providing advice on types of analyses that may be appropriate, SPRINGEX and the STATISTICAL NAVIGATOR are intended only to help choose an analysis method.
There are various software packages that incorporate “expertness” to a greater or lesser extent. Most of these must be viewed as preliminary or experimental programs. There are several articles in Hand (1993b) that provide a survey activities in developing expert systems for statistical data analysis, and Hand (1993a) discusses some of the future prospects.

### 3.4.2 Software Parts Technology

A good program is usually modular; that is, it is composed of smaller program units that perform specific tasks. Programs developed in this way can be understood and modified more easily. Once a module is working correctly to perform a given task, it can be used again for that task in a different setting.

![Modular Structure of a Program](image)

**Figure 3.25.** Modular Structure of a Program

In most fields of engineering and construction there are standard catalogs of parts, such as electronic components or plumbing systems. The parts are specified very precisely in terms of their sizes and their functions. The type of input and the type of output are exactly specified. Rice (1993) has promoted a similar scheme for software. The precise specification of the software is the main issue. The input, the functions, and the output must be clearly specified, as if the software part were an electronic component to be used in some larger system. Because the input is provided by the user (instead of the developer) and because the correctness of the function depends on the conformity of the input to the specifications, the software part must perform checks on the input to insure that it meets the specifications. This is a failing of many pieces of software. It is only in checking the input, however, that the robustness of the software part can be insured.

Another problem that software developers face is the differences in the environments in which the software will be operated. Major differences exist among the hardware and the system software required for proper execution of the program. The software developer generally seeks to make the software portable from one platform to another, but this is rarely achieved completely.
An example of a set of well-defined operations that lend themselves to implementation in a standard set of software modules are the basic linear algebraic operations. A set of routines called “basic linear algebra subprograms” (BLAS) to perform these standard operations was defined by Lawson et al. (1979) and incorporated in the widely used LINPACK software package (Dongarra et al., 1979). These routines have standard and precise specifications of the input and the output. The specification is the same on all computing platforms, but the actual coding may be quite different. There was an emphasis on speed, so the routines do not do much checking of the user’s input. The original set of routines were all for vector operations; and matrix operations, such as multiplying two matrices were generally built using the BLAS. A set of BLAS, called BLAS-2, for operations involving a matrix and a vector were defined by Dongarra et al. (1988), and a set called BLAS-3 for operations involving two matrices were defined by Dongarra et al. (1990). (The original set of BLAS is now called BLAS-1.)

Another type of operation that should be standardized is message passing in a distributed-memory parallel processing environment. The MPI (message passing interface) standard is being developed (primarily at Argonne National Labs) for the purpose of standardizing message passing across languages and systems. Use of the MPI standard will help make parallel programs more portable (see Gropp, Lusk, and Skjellum, 1999, and Gropp, Lusk, and Thakur, 1999). Pacheco (1997) discusses program development using MPI.

IBM has built the Message Passing Library (MPL) in both Fortran and C that provides message passing kernels. MPI bindings are being defined for Fortran 95 and C++. Several commercial software packages are beginning to adopt MPI as a parallel programming model.

Much as LINPACK helped to bring the BLAS into common usage, the Scalable Linear Algebra Package, ScaLAPACK (see Blackford et al., 1997), is helping to standardize routines for use in distributed memory computers and networks of workstations.

### 3.4.3 Object-Oriented Software

Another approach to the problem of software reuse is an “object orientation” (OO). The object orientation also seeks to help both the developer and the user by putting the emphasis on the types of “objects” that are being manipulated. Recognizing the nature of the object will help the programmer or the user to avoid misusing the object. It may also help to simplify the design and use, because once the object is understood clearly, the kinds of manipulation appropriate for that kind of object are known.

Object-oriented programming (OOP) is based on the concepts of methods and data types. The basic idea is that methods (procedures, or groups of instructions) apply in specific ways to certain data types and that the emphasis should be on the things themselves. Object-oriented programming thus deals primarily with things (nouns). Some people say that “traditional” programming deals primarily with methods or functions (verbs).

A task that is identified in a single way may actually be a different task for different types of data. This has advantages in simplicity of design, but has disadvantages in usage because the user must remember precisely the nature of the object. The
A fundamental concept in the object orientation is called a class, which is a collection of methods and the types of data the methods operate on. The definition of a class specifies how its members behave, how they are created, how they are manipulated, and how they are destroyed. An object is an instance of a class.

Some of the characteristics of the object orientation are the following.

Encapsulation. The information about how methods apply to specific data types.

This information is the essential part of the definition of a class. The programming language must make provision for supplying this information.

Inheritance. Building new derived classes from existing base classes. The derived classes inherit the properties of the base classes, but may include additional action and data. This results in a hierarchy of classes.

Polymorphism. Different meanings of functions depending on the operand objects.

The operation is overloaded.

We have seen examples of overloading in such trivial instances as the operator “/” that simulates division. The operator has different meanings depending on whether the operands are fixed-point or floating-point numbers. Operators in an object-oriented system are data-driven; that is, the precise definition of the operation depends on the nature of the data to which it is being applied.

The user of an overloaded operator must be careful to remember the meaning of the operator in the specific instance in which it is used. So long as the user is aware of this, the overloading brings a simplicity to the process. If the user forgets the class of the operands, however, the results may not be as expected.

An example of a very useful overloaded operation is “Print”. In a system that supports various types of data, such as matrices, time series, and tables for example, the “Print” command can result in quite different, but entirely appropriate output.

In the object orientation there is a strong emphasis on abstract data types.

Traditional data types may emphasize the way data are stored in computers, fixed-point or floating-point numbers, for example. Data abstraction allows the definition of data types that correspond to specific tasks. An abstract data type could be a statistical data set, a purchase order, or a spreadsheet, for example. An OOP language allows newly-defined abstract data types to be treated to some extent as if they were built into the language. (There are often restrictions on the extensibility of the language; for example, the precedence of operators in C++ is fixed, so operators the user may add must respect a precedence that is independent of the meaning the user attaches to the operator.) The definition of a simple abstract data type usually begins as an assembly of built-in data types. Higher-level abstract data types allow composite objects, that is, objects that contain other objects.

Some languages or software systems that support an object orientation are Simula-67, SmallTalk, C++, and R. Some of the ideas are present to a greater or lesser extent in other languages, such as Fortran 95 and Ada. Some distinctions among OOP languages and systems are:

- pure/hybrid, that is, whether the object orientation is grafted onto an existing language (C++ and R are hybrids; SmallTalk is not)
- mono-rooted/multi-rooted, that is, whether all classes inherit from a single, universal class (C++ is multi-rooted)
3.4 Scientific Software

- single/multiple inheritance, that is, whether a subclass can inherit from more than one class. (C++ can)
- compile-time/run-time, that is, when the overloading is resolved. (C++ resolves at compile time, R resolves at run-time.)

Chambers (1992) discusses the object orientation in the design of the S (or S-Plus and R) statistics software. Classes of objects in S include matrices, data frames, and fits. A single task, such as to print, may be resolved in several different ways. When a matrix is printed, for example, it is printed as a rectangular array. When a “fit” is printed, coefficient estimates and various sums of squares may be printed together with descriptive labels. (A “fit” object is the result of using observed data to fit some model according to some criterion.)

The object orientation may simplify construction of programs for parallel processing by letting the system handle multi-data in a transparent fashion. Processes can be treated as objects. One object communicates with another by message-passing. The objects themselves are called servers. The articles edited by Bruaset, Langtangen, and Quak (2000) describe various applications in scientific computing in which an object-oriented approach successfully addressed the problem. One example is the Matrix Template Library (Siek and Lumsdaine, 2000).

3.4.4 Software Development

Compared to other construction activities, there is an extremely wide range in the scale of software development. At one end of the scale are the small programs written for narrowly defined, and relatively simple tasks. At the other end are the massive projects that control very complex systems, such as a communications network or a space station. A program to do a least squares regression and compute the usual summary statistics may consist of only about a hundred lines of code in a language such as Fortran or C. Although this program may interact with other programs for data manipulation or graphics, it is relatively self-contained. The code for Windows 2000, which contains many tightly integrated modules, consists of over thirty millions lines.

The cost of software development projects can be enormous, but more interesting, the difference in the costs based on development practices might be two orders of magnitude. Although an objective of software engineering is to define procedures for efficient software development, best practices in software development do not scale well; they depend on the size of the project.

Over the years a number of techniques and practices have been advocated for the improvement of software development. Some, such as the “chief programmer team” approach of the 1970’s relate more to the management issues; some, such as “structured programming” and the software parts approach, relate to code development; and some, such as an object-oriented approach, rely on the supporting software development system to a large extent. One of the main benefits of any of these approaches, regardless of its intrinsic value (or whether it is really “new”), is that the structure imposed causes designers and programmers to think. The open source movement is an approach to the improvement of software development that relies on the fact (or the hope) that if users are empowered to change software, the collective efforts of a very large number of users will improve the software.
Software Development Phases

Software development consists of a number of interrelated steps, beginning with an analysis and specification of the requirements. The next phase consists of design at an increasing level of detail. Design considerations feed back into the requirements specifications. After a detailed design is in place, coding and debugging can begin. This phase may result in some modifications to the design. As code is developed for the individual units of the software, those units are tested against their detailed design specifications. As program units are validated, their interactions are tested, and finally the system undergoes extensive testing. At this point the software can be put into use. It then enters a maintenance phase.

Without a rigid discipline, software development may proceed in a haphazard fashion. Because coding is easy, there is a tendency to rush into this phase before the design is complete. This leads to costly modifications to bring the code into conformity with the evolving design.

Although it is often the most neglected phase of the software development, the design activity is probably the most important. It is in this phase that questions of extensibility should be addressed and designed in. The rule should be “design the biggest and best system imaginable.” It will probably not be necessary or even feasible to implement that system — until the requirements change. And they will.

Software systems that are used for any extended period of time usually undergo several revisions. This does not necessarily mean that the first version was deficient. Sometimes it is because users’ needs change because of secular changes in technology.

The design of a software system specifies what is to be constructed immediately; that is, what meets the current requirements. The design also includes additional features. Even if the additional features are not to be implemented in the initial system, consideration of them has affected and probably improved its design. The design of the biggest and best system allows these features to be added later in a way that blends in with the overall system, rather than appearing as add-ons. Many of the additional features may never be implemented; but the consideration of them improved the overall design process and did not negatively affect the system delivered.

Brooks (1975), in an interesting collection of essays, emphasizes the evolving nature of a software project by admonishing the developer, “plan to throw one away; you will, anyhow”. A challenge of software engineering is to design and develop a system only once without any iteration. Most of the people who really believe this is possible are the ones who have either worked only on small projects or have not participated in more than one or two revisions of a system.

Software Reviews

In a software development project of any significant magnitude, it is necessary periodically to conduct a review of all phases of the project. This is particularly important in a software development project involving several people.

For the code itself there are three types of reviews corresponding to various levels of formality:

- walkthroughs
- readings
• inspections

A walkthrough is an informal review process in which a programmer leads the programming team through the code. Each member of the team tries to understand the design and how the code functions. The fresh viewpoints taken by programmers who did not write are effective in uncovering errors in the code. A walkthrough generally should not last more than an hour.

For a code reading, each programmer on the team is given a segment of code to read prior to a meeting to discuss the code. The meeting is led by a programmer, and all members of the team discuss the segments of code they were assigned.

Code inspections are very formal reviews led by a moderator. Several members of the team are assigned duties to take notes, and one member of the team is assigned the task of preparing a written description of the inspection.

See McConnell (1993) for more discussions of software reviews.

Software Development Tools: UML and PDL

A variety of methods and systems have been developed over the years to facilitate the process of software development. Reading current literature on the subject often gives one a sense of having encountered the same idea before, possibly with a different name. Two systems currently in use are worth mentioning. The Unified Modeling Language (UML) is an object-oriented approach for specifying, visualizing, constructing, and documenting systems by use of a formal language, with a well-defined syntax. In addition to standard text, the language has a set of icons to express actions and relationships, somewhat similar to the formal diagrams once used in flow charts. UML is used for higher-level definition of a system. A supporting language for more precise definition of processes is called the Object Constraint Language (OCL). See Alhir (1998) for a description of UML and OCL, as well as some examples of their applications.

At a lower level, the Program Development Language (PDL) is useful for building programs. PDL is an English, or other natural language, specification of the steps in accomplishing a task. A program written in PDL would be at a level slightly higher or more task-oriented than the style in which algorithms are normally described in this book. For example, a PDL version of the program described in Algorithm 13.1 (page 576) is shown in Figure 3.26. See McConnell (1993) for more discussions of PDL and examples of its usage.

Form a set of the edges of the graph, together with their associated distances.
Check that the edges form a connected graph.
Choose the edge with smallest distance,
  remove it from the set of edges and put it in the tree.
If another edge is available that is not in the tree,
  Find shortest edge and remove it from the set of edges.
  If it does not create a cycle in the tree, add it to the tree.
Endif

Figure 3.26. Minimal Spanning Tree Algorithm in PDL
Automatic Software Development

Far more interesting approaches to software development are those involving automated program building. Ideally, the scientist would describe the problem to be solved in the natural language for the science. The computer, understanding the description in the language of the appropriate scientific domain, would solve the problem. Actually, this would not be sufficient. We would also require that the computer provide a description of the method of the solution. Ideally, we would also require that the program built be scalable for other problems.

An interesting idea for automatically developing programs from a problem specification has been explored by Koza (1992, 1994b) and Koza, Bennett, and Andre (1999). They define a relatively simple set of activities or program fragments, and a set of ways of performing those activities, such as looping, iteration, and recursion, or incorporation into a separate program module. The set of basic program fragments is sometimes called the “primordial soup”. They then define a measure of the difference of the output of a given program and the problem specifications. They then form programs randomly (but according to fixed rules) from the set of possible activities. In this process the original set of program fragments grows to include fragments with control structures and complete program modules. The output of a given program is compared to the problem specification, and the “fitness” of that program to solve the problem is computed. If the fitness is better than other programs that have been tried, the given program is accepted as the current best. Even if the fitness not better, the given program may replace the current best with a nonzero probability. This method is a genetic algorithm (see Section 7.2.2), and the method of program creation is called genetic programming.

For example, if the problem specification may be met by a program such as shown in Figure 3.25, at some point in the random search for a working program, we may have the program shown in Figure 3.27. At that point, a node is chosen randomly (the node corresponding to “Sub A”), and modification is chosen randomly. Suppose it is decided to add an activity in Sub A. A new node is formed (marked “????”), and a new module is placed at that node. The random selection may result in Sub D being placed at that node. After the fitness of the resulting program is evaluated, it may be decided to retain Sub D at that node. If so, the process continues, and ultimately will come back to a situation like Figure 3.27 again (unless the process is terminated). At that time, Sub B may be selected. When the fitness of the resulting program is evaluated, that program will be selected. The rules for generating programs (reproductions, mutations, and so on), for evaluating the fitness, and for termination of the process must all be carefully specified.

Koza et al. (1999) report the development of many successful programs by this method. Many of the applications they have studied are in filter design. The ideas and methods will have far-reaching effects in many areas of software development.

3.4.5 Software Testing

Although we discuss software testing near the end of the section on software, testing should be considered at every phase of software development. The design of the software should include designs for testing.

The problem of testing mathematical software is as old as mathematical software, and formal work on the problem is almost as old as any systematic work on
the development of mathematical software. It was part of the work on computer implementation technology at the National Bureau of Standards in 1952 (see Hoffman et al., 1953).

Despite advances in testing methodologies, there is no “magic bullet”. Testing is part of what has been called “experimental computer science” (Denning, 1980), and it retains many of the hallmarks of an experimental science.

Testing software is somewhat like quality control by acceptance sampling. There is a large literature on acceptance sampling, in which the hypergeometric distribution is contorted mightily to yield significance levels and confidence limits. W. Edwards Deming preached a different approach to quality improvement: focus on the front-end of the process, rather than on the back-end. The Deming approach can be effectively used in software development.

Just as in other industries, many issues of testing and development of high-quality software have more to do with management and design, rather than analysis of algorithms or sampling of the performance of the algorithms. Quality in software is most often a result of an organized, systematic approach, in which design is not relegated to a role secondary to development.

As we mentioned previously, one of the main reasons for the existence of poor software is that the software was developed ad hoc, but fell into general usage. The first rule in design is to know the nature of the applications.

Computer testing must be designed into the development process, so that it occurs at every stage. The various components of software, the algorithms (that is, the mathematical processes), the computer implementation of the computational algorithms, and the user interface must all be considered both separately and together.

Testing algorithms is the least experimental of the component testing. Convergence results, stability analyses, and other purely analytic results may be used to identify good algorithms, and the good algorithms become part of the design specifications. Testing is both a low-level and a high-level process. Testing is performed at the level of each module.

The user interface must also be tested. As with other aspects of testing, it should begin in the early design phase. It is the most subjective, and is often best accomplished by focus groups.

A problem with any kind of software testing is that the manufacturer or developer will build to the test; that is, the product will perform well on whatever set of tests are publicized, but may not perform as well on other data sets.
Test Datasets

In general, there are two types of datasets for use in testing: artificially generated datasets; and sets of problems with characteristics that are more-or-less known, many of which arose in real-life applications. A set of test datasets is often called a “testbed”.

There are standard test sets for use in various types of applications. Optimization probably has the richest collection. The tradition of developing tests for optimization problems goes back to Hoffman et al. (1953). Bongartz et al. (1995) describe an extensive testbed for optimization called CUTE. Many of the test sets are available from netlib (see the Bibliography for information on netlib).

Standard test datasets are hoped to be representative of real-world applications because they arose in real-world applications. The question of how well they span the space of possible applications remains, however. One of the more famous datasets used in testing software for regression computations is due to Longley (1967). This dataset consists of six economic variables observed yearly from 1947 to 1962, plus a variable for the year. There has been much discussion of this dataset in the statistical literature. Many have questioned whether the regression model he used is an appropriate set of data. It probably is not, but the dataset and Longley’s article were important because they pointed out some serious numerical problems with some software for regression computations.

Artificially generated datasets are sometimes used systematically to probe the limits of numerical accuracy delivered by a program. An example is the set of polynomials used by Wampler (1970) and (1978) to assess the accuracy of regression software. Simon and Lesage (1989) describe methods for systematically generating data for assessing the accuracy of computations for analysis of variance. In order for an artificially generated dataset to be useful as a test dataset, its solution must be known. There are methods of generating datasets with known solutions for a variety of problems. The polynomial regression datasets of Wampler, for example, are generated with zero residuals, so the solution is known. Velleman and Allen (1976) give methods for generating more general least squares regression test problems with known solutions. Kennedy, Gentle, and Sposito (1977) show how to generate test problems for regression fitting under criteria other than least squares. For the problem of inverting a matrix, Ericksen (1985) shows how to generate test matrices with known inverses. (See Section 5.4.4, page 339, for a description of Ericksen’s method.)

Some artificially generated datasets are generated randomly. They also are constructed to cover a range of characteristics felt to be important and to probe the limits of the software/algorithm. For example, the test datasets may span a range of increasing condition numbers for solution of linear systems.

Gentle, Narula, and Sposito (1992) describe how to generate random datasets that span a certain class of problems by using a testbed of fixed, standard data sets. They use the idea of data-based random number generation. (The method implicitly estimates a probability density and then samples from the density.) The idea is that the standard testbed represents a sample from some population of problems. The population has variables that include (within reason) the size of the problem and other characteristics in addition to the data (that is, data plus metadata). A random dataset is chosen; “nearby” datasets are identified, and random linear combinations of metadata (if practical) and of data are generated.
The National Institute for Standards and Technology (NIST) is collecting datasets with certified values for a variety of statistical methods. The collection is called Statistical Reference Datasets (StRD). The collection includes both artificial and real-world data of varying levels of difficulty. Generated datasets are designed to challenge specific computations. These include the classic Wampler datasets for testing linear regression algorithms and the Simon and Lesage datasets for testing analysis of variance algorithms. Real-world data include challenging datasets such as the Longley data for linear regression, and more benign datasets such as the Daniel and Wood data for nonlinear regression. Certified values are “best-available” solutions. The certification procedure is described in the web pages for each statistical method. See

www.itl.nist.gov/div898/strd

See McCullough (1998, 1999) for further discussions of these datasets and other tests for software.

Assessing the Accuracy of a Computed Result

In real-life applications, the correct solution is not known. (This is also often the case for randomly generated test datasets.) We would like to have some way of assessing the accuracy using the data themselves. Use of interval arithmetic (see page 52) can be used to bound the correct result. Corliss and Rall (1987) and Wang and Kennedy (1992) have described “self-validating” approaches for numerical quadrature that are based on interval arithmetic.

Sometimes a convenient way of assessing the accuracy of the computations in a given problem is to perform internal consistency tests. An internal consistency test may be an assessment of the agreement of various parts of the output. Relationships among the output are exploited to insure that the individually computed quantities satisfy these relationships. A simple example of this is the decomposition of the sum of squares in an analysis of variance program. (The infamous negative $F$-value may result if the sums are not computed accurately.) Other internal consistency tests may be performed by comparing the results of the solutions of two problems with a known relationship. Mullet and Murray (1971) describe consistency tests for regression software. Consider fitting the regression model

$$y = \beta_0 + \beta_1 x_1 + \cdots \beta_m x_m.$$ 

If we also regress $y + dx_j$ on $x_1, \ldots, x_m$, we know what relationships should exist among the two sets of outputs. If these relationships do not obtain, the analyst has strong reason to doubt the accuracy of the computations. Another perturbation of a regression problem with a known exact effect is the permutation of the rows or columns.

Known or computable relationships among results from different inputs can be used to identify problems in computations. If a function is known to be monotonic, the results can be inspected for monotonicity. Second or higher order differences may be used if any properties of the derivatives are known. Cody and Stoltz (1991) describe methods using Taylor expansions for assessing accuracy.

Another simple internal consistency test is to use two levels of precision in the computations. In using this test, one must be careful to make sure that the input
data are the same. Rounding of the input data may cause incorrect output to result, but that is not the fault of the computational algorithm.

Internal consistency tests do not confirm that the results are correct; they can only give an indication that the results are incorrect.

**Timing Program Execution**

In many programs about 80% of the execution time is spent in about 20% of the code. (This is called the Pareto Principle, and it applies to usage of many other resources as well.) If the parts of the program that are most heavily used are identified, special effort can be put into those parts to make them faster. Also, sometimes when the execution time is greatly unbalanced, it may be possible to restructure the program so as to relieve bottlenecks.

In programming languages like Fortran and C it is possible to instrument the source program so that a program called a profiler will measure the amount of time spent in each program module. A profiler should be run on any program that is to be used extensively. Profiling is generally not very useful during the very early stages of software development, but it can be very informative on code during the middle stages of development.

It is also of interest to know how fast a given computer runs. Because different computers run faster on some types of problems than others, measuring the speed of a computer is not a simple task. There are various sets of tests to assess the speed of software and computers. Some of the more famous of these timing benchmarks are

- Whetstone
- Dhrystone
- Linpack (based on the LINPACK software package, Dongarra et al., 1979)

The Linpack benchmarks are available in Java and can be invoked from [http://www.netlib.org/benchmark/linpackjava/](http://www.netlib.org/benchmark/linpackjava/)

- NAS Kernels and Parallel Benchmarks (from the Numerical Aerodynamic Simulation Program at NASA Ames Research Center)
- Livermore Loops
- Specmark (various sets of programs for measuring speed of fixed-point, floating-point, or mixed computations developed by the Standard Performance Evaluation Corporation, Manassas, Virginia)
- Slalom (Scalable language-independent Ames Laboratory One-Minute Measurement)

Hardware and software manufacturers and consumers use these tests to compare products. The major consideration in any kind of efficiency testing is the mix of operations to include in the benchmark.

Hockney (1995) provides a general introduction and overview of benchmarking, along with a description of a specific activity called PARKBENCH.

### 3.5 Scientific and Statistical Databases

Statistical applications of computers have always been as much for management of data as for the computations on the data. Management of large amounts of data
requires efficient algorithms for storage and retrieval and intelligent user interfaces for access. Research in the general area of databases has constituted a major subdiscipline of computer science, and has resulted in a relatively mature set of software for database management.

3.5.1 Data Structures and Characteristics of Statistical Data

For many datasets occurring in scientific applications, the rectangular array of a “flat file” is adequate. The array is a two-dimensional table. In the rectangular array for statistical data, the columns generally represent variables, and the rows, observations. Such data constructs were implemented in the very early statistical analysis packages of the 1960’s, such as Omnitab. The elements in a given row of the table are related, because they belong to a given item or observation. In many software systems the standard paradigm is the two-dimensional table in which columns represent variables and rows represent observations. A certain amount of metadata, such as variable names, ranges, and so on, must also be provided somehow. These are often contained in separate associated arrays. In systems such as C, Fortran, or Matlab, when the variables themselves are of different types, some mechanism for forming a more general structure for linking the tables is necessary. The individual arrays are a simple type of relational database. The more general relational database model is a collection of tables (see Codd, 1990).

Many scientific and statistical databases are far too complicated to be represented in a flat file. The relationships among variables may require complicated trees to describe them. Individual elements may not have the same level of reliability. In extreme cases, the elements may even be missing, but must be dealt with in an analysis. Because of the special needs of statistical data analysis, often a general-purpose database management system (DBMS) is not adequate. Instead, statistical data are generally managed by special-purpose software, usually incorporated in a statistical analysis package (see McCarthy, 1979, and Gentle and Bell, 1984).

Shoshani (1991) identifies several special needs of statistical databases that transcend the abilities generally available in a relational DBMS. One major advantage of a DBMS is that it provides a data model or abstraction for representing the structure and semantics of the data. Shoshani shows, however, that the data models appropriate for statistical databases are often much more complex than those available in a general-purpose DBMS, and gives as examples complex tables, images, geometric structures, and sequences such as DNA or temporal sequences. Even statistical database software provided by a data analysis package could not handle some of these kinds of databases. The special characteristics of scientific and statistical databases and methods for dealing with them are the subjects of a series of conferences whose proceedings are published by the IEEE Computer Society (Proceedings of the xxth International Conference on Scientific and Statistical Databases; the 10th symposium was held in 1998).

The kinds of needs for a database specified by Shoshani are not unique to scientific or statistical databases. They do, however, go beyond what is generally provided in commercial relational databases, which tend to be oriented toward handling transactions for the items. Object-oriented database systems may be more appropriate for working with the more complicated models.

R or S-Plus defines a dataframe object that can be built up from more elementary data objects. The single-variable data objects include time series, factors, and so on,
as well as the types that depend on whether numeric or character data are being represented. The dataframe is a composition of the individual variables, together with metadata including the names of the variables, for example.

**Missing Values**

Another consideration for statistical databases that requires special management capabilities is missing values. The first issue is how to represent missing values (they are not just 0’s or blanks). If there is only one type of missing value, as is usually the case, NaN (“not-a-number”) is usually a good choice for the missing value. NaNs may not be portable, however, even over machines that implement the IEEE floating-point standard, so occasionally some valid floating-point number that does not occur in the actual data must be used, at least in the porting process.

A more important issue are the statistical questions of how to handle the missing values in an analysis. For simple univariate computations, the obvious way to deal with missing data is just to leave them out of the computations. The statistics are then computed using the available data, with the obvious adjustments to the sample size and the degrees of freedom. The problem with this approach is that it implicitly assumes that the missing values have the same characteristics as the values that are present. This is often not the case, because, especially in observational studies, the missing values occur because of a subject’s refusal to supply data. Persons being surveyed who refuse to respond often are different in some way that is material for the survey from those who are willing respond. Software for statistical databases needs to be able to distinguish between different kinds of missing values, but the available database software usually does not do this (see Klemsin, 1991). In this case the user must devise special codes for the missing values.

There are other cases in which a statistical design assumes some kind of balance among various factors being investigated. When some data are missing, the analysis often must be changed because the data are no longer in balance.

**Standards for Scientific Datasets**

The content and structure of datasets vary so much that it is difficult to define common structures. A useful approach is to develop standards for metadata, that is, information about the dataset, such as descriptions of the meanings of the variables, how they are stored, and so on. In some cases the metadata are part of the same file as the dataset itself, and in other cases the metadata are stored in a linked file. Brown et al. (1993) describe some of the efforts toward standardization of scientific datasets.

Two standards that are widely used for scientific databases are the Common Data Format (CDF) and the Hierarchical Data Format (HDF). CDF, which was developed by the NASA National Space Science Data Center, is based on a multi-dimensional array model. HDF, developed at the National Center for Supercomputing Applications, is based on hierarchical relationships and dependencies among data. CDF and HDF are both scientific data management software packages and format specifications. Although the two models differ in the way in which their inherent structures are defined and accessed, there is a large overlap in the types of scientific data that each can support, and in fact, HDF is widely used for NASA data.
More information about these can be obtained from the Web sites. The top-level Web sites at both organizations are stable, and provide links to the Web pages describing the formats. The National Space Science Data Center is a part of the NASA Goddard Space Flight Center. Links to information about CDF and other data standards are available at

http://nssdc.gsfc.nasa.gov

Information about HDF can be obtained from links at the National Center for Supercomputing Applications site

http://www.ncsa.uiuc.edu

In the CDF structure, a collection of data elements is defined in terms of a variable. Variables can be completely independent of one another, or can be grouped together to express dependencies. CDF supports extensive metadata capabilities called attributes, which allow descriptions of the contents of a CDF file. CDF supports two interfaces from which a CDF file can be accessed. The Internal Interface provides access to all of the functionality supported in CDF through subroutine calls. The Standard Interface is built on top of the Internal Interface and consists of a smaller number of subroutines. The Standard Interface allows novice programmers to create a CDF data file quickly and easily.

NetCDF (network Common Data Form) is an interface for array-oriented data access and a library that provides an implementation of the interface. The NetCDF library also defines a machine-independent format for representing scientific data. Together, the interface, library, and format support the creation, access, and sharing of scientific data. The NetCDF software was developed at the Unidata Program Center in Boulder and is freely available. Information on NetCDF is available at

http://www.unidata.ucar.edu/packages/netcdf/

The Unidata home page contains more general information on data sharing and the standards to facilitate it. It is at

http://www.unidata.ucar.edu/

A useful tool for accessing data in NetCDF format and displaying and analyzing the data is the Grid Analysis and Display System (GrADS), developed by the Institute of Global Environment and Society (IGES). Information on GrADS is available at

http://iges.org/grads/

The HDF structure is based on a tagged format, in which each data object has a tag identifier. The basic structure of HDF consists of an index with the tags of the objects in the file, pointers to the data associated with the tags, and the data themselves. HDF supports a set of interface routines for each supported object.

An extension of HDF, HDF5, provides groups that contain other HDF5 objects. A group has two parts, a header, which contains a group name and a list of group
attributes, and a symbol table, which is a list of the HDF5 objects that belong to the group. An HDF5 dataset is stored in a file in two parts, a header and a data array.

The header contains information that is needed to interpret the array portion of the dataset, as well as metadata (or pointers to metadata) that describes or annotates the dataset. Header information includes the name of the object, its dimensionality, its data type, information about how the data itself is stored on disk, and other information used to speed up access to the dataset or maintain the integrity of the file.

There are two categories of data types, atomic data types and compound data types. Atomic datatypes are those that are not decomposed at the data type interface level, such as integers and floats. Some data types, called native types, are systemspecific instances of atomic data types. Compound data types are made up of atomic data types. Named data types are either atomic or compound data types that are have been specifically designated to be shared across datasets.

Properties of integer types include number of bits in the representation, order (endian-ness), and signed-ness (signed/unsigned). Properties of float types include the representation, either IEEE 32-bit and 64-bit, location of the exponent and mantissa, and the location of the sign bit.

Some software packages, such as PV-Wave, have extensive input/output facilities for HDF data.

3.5.2 Datasets in the Sciences

Commonalities in the nature of data in a scientific discipline, such as astronomy or genetics, allow widespread exchange of dataset among scientists working in those areas. Large-scale projects, such as the International Solar-Terrestrial Physics (ISTP) project and the genome project, have resulted in considerable attention being paid to standard methods of storing data in those particular fields. A very large proportion of scientific and statistical data from all fields have a location component. Sharing of data that were collected at the same location but for different purposes is often difficult because the format of the spatial component is different.

Spatial Data

The location component in a dataset may be in the form of variables on the dataset, usually the latitude and longitude, or it may be grouped into geographic regions, usually corresponding to political or administrative subdivisions. The U.S. Census Bureau uses a format called Topologically Integrated Geographically Encoded Reference (TIGER) to provide a standard method for geographical location within the United States. The TIGER files uses census tracts and street maps to provide location information. Demographic, economic, and environmental data can be associated with geographic locations in the TIGER files.

Geographical features, such as streets, political boundaries, topography, and other terrestrial features, are usually represented in one of two ways: as a raster grid or as a set of vectors or arcs. A raster image is a fixed set of pixels. It is resolution-dependent, so if it is displayed at a higher resolution, or if its size is increased, jagged edges may appear. A vector dataset or arc node dataset is made
3.5 Scientific and Statistical Databases

up of mathematically defined lines and curves. The definitions do not depend on the resolution. Modifications to the image, such as moving it or changing its size, are relatively simple and scalable because they are made to the mathematical definition.

A class of software designed to process spatial data or to incorporate location information in data analysis is called a geographic information system (GIS). A widely-used GIS is ArcView (see Hutchinson and Daniel, 1997, for a good introduction). Some statistical analysis packages such as S-Plus provide mechanisms for importing and exporting data from and to ArcView.

GIS software provides methods for processing digitized data representing geographic features. ArcView uses a format called shapefile to define regional boundaries and to produce maps. The R `polygon` function can also be used to draw maps from vectors representing latitude and longitude (or XY coordinates) of points on the boundary of a region.

**Earth Environmental Data**

Global climate modeling and intermediate to long range weather forecasting present some of the most challenging scientific problems. Modeling and prediction require massive amounts of data from all over the globe.

The ISTP project has a Coordinated Data Analysis Web site that provides access to a wide range of both global and space data at

http://cdaweb.gsfc.nasa.gov

The ISTP project promotes the use of the Common Data Format (CDF).

**Genomic Data**

Two of the most significant scientific challenges are the complete identification of genomes, most importantly of course the human genome, and the understanding of the DNA coding that produces proteins. DNA, RNA, and proteins are macromolecules that can be characterized by sequences of fixed sets of smaller molecules. In the case of DNA and RNA there are 4 such smaller molecules (nucleotides), and for proteins there are 20 smaller molecules (amino acids).

The human genome contains sequences of order $10^9$ nucleotides. A major task in the identification of the sequences is the construction of a very large database. Because of its size and the wet biology involved in its construction, teams from various institutes around the world are participating. The scientific skills are multidisciplinary, involving computer science, statistics, biology, and chemistry. This field is sometimes called bioinformatics (or just informatics). Waterman (1995) gives a good introduction to the field.

The most important evolving database of gene sequences is called GenBank. GenBank, which started in 1982, contains DNA sequences for many species in addition to humans. It is available over the World Wide Web from the National Center for Biotechnology Information of the National Library of Medicine at the National Institutes of Health. The URL is


The Web site provides various ways of accessing the data. Although the database is not well curated, its size makes it very useful.
An increasing proportion of stored computer information consists of images, both still images and video clips. This kind of data has a very different format and structure from numeric or character data. Cataloging, sorting, and searching non-text-based data presents major new challenges. Techniques in visual information retrieval are discussed extensively by Del Bimbo (1999).

### 3.5.3 Managing Data Collected from Various Sources

A clear description of the contents of a dataset is necessary in order for the data to be used effectively. Such metadata must define the quantities measured. It may also necessary to know how and when the measurements were taken. In some types of datasets it is necessary to know whether there are limitations on the use of the data.

#### Linking Disparate Databases

Data collected at different times and in different ways, and then stored in different formats can provide information about the same subject. The first challenge, of course, is just to become aware of the existence of the data. The Internet has helped in this respect. The next problem is the meaning of the metadata, that is, the information about what the data represent and how they were collected. Too often measurements in two different datasets are assumed to represent similar object classes, when in fact the data were collected in completely different ways. The metadata associated with the two datasets may indicate greater commonality than actually exists. In surveys in the social sciences, for example, the variable measured may be the response to a question that is an interrogatory sentence composed of twenty words. Converting the response to the twenty-word question to a count for a variable labeled “Favor X” can be (and is) done easily, but not necessarily accurately. A different phrasing of the same question could result in a different count for the simple variable “Favor X”. This, of course, is an inherent problem in social studies. The problem becomes even greater when different datasets of survey results are to be combined.

Except for measurements of very simple objects, this problem may occur in datasets in any of the sciences. Examples can easily be found in public health data, financial data, terrestial and environmental data, and agricultural data. The problems of differences in meanings of measured quantities are especially difficult when the data come from more than one country.

Another issue is what has been done with incomplete records, or more generally, how the data have been cleaned or postprocessed. The postprocessing can result in the dataset not providing a representative picture of the phenomenon being studied.

Beyond the problems of attaching common meaning to disparate data, there are the formatting problems that make analyses over different dataset more difficult. The use of common formats, as discussed earlier, helps to alleviate this problem. In any event, formatting differences are much easier to address than differences in meaning.

All of these problems, of course, arise in any data analysis. The problems are magnified, however, when the datasets in the analysis arise from disparate sources.
In the warehousing of data attempts are often made to resolve the disparities prior to the subsequent analyses using the warehouse.

**Security and Confidentiality of Data**

Data security and confidentiality are major concerns of database theory and methods in general. The data must be protected from unauthorized access and from malicious or accidental alteration. These needs must be addressed both by the DBMS and by the management practices of the owner of the database.

It is often of interest to compute summary statistics from a database while protecting the identity of the sources of the individual records. An example is computation of an average account balance, while being denied access to any individual’s account balance. There are, of course, many ways to do this either by the DBMS denying access to certain data fields that would identify the record or by encrypting some of the fields.

Problems of data confidentiality often arise when statistics are requested for a singleton subset. For example, if the average account balance is computed for all customers with a given postal code and there is only one customer with that given postal code, that customer’s balance could be determined. Subsets with only one member or a small number of members can often be formed in clever ways as the intersection of two subsets. The confidentiality problem that can arise in this way can be solved by the DBMS restricting access to any subset that has fewer than some specified number of members. The problem arising from computing statistics from two subsets with a small intersection may remain, however, because the computations can be made in two completely separate invocations of the DBMS.

Another way of dealing with confidentiality is data doping or pollution, which is the random alteration of data in such a way as to preserve the validity of most statistical summaries. See Silberschatz, Korth, and Sudershan (1998) for general discussions of methods of security and confidentiality of data in databases. The article by Fienberg and Willenborg (1998) and the special issue of *Journal of Official Statistics* in which it appeared discuss methods of protecting confidentiality of information in statistical databases.

**Exercises**

3.1. An important attitude in the computational sciences is that the computer is to be used as a tool of exploration and discovery. The computer should be used to check out “hunches” or conjectures, which then later should be subjected to analysis in the traditional manner. There are limits to this approach, however. An example is in limiting processes. Because the computer deals with finite quantities, the results of a computation may be misleading. Explore each of the situations below, using C or Fortran. A few minutes or even seconds of computing should be enough to give you a feel for the nature of the computations. In these exercises, you may write computer programs in which you perform tests for equality. A word of warning is in order about such tests. If a test involving a quantity \( x \) is executed soon after the computation of \( x \), the test may be invalid within the set of floating-point numbers with which the computer nominally
works. This is because the test may be performed using the extended precision of the computational registers.

a) Consider the question of the convergence of the series

\[ \sum_{i=1}^{\infty} \frac{1}{i} . \]

Obviously, this series does not converge in \( \mathbb{R} \). Suppose, however, that we begin summing this series using floating-point numbers. Will the computations overflow? If so, at what value of \( i \) (approximately)? Or will the series converge in \( \mathbb{F} \)? If so, to what value, and at what value of \( i \) (approximately)? In either case, state your answer in terms of the standard parameters of the floating-point model, \( b, p, e_{\text{min}}, \) and \( e_{\text{max}} \) (page 32).

b) Consider the question of the convergence of the series

\[ \sum_{i=1}^{\infty} 2^{-2i} . \]

(Same questions as above.)

c) Consider the question of the convergence of the series

\[ \sum_{i=1}^{\infty} \frac{1}{i} . \]

(Same questions.)

d) Consider the question of the convergence of the series

\[ \sum_{i=1}^{\infty} \frac{1}{i^x} , \]

for \( x \geq 1 \). (Same questions, except address the variable \( x \).)

3.2. We know, of course, that the harmonic series in Exercise 3.1c does not converge (although the naive program to compute it does). It is, in fact, true that

\[ H_n = \sum_{i=1}^{n} \frac{1}{i} = f(n) + \gamma + o(1), \]

where \( f \) is an increasing function and \( \gamma \) is Euler’s constant. For various \( n \), compute \( H_n \). Determine a function \( f \) that provides a good fit and obtain an approximation of Euler’s constant.

3.3. Machine characteristics.

a) Write a program to determine the smallest and largest relative spacings. Use it to determine them on the machine you are using.

b) Write a program to determine whether your computer system implements gradual underflow.

c) Write a program to determine the bit patterns of \(+\infty, -\infty,\) and NaN on a computer that implements the IEEE binary standard. (This may be more difficult than it seems.)
d) Obtain the program MACHAR (Cody, 1988b) and use it to determine the smallest positive floating-point number on the computer you are using. (MACHAR is included in CALGO, which is available from netlib. See the bibliography.)

3.4. Write a program in Fortran or C to determine the bit patterns of fixed-point numbers, of floating-point numbers, and of character strings. Run your program on different computers and compare your results with those shown in Figures 3.1 through 3.3 and Figures 3.11 through 3.13.

3.5. What is the numerical value of the rounding unit (1/2 ulp) in the IEEE Standard 754 double precision?

3.6. Consider the standard model (3.1) for the floating-point representation:

$$\pm 0.d_1d_2\cdots d_p \times b^e,$$

with $e_{\min} \leq e \leq e_{\max}$. Your answers may depend on an additional assumption or two. Either choice of (standard) assumptions is acceptable.

a) How many floating-point numbers are there?

b) What is the smallest positive number?

c) What is the smallest number larger than 1?

d) What is the smallest number $X$, such that $X + 1 = X$?

e) Suppose $p = 4$ and $b = 2$ (and $e_{\min}$ is very small and $e_{\max}$ is very large). What is the next number after 20 in this number system?

3.7. a) Define parameters of a floating-point model so that the number of numbers in the system is less than the largest number in the system.

b) Define parameters of a floating-point model so that the number of numbers in the system is greater than the largest number in the system.

3.8. Suppose that a certain computer represents floating point numbers in base 10, using eight decimal places for the mantissa, two decimal places for the exponent, one decimal place for the sign of exponent, and one decimal place for the sign of the number.

a) What is the “smallest relative spacing” and the “largest relative spacing”?

(Your answer may depend on certain additional assumptions about the representation; state any assumptions.)

b) What is the largest number $g$, such that $417 + g = 417$?

c) Discuss the associativity of addition using numbers represented in this system. Give an example of three numbers, $a$, $b$, and $c$, such that using this representation, $(a + b) + c \neq a + (b + c)$, unless the operations are chained. Then show how chaining could make associativity hold for some more numbers, but still not hold for others.

d) Compare the maximum rounding error in the computation $x + x + x + x$ with that in $4 \times x$. (Again, you may wish to mention the possibilities of chaining operations.)

3.9. Consider the same floating-point system of Exercise 3.8.

a) Let $X$ be a random variable uniformly distributed over the interval

$$[1 - .000001, 1 + .000001].$$

Develop a probability model for the representation $[X]_c$. (This is a discrete random variable with 111 mass points.)
b) Let $X$ and $Y$ be random variables uniformly distributed over the same interval as above. Develop a probability model for the representation $[X + Y]_c$. (This is a discrete random variable with 121 mass points.)
c) Develop a probability model for $[X]_c + [Y]_c$. (This is also a discrete random variable with 121 mass points.)

3.10. Give an example to show that the sum of three floating-point numbers can have a very large relative error.

3.11. Write a single program in Fortran or C to compute

a) $$\sum_{i=0}^{5} \binom{10}{i} 0.25^i 0.75^{20-i}$$

b) $$\sum_{i=0}^{10} \binom{20}{i} 0.25^i 0.75^{20-i}$$

c) $$\sum_{i=0}^{50} \binom{100}{i} 0.25^i 0.75^{20-i}$$

3.12. In standard mathematical libraries there are functions for $\log(x)$ and $\exp(x)$, called $\log$ and $\exp$ respectively. There is a function in the IMSL Libraries to evaluate $\log(1 + x)$ and one to evaluate $(\exp(x) - 1)/x$. (The names in Fortran, for single precision, are $\text{alnrel}$ and $\text{exprl}$.)
a) Explain why the designers of the libraries included those functions, even though $\log$ and $\exp$ are available.
b) Give an example in which the standard log loses precision. Evaluate it using $\log$ in the standard math library of Fortran or C. Now evaluate it using a Taylor series expansion of $\log(1 + x)$.

3.13. Suppose you have a program to compute the cumulative distribution function for the chi-squared distribution. The input for the program is $x$ and $df$, and the output is $\Pr(X \leq x)$. Suppose you are interested in probabilities in the extreme upper range and high accuracy is very important. What is wrong with the design of the program for this problem?

3.14. Write a program in Fortran or C to compute $e^{-12}$ using a Taylor’s series directly, and then compute $e^{-12}$ as the reciprocal of $e^{12}$, which is also computed using a Taylor’s series. Discuss the reasons for the differences in the results. To what extent is truncation error a problem?

3.15. Errors in computations.
a) Explain the difference in truncation and cancellation.
b) Why is cancellation not a problem in multiplication?

3.16. Assume we have a computer system that can maintain 7 digits of precision. Evaluate the sum of squares for the data set $\{9000, 9001, 9002\}$.
a) Use the algorithm in (3.5), page 59.
b) Use the algorithm in (3.6), page 59.
c) Now assume there is one guard digit. Would the answers change?

3.17. Develop algorithms similar to (3.7) on page 59 to evaluate the following.
a) The weighted sum of squares: $$\sum_{i=1}^{n} w_i (x_i - \bar{x})^2$$
b) The third central moment:
\[ \sum_{i=1}^{n} (x_i - \bar{x})^3 \]

c) The sum of cross products:
\[ \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y}) \]

Hint: Look at the difference in partial sums,
\[ \sum_{i=1}^{j} (\cdot) - \sum_{i=1}^{j-1} (\cdot) \]

3.18. Given the recurrence relation
\[ t(n) = pt(n/p) + cn, \]
for \( p \) positive and \( c \) nonnegative. Show that \( t(n) \) is \( O(n \log n) \). Hint: First assume \( n \) is a power of \( p \).

3.19. In statistical data analysis, it is common to have some missing data. This may be because of nonresponse in a survey questionnaire or because an experimental or observational unit dies or discontinues participation in the study. When the data are recorded, some form of missing-data indicator must be used. Discuss the use of NaN as a missing-value indicator. What are some advantages and disadvantages?

3.20. Suppose all of the \( n \) addends in a summation are positive. Why is the computation of the sum by a fan-in algorithm likely to have less roundoff error than computing the sum by a standard serial algorithm? (This does not have anything to do with the parallelism, and the reason does not involve catastrophic cancellation.)

3.21. Consider the problem of computing \( w = x - y + z \), where each of \( x, y, \) and \( z \) is nonnegative. Write a robust expression for this computation.

3.22. Sorting and merging.

a) Consider a very simple sorting method for an array \( a \), consisting of \( n \) elements, which is to be sorted in place. (This algorithm is not a good one; it is just used for illustration.)
   0. Set \( i = 1 \).
   1. Set \( r = i \).
   2. For \( j = i + 1 \) to \( n \), if \( a(i) > a(j) \), then \( r = j \).
   3. If \( r > i \), then
      interchange \( a(i) \) and \( a(r) \) and go to step 1;
      otherwise
      if \( i < n \), then
      set \( i = i + 1 \) and go to step 1,
      otherwise end.
Describe carefully how you might implement this algorithm on \( k \) processors with a shared memory. What is the order of this algorithm? Discuss the algorithm critically. Consider the case in which \( k \approx n \).

b) Now consider the problem of merging two sorted lists. The arrays \( b \) and \( c \) are each sorted and we wish to merge them into a new list \( d \) that is sorted. Describe carefully how you might do this using \( k \) processors with a shared memory.

3.23. Design and write either a C function or a Fortran subroutine that accepts two addends in vectors and a required precision level, and that returns the sum in a vector to the precision required. (Think of the elements of the vectors as being digits in some base.) Write a user-oriented description of your module. Compare your design with that of the modules of Brent’s multiple precision package (Brent, 1978).

3.24. Develop a set of test programs that will probe the accuracy of a given module to compute a sample sum of squares. You should consider various ways that the given module could go wrong and the various types of data that could cause it to have problems, and provide tests for all of them.

3.25. Assume the problem \( P \) has solution \( s \) (unknown, of course). An algorithm/program \( F \) is available to solve \( P \). Ideally, of course, \( F(P) = s \). Discuss the issues and the methods you would employ to determine that \( F(P) \) is an adequate approximation to \( s \).

3.26. Write either a C function or a Fortran subroutine that accepts the three coefficients of a quadratic polynomial and evaluates one of the roots by means of the quadratic formula, equation (3.3), and computes the other root in an appropriate manner. Write your function or subroutine as a standard software part. Write the part specification very carefully, but succinctly, as comments in the programming language. (What do you do if \( b^2 < 4ac \)? You do not have to provide a solution in this case, i.e., complex roots, but your software part must handle that case.

3.27. Design and write either a C function or a Fortran subroutine that uses a real-time algorithm for the method you developed in Exercise 3.17c to compute the mean vector and variance-covariance matrix for multivariate data in the standard rectangular paradigmatic structure displayed in (3.14), on page 71. Your program module should accept as input

- the number of variables (\( m \) in the notation of (3.14))
- the number of observations input in the current invocation (this is a number between 0 and \( n \)) – call this number \( n_i \) (a 0 value only makes sense if the current invocation is the final one for the given problem, and only wrap-up operations are to be performed)
- a subset of the rows of the overall data matrix (this is an \( n \times m \) matrix)
- an indicator of whether this invocation is the first one for the given problem, an intermediate one, or the final one
- the total number of observations that have been processed before the current invocation – and output as the updated total, to include the observations in the current invocation
- the vector of means of the observations that have been processed before the current invocation – and output as the updated means, to include the observations in the current invocation
The matrix of sums of squares and cross-products of the observations that have been processed before the current invocation – and output as the updated sums of squares and cross-products to include the observations in the current invocation. On the final invocation, the sums of squares and cross-products should be scaled by the appropriate divisor to form variances and covariances.

3.28. Discuss design issues for your program module of Exercise 3.27 if the data may contain missing values.
Random Number Generation

The role of Monte Carlo methods and simulation in all of the sciences has increased in importance during the past several years. These methods are at the heart of the rapidly developing subdisciplines of computational physics, computational chemistry, and the other computational sciences. The growing power of computers and the evolving simulation methodology have led to the recognition of computation as a third approach for advancing the natural sciences, together with theory and traditional experimentation. At the kernel of a Monte Carlo or simulation method is random number generation.

Various methods for generation of random numbers are available. Sometimes processes that are considered random are used, but for Monte Carlo methods, which depend on millions of random numbers, a physical process as a source of random numbers is generally cumbersome. In addition, if we wanted to be able to reproduce the Monte Carlo results exactly, we would have to save the entire set of realizations of the underlying random process. Instead of “random” numbers, most applications use “pseudorandom” numbers, which are deterministic but “look like” they were generated randomly. In this chapter we discuss methods for generation of sequences of pseudorandom numbers that simulate a uniform distribution over the unit interval (0, 1). These are the basic sequences from which are derived pseudorandom numbers from other distributions, pseudorandom samples, and pseudostochastic processes.

The literature on the topic of random number generation and Monte Carlo methods is vast. This chapter follows the discussion in Gentle (2003), to which the reader is referred for more details.

The use of random numbers has expanded beyond random sampling or random assignment of treatments to experimental units. More common uses now are in simulation studies of physical processes, of analytically intractable mathematical expressions, or of a population resampling from a given sample from that population. Although we do not make precise distinctions among the terms, these three general areas of application are sometimes called “simulation”, “Monte Carlo”, and “resampling”.

Randomness

The digital computer cannot generate random numbers, and it is generally not convenient to connect the computer to some external source of random events. For most
applications in statistics, this is not a disadvantage if there is some source of pseudo-
random numbers, samples of which seem to be randomly drawn from some known
distribution. There are many methods that have been suggested for generating such
pseudorandom numbers. It should be noted that there are two issues: randomness
and knowledge of the distribution. Although, at least heuristically, there are many
external physical processes that could perhaps be used as sources of random num-
bers — rather than pseudorandom numbers — there would still be the issue of what
is the distribution of the realizations of that external random process. For random
numbers to be useful in general applications, their distribution must be known.
Other issues to consider for an external process are the independence of consecutive
realizations and the constancy of the distribution. For random numbers to be useful
they must be identically and independently distributed (i.i.d.).

Deterministic generators yield numbers in a fixed sequence such that the previous
$k$ numbers (usually just the single previous number) determine(s) the next number.
Therefore, because the set of numbers usable in the computer is finite, the sequence
will repeat. The length of the sequence prior to beginning to repeat is called the
“period”, or cycle length. (Sometimes it is necessary to be more precise in defining
the period to account for the facts that with some generators, different starting
subsequences will yield different periods, and that the repetition may begin without
returning to the initial state.)

The most commonly used generator that is truly random according to generally
accepted understandings of that concept is a substance undergoing atomic decay.
The subatomic particles comprising the decaying substance transmute into other
particles at random points in time. At a macro level, that is, given an amount
of the substance that contains a very large number atoms, both theory and empirical
observations suggest that there are no dependencies among consecutive events, and
that the process is constant over sufficiently short time intervals. (“Sufficiently short”
can be several years.) If we can measure the times between the events, and if the
process is stationary, we can form a random variable with a known distribution. For
observed intervals between events, $s_1, s_2, \ldots$, let

$$X = 1, \quad \text{if } s_{2i-1} < s_{2i};$$

$$= 0, \quad \text{otherwise.}$$

Then $X$ is a random variable with a Bernoulli distribution with probability param-
eter 0.5. This random variable can be transformed into other random variables
easily. For example, a discrete uniform over the set of all numbers between 0 and
1 that have a $d$-bit terminating binary representation can be generated by taking
successive realizations of the Bernoulli.

The difficulty in using a generator based on atomic decay of course is measuring
the time intervals and inputting those measurements into the computer. John Walker
at his Fourmilab has assembled a radiation source (krypton-85), a sensor/timer, and
a computer to obtain realizations of Bernoulli random variables. A file containing a
random sample can be obtained at

http://www.fourmilab.ch/hotbits/

Each sample is generated in response to the user’s request, so the samples are unique.
Random number generation has applications in cryptography, where the requirements for “randomness” are generally much more stringent than for ordinary applications in simulation. In cryptography, the objective is somewhat different, leading to a dynamic concept of randomness that is essentially one of predictability: a process is “random” if the known conditional probability of the next event, given the previous history (or any other information, for that matter) is no different from the known unconditional probability. (The condition of being “known” in such a definition is a primitive, i.e., undefined, concept.) This kind of definition leads to the concept of a “one-way function”. A one-way function is a function \( f \), such that for any \( x \) in its domain, \( f(x) \) can be computed in polynomial time, and given \( f(x) \), \( x \) cannot be computed in polynomial time. (“Polynomial time” means that the time required can be expressed as or bounded by a polynomial in some measure of the size of the problem.) The existence of a one-way function has not been proven. In random number generation, the function of interest yields a stream of “unpredictable” numbers; that is, 

\[
x_i = f(x_{i-1}, x_{i-2}, \ldots, x_{i-k})
\]

is easily computable; but \( x_{i-1} \), given \( x_i, x_{i-2}, \ldots, x_{i-k} \), is not easily computable. The generator of Blum, Blum, and Shub (1986) is unpredictable under certain assumptions.

**Terminology**

Although we understand that the generated stream of numbers is really only pseudorandom, in this book we usually use just the term “random”, except when we want to emphasize the fact that the process is not really random, and then we use the term “pseudorandom”. Pseudorandom numbers are meant to simulate random sampling. Generating pseudorandom numbers is the subject of this section. On page 164 we consider an approach that seeks to insure that, rather than appear to be a random sample, the generated numbers are more uniformly spread out over their range. Such a sequence of numbers is called a quasirandom sequence.

We use the terms “random number generation” (or “generator”) and “sampling” (or “sampler”) interchangeably.

Another note on terminology: Some authors distinguish “random numbers” from “random variates”. In their usage, the term “random numbers” applies to pseudorandom numbers that arise from a uniform distribution, and the term “random variates” applies to pseudorandom numbers from some other distribution. Some authors use the term “random variates” only when those numbers resulted from transformations of “random numbers” from a uniform distribution. I do not understand the motivation for these distinctions, so I do not make them. In this book “random numbers” and “random variates”, as well as the additional term “random deviates”, are all used interchangeably. We generally use the term “random variable” with its usual meaning, which is different from the meaning of the other terms. Random numbers or random variates simulate realizations of random variables. I will also generally follow the notational convention of using capital Latin letters for random variables and corresponding lower case letters for their realizations.
4.1 Simulating Random Numbers from a Uniform Distribution

In most cases we want the generated pseudorandom numbers to simulate a uniform distribution over the unit interval \((0, 1)\), that is, the distribution with the probability density function,

\[
p(x) = 1, \quad \text{if} \quad 0 < x < 1; \\
= 0, \quad \text{otherwise}.
\]

We denote this distribution by \(U(0,1)\). More generally, we use the notation \(U(a,b)\) to denote the absolutely continuous uniform distribution over the interval \((a,b)\). The uniform distribution is a convenient one to work with because there are many techniques available to transform the uniform samples into samples from other distributions of interest.

There are currently two basic techniques in common use for generating uniform random numbers: congruential methods and feedback shift register methods. In both cases, usually random integers over some fixed range are first generated and then scaled into the interval \((0,1)\). If the range of the integers is large enough, the resulting granularity is of little consequence in modeling a continuous distribution. (The granularity of random numbers from good generators is no greater than the granularity of the numbers with which the computer ordinarily works.)

Modular Arithmetic

Both basic methods, congruential and feedback shift registers, use modular arithmetic, so we now describe a few of its properties.

The basic relation of modular arithmetic is \(\text{equivalence modulo } m\), also called \(\text{congruence modulo } m\), where \(m\) is some integer. Two numbers are said to be equivalent, or congruent, modulo \(m\) if their difference is an integer evenly divisible by \(m\). For \(a\) and \(b\), this relation is written as

\[
a \equiv b \mod m.
\]

For example, 5 and 14 are congruent modulo 3 (or just “mod 3”); 5 and \(-1\) are also congruent mod 3. Likewise 1.33 and 0.33 are congruent mod 1. It is clear from the definition that congruence is

- symmetric:
  \(a \equiv b \mod m\) implies \(b \equiv a \mod m\)
- reflexive:
  \(a \equiv a \mod m\) for any \(a\)
- transitive:
  \(a \equiv b \mod m\) and \(b \equiv c \mod m\) implies \(a \equiv c \mod m\)

that is, congruence is an “equivalence relationship”.

A basic operation of modular arithmetic is \(\text{reduction modulo } m\); that is, for a given number \(b\), find \(a\) such that \(a \equiv b \mod m\) and \(0 \leq a < m\). If \(a\) satisfies these two conditions, \(a\) is called the \(\text{residue of } b \mod m\).

Reduction of \(b\) modulo \(m\) can also be defined as
where the floor function $\lfloor \cdot \rfloor$ is the greatest integer less than or equal to the argument.

From the definition of congruence, we see that the numbers $a$ and $b$ are congruent modulo $m$ if and only if there exists an integer $k$ such that

$$km = a - b.$$  

(In this expression $a$ and $b$ are not necessarily integers, but $m$ and $k$ are.) This consequence of congruence is very useful in determining equivalence relationships. For example, using this property it is easy to see that modular reduction distributes over both addition and multiplication:

$$(a + b) \mod m \equiv a \mod m + b \mod m$$

and

$$ab \mod m \equiv (a \mod m)(b \mod m).$$

A system of modular arithmetic is usually defined on the nonnegative integers. Modular reduction together with the two operations of the ring results in a finite field on a set of integers. In using congruential random number generators it is common to work with a finite field of integers consisting of the nonnegative integers that are directly representable in the computer, that is, of about $2^{31}$ integers (see Section 3.1.1, page 29).

Because the pseudorandom numbers we wish to generate are between 0 and 1, in some algorithms reduction modulo 1 is used. The resultants are the fractional parts of real numbers.

Modular arithmetic has some useful applications with true random variables also. An interesting fact, for example, is that if $R$ is a random variable distributed as $U(0,1)$ and

$$S \equiv (kR + c) \mod 1$$

where $k$ is an integer constant not equal to 0, and $c$ is a real constant, then $S$ has a $U(0,1)$ distribution. (You are asked to show this in Exercise 4.1a, page 222.) Modular arithmetic can also be used to generate two independent random numbers from a single random number. If

$$\pm 0.d_1d_2d_3\cdots$$

is the representation, in a given base, of a uniform random number $R$, then any subsequence of the digits $d_1, d_2, \ldots$, etc., can be used to form other uniform numbers. (If the subsequence is finite, as of course it is in computer applications, the numbers are discrete uniform; but if the subsequence is long enough the result is considered continuous uniform.) Furthermore, any two disjoint subsequences, can be used to form independent random numbers.

The sequence of digits $d_1, d_2, \ldots$ can be rearranged to form more than one uniform variate, for example,

$$\pm 0.d_1d_3d_5\cdots$$

and

$$\pm 0.d_2d_4d_6\cdots$$

The use of subsequences of bits in a fixed-point binary representation of pseudorandom numbers to form other pseudorandom numbers is called bit stripping.
Modular reduction is a binary operation, or a function with two arguments. In the C programming language the operation is represented as “b%m”. (There is no obvious relation of the symbolic value of “%” to the modular operation. No committee passed judgment on this choice before it became a standard part of the language. Sometimes design by committee helps.) In Fortran the operation is specified by the function “mod(b,m)”; in Matlab, the function “rem(b,m)”; and in Maple, “b mod m”. There is no modulo function in R, but the operation can be implemented using the “floor” function as was shown above.

Modular reduction can be performed by using the lower-order digits of the representation of a number in a given base. For example, taking the two lower-order digits of the ordinary base-ten representation of a negative integer yields the decimal representation of the number reduced modulo 100. When numbers represented in a fixed-point scheme in the computer are multiplied, except for consideration of a sign bit, the product when stored in the same fixed-point scheme is the residue of the product modulo the largest representable number. In a two-complement representation, if the sign bit is changed, the meaning of the remaining bits is changed. For positive integers $x$ and $y$ represented in the fixed-point variables $ix$ and $iy$ in 32-bit two-complement, the product

$$iz = ix * iy$$

contains either $xy \mod 2^{31}$ or $xy \mod 2^{31} - 2^{31}$, which is negative.

### 4.1.1 Sources of Uniform Random Numbers

There are a

**************

### Linear Congruential Generators

D. H. Lehmer in 1948 (see Lehmer, 1951) proposed the *linear congruential generator* as a source of random numbers. In this generator, each single number determines its successor. The form of the generator is

$$x_i \equiv (ax_{i-1} + c) \mod m, \quad \text{with} \quad 0 \leq x_i < m; \quad (4.1)$$

$a$ is called the “multiplier”; $c$ is called the “increment”; and $m$ is called the “modulus” of the generator. Often $c$ in (4.1) is taken to be 0, and in this case, the generator is called a “multiplicative congruential generator”:

$$x_i \equiv ax_{i-1} \mod m, \quad \text{with} \quad 0 < x_i < m. \quad (4.2)$$

For $c \neq 0$, the generator is sometimes called a “mixed congruential generator”. The starting value in the recursion, $x_0$, is called the “seed”. A sequence resulting from the recursion (4.1) is called a *Lehmer sequence*. Each $x_i$ is scaled into the unit interval $(0,1)$ by division by $m$, that is,

$$u_i = x_i / m.$$ 

If $a$ and $m$ are properly chosen, the $u_i$’s will “look like” they are randomly and uniformly distributed between 0 and 1.

The recurrence in (4.2) for the integers is equivalent to the recurrence
4.1 Simulating Random Numbers from a Uniform Distribution

\[ u_i \equiv au_{i-1} \mod 1, \quad \text{with} \quad 0 < u_i < 1. \]

This recurrence has some interesting relationships to the first-order linear autoregressive model

\[ U_i = \rho u_{i-1} + e_i, \]

where \( \rho \) is the autoregressive coefficient and \( e_i \) is a random variable with a U(0, 1) distribution (see Lawrance, 1992).

Because \( x_i \) is determined by \( x_{i-1} \) and since there are only \( m \) possible different values of the \( x \)'s, the maximum period or cycle length of the linear congruential generator is \( m \). Also, since \( x_{i-1} = 0 \) cannot be allowed in a multiplicative generator, the maximum period of the multiplicative congruential generator is \( m - 1 \).

When computing was expensive, values of \( m \) used in computer programs were often powers of 2. Such values could result in faster computer arithmetic. The maximum period of multiplicative generators with such moduli is \( m/4 \); and, interestingly, this period is achieved for any multiplier that is \( \pm 3 \mod 8 \) (see Knuth, 1998). The bits in the binary representations of the sequences from such generators have very regular patterns. The period of the lowest order bit is at most 1 (i.e., it is always the same); the period of the next lowest order bit is at most 2; the period of the next lowest order bit is at most 4; and so on. In general, low order bits in the streams resulting from a composite modulus will have periods corresponding to the factors. The small periods can render a bit stripping technique completely invalid with random number generators with a modulus that is a power of 2.

Currently the numbers used as moduli in production random number generators are often primes in particular, Mersenne primes, which have the form \( 2^p - 1 \). (For any prime \( p \leq 31 \), numbers of that form are prime except for the three values: \( p = 11, 23, \) and 29. Most larger values of \( p \) do not yield primes. A large one that does yield a prime is \( p = 859 \, 433 \).) If (and only if) \( m \) is a prime and the multiplier, \( a \), is a primitive root modulo \( m \), then the generator has a maximal period of \( m - 1 \). (A primitive root, \( a \), modulo a prime, \( m \), is a number such that the smallest positive \( k \) satisfying

\[ a^k \equiv 1 \mod m \]

is \( m - 1 \). See Shockley, 1967, for a general discussion; see Fuller, 1976, for methods to determine if a number is a primitive root; and see Exercise 4.11, page 224 for some computations.) For example, consider \( m = 31 \) and \( a = 7 \) and begin with \( x_0 = 19 \). The next integers in the sequence are

\[ 9, 1, 7, 18, 2, 14, 5, 4, 28, 10, 8, 25, 20, 16, 19, \]

so, of course, at this point the sequence begins to repeat. The period is 15; 7 is not a primitive root modulo 31. If, on the other hand, we take \( a = 3 \) and begin with \( x_0 = 19 \), we go through 31 numbers before we get back to 19; 3 is a primitive root modulo 31.

The most commonly used modulus is probably the Mersenne prime \( 2^{31} - 1 \). A common multiplier for that modulus is \( 7^3 \) (see the discussion of the “minimal standard” on page 130). The Mersenne prime \( 2^{61} - 1 \) is also used to some extent. Wu (1997) suggests multipliers of the form \( \pm 2^{31} \pm 2^{29} \) because they result in particularly simple computations, yet seem generally to have good properties. Wu suggests \( 2^{15} - 2^{10} \) and \( 2^{16} - 2^{21} \) for a modulus of \( 2^{31} - 1 \), and \( 2^{10} - 2^{19} \) and \( 2^{42} - 2^{31} \) for a modulus of \( 2^{61} - 1 \). The computational efficiency of such multipliers results from the period of random number generator cycle length of random number generator bit stripping

Mersenne prime

“minimal standard”
generator
fact that multiplication by a multiplier of the form $2^q$ and followed by a modular reduction with a modulus of the form $2^p - 1$ results in an exchange of the block of the $q$ most significant bits and the block of the $p - q$ least significant bits. Multipliers of the form suggested by Wu, effectively do this kind of exchange twice and then add the results. L’Ecuyer and Simard (1999) point out, however, that an operation consisting of two exchanges of two blocks of bits followed by addition of the results tends to yield a value whose binary representation has a similar the number of 1’s to the number of 1’s in the original value. The number of 1’s in the binary representation of a value is called its “Hamming weight”. (Compare “Hamming distance”, page 571.) L’Ecuyer and Simard (1999) define a test of independence of Hamming weights of successive values in the output streams of random number generators and, in applying the test to generators with multipliers of the form $\pm 2^{q_1} \pm 2^{q_2}$, find that such generators perform poorly with respect to this criterion.

For a random number generator to be useful in most practical simple applications, the period must be of the order of at least $10^9$ or so, which means that the modulus in a linear congruential generator must be at least that large. The values of the moduli in common use range in order from about $10^9$ to $10^{15}$. Even so, the period of such generators is relatively short because of the speed with which computers can cycle through the full period and in view of the very large sizes of some simulation experiments.

Structure in the Generated Numbers

In addition to concern about the length of the period, there are several other considerations. It is clear that if the period is $m$, the output of the generator over a full cycle will be evenly distributed over the unit interval. If we ignore the sequential order of a full-period sequence from a congruential generator, it will appear to be $U(0,1)$; in fact, the sample would appear too much like a sample from $U(0,1)$.

A useful generator, however, must generate subsamples of the full cycle that appear to be uniformly distributed over the unit interval. Furthermore, the numbers should appear to be distributionally independent of each other; that is, the serial correlations should be small.

Unfortunately, the structure of a sequence resulting from a linear congruential generator is very regular. Marsaglia (1968) pointed out that the output of any congruential generator lies on a simple lattice in a $k$-space with axes representing successive numbers in the output. This is fairly obvious upon inspection of the algebra of the generator. How bad this is (that is, how much this situation causes the output to appear nonrandom) depends on the structure of the lattice. A lattice is defined in terms of integer combinations of a set of “basis vectors”. Given a set of linearly independent vectors $\{v_1, v_2, \ldots, v_d\}$ in $\mathbb{R}^d$, a lattice is the set of vectors $w$ of the form $\sum_{i=1}^{d} z_i v_i$, where $z_i$ are integers. The set of vectors $\{v_i\}$ is a basis for the lattice. Figure 4.1 shows a lattice in two-dimensions, with basis $\{v_1, v_2\}$.

For an example of the structure in a stream of pseudorandom numbers produced by a linear congruential generator, consider the output of the generator (4.2) with $m = 31$ and $a = 3$ that begins with $x_0 = 9$. The next integers in the sequence are 27, 19, 26, 16, 17, 20, 29, 25, 13, 8, 24, 10, 30, 28, 22, 4, 12, 5, 15, 14, 11, 2, 6, 18, 23, 7, 21, 1, 3, 9,
and, of course, at this point the sequence begins to repeat. The period is 30; we know 3 is a primitive root modulo 31. A visual assessment or even computation of a few descriptive statistics does not raise serious concerns about whether this represents a sample from a discrete uniform distribution over the integers from 1 to 30. The scaled numbers (the integers divided by 30) have a sample mean of 0.517 and a sample variance of 0.86. Both of these values are consistent with the expected values from a $U(0, 1)$ distribution. The autocorrelations for lags 1 through 5 are

$$0.27, 0.16, -0.10, 0.06, 0.07.$$ 

Although the lag 1 correlation is somewhat large, for a sample of this size, these values are not inconsistent with a hypothesis of independence.

However, when we plot the successive (overlapping) pairs,

$$(27, 19), (19, 26), (26, 16), \ldots$$

as in Figure 4.2, a disturbing picture emerges. All points in the lattice of pairs lie along just 3 lines, each with a slope of $3$. (There are also 10 lines with slope $-rac{1}{10}$; and 10 lines with slope $\frac{2}{11}$, if we count as lines the continuation by “wrapping” modulo 31.) Even though we have worked with overlapping pairs, it should be obvious that nonoverlapping pairs have the same structure; there are just half as many points.

This pattern is in fact related to the relatively large correlation at lag 1. Although the correlation may not appear so large for the small sample size, even if we were to increase the sample size by generating more random numbers, that large value of the correlation would persist, because the random
Random Number Generation

numbers would just repeat themselves. It is easy to see that this kind of pattern results from the small value of the multiplier. The same kind of problem would also result from a multiplier that is too close to the modulus, such as $a = 27$, for example.

There are 8 primitive roots modulo 31, so we might try another one, say 12. Let $a = 12$ and again begin with $x_0 = 9$. The next integers in the sequence are

15, 25, 21, 4, 17, 18, 30, 19, 11, 8, 3, 5, 29, 7, 22, 16, 6, 10, 27, 14, 13, 1, 12, 20, 23, 28, 26, 2, 24, 9.

A visual assessment does not show much difference in this sequence and the sequence of numbers from the sequence generated by $a = 3$. The numbers themselves are exactly the same as those before, so the static properties of
mean and variance are the same. The autocorrelations are different, however. For lags 1 through 5 they are

\[-0.01, -0.07, -0.17, -0.15, 0.03, 0.35.\]

The smaller value for lag 1 indicates that the structure of successive pairs may be better; and in fact, the points do appear better distributed, as we see in Figure 4.3. There are 6 lines with slope \(-\frac{2}{7}\) and 7 lines with slope \(\frac{5}{3}\).

A quantitative measure of the severity of the lattice structure is the distance between the lines — specifically, the shortest distance between two sides of the maximal volume parallelogram formed by four points and not enclosing any points. The distance between the lines with slope of \(\frac{2}{7}\) is 6.96, as shown in Figure 4.3. The distance between the lines with slope of \(-\frac{5}{3}\) is 5.76. Dieter (1975) discusses the general problem of determining the distance between the lattice lines. We encounter similar structural problems later in this section and discuss the identification of this kind of structural problem beginning on page 153.

Figures 4.2 and 4.3 show all of the points from the full period of those small generators. For a generator with a larger period, we obviously would get more points; but with a poor generator, they could still all lie along a small number of lines.

It is a good idea to view a similar plot for a sample of points from any random number generator that we plan to use. For example, the R commands

```r
xunif <- runif(1000)
plot(xunif[1:999],xunif[2:1000])
```

yield the plot shown in Figure 4.4. There does not appear to be any obvious pattern in those points generated by the standard R generator. We discuss random number generation in R further in Section 4.4, page 218.

The two-dimensional patterns are related to the autocorrelation of lag 1, as we have seen. Autocorrelations at higher-order lags are also of concern. The lattice structure in higher dimensions is related to, but more complicated than, simple bivariate autocorrelations. In \(d\) dimensions, it is the pattern of the subsequences \((x_i, \ldots, x_{i+d})\). Among triplets, for example, we could observe a three-dimensional lattice on which the points would all lie. As in the two-dimensional case, the quality of the lattice structure depends on how many lattice points are covered, and in what order they are covered. The lattice structure is related to the correlation at a lag corresponding to the dimension of the lattice; so large correlations of lag 2, for example, would suggest that a three-dimensional lattice structure would not cover three-dimensional space well.

Correlations of lag 1 in a sequence produced by a multiplicative congruential generator will be small if the square of the multiplier is approximately equal to the modulus. In this case, however, the correlations of lag 2 are
likely to be large, and consequently the three-dimensional lattice structure will be poor. (In our examples in Figures 4.2 and 4.3, we had $a^2 \equiv 9 \mod 31$ and $a^2 \equiv 28 \mod 31$, respectively, so the generator used in Figure 4.3 would have poor lattice structure in three dimensions.) An example of a generator with very good two-dimensional lattice structure yet very poor three-dimensional lattice structure is implemented in the program RANDU, which for many years was the most widely used random number generator in the world.

The generator in RANDU is essentially

$$x_i \equiv 65539x_{i-1} \mod 2^{31}. \quad (4.3)$$

(RANDU was coded to take advantage of integer overflow (see Section 3.1.2, page 44) and made an adjustment when overflow occurred, so the generator is not exactly the same as (4.3).)
The generator in (4.3) can be written to express the relationship among three successive members of the output sequence:

\[ x_i \equiv (65539)^2 x_{i-2} \mod 2^{31} \]
\[ \equiv (2^{16} + 3)^2 x_{i-2} \mod 2^{31} \]
\[ \equiv (6x_{i-1} - 9x_{i-2}) \mod 2^{31} \]

That is,

\[ x_i - 6x_{i-1} + 9x_{i-2} = c2^{31}, \] (4.4)

where \( c \) is an integer. Since \( 0 < x_i < 2^{31} \), all such triplets must lie on no more than 15 planes in \( \mathbb{R}^3 \). Lewis and Orav (1989) show a slice of a cube in which is plotted a sample from this generator. As might be expected, all of the points in the slice appear in only a small number of linear regions. (See Exercise 4.4, page 223.) Huber (1985) and later Cabrera and Cook (1992) use data generated by \textsc{randu} to illustrate a method of projection pursuit, and their examples expose the poor structure of the output of \textsc{randu}.

Although it had been known for many years that the generator (4.3) had problems (see Coldwell, 1974), and even the analysis represented by equa-
tion (4.4) had been performed, the exact nature of the problem has sometimes been misunderstood. For example, as James (1990) states “We now know that any multiplier congruent to 5 mod 8 ... would have been better ....” Being congruent to 5 mod 8 does not solve the problem. Such multipliers have the same problem if they are close to $2^{16}$, for the same reason (see Exercise 4.5, page 223).

RANDU is still available at a number of computer centers and is used in some statistical analysis and simulation packages.

The lattice structure of the common types of congruential generators can be assessed by the spectral test of Coveyou and MacPherson (1967) (see Knuth, 1998, pages 93–114) or by the lattice test of Marsaglia (1972a). We discuss these types of tests beginning on page 153.

### Tests of Linear Congruential Generators

Any number of statistical tests could be developed to be applied to the output of a given generator. The simple underlying idea is to form any transformation on the subsequence, determine the distribution of the transformation under the null hypothesis of independent uniformity of the sequence, and perform a goodness-of-fit test of that distribution. Useful transformations can be multivariate; the simplest is just the identity transformation on a subsequence of a chosen length. Another simple transformation is just to add successive terms. (Adding two successive terms should yield a triangular distribution.) With a little imagination, the reader should be able to devise chi-squared tests for uniformity in any dimension, chi-squared tests for triangularity of sums of two successive numbers, and so on. Tests for serial correlation of various lags, and various sign tests are other possibilities that should come to mind to anyone with some training in statistics.

For some specific generators or families of generators there are extensive empirical studies reported in the literature. For $m = 2^{31} - 1$, for example, empirical studies by Fishman and Moore (1982, 1986) indicate that different values of multipliers, all of which perform well under the lattice test and the spectral test (see page 153), may yield samples statistically distinguishable from samples from a true uniform distribution.

Park and Miller (1988) summarize some problems with random number generators commonly available, and propose a “minimal standard” for a linear congruential generator. The generator must perform “at least as well as” one with $m = 2^{31} - 1$ and $a = 16807$, which is a primitive root. (The smallest primitive root of $2^{31} - 1$ is 7; and $7^5 = 16807$ is the largest power of 7 such that $7^p \cdot x$, for the largest value of $x$ (which is $2^{32} - 2$), can be represented in the common 64-bit floating point format.)

This choice of $m$ and $a$ was made by Lewis, Goodman, and Miller (1969), and is very widely used. Results of extensive tests by Learmonth and Lewis (1973) are available for it. It is provided as one option in the IMSL Libraries. Fishman and Moore (1986) found the value of 16807 to be marginally acceptable
as the the multiplier, but there were several other multipliers that performed better in their battery of tests.

The article by Park and Miller generated extensive discussion; see the “Technical Correspondence” in the July 1993, issue of Communications of the ACM, pages 105 through 110. It is relatively easy to program the minimal standard, as we see in the next section, but the algorithm given by Carta (1990) ostensibly to implement the minimal standard should be avoided.

Ferrenberg, Landau, and Wong (1992) used some of the generators that meet the Park and Miller minimal standard to perform several simulation studies in which the correct answer was known. Their simulation results suggested that even some of the “good” generators could not be relied on in some simulations. Vattulainen, Ala-Nissila, and Kankaala (1994) likewise used some of these generators as well as generators of other types and found that their simulations often did not correspond to the processes they were modeling. The point is that the “minimal standard” is minimal.

Deng and Lin (2000) contend that the “minimal standard” is not acceptable for serious work. Its relatively short period and its lattice structure are unacceptable. They suggest use of matrix congruential generators (see generators (4.9) and (4.10) and discussion on page 140).

We discuss general empirical tests of random number generators beginning on page 155.

**Skipping Ahead in Linear Congruential Generators**

Sometimes it is useful to generate separate, independent subsequences with the same generator. This may be because computations are being performed in parallel or because a Monte Carlo experiment is being run in blocks.

To generate separate subsequences, it is generally not a good idea to choose two seeds arbitrarily for the subsequences because we generally have no information about the relationships between them. In fact, an unlucky choice of seeds could result in a very large overlap of the subsequences. A better way is to fix the seed of one subsequence and then to skip a known distance ahead to start the second subsequence.

The basic equivalence relation of the generator

\[ x_{i+1} \equiv ax_i \mod m \]

implies

\[ x_{i+k} \equiv a^k x_i \mod m. \]

This provides a simple way of skipping ahead in the sequence generated by a linear congruential generator. This may be useful in parallel computations, where we may want one processor to traverse the sequence

\[ x_s, x_{s+1}, x_{s+2}, \ldots \]

and a second processor to traverse the nonoverlapping sequence,
seed
leapfrogging, in
random number
generation
antithetic variates
variance reduction
seed

\[ x_{s+k}, x_{s+k+1}, x_{s+k+2}, \ldots \]

The seed for the second stream can be generated by

\[ x_{s+k-1} \equiv bx_0 \mod m, \]

where

\[ b \equiv a^k \mod m. \]

Another interesting subsequence that is “independent” of the first sequence is

\[ x_s, x_{s+k}, x_{s+2k}, \ldots \quad (4.5) \]

This sequence is generated by

\[ x_{i+1} \equiv bx_i \mod m, \]

where \( b \) is as before. This method of generating independent streams is called **leapfrogging** by \( k \). Different “independent” subsequences can be formed using various leapfrog distances. (The distances must be chosen carefully, of course. A minimum requirement is that the distances be relatively prime.) We could let one processor leapfrog beginning at \( x_s \) and let a second processor leapfrog beginning at \( x_{s+1} \) and using the same leapfrog distance.

A certain amount of care is necessary in choosing the seed and the distance to skip ahead. Anderson and Titterington (1993) show that for a multiplicative congruential generator, the correlation between the sequences \( (x_i, \ldots) \) and \( (x_{i+k}, \ldots) \) is approximately asymptotically \( (n_1n_2)^{-1} \), where \( x_{i+k} = (n_1/n_2)x_i \) and \( n_1 \) and \( n_2 \) are relatively prime. Figure 4.5 shows a plot of two subsequences, each of length 100, whose seeds differ by a factor of 5. The correlation of these two subsequences is 0.375. (See also Exercise 4.12, page 224.)

This would indicate that when choosing to skip ahead \( k \) steps, the seed should not be a small multiple of, or a small fraction of, \( k \).

Sometimes, instead of “independent” sequences, we may want a sequence that is strongly negatively correlated with the first sequence. (We call such sequences “antithetic” They may be useful in variance reduction; we discuss them further in Section 2.1.4, page 12.) If one sequence begins with the seed \( x_0 \), and another sequence begins with \( m - x_0 \), then the \( i \)th term in the two sequences will be \( x_i \) and \( m - x_i \). The sequences are perfectly negatively correlated.

Another pair of sequences that may be of interest are ones that go in reverse order; that is, if one sequence is

\[ x_1, x_2, x_3, \ldots, x_k, \ldots, \]

the other sequence is

\[ x_k, x_{k-1}, x_{k-2}, \ldots, x_3, x_2, x_1, \ldots. \]
4.1 Simulating Random Numbers from a Uniform Distribution

If the first sequence is generated by $x_{i+1} \equiv ax_i \mod m$, the second sequence is generated by $y_{i+1} \equiv by_i \mod m$, where

$$b \equiv a^{c-1} \mod m,$$

and $c$ is the period.

**Shuffling the Output Stream**

MacLaren and Marsaglia (1965) suggest that the output stream of a linear congruential random number generator be *shuffled* by using another, perhaps simpler, generator to permute subsequences from the original generator. This shuffling can increase the period (because it is no longer necessary for the same value to follow a given value every time it occurs) and it can also break up the lattice structure. (There will still be a lattice, of course; it will just have a different number of planes.)

Because a single random number can be used to generate independent random numbers ("bit stripping", see page 121), a single generator can be used to shuffle itself.

Bays and Durham (1976) describe a method of using a single generator to fill a table of length $k$ and then using a single stream to select a number from the table and to replenish the table. After initializing a table $T$ to contain $x_1, x_2, \ldots, x_k$, set $i = k + 1$ and generate $x_i$ to use as an index to the table.
134 4 Random Number Generation

Then update the table with \( x_{i+1} \). The method is shown in Algorithm 4.1 to generate the stream \( y_i \) for \( i = 1, 2, \ldots \).

**Algorithm 4.1 Bays-Durham Shuffling of Uniform Deviates**

0. Initialize the table \( T \) with \( x_1, x_2, \ldots, x_k, i = 1 \), generate \( x_{k+i} \), and set

\[
y_i = x_{k+i}.
\]

1. Generate \( j \) from \( y_i \) (use bit stripping or \( \text{mod } k \)).
2. Set \( i = i + 1 \).
3. Set \( y_i = T(j) \).
4. Generate \( x_{k+i} \), and refresh \( T(j) \) with \( x_{k+i} \).

The period of the generator may be increased by this shuffling. Bays and Durham (1976) show that the period under this shuffling is \( O(k!c^2) \), where \( c \) is the cycle length of the original, unshuffled generator. If \( k \) is chosen so that \( k! > c \), the period is increased.

For example, with the generator used in Figure 4.2 (\( m = 31, a = 3 \) and beginning with \( x_0 = 9 \)), which yielded the sequence

\[
27, 19, 26, 17, 20, 29, 25, 13, 8, 24, 10, 30, 28, 22, 4, 12, 5, 15, 14, 11, 2, 6, 18, 23, 7, 21, 1, 3, 9,
\]

we select \( k = 8 \), and initialize the table as

\[
27, 19, 26, 17, 20, 29, 25.
\]

We then use the next number, 13, as the first value in the output stream, and also to form a random index into the table. If we form the index as \( 13 \text{ mod } 8 + 1 \), we get the sixth tabular value, 20, as the second number in the output stream. We generate the next number in the original stream, 8, and put it in the table, so we now have the table

\[
27, 19, 26, 17, 8, 29, 25.
\]

Now we use 20 as the index to the table and get the fifth tabular value, 17, as the third number in the output stream. By continuing in this manner to yield 10,000 deviates, and plotting the successive pairs, we get Figure 4.6. The very bad lattice structure shown in Figure 4.2 has diminished. (Remember there are only 30 different values, however.)

**Computer Implementation of Linear Congruential Generators**

There are many important issues to consider in writing computer software to generate random numbers. Before proceeding to discuss other types of random number generators, we consider the computer implementation of linear congruential generators. These same kinds of considerations apply to other generators as well.

As a problem in numerical analysis, the basic recurrence of the linear congruential generator is somewhat ill-conditioned. This is because full precision must be maintained throughout; if any term in the sequence is not exact in the last place, all subsequent terms will differ radically from the true sequence.
4.1 Simulating Random Numbers from a Uniform Distribution

"minimal standard" generator

Figure 4.6. Pairs of Successive Numbers from a Shuffled Version of $x_i \equiv 3x_{i-1} \mod 31$. Compare this with Figure 4.2 grr006

Insuring Exact Computations

Because of the number of significant digits in the quantities in this recurrence, even the novice programmer learns very quickly that special steps may be required to provide the complete precision required.

If the multiplier $a$ is relatively large, a way of avoiding the need for higher precision is to perform the computations for $x_i \equiv ax_{i-1} \mod m$ as

$$x_i \equiv (b(cx_{i-1} \mod m) + dx_{i-1} \mod m) \mod m,$$

where $a = bc + d$. The values of $b$, $c$, and $d$ are chosen so that all products are exactly representable. This is possible as long as the available precision is at least $1.5\log_B m$ places in the arithmetic base $B$. The origins of the idea of using the decomposition (4.6), as so many of the ideas useful in numerical computations, are lost in antiquity. The idea has been rediscovered many times.

Even if $a$ is relatively small, as in the Park and Miller (1988) “minimal standard”, the computations cannot be performed directly in ordinary floating-point words on computers with 32-bit words. In C we can use `double (fmod)`, and in Fortran we can use `double precision`, as shown in Figure 4.7.

Another way of avoiding the need for higher precision for relatively small multipliers, that is, multipliers smaller than $\sqrt{m}$ (as in the case shown in Figure 4.7), is to perform the computations in the following steps given by Bratley, Fox, and Schrage (1987). Let
subroutine rnlcm (dseed, nr, u)
  double precision dseed, dm, dmp
  real u(*)
  data dm/2147483647.d0/, dmp/2147483655.d0/
    !(dmp is computer specific)
  do 10 i = 1, nr
    dseed = dmod (16807.d0*dseed, dm)
    u(i) = dseed/dmp
 10 continue
end

Figure 4.7. A Fortran Program Illustrating a Congruential Generator for a Machine with 32-Bit Words

\[ q = \lfloor m/a \rfloor \]

and

\[ r \equiv m \mod a. \]

Then

\[
ax_{i-1} \mod m = (ax_{i-1} - \lfloor x_{i-1}/q \rfloor m) \mod m \\
= (ax_{i-1} - \lfloor x_{i-1}/q \rfloor (aq + r)) \mod m \\
= (a(x_{i-1} - \lfloor x_{i-1}/q \rfloor)q - \lfloor x_{i-1}/q \rfloor r) \mod m \\
= (a(x_{i-1} \mod q) - \lfloor x_{i-1}/q \rfloor r) \mod m. \tag{4.7}
\]

For the operations of equation (4.7), L’Ecuyer (1988) gives coding of the following form:

\[
k = x/q \\
x = a*(x - k*q) - k*r \\
\text{if } (x < 0) \text{ } x = x + m
\]

As above, it is often convenient to use the fact that the modular reduction is equivalent to

\[ x_i = ax_{i-1} - m \lfloor ax_{i-1}/m \rfloor. \]

But in this equivalent formulation, the floor operation must be performed simultaneously with the division operation (that is, there can be no rounding between the operations). This may not be the case in some computing systems, and this is just another numerical “gotcha” for this problem.

**Restriction that the Output Be in the Open Interval (0, 1)**

There is another point that is often overlooked in general discussions of random number generation. We casually mentioned earlier in the chapter that we want to simulate a uniform distribution over (0, 1). Mathematically, that is the same as a uniform distribution over [0, 1]; it is not the same on the computer, however. Using the methods we have discussed, we must sometimes be
4.1 Simulating Random Numbers from a Uniform Distribution

careful to exclude the endpoints. Whenever the mixed congruential generator is used (i.e., \( c \neq 0 \) in the recurrence \((4.1)\)), we may have to take special steps to handle the case when \( x_i = 0 \). For the multiplicative congruential generator \((4.2)\), we do not have to worry about a 0. The normalization \( u_i = x_i/m \) will not yield a 0.

The normalization, however, can yield a 1, whether the generator is mixed or multiplicative. To avoid that we choose a different normalizer, \( \tilde{m} (> m) \). See the code in Figure 4.7, and see Exercise 4.10, page 224.

**Efficiency Considerations**

In some computer architectures, operations on fixed-point numbers are faster than those on floating-point numbers. If the modulus is a power of 2, it may be possible to perform the modular reduction by simply retaining only the low-order bits of the product. Furthermore, if the power of 2 corresponds to the number of numeric bits, it may be possible to use the fixed-point overflow to do the modular reduction. (See Section 3.1.2, page 44.) This is an old “trick”, which was implemented in many of the early generators, including \texttt{RANDU}. This trick can also be implemented using a bitwise “and” operation. (The Fortran intrinsic \texttt{land} does this.) This can have the same effect but without causing the overflow. Overflow may be considered an arithmetic error. A power of 2 is generally not a good modulus, however, as we have already pointed out.

The fixed-point overflow trick can be modified for doing a modular reduction for \( m = 2^p - 1 \). Let

\[
\tilde{x}_i \equiv (ax_{i-1} + c) \mod (m + 1),
\]

and if \( \tilde{x}_i < m + 1 \), then

\[
x_i = \tilde{x}_i;
\]

otherwise,

\[
x_i = \tilde{x}_i - m.
\]

This trick can be implemented three times if the multiplier \( a \) is large, and the representation of equation \((4.6)\) is used.

Some multiplications, especially in fixed-point, are performed more rapidly if one of the multiplicands is a power of 2. This fact provides another opportunity for accelerating computations that make use of equation \((4.6)\). The decomposition of the multiplier \( a \) can have two components that are powers of 2.

**Vector Processors**

Because a random number generator may be invoked millions of times in a single program, it is important to perform the operations efficiently. Brophy et al. (1989) describe an implementation of the linear congruential generator...
for a vector processor (see Section 3.3, page 75, for a discussion of vector architectures). If the multiplier is \( a \) and the modulus is \( m \), and the vector register length is \( k \), the quantities

\[ a_j \equiv a^j \mod m, \text{ for } j = 1, 2, \ldots, k, \]

are precomputed and stored in a vector register. The modulo reduction operation is not (usually) a vector operation, however. For the particular generator they considered, \( m = 2^{31} - 1 \), and because

\[ \frac{x}{2^{31} - 1} = x2^{31} + x2^{62} + \ldots, \]

they did the modular reduction as \( a_jx - (2^{31} - 1)a_j2^{31}x \). For a given \( x_i \), the \( k \) subsequent deviates are obtained with a few operations in a vector register.

Anderson (1990) provides a survey of the methods for vector processors and other high-performance architectures.

How to implement a random number generator in a given environment depends on such things as

- the type of architecture (vector, parallel, etc.)
- the precision available
- whether integer arithmetic is available
- whether integer overflow is equivalent to modular reduction
- the base of the arithmetic
- the relative speeds of multiplication and division
- the relative speeds of MOD and INT.

The quality of the generator should never be sacrificed in the interest of efficiency, no matter what type of computer architecture is being used.

### Other Congruential Generators

There are several other variations of the basic congruential generator. They share the fundamental operation of modular reduction that provides the “randomness” or the “unpredictability”. In general, the variations that have been proposed increase the complexity of the generator. For a generator to be useful, however, we must be able to analyze and understand its properties at least to the point of having some assurance that there is no deleterious nonrandomness lurking in the output of the generator. We should also be able to skip ahead in the generator a fixed distance. More study is needed for most of the generators mentioned in this section.

### Multiple Recursive Generators

A simple extension of the multiplicative congruential generator is to use multiples of the previous \( k \) values to generate the next one:
4.1 Simulating Random Numbers from a Uniform Distribution

\[ x_i \equiv (a_1 x_{i-1} + a_2 x_{i-2} + \cdots + a_k x_{i-k}) \mod m. \]

This is sometimes called a “multiple recursive” multiplicative congruential generator. A multiple recursive generator can have a much longer period than a simple multiplicative generator. Computational efficiency can be enhanced by choosing some of the \(a_j\) as 0 or \(\pm 1\).

L’Ecuyer, Blouin, and Couture (1993) study some of the statistical properties of multiple recursive generators, and make recommendations for the multipliers for some specific moduli, including the common one, \(2^{31} - 1\). Their assessment of the quality of the generators is based on the Beyer ratio (see page 155). They also discuss programming issues for such generators, and give code for a short C program for a multiple recursive generator with \(k = 5\). Their code is portable and will not return 0 or 1. They also indicate how to skip ahead in multiple recursive generators.

**Lagged Fibonacci**

A simple Fibonacci sequence has \(x_{i+2} = x_{i+1} + x_i\). Reducing the numbers mod \(m\) produces a sequence that looks somewhat random, but actually does not have satisfactory randomness properties.

We can generalize the Fibonacci recursion in two ways. First, instead of combining successive terms, we combine terms at some greater distance apart, so the **lagged Fibonacci congruential generator** is

\[ x_i \equiv (x_{i-j} + x_{i-k}) \mod m. \] (4.8)

A lagged Fibonacci congruential generator requires an initial sequence, rather than just a single seed.

If \(j, k, \text{ and } m\) are chosen properly, the lagged Fibonacci generator can perform well. If \(m\) is a prime and \(k > j\), the period can be as large as \(m^k - 1\). More important, this generator is the basis for other generators, such as discussed in Section 4.1.1, page 145.

Altman (1989) has shown that care must be exercised in selecting the initial sequence, but for a carefully chosen initial sequence, the bits in the output sequence seem to have better randomness properties than those from congruential and Tausworthe generators (page 141).

A second way of generalizing the Fibonacci recursion is to use more general binary operators, instead of addition modulo \(m\). In a **general** lagged Fibonacci generator we start with \(x_1, x_2, \ldots, x_k\) (“random” numbers), and let

\[ x_i = (x_{i-j} \circ x_{i-k}), \]

where \(\circ\) is some binary operator, with

\[ 0 \leq x_i \leq m - 1, \quad \text{and} \quad 0 < j < k < i. \]
Matrix Congruential Generators

A generalization of the scalar linear congruential generator to a generator of pseudorandom vectors is straightforward:

\[ x_i \equiv (Ax_{i-1} + C) \mod m, \]  

(4.9)

where the \(x_i\) are vectors of length \(d\), and \(A\) and \(C\) are \(d \times d\) matrices. The elements of the vectors and of the matrices are integers between 1 and \(m - 1\). The vector elements are then scaled into the interval \((0, 1)\) to simulate \(U(0, 1)\) deviates. Such a generator is called a matrix congruential generator. As with the scalar generators, \(C\) is often chosen as 0 (a matrix with all elements equal to 0). Reasons for using a matrix generator are to generate parallel streams of pseudorandom deviates or to induce a correlational structure in the random vectors.

If \(A\) is a diagonal matrix, the matrix generator is essentially a set of scalar generators with different multipliers. For more general \(A\), the elements of the vectors are correlated. Rather than concentrating directly on the correlations, most studies of the matrix generators (e.g., Afflerbach and Grothe, 1988) have focused on the lattice structure. Choosing \(A\) as the Cholesky factor (see Section 5.2.2, page 283) of a target variance-covariance matrix may make some sense, and there may be other situations in which a matrix generator would be of some value. In Exercise 4.3, page 223, you are asked to experiment with a matrix generator to study the correlations. Generally, however, any desired correlational structure should be simulated by transformations on an independent stream of uniforms, as we discuss in Section 4.3.3, page 208, rather than trying to induce it in the congruential generator.

Analysis of the period of the matrix congruential generator is somewhat more complicated than that of the scalar generator (see Grothe, 1987). It is clear that \(A\) should be nonsingular because otherwise it is possible to generate a zero vector, at which point all subsequent vectors are zero.

A straightforward extension of the matrix congruential generator is the multiple recursive matrix random number generator:

\[ x_i \equiv (A_1 x_{i-1} + A_2 x_{i-2} + \cdots + A_k x_{i-k}) \mod m. \]  

(4.10)

This is the same idea as in the multiple recursive generator for scalars, as considered above. Also, as mentioned above, computational efficiency can be enhanced by choosing some of the \(A_j\) as 0 matrices, identity (or negative identity) matrices, or simply scalars. Deng and Lin (2000) suggest a fast implementation with all multipliers equal to 0 except the first, which is the constant \(-1\) (that is, \(A_1 = -I\)), and the last, which is a scalar \(a_k\) (that is, \(A_k = a_k I\)).

Feedback Shift Register Generators

Tausworthe (1965) introduced a generator based on a sequence of 0’s and 1’s generated by a recurrence of the form

\[ a_i \equiv (c_p a_{i-p} + c_{p-1} a_{i-p+1} + \cdots + c_1 a_{i-1}) \mod 2, \tag{4.11} \]

where all variables take on values of either 0 or 1. A similar generator could, of course, be devised with a modulus \( m \), not necessarily 2.

If the modulus is a prime, the generator can be related to a polynomial

\[ f(z) = z^p - (c_1 z^{p-1} + \cdots + c_{p-1} z + c_p) \tag{4.12} \]

over the Galois field defined over the integers 0, 1, \ldots, \( m-1 \) with the addition and multiplication being defined in the usual way followed by a reduction modulo \( m \). We will denote a Galois field over a set with \( m \) elements as \( \mathbb{G}(m) \).

An important result from the theory developed for such polynomials is that, so long as the initial vector of 1’s are not all 0’s, the period of the recurrence (4.11) is \( m^p - 1 \) if and only if the polynomial (4.12) is irreducible (i.e., cannot be factored) over \( \mathbb{G}(m) \). An irreducible polynomial is also called a primitive polynomial. See Knuth (1998) for further discussion of these properties as well as for a method to determine primitive polynomials. It is easy to see that the maximal period is \( m^p - 1 \), because if any \( p \)-vector of 1’s repeats, all subsequent values are repeats.

For computational efficiency, the modulus in (4.11) should be 2 and most of the \( c \)’s should be zero. For \( m = 2 \), there is only one binomial that is irreducible; and it is \( z + 1 \), which would yield an unacceptable period. There are, however, many trinomials and higher degree polynomials. See Zierler and Brillhart (1968, 1969) for a list of all primitive trinomials modulo 2 up to degree 1,000. Kumada et al. (2000) give additional trinomials as well as several fifth degree polynomials.

The recurrence (4.11) often has the form

\[ a_i \equiv (a_{i-p} + a_{i-p+q}) \mod 2, \tag{4.13} \]

resulting from a trinomial. Addition of 0’s and 1’s modulo 2 is the binary exclusive-or operation, denoted by \( \oplus \); thus we may write the recurrence as

\[ a_i = a_{i-p} \oplus a_{i-p+q}. \tag{4.14} \]

After this recurrence has been evaluated a sufficient number of times, say \( l \) (with \( l \leq p \)), the \( l \)-tuple of 1’s is interpreted as a number in base 2. This is referred to as an \( l \)-wise decimation of the sequence of 1’s. If \( l \) is relatively prime to \( 2^p - 1 \) (the period of the sequence of 1’s), the period of the \( l \)-tuples will also be \( 2^p - 1 \). In this case the decimation is said to be proper. Note that the recurrence of bits is the same recurrence of \( l \)-tuples,

\[ x_i = x_{i-p} \oplus x_{i-p+q}, \tag{4.15} \]
where the $x$'s are the numbers represented by interpreting the $l$-tuples as binary notation, and the exclusive-or operation is performed bit-wise. The underlying trinomial is

$$x^r + x^r + 1,$$

where $r = p - q$. A random number generator built on this recurrence is sometimes denoted by $R(r, p)$.

As an example, consider the trinomial

$$x^4 + x + 1,$$

and begin with the bit sequence

$$1, 0, 1, 0.$$

For this polynomial, $p = 4$ and $q = 3$ in the recurrence (4.14). Operating the generator, we obtain

$$1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0,$$

at which point the sequence repeats; its period is $2^4 - 1$. A 4-wise decimation using the recurrence (4.15) yields the numbers

$$12, 8, 15, 5, \ldots$$

(in which the 5 required an additional bit in the sequence above). We could continue in this way to get 15 (that is, $2^4 - 1$) integers between 1 and 15 before the sequence began to repeat.

As with the linear congruential generators, different values of the $c$'s and even of the starting values of the $a$'s can yield either good generators (i.e., ones whose outputs seem to be random samples of a uniform distribution) or bad generators.

This recurrence operation can be performed in a feedback shift register, which is a vector of bits that is shifted, say, to the left, one bit at a time, and the bit shifted out is combined with other bits in the register to form the rightmost bit. The operation can be pictured as shown in Figure 4.8, where the bits are combined using $\oplus$.

The two sources of bits to shift into the right side of the register are called “taps”. Thus, Figure 4.8 represents a two-tap generator. The basic idea of the feedback shift register can of course easily be extended to more than two taps.

**Generalized Feedback Shift Registers and Variations**

Feedback shift registers have an extensive theory (see, e.g., Golomb, 1982) because they have been used for some time in communications and cryptography.
A variation on the Tausworthe generator is suggested by Lewis and Payne (1973), who call their modified generator a generalized feedback shift register (GFSR) generator. In the GFSR method, the bits of the sequence from recurrence (4.13) form the bits in one binary position of the numbers being generated. The next binary position of the numbers being generated is filled with the same bit sequence but with a delay. By using the bit stream from the trinomial $x^4 + x + 1$ and the starting sequence we considered before, and again forming 4-bit words by putting the bits into a fixed binary position with a delay of 3 between binary positions, we have

\[
\begin{align*}
1010 &= 10 \\
1110 &= 14 \\
0011 &= 3 \\
0101 &= 5 \\
1111 &= 15 \\
0001 &= 1 \\
0010 &= 2 \\
0111 &= 7 \\
1000 &= 8 \\
1001 &= 9 \\
1011 &= 11 \\
1100 &= 12 \\
0100 &= 4 \\
1101 &= 13 \\
0110 &= 6
\end{align*}
\]

at which point the sequence repeats. Notice that the recurrence (4.15),

\[x_i = x_{i-p} \oplus x_{i-p+q},\]

still holds, where the $x$'s are the numbers represented by interpreting the $l$-tuples as binary notation, and the exclusive-or operation is performed bitwise.

Lewis and Payne (1973) discussed methods of initializing the generator, and gave programs both for the initialization and for the generation. Kirkpatrick and Stoll (1981) presented a faster way of initializing the generator.
and developed a program, R250, implementing a generator with \( p = 250 \) and \( q = 103 \) (that is, an \( R(103, 250) \)). This program is widely used among physicists. Empirical results for this generator are mixed. The generator performed well in tests reported by L’Ecuyer (1997), but other tests based on physical simulations have indicated some problems (see, for example, Ferrenberg, Landau, and Wong, 1992; and Selke, Talapov, and Shchur, 1993).

Fushimi (1990) has studied GFSR methods in general, and has described some particular GFSR’s, which he showed to have good properties, both theoretically and empirically. One particular case studied by Fushimi had \( p = 521 \), with \( c_{521} = c_{32} = 1 \) (with a slight modification in the definition of the recursion as \( x_i = x_{i-3p} \oplus x_{i-3q} \)). The generator, which is available in the IMSL Libraries, has a period of \( 2^{521} - 1 \). This generator also performed well in tests reported by L’Ecuyer (1997), but not so well in one test by Matsumoto and Kurita (1994).

Ziff (1998) suggests that the problems arising in two-tap shift registers can be ameliorated by use of four-tap shift registers, that is, generators based on the recurrence

\[
x_i = x_{i-a} \oplus x_{i-b} \oplus x_{i-c} \oplus x_{i-d}.
\]

He shows that four-tap sequences can also be constructed by decimation of two-tap sequences. See also Compagner (1991, 1995) for further discussion of multi-tap rules.

Another way of addressing the problems of two-tap shift registers has been proposed by Matsumoto and Kurita (1992, 1994). They modify the GFSR in the recurrence (4.15), by “twisting” the bit pattern in \( x_{i-p+q} \). This is done by viewing the \( x \)’s as \( l \)-vectors of zeros and ones and multiplying \( x_{i-p+q} \) by an \( l \times l \) matrix \( A \). The recurrence then becomes

\[
x_i = x_{i-p} \oplus Ax_{i-p+q}.
\]

They call this a twisted GSFR generator. Matsumoto and Kurita (1994) show how to choose \( A \) and modify the output so as to achieve good uniformity properties. They also gave a relatively short C program implementing the method. Further empirical studies are needed for this generator. The idea of a twisted GFSR generator is related to the matrix congruential generators discussed on page 140.

There is some evidence that a polynomial with more nonzero terms may be better than a trinomial in the recurrence (4.11). Matsumoto and Nishimura (1998) describe a twisted generator based on the recurrence (4.17) that has approximately 100 terms in the characteristic polynomial of the matrix \( A \). They give a C program for a version of this generator, called MT19937, that has a period of \( 2^{19937} - 1 \) and apparently very good uniformity properties. Additional empirical studies are needed for this generator.

In our discussion of computer implementation of congruential generators beginning on page 134, we mentioned a technique to insure that the result of the generator be less than 1. The same technique, of course, works for GFSR
4.1 Simulating Random Numbers from a Uniform Distribution

generators, but in these generators we have the additional concern of a 0. By using the obvious bit operations, which are inexpensive, it is usually fairly easy to prevent 0 from occurring, however.

**Skipping Ahead in GFSR Generators**

Golomb (1982) noted that the basic recurrence (4.15),

\[ x_i = x_{i-p} \oplus x_{i-p+q}, \]

implies

\[ x_i = x_{i-2^e p} \oplus x_{i-2^e p+2^e q}, \]

for any integer \( e \). This provides a method of skipping ahead in a generalized feedback shift register generator by a fixed distance that is a power of 2. Aluru, Prabhu, and Gustafson (1992) described a leapfrog method using this relationship and applied it to parallel random number generators. So for \( k \) a power of 2, we have

\[ x_i = x_{i-kp} \oplus x_{i-k(p-q)}, \]

and so we can generate a leapfrog sequence analogous to (4.5), page 132, for multiplicative congruential generators:

\[ x_s, x_{s+k}, x_{s+2k}, \ldots \]

**Other Sources of Uniform Random Numbers**

Many different mechanisms for generating random numbers have been introduced. Some are simple variations of the linear congruential generator or the feedback shift register, perhaps designed for microcomputers, or some other special environment. Various companies distribute proprietary random number generators, perhaps based on some physical process. Marsaglia (1995) uses some of those, and his assessment of the ones he used was that they were not of high quality. (Nevertheless, he combined them with other sources and produced pseudorandom streams of high quality.)

Digits in the decimal representations of approximations to transcendental numbers, such as \( \pi \) or \( e \), or simpler irrational numbers, such as \( \sqrt{2} \), have often been suggested as streams of independent discrete uniform numbers (see Dodge, 1996, for example). Most statistical tests of such streams have not detected nonzero correlations or departures from uniformity. See Jaditz (2000) and NIST (2000) for summaries of some tests of the randomness in the expansions of these irrational numbers.
Generators Based on Cellular Automata

John von Neumann introduced cellular automata as a way of modeling biological systems. A cellular automaton consists of a countable set of fixed sites, each with a given value chosen from a countable, usually finite, set. In discrete time steps, the values at all sites are updated simultaneously based on simple rules that use the values at neighboring sites. If the values at a time $k$ are $a_{-2}^{(k)}$, $a_{-1}^{(k)}$, $a_0^{(k)}$, $a_1^{(k)}$, $a_2^{(k)}$, ..., the update rule is some function $\phi$, and

$$
a_i^{(k+1)} = \phi(a_{i-r}^{(k)}, a_{i-r+1}^{(k)}, ..., a_{i}^{(k)}, ..., a_{i+r}^{(k)}).
$$

Wolfram (1994) provides extensive descriptions of ways of constructing cellular automata and discussions of their properties. Wolfram (1984) suggested a random number generator using the cellular automaton in which $a_i$ takes on the values 0 and 1, and

$$
a_i^{(k+1)} \equiv (a_{i-1}^{(k)} + a_{i}^{(k)} + a_{i+1}^{(k)} + a_{i+r}^{(k)}) \mod 2. \tag{4.18}
$$

In practice, of course, at any stage there would be relatively small number of $a$’s, fewer than 64 or so. (Edge effects in (4.18) are handled by assigning 0’s to $a$’s whose indices are out of range.) Wolfram describes some ways of mapping a vector of bits to an integer, and then, from a given integer characterized by a finite vector of bits, using the iteration (4.18) to generate a new number. A random number generator based on the cellular automaton (4.18) is available in Mathematica.

Generators Based on Chaotic Systems

Because chaotic systems exhibit irregular and seemingly-unpredictable behavior, there are many connections between random number generators and chaotic systems. Chaotic systems are generally defined by recursions, just as random number generators. It is difficult to define useful random processes based on chaotic systems, but some of the results of chaos theory may have some relevance to the understanding of pseudorandom generators.

Lüscher (1994) relates the Marsaglia-Zaman subtract-with-borrow generator to a chaotic dynamical system. This relationship allows an interpretation of the correlational defects of the basic generator as short-term effects, and provides a way for skipping ahead in the generator to avoid those problems. The Lüscher generator has a long period and performed well in the statistical tests reported by Lüscher. James (1994) provides a portable Fortran program, RANLUX, implementing this generator.

Other Recursive Generators

Another family of combination generators with long periods are the so-called ACORN (additive congruential random number) generators described by
4.1 Simulating Random Numbers from a Uniform Distribution

The \( k \)th order ACORN generator yields the sequence \( u_{k,1}, u_{k,2}, u_{k,3}, \ldots \) from the recursions

\[
\begin{align*}
  u_{0,i} &= u_{0,i-1}, & \text{for } i \geq 1 \\
  u_{j,i} &\equiv (u_{j-1,i} + u_{j,i-1}) \mod 1, & \text{for } i \geq 1, \text{ and } j = 1, \ldots, k.
\end{align*}
\]

The ACORN generators depend heavily on the starting values \( u_{j,0} \). Wikramaratna (1989) shows that initial values satisfying certain conditions can yield very long periods for this generator. Although the generator is easy to implement, the user must be concerned with proper choice of seeds.

It is easy to modify existing random number generators or to form combinations of existing generators, and every year several new generators are suggested in the scientific literature. The *Journal of Computational Physics* and *Computer Physics Communications* provide new generators in abundance. Another one proposed in the former journal is the Chebyshev generator, defined by Erber, Everett and Johnson (1979) as

\[
\begin{align*}
  x_i &= x_{i-1}^2/2 \\
  u_i &= \frac{1}{\pi} \arccos(x_i/2).
\end{align*}
\]

Although this generator has some desirable qualities, it does not hold up well in statistical tests. (See other articles in the same journal, for example, Hosack, 1986, and Wikramaratna, 1989.)

Some newly-proposed generators merit further study, but the scientist generally must be very careful in choosing a generator to use. It is usually better to avoid new generators or ones for which there is not a large body of empirical evidence regarding their quality.

**Tables of Random Numbers**

Before the development of reliable methods for generating pseudorandom numbers, there were available a number of tables of random numbers that had been generated by some physical process or that had been constructed by systematically processing statistical tables to access midorder digits of the data they contained. The most widely used of these were the “RAND tables” (RAND Corporation, 1955), which contain a million random digits generated by a physical process. The random digits in these tables have been subjected to many statistical tests, with no evidence that they did not arise from a discrete uniform distribution over the ten mass points.

Marsaglia (1995) has produced a CD-ROM that contains 4.8 billion random bits stored in sixty separate files. The random bits were produced by a combination of two or more deterministic random number (pseudorandom number) generators and three devices that use physical processes. Some of the files also have added bit streams from digitized music or pictures.
Combining Generators

Both the period and the apparent randomness of random number generators can often be improved by combining more than one generator. The shuffling methods of MacLaren and Marsaglia (1965) and others, described on page 133, may use two or more generators in filling a table and shuffling it.

Another way of combining generators is suggested by Collings (1987). His method requires a pool of generators, each maintaining its own stream. For each number in the delivered stream, an additional generator is used to select which of the pool of generators is to be used to produce the number.

The generators that are combined can be of any type. One of the early combined generators, developed by George Marsaglia and called “Super-Duper”, used a linear congruential generator and a Tausworthe generator. See L'Ecuyer and Lewis (1973) for a description of Super-Duper.

Wichmann/Hill Generator

Wichmann and Hill (1982 and corrigendum, 1984) describe a combination of three linear congruential generators that is easy to program and has good randomness properties. The generator is

\[
\begin{align*}
  x_i &\equiv 171x_{i-1} \mod 30269 \\
  y_i &\equiv 172y_{i-1} \mod 30307 \\
  z_i &\equiv 170z_{i-1} \mod 30323,
\end{align*}
\]

and

\[
u_i = \left(\frac{x_i}{30269} + \frac{y_i}{30307} + \frac{z_i}{30323}\right) \mod 1.
\]

This generator requires three seeds \((x_0, y_0, z_0)\) and yields numbers \(u_i\) in the interval \((0,1)\). The period is of the order of \(10^{12}\). Notice that the final modular reduction is equivalent to retaining only the fractional part of the sum of three uniform numbers. (See Exercise 4.1a, page 222.)

Zeisel (1986) showed that the Wichmann/Hill generator is the same as a single multiplicative congruential generator with a modulus of 27,817,185,604.

De Matteis and Pagnutti (1993) studied the higher-order autocorrelations in sequences from the Wichmann/Hill generator and found them to compare favorably with those from other good generators. This, together with the ease of implementation, makes the Wichmann/Hill generator a useful one for common applications. A straightforward implementation of this generator can yield a 0. (See Exercise 4.13, page 224.)

L’Ecuyer Combined Generators

L’Ecuyer (1988) suggests combining \(k\) multiplicative congruential generators that have prime moduli \(m_j\), such that \((m_j-1)/2\) are relatively prime, and with
multipliers that yield full periods. Let the sequence from the $j^{\text{th}}$ generator be $x_{j,1}, x_{j,2}, x_{j,3}, \ldots$. Assuming that the first generator is a relatively “good” one, and that $m_1$ is fairly large, we form the $i^{\text{th}}$ integer in the sequence as

$$x_i \equiv \sum_{j=1}^{k} (-1)^{j-1} x_{j,i} \mod (m_1 - 1).$$

The other moduli $m_j$ do not need to be large.

The normalization takes care of the possibility of a 0 occurring in this sequence:

$$u_i = \begin{cases} x_i/m_1 & \text{if } x_i > 0, \\ (m_1 - 1)/m_1 & \text{if } x_i = 0. \end{cases}$$

A specific generator suggested by L’Ecuyer (1988) is the following:

$$x_i \equiv 40014x_{i-1} \mod 2147483563$$
$$y_i \equiv 40692y_{i-1} \mod 2147483399 \quad (4.19)$$
$$z_i \equiv (x_i - y_i) \mod 2147483563,$$

and

$$u_i = 4.656613z_i \times 10^{-10}.$$ 

The period is of the order of $10^{18}$. L’Ecuyer presented results of both theoretical and empirical tests that indicate that the generator performs well.

Notice that the difference between two discrete uniforms modulo the larger of the ranges of the two random variables is also a discrete uniform. (See Exercise 4.1c, page 222.) Notice also the normalization always yields numbers less than 1 because the normalizing constant is larger than $2147483563$.

L’Ecuyer (1988) gives a portable program for the generator that uses the techniques we have discussed for keeping the results of intermediate computations small (see page 134 and following).

**Properties of Combined Generators**

The uniform distribution has a number of interesting properties. We have already referred to the fact that if $R$ is a random variable distributed as $U(0, 1)$ and

$$S \equiv (kR + c) \mod 1$$

where $k$ is an integer constant not equal to 0, and $c$ is a real constant, then $S$ has a $U(0, 1)$ distribution. More generally, the fractional part of the sum of independent random variables one of which is $U(0, 1)$ is also a $U(0, 1)$ random variable.

Deng and George (1990) consider sums of random variables whose distributions are “nearly” uniform. If $X_i$ are independently distributed with probability density functions $p_i$, and $|p_i(x) - 1| \leq \epsilon_i$ over $[0, 1]$, then for the density $p_X$ of the reduced sum, $X = (\sum X_i) \mod 1$, we have
over \([0, 1]\). This implies that the random variable \(X\) is more nearly uniform in some sense than any of the individual \(X_i\).

Deng and George (1992) also describe several other properties of uniform variates, some of which characterize the uniform distribution, that can be useful in improving uniform random number generators.

**Additional Considerations**

We have mentioned the important difference between the ranges \([0, 1]\) and \((0, 1)\) for the simulated uniform distribution. Combining generators exposes the naive implementer to this problem anew. Whenever two generators are combined, there is a chance of obtaining a 0 or a 1 even if the output of each is over \((0, 1)\). See Exercise 4.13.

If the streams in a combination generator suffer similar irregularities in their patterns, the combination may not be able to overcome the problems. Combining some generators can actually degrade the quality of the output.

**4.1.2 Independent Streams and Parallel Random Number Generation**

Many of the Monte Carlo methods are “embarrassingly” parallel; they consist of independent computations, at least at some level, that are then averaged. The main issue for Monte Carlo methods performed in parallel is that the individual computations (that is, all computations except the outer averaging loop) be performed independently. For the random number generators providing data for parallel Monte Carlo computations, the only requirement over and above those of any good random number generator is to satisfy this requirement of independence of the random number streams. This is exactly the same problem as blocking or renewal, which has long been a common practice in simulation (see, e.g., Mihram, 1972, and Maclaren, 1989). Blocks are used in simulation experiments in much the same way as in ordinary experiments: to obtain a better measure of the underlying variation. In each block or each simulation run, an independent stream of random numbers is required.

Although several methods have been proposed for generating independent streams of random numbers, there are basically two kinds of methods. One type is based on skipping ahead in the stream, either by using different starting points or by a leapfrog method, as discussed in Section 4.1.1 on page 132 for linear congruential generators and in Section 4.1.1 on page 145 for GFSR generators. Both kinds of skipping ahead can be done randomly or at a fixed distance. The other type of method uses a combination of generators, either by simply using completely different generators in each of the parallel streams or by using a combination, such as suggested in Section 4.1.1, page 148.
4.1 Simulating Random Numbers from a Uniform Distribution

Lehmer Trees

Frederickson et al. (1984) describe a way of combining linear congruential

generators to form what they call a Lehmer tree, which is a binary tree in

which all right branches or all left branches form a sequence from a Lehmer

linear congruential generator. The tree is defined by the two recursions, both

of which are the basic recursion (4.1) introduced by Lehmer,

\[
x_i^{(L)} \equiv (a_L x_{i-1} + c_L) \mod m,
\]

and

\[
x_i^{(R)} \equiv (a_R x_{i-1} + c_R) \mod m.
\]

At the \((i-1)\)th node in the tree, an ordinary Lehmer sequence with seed \(x_{i-1}\)

is generated by all right-branch nodes below it. A new sequence is initiated

by taking a left branch to the first node below it and then all right branches

from then on. The question is whether a (finite) sequence of all right-branch

nodes below a given node is independent from the (finite) sequence of all

right-branch nodes below the left-branch node immediately below the given

node. (“Independent” for these finite subsequences can be interpreted strictly

as having no elements in common.) Frederickson et al. gave conditions on

\(a_L, c_L, a_R, c_R,\) and \(m\) that would guarantee the independence for a fixed length

of the sequences.

Although Frederickson et al. (1984) concentrate on congruential generators

whose modulus is a power of 2, Bowman and Robinson (1987) show that

Lehmer trees constructed from generators using a modulus of that form had

very poor properties. The problem is the same that we pointed out in Sec-

tion 4.1.1, page 122, for generators with such moduli; the low-order bits have

very short periods.

If the modulus in the lagged Fibonacci generator (4.8), page 139,

\[
x_i \equiv (x_{i-j} + x_{i-k}) \mod m,
\]

is a power of 2, say \(2^p\), the maximum period possible is \((2^p - 1)2^{p-1}\). Mascagni

et al. (1995) describe a method of identifying and using exclusive substreams,

each with the maximal possible period.

Neither the congruential generator nor the generalized feedback register

generator has entirely satisfactory methods of skipping ahead for use in par-

allel random number generation. As we have already mentioned, the congru-

ential generator also suffers from a relatively short period.

Combination Generators

The ability to obtain independent streams and the long period and other good

properties of a GFSR generator can be achieved by a combination generator

in which a simple skip-ahead method is implemented in one of the generators,
either a congruential or a GFSR generator, and the other generator is a good GFSR generator. Wollan (1992) describes such a method, which combines a skipping multiplicative congruential generator with a lagged Fibonacci generator, by subtraction modulo 1 of the normalized output. For the skipping congruential generator based on \( x_i \equiv a x_{i-1} \mod m \), the seed \( a^j x_0 \) is used for the \( j \)th processor, and the recurrence \( x_i \equiv a^k x_{i-1} \mod m \) is used in each processor. The same Fibonacci generator is used in all processors. Wollan also allowed for a single process to spawn a child process. Both the parent and the child process begin using \( a^2 \) in place of \( a \) (i.e., the multiplier for both is \( a^{2k} \) when one of the original \( k \) processes spawns a new process), and the child process skips from the current value \( x_i \) to \( ax_i \).

**Monte Carlo on Parallel Processors**

Eddy (1990) and Mascagni et al. (1993) provide overviews of some of the issues and some of the proposed methods for generating random numbers on parallel processors, especially methods. Anderson (1990) discusses general methods for high-performance architectures. His emphasis is on vector processors, however.

There are basically two ways to approach the problem. One is to use blocking or leapfrogging, as we discuss on page 131 for linear congruential generators and on page 145, for feedback shift register methods. The other is to use different generators on different processors. This method is called parametrization. Mascagni and Srinivasan (2000) describe and give C code for a library of generators that operate in parallel by choosing different generators on different processors. The library, called SPRNG, includes linear congruential generators, feedback shift register generators, and lagged Fibonacci generators.

Assessing the quality of random number generators operating in parallel can be quite difficult. Hanxleden and Scott (1992) discuss ways of determining correctness of a random number generator operating in parallel, and they describe a dynamic way of choosing seeds in the different processors. (“Correctness” in this case means that the computer is executing a program that implements the algorithm correctly.) See also Cuccaro, Mascagni, and Pryor (1994) for some of the issues involved.

**4.1.3 Quality of Random Number Generators**

Ziff (1992) describes a simulation requiring a total of \( 6 \times 10^{12} \) random numbers and using a few months computing time on about a dozen workstations running simultaneously. Research work like this that depends so heavily on random numbers emphasizes the need for high-quality random number generators. Yet, as we have seen, not all random number generators are good ones (Park and Miller, 1988, and Ripley, 1988).
4.1 Simulating Random Numbers from a Uniform Distribution

Only random number generators that have solid theoretical properties and that have also successfully passed a battery of tests should be used. Even so, often in Monte Carlo applications it is appropriate to construct ad hoc tests that are sensitive to departures from distributional properties that are important in the given application. For example, in using Monte Carlo methods to evaluate a one-dimensional integral, autocorrelations of order one may not be harmful, but they may be disastrous in evaluating a two-dimensional integral. If the routines for generating random deviates from nonuniform distributions use exact methods, their quality depends almost solely on the quality of the underlying uniform generator. Nevertheless, it is often advisable to employ an ad hoc goodness-of-fit test for the transformations that are to be applied.

Analysis of the Algorithm

An initial consideration for any generator is its period. As the number and complexity of Monte Carlo studies increase, our requirement for the period of the generator increases. The minimal standard generator of Park and Miller (1988) (see page 130) has a period of $2^{31} - 1$. As Deng and Lin (2000) argue, this is no longer sufficient for most serious work.

For a generator that yields $m$ different values, $k$-tuples of successive numbers in the output would ideally fall on $m^k$ points uniformly spaced through a $k$-dimensional hypercube. As we have seen, however, $k$-tuples of the output from pseudorandom number generators lie on somewhat restrictive lattices in a $k$-dimensional hypercube.

In a simple linear congruential generator with period $m$, no point can repeat within a single period; therefore, there can be only $m$ points in the lattice generated by such a generator. Although, as we have seen, there is no reason to use only simple linear congruential generators, they are widely used, and so assessing their lattice structure is of interest. The lattice structure of these common types of congruential generators can be assessed by means of two analytic “tests” of the generator.

The spectral test determines the maximum distance between adjacent parallel hyperplanes defined by the lattice. The smaller the maximum distance is, the better the output of the generator covers the hypercube.

The spectral test for a linear congruential generator does not use the output of the generator, but rather computes values analogous to a discrete Fourier transform of the output using the recurrence formula of the congruential relationship itself. For a linear congruential generator with multiplier $a$ and modulus $m$, let $d_k(m,a)$ represent the maximum distance between adjacent parallel hyperplanes. Now, for any $k$-dimensional lattice with $m$ points, there is a lower bound, $d^*_k(m)$, on $d_k(m,a)$ (see Knuth, 1998); hence, a useful measure for a given multiplier is the ratio

$$S_k(m,a) = \frac{d^*_k(m)}{d_k(m,a)}.$$  (4.20)
The closer this ratio is to 1, the better the generator is with respect to this measure. Coveyou and MacPherson (1967) describe a spectral test method that computes the denominator in equation (4.20). Their method was improved by Dieter (1975) and Knuth (1998), and is described in detail on pages 98 through 103 of the latter reference. Hopkins (1983) gives a computer program to implement the Coveyou/MacPherson test.

L’Ecuyer (1988) computes values of $S_k(2^{31} - 1, a)$ for some specific values of $k$ and $a$. For example, for the multiplier 16 807 (see Section 4.1.1),

$$
\begin{array}{cccccc}
 k & 2 & 3 & 4 & 5 & 6 \\
 S_k(2^{31} - 1, 16 807) & .34 & .44 & .58 & .74 & .65 \\
\end{array}
$$

and for the multiplier 950 706 376 (see Exercise 4.7, page 223),

$$
\begin{array}{cccccc}
 k & 2 & 3 & 4 & 5 & 6 \\
 S_k(2^{31} - 1, 950 706 376) & .86 & .90 & .87 & .83 & .83 \\
\end{array}
$$

(Although often one sees values of $S_k(m, a)$ given to several digits of precision, maybe because the computational algorithms used are probably accurate to 6 or 7 digits, anything beyond 1 or 2 digits is not very meaningful.)

Another assessment of the badness of the lattice is the lattice test of Marsaglia (1972a), which is ratio of the longest edge of the smallest parallelepiped in $k$ dimensions to that of the shortest edge. Atkinson (1980) implemented Marsaglia’s lattice test and illustrated its use on some generators. Eichenauer and Niederreiter (1988), however, give examples that show that the lattice test is quite weak.

Other ways of examining the structure are to perform statistical analyses of the output of the generator. Neither the spectral test nor the lattice test use statistical analysis.

The general problem of long-range correlations in sequences from multiplicative congruential generators has been studied by Eichenauer-Herrmann and Grothe (1989) and by De Matteis and Pagnutti (1990). In general, for most generators the correlations die out very early, that is, at very low lags; and then the correlations grow again at large lags. At a lag equal to the period, the correlation becomes 1, of course. For some generators, the correlation may become relatively large at some intermediate lags. In most cases, the correlations are not large enough to be statistically significant if the generated sequence would be assumed to arise from a random (instead of pseudorandom) process. The fact that the correlations are not statistically significant does not mean that these systematic correlations would not be so bad as to invalidate a Monte Carlo study.

Beyer, Roof, and Williamson (1971) and Beyer (1972) describe a measure of the uniformity of the lattice of the output of a random number generator in terms of the ratio of the shortest vector to the longest vector in the
Minkowski reduced basis of the lattice. A basis for a lattice in \( \mathbb{R}^d \) is a linearly-independent generating set of the lattice. Given a set of linearly independent vectors \( \{v_1, v_2, \ldots, v_d\} \) in \( \mathbb{R}^d \), a lattice is the set of vectors \( w \) of the form \( \sum_{i=1}^{d} z_i v_i \), where \( z_i \) are integers. (See Figure 4.1, page 125.) The set of vectors \( \{v_1\} \) is a basis for the lattice. The basis is a Minkowski reduced basis if

\[
\|v_k\| \leq \left\| \sum_{i=1}^{d} z_i v_i \right\|, \quad \text{for } 1 \leq k \leq d,
\]

for all sets of \( d \) integers \( z_i \) such that the greatest common divisor of the set \( \{z_k, z_{k+1}, \ldots, z_d\} \) is 1. (The symbol \( \|v\| \) denotes the Euclidean norm of the vector \( v \); see Section 5.1.1.) The ratio of the length of the shortest vector to the longest length,

\[
\frac{\|v_d\|}{\|v_1\|}.
\]

is called the Beyer ratio. A large Beyer ratio (close to 1) indicates good uniformity. It is not easy to compute the Beyer ratio for a given generator. Afflerbach and Grothe (1985) give an algorithm for computing the ratio for linear congruential generators. They used the algorithm up to dimension of 20. Eichenauer-Herrmann and Grothe (1990) give upper bounds for the ratio for linear congruential generators that are applicable to higher dimensions. The problem of determining the shortest vectors defining a lattice has been shown to be NP-complete. (See Chapter 3.)

The discrepancy of a set of points \( P = \{p_1, p_2, \ldots, p_n\} \) in the unit hypercube is

\[
D_n(P) = \sup_{J} |F_n(J) - V(J)|,
\]

(4.21)

where the supremum is taken over all subintervals \( J \) of the unit hypercube, \( F_n(J) \) is the number of points in \( P \cap J \) divided by \( n \), and \( V(J) \) is the volume of \( J \). The discrepancy is a measure of the uniformity of spread of a given set of points within another set of points.

For a given generator of points, we may consider the discrepancy of arbitrary sets of points within an \( s \)-dimensional unit hypercube. In that case we use the notation \( D^{(s)} \).

**Statistical Tests of Random Number Generators**

In addition to analyzing the algorithm itself, we may evaluate a specific generator by analyzing the output stream it produces. This approach has the additional advantage of assessing the program that is actually used, so the question of whether the program correctly implements the algorithm does not arise. There are basically two general kinds of properties of a generator to assess: the elements in the set of deviates without regard to the order in which they were generated, that is, the “static” properties; and the patterns in the sequences of the deviates, that is, the “dynamic” properties.
In Section 4.1.1, beginning on page 122, we discussed some ways of assessing the output of a linear congruential random number generator. Those tests addressed the structure of the output and how well the output covered the space. Other kinds of tests are statistical tests applied to a sample of the output.

An important source of information about the quality of a generator are specific simulations. Whatever is known about the physics of a problem that is simulated can be compared with the results of a simulation. It was this kind of study that led to the discovery of problems with RANDU (see Coldwell, 1974).

Empirical tests of random number generators include both statistical tests of the output stream and the anecdotal evidence of specific simulations.

**Statistical Tests**

The static statistical tests are goodness-of-fit tests for various combinations and transformations of the numbers in the output stream. These tests are of infinite variety.

The two most common types of goodness-of-fit tests are chi-squared tests and Kolmogorov-Smirnov tests. In a chi-squared goodness-of-fit test, observed counts of events are compared with counts of occurrences of those events that would be expected under the null hypothesis to be tested. If the null hypothesis, for example, is that a random variable has a uniform (0, 1) distribution, we may take 100 observations on the random variable, count how many are in each of the ten intervals (0, 0.1], (0.1, 0.2], ..., (0.9, 1.0], and compare those counts with the expected numbers, which in this case would be 10 in each interval. If the observed numbers are significantly different from the expected numbers, we have reason to reject the null hypothesis. Formally, we compute

\[ \chi^2_c = \sum_{i=1}^{k} \frac{(o_i - e_i)^2}{e_i}, \]

where \( k \) is the number of intervals (10, in this case), and \( o_i \) and \( e_i \) are, respectively, the observed and expected numbers in the \( i \)th interval. Under the hypothesis of independent sampling from a uniform (0, 1) distribution, \( \chi^2_c \) is a realization of a random variable that has an approximate chi-squared distribution with \( k - 1 \) degrees of freedom. Large values of \( \chi^2_c \) provide evidence that the observed counts differ by large amounts from the expected counts; that is, evidence that the hypothesized distribution is not the true one. The test is performed by computing the probability that a chi-squared random variable with \( k - 1 \) degrees of freedom would be as large as the observed value, \( \chi^2_c \). This probability, which is called the “p-value” can be computed by use of many software packages. If we let \( x \) take the value of \( \chi^2_c \), using the R function, `pchisq`, for example, the probability is
4.1 Simulating Random Numbers from a Uniform Distribution

\[ 1 - \text{pchisq}(x, k - 1); \]

or, using the IMSL Fortran function, \text{chidf}, the probability is

\[ 1 - \text{chidf}(x, k - 1). \]

If the p-value is smaller than some preset value, say 0.05, the null hypothesis is rejected.

A Kolmogorov-Smirnov test (or just “Kolmogorov test”) compares the empirical cumulative distribution function, or ECDF, with the hypothesized cumulative distribution function, or CDF. The CDF of a random variable \( X \), denoted by \( P(x) \), is defined by

\[ P(x) = \Pr(X \leq x), \]

where “Pr” represents “probability”. For a sample of size \( n \), the ECDF is

\[ P_n(x) = \frac{1}{n} \sum_{i=1}^{n} I_{(-\infty,x]}(x_i), \]

for the indicator function \( I_{(-\infty,x]}(\cdot) \). The comparison of the two functions used by the Kolmogorov-Smirnov test is based on

\[ \sup_x |P_n(x) - P(x)|, \]

which is the Kolmogorov distance between two probability distributions. (In this case, one of the probability distributions is the discrete uniform distribution with probability 1/n at each of the points in the sample.) The discrepancy in equation (4.21) is a multivariate Kolmogorov distance. R provides a function \text{ks.gof} and the IMSL Fortran Library provides a subroutine \text{ksone} that compute the Kolmogorov-Smirnov statistic and determine its p-value. For further discussion of these tests and their properties and limitations, the reader is referred to any comprehensive text on applied statistics.

An easy statistical test for a basic random number generator is just a goodness-of-fit test for uniformity in various dimensions. A test for uniformity in one dimension is just based on the one-dimensional histogram. For higher dimensions, the bins are usually made to correspond to groups formed by subsequences of the output of the generator; for example, in two dimensions, each successive pair of numbers constitutes a single bivariate observation. There are obviously other ways of forming multivariate observations by grouping variates in a univariate sequence.

Dynamic statistical tests are for the independence of the output. These tests address patterns in the output stream; an example is a test for serial correlation of various lags. Kennedy and Gentle (1980) describe a variety of statistical tests of both static and dynamic types.

Fishman and Moore (1982, 1986) developed an extensive suite of goodness-of-fit tests based on various transformations of the sample. They applied the
DIEHARD tests for random number generators
birthday spacing test
overlapping permutation test
binary matrix rank test
tests to linear congruential generators with the same modulus but different multipliers, in order to identify good multipliers.

All it takes to devise an empirical test is a little imagination. Many tests are based on familiar everyday random processes, such as occur in gaming. A “poker test”, for example, is a chi-squared goodness-of-fit test for simulated poker hands. The idea is to devise a scheme to use a random number generator for simulating deals from a poker deck, and then to compare the sampled frequency of various poker hands with the expected frequency of such hands.

There are two important suites of statistical tests for random number generators, the DIEHARD tests and the NIST Test Suite.

DIEHARD Tests

Marsaglia (1985) describes a battery of eighteen goodness-of-fit tests called “DIEHARD”. Source code for these tests is available in a CD-ROM (Marsaglia, 1995) and at

http://stat.fsu.edu/~geo/diehard.html

Most of these tests are performed on integers in the interval \((0, 2^{31} - 1)\) that are hypothesized to be realizations of a discrete uniform distribution with mass points being the integers in that range. These tests indicate a variety of possibilities for assessing randomness of particular features exhibited by a sequence of numbers. The tests are:

- birthday spacings test
  For this test, choose \(m\) birthdays in a year of \(n\) days. List the spacings between the birthdays. If \(j\) is the number of values that occur more than once in that list, then \(j\) has an asymptotic Poisson distribution with mean \(m^3/(4n)\). Various groups of bits in the binary representation of the hypothesized uniform integers are used to simulate birthday spacings; goodness-of-fit tests are performed, yielding \(p\)-values; other groups of bits are used and tests performed to yield more \(p\)-values; and so on. Then a goodness-of-fit test is performed on the \(p\)-values.

- overlapping 5-permutation test
  Each set of five consecutive integers can be in one of 120 states, for the 5! possible orderings of five numbers. The test is on the observed numbers of states and transition frequencies.

- binary rank test for \(31 \times 31\) matrices
  The leftmost 31 bits of 31 random integers from the test sequence are used to form a \(31 \times 31\) binary matrix over the field \(\mathbb{G}(2)\). The rank of the matrix is determined. Each time this is done, counts are accumulated for matrices with ranks of 31, of 30, of 29 and of 28 or less. A chi-squared test is performed for these four outcomes.

- binary rank test for \(32 \times 32\) matrices
  This is similar to the test above.
4.1 Simulating Random Numbers from a Uniform Distribution

- **binary rank test for $6 \times 8$ matrices**
  
  This is similar to the tests above, except that the rows of the matrix are specified bytes in the integer words.

- **bitstream test**
  
  Using the stream of bits from the random number generator, form 20-bit words, beginning with each successive bit, that is, the words overlap. The bitstream test compares the observed number of missing 20-bit words in a string of $2^{21}$ overlapping 20-bit words with the approximate distribution of that number.

- **overlapping-pairs-sparse-occupancy test**
  
  For this test 2-letter “words” are formed from an alphabet of 1024 letters. The letters in a word are determined by a specified ten bits from a 32-bit integer in the sequence to be tested, and the bits defining the letters overlap. The test counts the number of missing words, that is, combinations that do not appear in the entire sequence being tested. The count has an asymptotic normal distribution, and that is the basis for the goodness-of-fit test, when many sequences are used.

- **overlapping-quadruples-sparse-occupancy test**
  
  This is similar to the test above. The null distribution of the test statistic is very complicated; interestingly, a parameter of the null distribution of the test statistic was estimated by Monte Carlo.

- **DNA test**
  
  This is similar to the tests above, except that it uses 10-letter words built on a 4-letter alphabet (the DNA alphabet).

- **count-the-1’s test on a stream of bytes**
  
  This test is based on the binomial $(8, 0.5)$ distribution of 1’s in uniform random bytes. Rather than testing this directly, however, counts of 1’s are grouped into 5 sets $\{0, 1, 2\}$, $\{3\}$, $\{4\}$, $\{5\}$, and $\{6, 7, 8\}$; that is, if a byte has no 1’s, exactly one 1, or exactly two 1’s, it is counted in the first group.

  Next, overlapping sequences of length 5 are formed and the counts of each of the 5^5 combinations are obtained. A chi-squared test is performed on the counts. The test takes into account the covariances of the counts, and so is asymptotically correct. If the groups of counts of 1’s are thought of as 5 letters, and the groups of bytes thought of as 5-letter words, the output of the sequence under test is similar to the typing of a “monkey at a typewriter hitting five keys” (with rather strange probabilities), so the test is sometimes called the “monkey at a typewriter test”.

- **count-the-1’s test for specific bytes**
  
  This is similar to the test above.

- **parking lot test**
  
  This test is based on the results of randomly packing circles of radius 1 about a center located randomly within a square of side 100. The test is based on the number of “cars packed” (i.e., non-overlapping circles packed) after a large number of attempts. The distribution of the test statistic, and hence its critical values, are determined by simulation.
minimum distance test
3-D sphere test
squeeze test
overlapping sums test
craps test
runs test
NIST Test Suite, for random number generators

- minimum distance test
  This test is based on the minimum distance between a large number of random points in a square. If the points are independent uniform, then the square of the minimum distance should be approximately exponentially distributed with mean dependent on the length of the side of the square and the number of points. A chi-squared goodness-of-fit test is performed on the $p$-values of a large number of tests for exponentiality.

- 3-D spheres test
  In this test, 4000 points are chosen randomly in a cube of edge 1000. At each point, a sphere is centered large enough to reach the next closest point. The volume of the smallest such sphere is approximately exponentially distributed with mean $\frac{120\pi}{3}$. Thus the radius cubed is exponential with mean 30. (The mean was obtained by extensive simulation.) A chi-squared goodness-of-fit test is performed on the $p$-values of a large number of tests for exponentiality.

- squeeze test
  This test is performed on floating-point numbers hypothesized to be from a $U(0,1)$ distribution. Starting with $k = 2^{31}$, the test finds $j$, the number of iterations necessary to reduce $k$ to 1, using the reduction $k = \lceil k \ast U \rceil$, with $U$ from the stream being tested. Such $j$'s are found 100,000 times, then counts for the number of times $j$ was $\leq 6,7,\ldots,47,\geq 48$ are used to provide a chi-squared test.

- overlapping sums test
  This test is performed on floating-point numbers hypothesized to be from a $U(0,1)$ distribution. Sums of overlapping subsequences of 100 uniforms are tested for multivariate normality by a chi-squared test several times and then a chi-squared goodness-of-fit test is performed on the $p$-values of the chi-squared tests.

- craps test
  This test simulates games of craps and counts the number of wins and the number of throws necessary to end each game, and then performs chi-squared goodness-of-fit tests on the observed counts.

- runs test
  This is the test of runs up and runs down described on page 161.

The NIST Test Suite

The NIST Test Suite, described in NIST (2000), includes sixteen tests. Descriptions of these tests, examples of their use, and source code are available at

http://csrc.nist.gov/rng/

Most of these tests are performed on sequences of bits that are hypothesized to be independent realizations of a Bernoulli process.
4.1 Simulating Random Numbers from a Uniform Distribution

- frequency test
- frequency tests within blocks
- runs test
  (based on the lengths of runs of bits)
- test for longest run of ones in a block
- random binary matrix rank test
- discrete Fourier transform test
- nonoverlapping template matching test
- overlapping template matching test
- Maurer’s universal statistical test
  Maurer (1992)
- Lempel-Ziv complexity test
  Ziv and Lempel (1977)
- linear complexity test
- serial test
- approximate entropy test
- cumulative sum test
- random excursions test
- random excursions variant test

**Runs Test**

One of the most effective of the dynamic tests is the runs test. A runs test uses the number and/or length of successive runs in which the values of the sequence are either increasing or decreasing. There are different ways these can be counted. To illustrate one way of counting runs, consider the sequence:

\[ 1, 5, 4, 1, 3, 1, 3, 4, 7 \]

There are 5 different runs in this sequence:

\[ \text{up : 1, 5}; \quad \text{down : 5, 4, 1}; \quad \text{up : 1, 3}; \quad \text{down : 3, 1}; \quad \text{up : 1, 3, 4, 7} \]

The number of runs could be used as a test statistic, but a more powerful test can be constructed using the lengths of the runs up and/or the runs down. The lengths of the runs up as shown above are 2, 2, and 4. With small integers such as in this example, an issue may be how to handle ties, that is, two adjacent numbers that are equal. This issue is not relevant when the test is applied to the output of a generator that simulates a continuous uniform distribution.

A more common way of counting the lengths of the runs up (and analogously, the runs down) is to identify breakpoints where the sequence no longer continues increasing (or decreasing). The lengths of runs are then the lengths of the subsequences between breakpoints. For the sequence above we have breakpoints as indicated below:
This gives 1 run-up of length 1; 2 runs-up of length 2; and 1 run-up of length 4. The runs test (more properly, the runs-up test and the runs-down test) is based on the covariance matrix of the random number of runs of each length. Usually only runs up to some fixed length, such as 7, are considered. (Runs greater than that are lumped together; the covariance matrix can be adjusted to fix the test.) The test statistic is the quadratic form in the counts with the covariance matrix, which has an asymptotic chi-squared distribution with degrees of freedom equal to the number of different lengths of runs counted. Grafton (1981) gives a Fortran program to compute the chi-squared test statistic for the runs test (with runs of length 1 through 6) and the significance level of the test statistic. The program is available from statlib as the Applied Statistics algorithm AS 157.

Comparisons of Simulated Results with Statistical Models

Another way that random number generators are tested is in their usage in simulations in which some of the output can be compared with what established theory would suggest. There are several physical processes in statistical mechanics that could be used for testing random number generators. One of the most widely-studied models in computational physics is the Ising model (see Section 2.1.6). This model can be solved analytically in two dimensions.

Ferrenberg, Landau, and Wong (1992) use some of the generators that meet the Park and Miller (1988) minimal standard (see page 130) to perform several simulation studies in which the correct answer was known. Their simulation results suggest that even some of the “good” generators could not be relied on in some simulations. In complex studies it is difficult to trace unexpected results to errors in the computer code or to some previously unknown phenomenon. Because of sampling error in simulations, there is always a question of whether results are due to a sample that is unusual. Vattulainen, Ala-Nissila, and Kankaala (1994) studied the methods of Ferrenberg, Landau, and Wong, and determined that the anomalous results were indeed due to defects in the random number generator.

Vattulainen, Ala-Nissila, and Kankaala conducted further studies on these generators as well as generators of other types and found that their simulations often did not correspond to reality. Selke, Talapov, and Shchur (1993), however, found that the Park-Miller minimum standard generator performed much better than the R250 generator in a specific simulation of particle behavior. Vattulainen et al. (1995) study eight widely-available generators. For the tests in their study, the Park-Miller minimum standard generator, which is the default generator in the IMSL Library; R250; and G05FAF from the Nag Library were found to be acceptable. The generally inconclusive results lead us to repeat the advice given above to employ an ad hoc goodness-of-fit test for the specific application.
Tests of Random Number Generators Used in Parallel

A major problem in random number generation in parallel is the inability to synchronize the computations. This is easy to appreciate even in a simple case of acceptance/rejection; if one processor must do more rejections than another, that processor will get behind the other one. Cuccaro, Mascagni, and Pryor (1994) describe some approaches for testing parallel random number generators.

Second-Order Statistical Tests

For a given test, the significance level of the test statistic, under the null hypothesis is $U(0, 1)$ (recall our discussion of the inverse CDF technique). This fact gives rise to a second-order test: perform a selected goodness-of-fit test many times and then perform a goodness-of-fit test on the $p$-values. This fact gives rise to a third-order test, and so on, ad infinitum.

Interpretation of Statistical Tests

Because the empirical tests are statistical tests, the ordinary principles of hypothesis testing apply. The results of a battery of tests must be interpreted carefully. As Marsaglia (1995) states (sic):

> By all means, do not, as a Statistician might, think that a $p < .025$ or $p > .975$ means that the RNG has “failed the test at the .05 level”. Such $p$’s happen among the hundreds that DIEHARD produces, even with good RNG’s. So keep in mind that “$p$ happens”.

Fortunately, a “Statistician” understands statistical hypothesis testing better than some other testers of random number generators. If the null hypothesis is true, and if the statistical test is exact, we expect $|p| < 0.05$ 5% of the time.

In some cases, the kind of departure from randomness that a particular test addresses is obvious; in other cases, however, it is not so clear what kind of nonrandomness a given test may detect. In addition, the relationships of various tests to each other are not well understood. Banks (1998) approached this question in an interesting way. He applied a number of tests to a variety of generators and sequences from those generators and then performed a factor analysis on the results to study the extent of independence of the tests.

Anecdotal Evidence

The null hypothesis in the statistical tests of random number generators is that “the generator is performing properly”; therefore, failure to reject is not confirmation that the generator is a good one. There have been many statistical studies of the properties of given sets of random number generators. The tests reported in the literature have often been inconclusive, however.
Every Monte Carlo study in which there is additional information available about expected results provides evidence about the quality of the random number generators used. We call this anecdotal evidence. The first indication that the RANDU generator had problems came from anecdotal evidence in a simulation whose results did not correspond to theory (see Coldwell, 1974). Although statistical measures such as standard deviations are sometimes used in comparing simulation results with theoretical results, there is often no attempt to quantify the probability of a type I error.

**Quasirandom Numbers**

The statistical idea of stratification to reduce sampling variance can be extended; as the number of strata increases, the sample size in each stratum decreases. In the limit there is no randomness remaining in the sampling.

In our discussion of the Monte Carlo estimator in equation (2.2) on page 8, we indicated that the expressions for the expectation and variance were for samples from a random variable. For a deterministic sequence, instead of the expression of the estimator, we might consider its limit; and so we might take into account the discrepancy (equation (4.21), page 155).

For pseudorandom numbers generated by any of the methods we have discussed, the limit is taken over a cyclic finite set, and the supremum in the discrepancy is a maximum. Suppose, instead of the pseudorandom numbers resulting from the generators we have discussed, we explicitly address the problem of discrepancy. Then, instead of being concerned with $E(\hat{\theta})$, we might consider

$$\lim_{n \to \infty} \hat{\theta} = \lim_{n \to \infty} \left( b - a \right) \frac{\sum f(y_i)}{n}.$$ 

A moment’s pause, of course, tells us this kind of limit does not apply to the real world of computing, with its finite set of numbers. Nevertheless, it is useful to consider a deterministic sequence with low discrepancy. The objective is that any finite subsequence fill the space uniformly.

Several such sequences have been proposed, such as van der Corput sequences, Halton sequences (Halton, 1960); Faure sequences; Sobol’ sequences (Sobol’, 1967, 1976), program available in Bratley and Fox (1988); and Niederreiter sequences (Niederreiter, 1988), program available in Bratley, Fox, and Niederreiter (1994). These sequences are called *quasirandom sequences*. Whereas pseudorandom sequences or pseudorandom generators attempt to simulate randomness, quasirandom sequences are decidedly *not* random. The objective for a (finite) pseudorandom sequence is for it to “look like” a sequence of realizations of i.i.d. uniform random variables; but for a (finite) quasirandom sequence the objective is that it fill a unit hypercube as uniformly as possible.

Quasirandom sequences correspond to samples from a $U(0, 1)$ distribution. (Contrast this statement with the statement that “pseudorandom sequences *simulate random samples* from a $U(0, 1)$ distribution”.) The techniques of
Sections 4.2 can therefore be used to generate quasirandom sequences that correspond to samples from nonuniform distributions. For the methods that yield one nonuniform deviate from each uniform deviate, such as the inverse CDF method, everything is straightforward. For other methods that use multiple independent uniforms for each nonuniform deviate, the quasirandom sequence may be inappropriate. The quasirandom method does not simulate independence.

**Halton Sequences**

Halton sequences are generalizations of van der Corput sequences. A Halton sequence is formed by reversing the digits in the representation of some sequence of integers in a given base. Although this can be done somewhat arbitrarily, a straightforward way of forming a \(d\)-dimensional Halton sequence \(x_1, x_2, \ldots\), where \(x_i = (x_{i1}, x_{i2}, \ldots, x_{id})\) is first to choose \(d\) bases, \(b_1, b_2, \ldots, b_d\), perhaps the first \(d\) primes. The \(j^{th}\) base will be used to form the \(j^{th}\) component of each vector in the sequence. Then begin with some integer \(m\) and

1. choosing \(t_{mj}\) suitably large, represent \(m\) in each base:

   \[ m = \sum_{k=0}^{t_{mj}} a_{mk} b_j^k, \quad j = 1, \ldots, d, \]

2. form

   \[ x_{ij} = \sum_{k=0}^{t_{mj}} a_{mk} b_j^{k-t_{mj}-1}, \quad j = 1, \ldots, d, \]

3. set \(m = m + 1\) and repeat.

Suppose, for example, \(d = 3\), \(m = 15\), and we use the bases 2, 3, and 5. We form 15 = 1111\(_2\), 15 = 120\(_3\), and 15 = 30\(_5\), and deliver the first \(x\) as (0.1111\(_2\), 0.0213\(_3\), 0.035\(_5\)), or (0.937500, 0.259259, 0.120000).

The Halton sequences are acceptably uniform for lower dimensions, up to about 10. For higher dimensions, however, the quality of the Halton sequences degrades rapidly because the two-dimensional planes occur in cycles with decreasing periods. Generalized Halton sequences have been proposed and studied by Braaten and Weller (1979), Hellekalek (1984), Faure (1986), and Krommer and Ueberhuber (1994). The basic idea is to permute the \(a_{mk}\)'s in step 2. The Faure sequences are formed in this way.

Kocis and Whiten (1997) suggest a “leaped Halton sequence”. In this method the cycles of the Halton sequence are destroyed by using only every \(l^{th}\) Halton number, where \(l\) is a prime different from all of the bases \(b_1, b_2, \ldots, b_d\).
Sobol’ Sequences

A Sobol’ sequence is based on a set of “direction numbers”, \{v_i\}. The \( v_i \) are

\[
v_i = \frac{m_i}{2^i},
\]

where the \( m_i \) are odd positive integers less than \( 2^i \); and the \( v_i \) are chosen so that they satisfy a recurrence relation using the coefficients of a primitive polynomial in the Galois field \( \mathbb{F}(2) \),

\[
f(z) = z^p + c_1 z^{p-1} + \cdots + c_{p-1} z + c_p
\]

(compare equation (4.12), page 141). For \( i > p \), the recurrence relation is

\[
v_i = c_1 v_{i-1} \oplus c_2 v_{i-2} \oplus \cdots \oplus c_p v_{i-p} \oplus \lfloor v_{i-p}/2^p \rfloor,
\]

where \( \oplus \) denotes bitwise binary exclusive-or. An equivalent recurrence for the \( m_i \) is

\[
m_i = 2c_1 m_{i-1} \oplus 2^2 c_2 m_{i-2} \oplus \cdots \oplus 2^p c_p m_{i-p} \oplus m_{i-p}.
\]

As an example, consider the primitive polynomial (4.16) from page 142,

\[x^4 + x + 1\]

The corresponding recurrence is

\[m_i = 8m_{i-3} \oplus 16m_{i-4} \oplus m_{i-4}.
\]

If we start with \( m_1 = 1, m_2 = 1, m_3 = 3, \) and \( m_4 = 13 \), for example, we get

\[
m_5 = 8 \oplus 16 \oplus 1
\]

\[
= 01000(\text{binary}) \oplus 10000(\text{binary}) \oplus 00001(\text{binary})
\]

\[
= 11001(\text{binary})
\]

\[
= 25.
\]

The \( i^{th} \) number in the Sobol’ sequence is now formed as

\[x_i = b_1 v_1 \oplus b_2 v_2 \oplus b_3 v_3 \oplus \cdots,
\]

where \( \cdots b_3 b_2 b_1 \) is the binary representation of \( i \).

Antonov and Saleev (1979) show that equivalently the Sobol’ sequence can be formed as

\[x_i = g_1 v_1 \oplus g_2 v_2 \oplus g_3 v_3 \oplus \cdots, \quad (4.22)
\]

where \( \cdots g_3 g_2 g_1 \) is the binary representation of a particular Gray code evaluated at \( i \). (A Gray code is a function, \( G(i) \), on the nonnegative integers such that the binary representations of \( G(i) \) and \( G(i+1) \) differ in exactly one bit;
that is, with a Hamming distance of 1.) The binary representation of the Gray code used by Antonov and Saleev is
\[ \cdots g_3 g_2 g_1 = \cdots b_3 b_2 b_1 \oplus \cdots b_4 b_3 b_2. \]
(This is the most commonly used Gray code, which yields function values 0, 1, 3, 2, 6, 7, 5, 4, \ldots) The Sobol’ sequence from (4.22) can be generated recursively by
\[ x_i = x_{i-1} \oplus v_r, \]
where \( r \) is determined so that \( b_r \) is the rightmost zero bit in the binary representation of \( i - 1 \).

Bratley and Fox (1988) discuss criteria for starting values, \( m_1, m_2, \ldots \).
(The starting values used in the example with the primitive polynomial above satisfy those criteria.)

**Comparisons**

Empirical comparisons of various quasirandom sequences that have been reported in the literature are somewhat inconclusive. Sarkar and Prasad (1987) compare the performance of pseudorandom and quasirandom sequences in the solution of integral equations by Monte Carlo methods and find no difference in the performance of the two quasirandom sequences they studied: the Halton and Faure sequences. Fox (1986), on the other hand, finds the performance of Faure sequences to be better than that of Halton sequences. This is confirmed by Bratley and Fox (1988), who also find that the performance of the Sobol’ sequence is roughly the same as that of the Faure sequence. The empirical results reported in Bratley, Fox, and Niederreiter (1992) show inconclusive differences between Sobol’ sequences and Niederreiter sequences.

Although in some cases it is simple to make comparisons between the performance of pseudorandom and quasirandom sequences, there is a fundamental difference in the nature of the error bounds appropriate for Monte Carlo methods and for other numerical algorithms. This is a topic we will return to in Chapter 8. Hickernell (1995) compares a certain type of error bounds for quadrature using pseudorandom and quasirandom sequences, and shows that the quasirandom sequences resulted in smaller bounds for errors. Bouleau and Lépingle (1994), quoting Pagès and Xiao, give comparative sample sizes required by pseudorandom and Halton and Faure sequences to achieve the same precision for quadrature in various dimensions from 2 to 20. Precision, in this case, is approximated using the asymptotic convergence of the quadrature formulas.

**Variations**

Quasirandom sequences that cover domains other than hypercubes have also been studied. Beck and Chen (1987) review work on low-discrepancy sequences over various types of regions.
Combination generators (see Section 4.1.1, page 148) for quasirandom sequences can also be generated. It generally would not be useful to combine another sequence with a quasirandom sequence, except to shuffle the quasirandom sequence. The shuffling generator could be either a quasirandom generator or a pseudorandom generator, but a pseudorandom generator would be more likely to be effective. When a combination generator is composed of one or more quasirandom generators and one or more pseudorandom generators, it is called a hybrid generator. Braaten and Weller (1979) describe a hybrid method in which a pseudorandom generator is used to scramble a quasirandom sequence.

Some Examples of Applications

There are several applications of Monte Carlo reported in the literature that use quasirandom numbers. For example, Shaw (1988) uses quasirandom sequences instead of the usual pseudorandom sequences for evaluating integrals arising in Bayesian inference. Do (1991) uses quasirandom sequences in a Monte Carlo bootstrap. Joy, Boyle, and Tan (1996) empirically comparing the use of Faure sequences with pseudorandom sequences in valuing financial options of various types, find that the quasirandom sequences had better convergence rates in that application. Also in an application in financial derivatives, Papageorgiou and Traub (1996) compared the performance of a generalized Faure sequence with a Sobol’ sequence. They concluded that the Faure sequence was superior in that problem.

Computations

Halton sequences are easy to generate (Exercise ?? asks you to write a program to do so). Fox (1986) gives a program for Faure sequences; Bratley and Fox (1988) give a program for Sobol’ sequences (using Gray codes, as mentioned above); and Bratley, Fox, and Niederreiter (1994) give a program for Niederreiter sequences.

Summary

There are many issues to consider in assessing the quality of a random number generator. It is easy to state what is desired for the basic uniform generator. The first desideratum is obvious:

- the output should be essentially indistinguishable from a sample from a uniform distribution.

This has nothing to do with how the generator is implemented, whether it is in a computer program or in some other form. It does, however, have implications for the period of a deterministic, cyclic pseudorandom number generator.
implemented in a computer program. The period should be very long. The specific meaning of “long” in this context depends on the size of problems in which random number generators find application. The size of the problems grows from year to year, and so our criterion for “long” changes.

Another desideratum for a random number generator is not so obvious:

- a sequence should be reproducible on (almost) any computer.

This quality, of course, means that a pseudorandom number generator is to be preferred to a random one.

There are specific desirable qualities that arise from the use of the random number generator in the computer:

- neither 0 nor 1 should occur in the output
- the generator should be efficient in use of computing resources
- the generator should be easy to use.

The last two qualities listed above have additional specific implications:

- the generator should not require user input with qualities that are difficult to assess
- it should be possible to generate long substreams that do not overlap.

Although, as we have suggested, before using a random number generator, it is wise to apply ad hoc goodness-of-fit tests that may uncover problems in that application, it is desirable that the quality of the output of the generator not be dependent on specific transformations or on a specific seed.

The last quality listed above makes the generator more useful in parallel implementations.

Finally, an additional concern is the quality of the computer program:

- the computer program must faithfully implement the generator.

Bugs in computer programs are all too common.

4.2 Transformations of Uniform Deviates: General Methods

Sampling of random variates from a nonuniform distribution is usually done by applying a transformation to uniform variates. Each realization of the nonuniform random variable might be obtained from a single uniform variate or from a sequence of uniforms. Some methods that use a sequence of uniforms require that the sequence be independent; other methods use a random walk sequence, a Markov chain.

For some distributions there may be many choices of algorithms for generation of random numbers. The algorithms differ in speed, in accuracy, in storage requirements, and in complexity of coding. Some of the faster methods are approximate; but given the current cost of computing, the speed-up
After accuracy, the next most important criterion is speed. The speed of a random number generation algorithm has two aspects: the set-up time and the generation time. In most cases the generation time is the more important component to optimize. Whenever the set-up time is significant, the computer program can preserve the variables initialized, so that if the function is called again with the same parameters, the set-up step can be bypassed. In a case of relatively expensive set-up overhead, a software system may provide a second function for the same distribution with less set-up time.

The other two criteria mentioned above, storage requirements and complexity of coding, are generally of very little concern in selecting algorithms for production random number generators.

Another important issue is whether the algorithm can be implemented so as to be portable, as discussed in Section 4.4.

The methods discussed in this section are “universal” in the sense that they apply to many different distributions. Some of these methods are better than others for a particular distribution or for a particular range of the distribution. These techniques are used, either singly or in combination, for particular distributions. We discuss some of these methods in Section 4.3, beginning on page 193.

Some of these methods in this section, especially those that involve inverting a function, apply directly only to univariate random variables, whereas other methods apply immediately to multivariate random variables.

### 4.2.1 Inverse CDF Method

**Continuous Distributions**

If $X$ is a scalar random variable with a continuous cumulative distribution function (CDF) $P_X$, then the random variable

$$U = P_X(X)$$

has a U(0, 1) distribution. (This is easy to show; you are asked to do that in Exercise 4.15, page 224.) This fact provides a very simple relationship with a uniform random variable $U$ and a random variable $X$ with distribution function $P$:
Use of this straightforward transformation is called the inverse CDF technique. The reason it works can be seen in Figure 4.9; over a range for which the derivative of the CDF (the density) is large, there is more probability of realizing a uniform deviate.

**Figure 4.9.** The Inverse CDF Method to Convert a Uniform Random Number to a Number from a Continuous Distribution

The inverse CDF relationship exists between any two continuous (nonsingular) random variables. If \( X \) is a continuous random variable with CDF \( P_X \) and \( Y \) is a continuous random variable with CDF \( P_Y \), then

\[
X = P_X^{-1}(P_Y(Y))
\]

over the ranges of positive support. Use of this kind of relationship is a matching of “scores”, that is, of percentile points, of one distribution with those of another distribution. In addition to the uniform, as above, this kind of transformation is sometimes used with the normal distribution.

Whenever the inverse of the distribution function is easy to compute, the inverse CDF method is a good one. It also has the advantage that basic relationships among a set of uniform deviates (such as order relationships) may result in similar relationships among the set of deviates from the other distribution.

Because it is relatively difficult to compute the inverse of some distribution functions of interest, however, the inverse CDF method is not as commonly
used as its simplicity might suggest. Even when the inverse $P^{-1}$ exists in closed form, evaluating it directly may be much slower than use of some alternative method for sampling random numbers. On the other hand, in some cases when $P^{-1}$ does not exist in closed form, use of the inverse CDF method by solving the equation

$$P(x) - u = 0$$

may be better than the use of any other method. (Methods for solving this equation are discussed in Chapter 7.)

**Discrete Distributions**

The inverse CDF method also applies to discrete distributions, but of course we cannot take the inverse of the distribution function. Suppose the discrete random variable $X$ has mass points

$$m_1 < m_2 < m_3 < \ldots$$

with probabilities

$$p_1, p_2, p_3, \ldots,$$

and with the distribution function

$$P(x) = \sum_{i \geq m_i \leq x} p_i.$$ 

To use the inverse CDF method for this distribution, we first generate a realization $u$ of the uniform random variable $U$. We then deliver the realization of the target distribution as $x$, where $x$ satisfies the relationship

$$P(x(-)) < u \leq P(x). \quad (4.24)$$

This is illustrated in Figure 4.10.

Without loss of generality, we often assume that the mass points of a discrete distribution are the integers $1, 2, 3, \ldots$. The special case in which there are $k$ mass points and they all have equal probability is called the discrete uniform distribution, and the use of the inverse CDF method is particularly simple: the value is $[uk]$.

An example of a common and very simple application of the inverse CDF technique is for generating a random deviate from a Bernoulli distribution with parameter $\pi$, as in Algorithm 4.2. The probability function for the Bernoulli distribution with parameter $\pi$ is

$$p(x) = \pi^x (1-\pi)^{1-x}, \quad \text{for } x = 0, 1, \quad (4.25)$$

where $0 < \pi < 1$.

**Algorithm 4.2 Generating a Bernoulli by the Inverse CDF**
4.2 Transformations of Uniform Deviates: General Methods

Figure 4.10. The Inverse CDF Method to Convert a Uniform Random Number to a Number from a Discrete Distribution

1. Generate $u$ from a $U(0, 1)$ distribution.
2. If $u < \pi$, then
   2.a. deliver 0;
   otherwise
   2.b. deliver 1.

Multivariate Distributions

The inverse CDF method does not apply to a multivariate distribution, although marginal and conditional univariate distributions can be used in an inverse CDF method to generate multivariate random variates. If the CDF of the multivariate random variable $(X_1, X_2, \ldots, X_d)$ is decomposed as

$$P_{X_1, X_2, \ldots, X_d}(x_1, x_2, \ldots, x_d) = P_{X_1}(x_1)P_{X_2|X_1}(x_2|x_1) \cdots P_{X_d|X_1, X_2, \ldots, X_{d-1}}(x_d|x_1, x_2, \ldots, x_{d-1}),$$

and the functions are invertible, the inverse CDF method is applied sequentially, using independent realizations of a $U(0, 1)$ random variable, $u_1, u_2, \ldots, u_d$:

$$x_1 = P_{X_1}^{-1}(u_1)$$
$$x_2 = P_{X_2|X_1}^{-1}(u_2)$$
$$\ldots$$
$$x_d = P_{X_d|X_1, X_2, \ldots, X_{d-1}}^{-1}(u_d).$$
The modifications of the inverse CDF for discrete random variables described above can be applied if necessary.

**Transformations That Use More Than One Uniform Deviate**

Many of the methods discussed below require the use of more than one uniform deviate. For such methods we must be careful to avoid any deleterious effects of short-range correlations in the underlying generator. An example in the use of a congruential generator,

\[ x_i \equiv ax_{i-1} \mod m, \]

is the set of possible values for \( x_i \) when \( x_{i-1} \) is extremely small. These values are just \( ax_{i-1} \), with no modular reduction. A small value of \( x_{i-1} \) may correspond to some extreme intermediate value. When \( x_i \) is used to complete the transformation to the variate of interest, it may happen that the extreme values of that variate do not cover their appropriate range.

As a simple example, consider a method for generating a variate from a double exponential distribution. One way to do this is to use one uniform variate to generate an exponential variate (using one of the methods we discuss below), and then to use a second uniform variate to decide whether to change the sign of the exponential variate (with probability \( 1/2 \)). Some methods for generating an exponential variate yield an extremely large value if the underlying uniform is extremely small. If the next uniform from the basic generator is used to determine whether to change the sign, it may happen that all of the extreme double exponentials have the same sign.

Many such problems arise because of a poor uniform generator; a particular culprit is a multiplicative congruential generator with a small multiplier. Use of a high-quality uniform generator generally solves the problem. A more conservative approach may be to use a different uniform generator for each uniform used in the generation of a single deviate. For this to be effective, each generator must be of high quality of course.

**4.2.2 Mixtures of Distributions**

It is often useful to break up the range of the distribution of interest, using one density over one subrange and another density over another subrange. More generally, we may represent the distribution of interest as a mixture distribution. A mixture distribution is composed of proportions of other distributions. Suppose, for example, that the probability density or probability function of the random variable of interest, \( p \), can be represented as

\[ p(x) = w_1p_1(x) + w_2p_2(x), \]

(4.26)

where \( p_1 \) and \( p_2 \) are density functions or probability functions of random variables, the union of whose support is the support of the random variable of interest. Obviously,
4.2 Transformations of Uniform Deviates: General Methods

The random variable of interest has a mixture distribution.

In some cases for \( x \) in some range, either \( p_1(x) \) or \( p_2(x) \) is zero, even though \( p(x) \) is nonzero in that range. In such cases the mixture is a stratification.

To generate a random deviate from a mixture distribution, first use a single uniform to select the component distribution, and then generate a deviate from it. The mixture can consist of any number of terms. To generate a sample of \( n \) random deviates from a mixture distribution of \( d \) distributions, consider the proportions to be the parameters of a \( d \)-variate multinomial distribution. The first step is to generate a single multinomial deviate, then generate the required number of deviates from each of the component distributions.

Any decomposition of \( p \) into the sum of nonnegative integrable functions yields the decomposition in equation (4.26). The nonnegative \( w_i \) are chosen to sum to 1.

For example, suppose a distribution has density \( p(x) \), and for some constant \( c \), \( p(x) \geq c \) over \((a, b)\). Then the distribution can be decomposed into a mixture of a uniform over \((a, b)\) with proportion \( c(b-a) \) and some leftover part, say \( g(x) \). Now \( g(x)/(1 - c(b-a)) \) is a probability density function. To generate a deviate from \( p \):

- with probability \( c(b-a) \), generate a uniform \((a, b)\);
- otherwise generate from the density \( \frac{1}{1 - c(b-a)} g(x) \).

If \( c(b-a) \) is close to 1, we will generate from the uniform distribution most of the time, so even if it is difficult to generate from \( g(x)/(1 - c(b-a)) \), this decomposition of the original distribution may be useful.

Another way of forming a mixture distribution is to consider a density similar to (4.26) that is a conditional density,

\[
p(x|y) = yp_1(x) + (1 - y)p_2(x),
\]

where \( y \) is the realization of a Bernoulli random variable, \( Y \). If \( Y \) takes a value of 0 with probability \( w_1/(w_1 + w_2) \), then the density in (4.26) is the marginal density. This conditional distribution yields

\[
p_X(x) = \int p_{X,Y}(x, y) \, dy
= \sum_y p_{X|Y=y} \Pr(Y = y)
= w_1p_1(x) + w_2p_2(x),
\]
as in equation (4.26).

More generally, for any random variable \( X \) whose distribution is parameterized by \( \theta \), we can think of the parameter as being the realization of a

\[
w_1 + w_2 = 1.
\]
acceptance/rejection method for random number generation
majorizing density
instrumental density

random variable $\Theta$. Some common distributions result from mixing other distributions; for example, if the gamma distribution is used to generate the parameter in a Poisson distribution, a negative binomial distribution is formed. Mixture distributions are often useful in their own right; for example, the beta-binomial distribution, which can be used to model overdispersion (see Ahn and Chen, 1995).

4.2.3 Acceptance/Rejection Methods

For generating realizations of a random variable $X$, the *acceptance/rejection method* makes use of realizations of another random variable $Y$ whose probability density $g_Y$ is similar to the probability density of $X$, $p_X$. The random variable $Y$ is chosen so that we can easily generate realizations of it and so that its density $g_Y$ can be scaled to majorize $p_X$, using some constant $c$; that is, so that $c g_Y(x) \geq p_X(x)$ for all $x$. The density $g_Y$ is called the majorizing density and $c g_Y$ is called the majorizing function. The majorizing density is also sometimes called the “instrumental density”.

Unlike the inverse CDF method, the acceptance/rejection applies immediately to multivariate random variables.

**Algorithm 4.3 The Acceptance/Rejection Method to Convert Uniform Random Numbers**

1. Generate $y$ from the distribution with density function $g_Y$.
2. Generate $u$ from a U(0,1) distribution.
3. If $u \leq p_X(y)/cg_Y(y)$, then
   3.a. take $y$ as the desired realization;
   otherwise
   3.b. return to step 1.

It is easy to see that the random number delivered by Algorithm 4.3 has a density $p_X$. (In Exercise 4.16, page 224, you are asked to write the formal proof.)

Figure 4.11 illustrates the functions used in the acceptance/rejection method. (Actually, Figure 4.11 represents a specific case, namely, the beta distribution with parameters 3 and 2. In Exercise 4.17, page 224, you are asked to write a program implementing the acceptance/rejection method with the majorizing density shown.)

The acceptance/rejection method can be visualized as choosing a subsequence from a sequence of independently and identically distributed (i.i.d.) realizations from the distribution with density $g_Y$ in such a way the subsequence has density $p_X$, as shown in Figure 4.12.

Because of the comparison in step 3, there is a chance that an acceptance/rejection method may yield different streams on different computer systems. Even if the computations are carried out correctly, the program is inherently nonportable, because if a comparison on one system at a given
4.2 Transformations of Uniform Deviates: General Methods

portability of software

Figure 4.11. The Acceptance/Rejection Method to Convert Uniform Random Numbers

precision results in acceptance, and the comparison on another system results in rejection, the two output streams will be different. At best, the streams will be the same except for a few differences; at worst, however, because of how the output is used, the results will be different beginning at the point at which the acceptance/rejection decision is different. If the decision results in the generation of another random number (as in Algorithm 4.5, on page 180), the two output streams can become completely different.

\[
\begin{array}{c|cccccc}
\text{i.i.d. from } g_Y & y_i & y_{i+1} & y_{i+2} & y_{i+3} & \cdots & y_{i+k} & \cdots \\
\text{accept?} & \text{no} & \text{yes} & \text{no} & \text{yes} & \cdots & \text{yes} & \cdots \\
\text{i.i.d. from } p_X & x_j & x_{j+1} & \cdots & \text{yes} & \cdots
\end{array}
\]

Figure 4.12. Acceptance/Rejection

It is easy to see that the random variable corresponding to the number of passes through the steps of Algorithm 4.3 until the desired variate is delivered has a geometric distribution. If we ignore the time required to generate \( y \) from the dominating density \( g_Y \), the closer \( cg_Y(x) \) is to \( p_X(x) \), that is, the closer \( c \) is to its lower bound of 1, the faster the acceptance/rejection algorithm will be. When \( c \) is close to 1, \( p_X(x)/g_Y(x) \) is nearly constant. Selection of a majorizing function involves the principles of function approximation, such as we discuss in Section 8.2, page 458, with the added constraint that the approx-
imating function be a probability density from which it is easy to generate random variates. Often, \( g_Y \) is chosen to be a very simple density, such as a uniform or a triangular density. When the dominating density is uniform, the acceptance/rejection method is similar to the “hit-or-miss” method described in Exercise 2.1, page 21.

(In the following, we generally dispense with the subscripts on the densities, except in some cases where we want to emphasize the random variable. It is important to remember, however, that the notation used for the argument of a function does not identify the function, despite the use of this convention by some statisticians.)

Although the acceptance/rejection method can be used for multivariate random variables, in that case the majorizing distribution must also be multivariate. For higher dimensions, another problem is the relationship of the rejection region to the acceptance region. Consider the use of a normal with mean 0 and variance 2 as a majorizing density for a normal with mean 0 and variance of 1, as shown in Figure 4.13. A majorizing density like this whose shape more closely approximates that of the target density is more efficient. (An obvious problem in this case is that if we could generate deviates from the \( N(0, 2) \) distribution we could generate ones from the \( N(0, 1) \) distribution.) In

**Figure 4.13.** Normal \((0, 1)\) Density with a Normal \((0, 2)\) Majorizing Density grr40a

the one-dimensional case, as shown in Figure 4.13, the acceptance region is the area under the lower curve, and the rejection region is the thin shell between the two curves. In higher dimensions, even a thin shell contains most of the
4.2 Transformations of Uniform Deviates: General Methods

volume, so the rejection proportion would be high. See Section ??, page ??, and Exercise ??, page ??.

There are many variations of the acceptance/rejection method. The method described here uses a sequence of i.i.d. variates from the majorizing density. It is also possible to use a sequence from a conditional majorizing density. A method using a nonindependent sequence is called a Metropolis method (and there are variations of these, with their own names, as we see below).

Acceptance/Rejection for Discrete Distributions

There are various ways that acceptance/rejection can be used for discrete distributions. One advantage of these methods is that they can be easily adapted to changes in the distribution. Rajasekaran and Ross (1993) consider the discrete random variable \( X_s \) such that

\[
\Pr(X_s = x_i) = p_{si} = \frac{a_{si}}{a_{s1} + a_{s2} + \cdots + a_{sk}}, \quad i = 1, \ldots, k.
\]

(If \( \sum_{i=1}^{k} a_{si} = 1 \), the numerator \( a_{si} \) is the ordinary probability \( p_{si} \) at the mass point \( i \).) Suppose that there exists an \( a^*_s \) such that \( a^*_s \leq a_{si} \) for \( s = 1, 2, \ldots \), and \( b > 0 \) such that \( \sum_{i=1}^{k} a_{si} \geq b \) for \( s = 1, 2, \ldots \). Let

\[
a^* = \max\{a^*_s\},
\]

and let

\[
P_{si} = \frac{a_{si}}{a^*} \quad \text{for } i = 1, \ldots, k.
\]

The generation method for \( X_s \) is shown in Algorithm 4.4.

**Algorithm 4.4 Acceptance/Rejection Method for Discrete Distributions**

1. Generate \( u \) from a U(0, 1) distribution, and let \( i = \lceil ku \rceil \).
2. Let \( r = i - ku \).
3. If \( r \leq P_{si} \), then
   3.a. take \( i \) as the desired realization;
   otherwise
   3.b. return to step 1.

Suppose for the random variable \( X_{s+1} \), \( p_{s+1,i} \neq p_{si} \) for some \( i \). (Of course, if this is the case for mass point \( i \), it is also necessarily the case for some other mass point.) For each mass point whose probability changes, reset \( P_{s+1,i} \) to \( a_{s+1,i}/a^* \) and continue with Algorithm 4.4.

Rajasekaran and Ross (1993) also gave two other acceptance/rejection type algorithms for discrete distributions that are particularly efficient for use with distributions that may be changing. The other algorithms require slightly more preprocessing time, but yield faster generation times than Algorithm 4.4.
Other Applications of Acceptance/Rejection

The acceptance/rejection method can often be used to evaluate an elementary function at a random point. Suppose, for example, we wish to evaluate \(\tan(\pi U)\) for \(U\) distributed as \(U(-0.5, 0.5)\). A realization of \(\tan(\pi U)\) can be simulated by generating \(u_1\) and \(u_2\) independently from \(U(-1, 1)\), checking if \(u_1^2 + u_2^2 \leq 1\), and if so, delivering \(u_1/u_2\) as \(\tan(\pi u)\). (To see this, think of \(u_1\) and \(u_2\) as sine and cosine values.) Von Neumann (1951) gives an acceptance/rejection method for generating sines and cosines of random angles. An example of evaluating a logarithm can be constructed by use of the equivalence of an inverse CDF method and an acceptance/rejection method for sampling an exponential random deviate. (The methods are equivalent in a stochastic sense; they are both valid, but they will not yield the same stream of deviates.) Generally, if reasonable numerical software is available for evaluating special functions, it should be used rather than using Monte Carlo methods to estimate the function values.

Acceptance/Rejection for Multivariate Distributions

The acceptance/rejection method is one of the most widely applicable methods for random number generation. It is used in many different forms, often in combination with other methods. It is clear from the description of the algorithm that the acceptance/rejection method applies equally to multivariate distributions. (The uniform random number is still univariate, of course.)

As we have mentioned however, for higher dimensions, the rejection proportion may be high; and thus, the efficiency of the acceptance/rejection method may be low.

4.2.4 Other Methods

Mixtures and Acceptance Methods

In practice, in acceptance/rejection methods the density of interest \(p\) and/or the majorizing density are decomposed into mixtures.

Suppose, as above, the density of interest, \(p\), may be written as

\[
p(x) = w_1 p_1(x) + w_2 p_2(x),
\]

and suppose there is a density \(g\) that majorizes \(w_1 p_1\), that is, \(g(x) \geq w_1 p_1(x)\) for all \(x\).

Kronmal and Peterson (1981, 1984) consider this case and propose the following algorithm, which they call the acceptance/complement method.

Algorithm 4.5 The Acceptance/Complement Method to Convert Uniform Random Numbers

1. Generate \(y\) from the distribution with density function \(g\).
2. Generate \( u \) from a \( U(0, 1) \) distribution.
3. If \( u > \frac{w_1 p_1(y)}{g(y)} \), then generate \( y \) from the density \( p_2 \).
4. Take \( y \) as the desired realization.

**Use of Stationary Distributions of Markov Chains**

If the density of interest, \( p \), is the density of the stationary distribution of a Markov chain, correlated samples from the distribution can be generated by simulating the Markov chain. (A *Markov chain* is a sequence of random variables, \( X_1, X_2, \ldots \), such that the distribution of \( X_{t+1} \) given \( X_t \) is independent of \( X_{t-1}, X_{t-2}, \ldots \). A sequence of realizations of such random variables is also called a Markov chain. An aperiodic, irreducible, positive recurrent Markov chain is associated with a *stationary distribution* or *invariant distribution*, which is the limiting distribution of the chain. See Meyn and Tweedie, 1993, for relevant definitions and discussions of properties of Markov chains.) An algorithm based on a stationary distribution of a Markov chain is an *iterative method* because a sequence of operations must be performed until they converge.

A Markov chain is the basis for several schemes for generating random numbers. The interest is not in the sequence of the Markov chain itself. The elements of the chain are accepted or rejected in such a way as to form a different chain whose stationary distribution is the distribution of interest.

Following engineering terminology for sampling sequences, the techniques based on these chains are generally called “samplers”. The static sample, and not the sequence, is what is used. The objective in the Markov chain samplers is to generate a sequence of autocorrelated points with a given stationary distribution.

For a distribution with density \( p \), the Metropolis algorithm, introduced by Metropolis et al. (1953), generates a random walk and performs an acceptance/rejection based on \( p \) evaluated at successive steps in the walk. In the simplest version, the walk moves from the point \( y_i \) to a candidate point \( y_{i+1} = y_i + s \), where \( s \) is a realization from \( U(-a, a) \), and accepts \( y_{i+1} \) if

\[
\frac{p(y_{i+1})}{p(y_i)} \geq u,
\]

where \( u \) is an independent realization from \( U(0, 1) \). If the new point is at least as probable, that is, if \( p(y_{i+1}) \geq p(y_i) \), the condition (4.27) implies acceptance without the need to generate \( u \). The random walk of Metropolis et al. is the basic algorithm of *simulated annealing*, which is currently widely used in optimization problems. It is also used in simulations of models in statistical mechanics (see Section 2.1.6).

If the range of the distribution is finite, the random walk is not allowed to go outside of the range. Consider, for example, the von Mises distribution, with density,


\[ p(x) = \frac{1}{2\pi I_0(c)} e^{c\cos(x)}, \quad \text{for } -\pi \leq x \leq \pi, \quad (4.28) \]

where \( I_0 \) is the modified Bessel function of the first kind and of order zero. Notice, however, that it is not necessary to know this normalizing constant, because it is canceled in the ratio. The fact that all we need is a nonnegative function that is proportional to the density of interest is an important property of this method.

If \( c = 3 \), after a quick inspection of the amount of fluctuation in \( p \), we may choose \( a = 1 \). The following Matlab statements implement the Metropolis algorithm to generate \( n - 1 \) deviates from the von Mises distribution.

```matlab
while i < n
    yip1 = yi + 2*a*rand - 1;
    if yip1 < pi & yip1 > -pi
        i = i+1;
        if exp(c*(cos(yip1)-cos(yi))) > rand
            yi = yip1;
        else
            yi = x(i-1);
        end
    end
    x(i) = yip1;
end
```

The output for \( n = 1000 \) and a starting value of \( y_0 = 1 \) is shown in Figure 4.14. The output is a Markov chain. A histogram, which is not affected by the sequence of the output in a large sample, is shown in Figure 4.15.

The von Mises distribution is an easy one to simulate by the Metropolis algorithm. This distribution is often used by physicists in simulations of lattice gauge and spin models, and the Metropolis method is widely used in these simulations. Notice the simplicity of the algorithm: we did not need to determine a majorizing density, nor even evaluate the Bessel function that is the normalizing constant for the von Mises density.

The Markov chain samplers require a “burn-in” period, that is, a number of iterations before the stationary distribution is achieved. In practice, the variates generated during the burn-in period are discarded. The number of iterations needed varies with the distribution, and can be quite large, sometimes several hundred. The von Mises example shown in Figure 4.14 is unusual; no burn-in is required. In general, convergence is much quicker for univariate distributions with finite ranges such as this one.

It is important to remember what convergence means; it does not mean that the sequence is independent from the point of convergence forward. The deviates are still from a Markov chain.

The Metropolis acceptance/rejection sequence is illustrated in Figure 4.16. Compare this with the acceptance/rejection method based on independent variables, as illustrated in Figure ??.
Hastings (1970) developed an algorithm that uses a more general chain for the acceptance/rejection step. The Metropolis-Hastings sampler to generate deviates from a distribution with density \( p_X \) uses deviates from a Markov chain with density \( g_{Y_{t+1}|Y_t} \). The method is shown in Algorithm 4.6. The conditional density \( g_{Y_{t+1}|Y_t} \) is chosen so that it is easy to generate deviates from it.

**Algorithm 4.6 Metropolis-Hastings Algorithm**

0. Set \( k = 0 \).
1. Choose \( x^{(k)} \) in the range of \( p_X \). (The choice can be arbitrary.)
2. Generate \( y \) from the density \( g_{Y_{t+1}|Y_t}(y|x^{(k)}) \).
3. Set \( r \):
   \[
   r = p_X(y) \frac{g_{Y_{t+1}|Y_t}(x^{(k)}|y)}{p_X(x^{(k)})g_{Y_{t+1}|Y_t}(y|x^{(k)})}
   \]
4. If \( r \geq 1 \), then
   4.a. set \( x^{(k+1)} = y \);
   otherwise
   4.b. generate \( u \) from \( U(0,1) \) and
      if \( u < r \), then
      4.b.i. set \( x^{(k+1)} = y \),
      otherwise
      4.b.ii. set \( x^{(k+1)} = x^{(k)} \).
Figure 4.15. Histogram of the Output from the Metropolis Algorithm for a Von Mises Distribution grr044b

5. If convergence has occurred, then
   5.a. deliver $x = x^{(k+1)}$;
   otherwise
   5.b. set $k = k + 1$, and go to step 2.

Compare Algorithm 4.6 with the basic acceptance/rejection method in Algorithm 4.3, page 176. The analogue to the majorizing function in the Metropolis-Hastings algorithm is the reference function

$$\frac{g_{Y_{t+1}|Y_t}(x|y)}{p_X(x)g_{Y_{t+1}|Y_t}(y|x)}$$

In Algorithm 4.6, $r$ is called the “Hastings ratio”, and step 4 is called the “Metropolis rejection”. The conditional density, $g_{Y_{t+1}|Y_t}(\cdot|\cdot)$ is called the “proposal density” or the “candidate generating density”. Notice that because the reference function contains $p_X$ as a factor, we only need to know $p_X$ to within

<table>
<thead>
<tr>
<th>random walk</th>
<th>accept?</th>
<th>i.i.d. from $p_X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_i$</td>
<td>no</td>
<td>$x_i$</td>
</tr>
<tr>
<td>$y_{i+1} = y_i + s_{i+1}$</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>$y_{i+2} = y_{i+3} + s_{i+3}$</td>
<td>yes</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 4.16. Metropolis Acceptance/Rejection
a constant of proportionality. As we have mentioned already, this is an important characteristic of the Metropolis algorithms.

As with the acceptance/rejection methods with independent sequences, the acceptance/rejection methods based on Markov chains apply immediately to multivariate random variables.

We can see that this algorithm delivers realizations from the density $p_X$ by using the same method suggested in Exercise 4.16 (page 224), that is, determine the CDF and differentiate. The CDF is the probability-weighted sum of the two components corresponding to whether the chain moved or not. In the case in which the chain does move, that is, in the case of acceptance, for the random variable $Z$ whose realization is $y$ in Algorithm 4.6, we have

$$
Pr(Z \leq x) = \Pr \left( Y \leq x \mid U \leq p(Y) \frac{g(x_i | Y)}{p(x_i) g(Y | x_i)} \right)
$$

$$
= \frac{\int_{-\infty}^{x} \int_{0}^{p(t)g(x_i | t)/(p(x_i) g(t | x_i))} g(t | x_i) \, ds \, dt}{\int_{-\infty}^{\infty} \int_{0}^{p(t)g(x_i | t)/(p(x_i) g(t | x_i))} g(t | x_i) \, ds \, dt}
$$

$$
= \int_{-\infty}^{x} p_X(t) \, dt.
$$

We can illustrate the use of the Metropolis-Hastings algorithm using a Markov chain in which the density of $X_{t+1}$ is normal with a mean of $X_t$ and a variance of $\sigma^2$. Let us use this density to generate a sample from a standard normal distribution (that is, a normal with a mean of 0 and a variance of 1). We start with a $x_0$, chosen arbitrarily. We take logs and cancel terms in the expression for $r$ in Algorithm 4.6. The following simple Matlab statements generate the sample (and plot the sequence):

```matlab
x(1) = x0;
while i < n
    i = i + 1;
    yip1 = yi + sigma*randn;
    lr2 = yi^2 - yip1^2;
    if lr2 > 0
        yi = yip1;
    else
        u = rand;
        if lr2 > log(u)*2
            yi = yip1;
        else
            yi = x(i-1);
        end
    end
    x(i) = yi;
end
plot (x)
```
The sequential output for $n = 1000$, a starting value of $x_0 = 10 \ (y_1 = 10)$, and a variance of $\sigma^2 = 9 \ (\text{sigma} = 3)$ is shown in Figure 4.17. Notice that the values descend very quickly from the starting value, which would be a very unusual realization of a standard normal. In practice, we generally cannot expect such a short burn-in period. Notice also in Figure 4.17 the horizontal line segments where the underlying Markov chain did not advance.

There are several variations of the basic Metropolis-Hastings algorithm. See Bhanot (1988) and Chib and Greenberg (1995) for descriptions of modifications and generalizations. Also see Section 4.2.5 for two related methods, Gibbs sampling and hit-and-run sampling. (Because those methods are particularly useful in multivariate simulation, we defer the discussion to that section.) These algorithms are used extensively in a class of methods called Markov chain Monte Carlo, or MCMC.

Markov chain Monte Carlo has become one of the most important tools in statistics in recent years. Its applications pervade Bayesian analysis, as well as many Monte Carlo procedures in the frequentist approach to statistical analysis. See Gilks, Richardson, and Spiegelhalter (1996) for several examples.

Whenever a correlated sequence such as a Markov chain is used, variance estimation must be performed with some care. In the more common cases of positive autocorrelation, the ordinary variance estimators are negatively biased. The method of batch means or some other method that attempts to
4.2 Transformations of Uniform Deviates: General Methods

A modification is called an independence sampler, which is a Metropolis-Hasting sampler whose proposal density does not depend on $Y_t$: $g_{Y_{t+1}|Y_t}(\cdot|\cdot) = g_{Y_{t+1}}(\cdot)$. For this type of proposal density, it is more critical that $g_{Y_{t+1}}(\cdot)$ approximates $p_X(\cdot)$ fairly well, and that it can be scaled to majorize $p_X(\cdot)$ in the tails. Liu (1996) and Roberts (1996) discuss some of the properties of the independence sampler and its relationship to other Metropolis-Hastings methods.

Some of the most important issues in MCMC concern the rate of convergence, that is, the length of the burn-in, and the frequency with which the chain advances. In most applications of simulation, such as studies of waiting times in queues, there is more interest in transient behavior than in stationary behavior. This is not the case in random number generation using an iterative method. For general use in random number generation, the stationary distribution is the only thing of interest. (We often use the terms “Monte Carlo” and “simulation” rather synonymously; stationarity and transience, however, are often the key distinction between Monte Carlo applications and simulation applications. In simulation in practice, the interest is rarely in the stationary behavior, but it is in these Monte Carlo applications.)

The issue of convergence is more difficult to address in multivariate distributions. It is for multivariate distributions, however, that the MCMC method is most useful. This is because the Metropolis-Hastings algorithm does not require knowledge of the normalizing constants, and the computation of a normalizing constant may be more difficult for multivariate distributions.

To observe the performance of MCMC in higher dimensions, consider an example similar to that shown in Figure 4.17, except for a multivariate normal distribution instead of a univariate one. We use a $d$-dimensional normal with a mean vector $x_t$ and a variance-covariance matrix $\Sigma$ to generate $x_{t+1}$ for use in the Metropolis-Hastings method of Algorithm 4.6. Taking $d = 3$,

$$\Sigma = \begin{bmatrix} 9 & 0 & 0 \\ 0 & 9 & 0 \\ 0 & 0 & 9 \end{bmatrix},$$

and starting with $x_0 = (10, 10, 10)$, the first 1000 values of the first element (which should be a realization from a standard univariate normal) are shown in Figure 4.18.

The example shown was generated by the following simple Matlab statements:

```matlab
x(1,:) = x0;
while i < n
    i = i + 1;
    yip1 = yi + (half*randn(size(yi))
    lr2 = yi*yi - yip1*yip1;
```


Figure 4.18. Sequential Output of $x_1$ from a Trivariate Standard Normal Distribution Using a Markov Chain, $N(X, \Sigma)$

```matlab
if 1r2 > 0
    yi = yip1;
else
    u = rand;
    if 1r2 > log(u)*2
        yi = yip1;
    else
        yi = x(i-1,:);
    end
end
x(i,:) = yi;
end
plot (x(:,1))
```

Various diagnostics have been proposed to assess convergence and to speed up the convergence. Most of these diagnostics use multiple chains in one way or another. Use of batch means from separate streams can be used to determine when the variance has stabilized. (See Section 2.1.3 for a description of batch means.)

For a monotone chain (one whose transition matrix stochastically preserves orderings of state vectors) that has two starting state vectors $x^-_0$ and $x^+_0$ such that for all $x \in S$, $x^-_0 \leq x \leq x^+_0$, Propp and Wilson (1996) show that if the sequence beginning with $x^-_0$ and the sequence beginning with $x^+_0$
exact sampling of Markov chain stationary distribution
Markov chain Monte Carlo
MCMC (Markov chain Monte Carlo)
Gibbs method

Use of Conditional Distributions

If the density of interest, \( p_X \), can be represented as a marginal density of some joint density \( p_{XY} \), observations on \( X \) can be generated as a Markov chain whose elements have densities,

\[
P_{Y|X_{i-1}} \cdot P_{X_i|Y} \cdot P_{Y_{i+1}|X_i} \cdot P_{X_{i+1}|Y_{i+1}} \cdot \ldots
\]

This is a simple instance of the Gibbs algorithm, which we discuss beginning on page 190. Casella and George (1992) explain this method in general.

The usefulness of this method depends on identifying a joint density with conditionals that are easy to simulate. For example, if the distribution of interest is a standard normal, the joint density

\[
p_{XY}(x,y) = \frac{1}{\sqrt{2\pi}} \quad \text{for} \quad -\infty < x < \infty, \quad 0 < y < e^{-x^2/2},
\]

has a marginal density corresponding to the distribution of interest, and it has simple conditionals. The conditional distribution of \( Y|X \) is \( U(0, e^{-x^2/2}) \), and the conditional of \( X|Y \) is \( U(-\sqrt{-2 \log Y}, \sqrt{-2 \log Y}) \). Starting with \( x_0 \) in the range of \( X \), we generate \( y_1 \) as a uniform conditional on \( x_0 \), then \( x_1 \) as a uniform conditional on \( y_1 \), and so on. The auxiliary variable \( Y \) that we introduce just to use to simulate \( X \) is called a “latent variable”.

4.2.5 General Methods for Multivariate Distributions

Two simple methods of generating multivariate random variates make use of variates from univariate distributions. One way is to generate a vector of i.i.d. variates, and then apply a transformation to yield a vector from the desired multivariate distribution. Another way is to use the representation of the distribution function or density function as a product of the form

\[
p_{X_1X_2X_3\ldots X_d} = p_{X_1|X_2X_3\ldots X_d} \cdot p_{X_2|X_3\ldots X_d} \cdot p_{X_3|\ldots X_d} \cdots p_{X_d}.
\]

In this method we generate a marginal \( x_d \) from \( p_{X_d} \), then a conditional \( x_{d-1} \) from \( p_{X_{d-1}|X_d} \), and continue in this way until we have the full realization
We see two simple examples of these methods at the beginning of Section 4.3.3, page 208. In the first example in that section, we generate a $d$-variate normal with variance-covariance matrix $\Sigma$ either by the transformation $x = T^Tz$, where $T$ is a $d \times d$ matrix such that $T^TT = \Sigma$ and $z$ is a $d$-vector of i.i.d. $N(0,1)$ variates. In the second case, we generate $x_1$ from $N_1(0,\sigma_{11})$, then generate $x_2$ conditionally on $x_1$, then generate $x_3$ conditionally on $x_1$ and $x_2$, and so on.

As mentioned in discussing acceptance/rejection methods in Sections 4.2.3 and 4.2.4, these methods are directly applicable to multivariate distributions, so acceptance/rejection is a third general way of generating multivariate observations. As in the example of the bivariate gamma on page ??, however, this usually involves a multivariate majorizing function, so we are still faced with the basic problem of generating from some multivariate distribution.

An iterative method, somewhat similar to the use of marginals and conditionals, can also be used to generate multivariate observations. This method was used by Geman and Geman (1984) for generating observations from a Gibbs distribution (Boltzmann distribution), and so is called the Gibbs method. In the Gibbs method, after choosing a starting point, the components of the $d$-vector variate are generated one at a time conditionally on all others. If $p_X$ is the density of the $d$-variate random variable $X$, we use the conditional densities $p_{X_1|X_2,X_3,\ldots,X_d}$, $p_{X_2|X_1,X_3,\ldots,X_d}$, and so on. At each stage the conditional distribution uses the most recent values of all the other components. Obviously, it may require a number of iterations before the choice of the initial starting point is washed out.

The method is shown in Algorithm 4.7. (In the algorithms to follow, we represent the support of the density of interest by $S$, where $S \subseteq \mathbb{R}^d$.)

**Algorithm 4.7 Gibbs Method**

0. Set $k = 0$.

1. Choose $x^{(k)} \in S$.

2. Generate $x_1^{(k+1)}$ conditionally on $x_2^{(k)}, x_3^{(k)}, \ldots, x_d^{(k)}$.
   Generate $x_2^{(k+1)}$ conditionally on $x_1^{(k+1)}, x_3^{(k)}, \ldots, x_d^{(k)}$.
   \[ \ldots \]
   Generate $x_{d-1}^{(k+1)}$ conditionally on $x_1^{(k+1)}, x_2^{(k+1)}, \ldots, x_d^{(k)}$.
   Generate $x_d^{(k+1)}$ conditionally on $x_1^{(k+1)}, x_2^{(k+1)}, \ldots, x_{d-1}^{(k+1)}$.

3. If convergence has occurred, then
   3.a. deliver $x = x^{(k+1)}$;
   otherwise
   3.b. set $k = k + 1$, and go to step 2.

Casella and George (1992) give a simple proof that this iterative method converges; that is, as $k \to \infty$, the density of the realizations approaches $p_X$. The question of whether convergence has practically occurred in a finite number of iterations is similar in the Gibbs method to the same question in the
4.2 Transformations of Uniform Deviates: General Methods

Metropolis-Hastings method discussed in Section 4.2.4. In either case, to determine that convergence has occurred is not a simple problem.

Once a realization is delivered in Algorithm 4.7, that is, once convergence has been deemed to have occurred, subsequent realizations can be generated either by starting a new iteration with \( k = 0 \) in step 0, or by continuing at step 1 with the current value of \( x^{(k)} \). If the chain is continued at the current value of \( x^{(k)} \), we must remember that the subsequent realizations are not independent. This affects variance estimates (second order sample moments), but not means (first order moments). In order to get variance estimates we may use means of batches of subsequences or use just every \( m^\text{th} \) (for some \( m > 1 \)) deviate in step 3. (The idea is that this separation in the sequence will yield subsequences or a systematic subsample with correlations nearer 0. See Section 2.1.3 for a description of batch means.) If we just want estimates of means, however, it is best not to subsample the sequence; that is, the variances of the estimates of means (first order sample moments) using the full sequence is smaller than the variances of the estimates of the same means using a systematic (or any other) subsample (so long as the Markov chain is stationary.)

To see this, let \( \bar{x}_i \) be the mean of a systematic subsample of size \( n \) consisting of every \( m^\text{th} \) realization beginning with the \( i^\text{th} \) realization of the converged sequence. Now, following MacEachern and Berliner (1994), we observe that

\[
|\text{Cov}(\bar{x}_i, \bar{x}_j)| \leq V(\bar{x}_l)
\]

for any positive \( i, j, \) and \( l \) less than or equal to \( m \). Hence if \( \bar{x} \) is the sample mean of a full sequence of length \( nm \), then

\[
V(\bar{x}) = V(\bar{x}_l)/m + \sum_{i \neq j; i,j=1}^{m} \text{Cov}(\bar{x}_i, \bar{x}_j)/m^2
\]

\[
\leq V(\bar{x}_l)/m + m(m-1)V(\bar{x}_l)/m^2
\]

\[
= V(\bar{x}_l).
\]

See also Geyer (1992) for discussion of subsampling in the chain.

The paper by Gelfand and Smith (1990) was very important in popularizing the Gibbs method. Gelfand and Smith also describe a related method of Tanner and Wong (1987), called data augmentation, which Gelfand and Smith call substitution sampling. In this method, a single component of the \( d \)-vector is chosen (in step 1), and then multivariate subvectors are generated conditional on just one component. This method requires \( d(d-1) \) conditional distributions. The reader is referred to their article and to Schervish and Carlin (1992) for descriptions and comparisons with different methods. Tanner (1996) defines a chained data augmentation, which is the Gibbs method described above.

In the Gibbs method the components of the \( d \)-vector are changed systematically, one at a time. The method is sometimes called alternating conditional sampling to reflect this systematic traversal of the components of the vector.
Another type of Metropolis method is the hit-and-run sampler. In this method all components of the vector are updated at once. The method is shown in Algorithm 4.8, in the general version described by Chen and Schmeiser (1996).

Algorithm 4.8 Hit-and-Run Sampling

0. Set $k = 0$.
1. Choose $x^{(k)} \in S$.
2. Generate a random normalized direction $v^{(k)}$ in $\mathbb{R}^d$. (This is equivalent to a random point on a sphere, as discussed on page ??.)
3. Determine the set $S^{(k)} \subseteq \mathbb{R}$ consisting of all $\lambda \ni (x^{(k)} + \lambda v^{(k)}) \in S$. ($S^{(k)}$ is one-dimensional; $S$ is $d$-dimensional.)
4. Generate $\lambda^{(k)}$ from the density $g^{(k)}$, which has support $S^{(k)}$.
5. With probability $a^{(k)}$,
   5.a. set $x^{(k+1)} = x^{(k)} + \lambda^{(k)} v^{(k)}$;
   otherwise
   5.b. set $x^{(k+1)} = x^{(k)}$.
6. If convergence has occurred, then
   6.a. deliver $x = x^{(k+1)}$;
   otherwise
   6.b. set $k = k + 1$, and go to step 2.

Chen and Schmeiser (1996) discuss various choices for $g^{(k)}$ and $a^{(k)}$. One choice is

$$g^{(k)}(\lambda) = \begin{cases} 
\frac{p(x^{(k)} + \lambda v^{(k)})}{\int_{S^{(k)}} p(x^{(k)} + \lambda v^{(k)}) \, d\lambda} & \text{for } \lambda \in S^{(k)}, \\
0 & \text{otherwise},
\end{cases}$$

and

$$a^{(k)} = 1.$$

Another choice is $g$ uniform over $S^{(k)}$ if $S^{(k)}$ is bounded, or else some symmetric distribution centered on 0 (such as a normal or Cauchy), together with

$$a^{(k)} = \min\left(1, \frac{p(x^{(k)} + \lambda^{(k)} v^{(k)})}{p(x^{(k)})}\right).$$


Berbee et al. (1987) compare the efficiency of hit-and-run methods with acceptance/rejection methods and find the hit-and-run methods to be more efficient in higher dimensions. Gilks, Roberts, and George (1994) describe a generalization of the hit-and-run algorithm called adaptive direction sampling.
In this method a set of current points is maintained, and only one, chosen at random from the set, is updated at each iteration (see Gilks and Roberts, 1996).

Both the Gibbs method and hit-and-run are special cases of the Metropolis/Hastings method in which the $r$ of step 2 in Algorithm 4.6 (page 183) is exactly 1, so there is never a rejection.

The same issues of convergence that we encountered in discussing the Metropolis/Hastings methods must be addressed when using the Gibbs or hit-and-run methods. The need to run long chains can increase the number of computations to unacceptable levels. Schervish and Carlin (1992) and Cowles and Carlin (1996) discuss general conditions for convergence of the Gibbs sampler. Dellaportas (1995) discusses some issues in the efficiency of random number generation using the Gibbs method. See Casella and George (1992) for a description of the Gibbs sampler; and see Chen and Schmeiser (1993) for some general comparisons of Gibbs, hit-and-run, and variations. Generalizations about the performance of the methods are difficult; the best method often depends on the problem.

### 4.3 Simulating Random Numbers from Specific Distributions

For the important distributions, specialized algorithms based on the general methods discussed in the previous section are available. The important difference in the algorithms is their speed. A secondary difference is the size and complexity of the program to implement the algorithm. Because all of the algorithms for generating from nonuniform distributions rely on programs to generate from uniform distributions, an algorithm that uses only a small number of uniforms to yield a variate of the target distribution may be faster on a computer system on which the generation of the uniform is very fast. As we have mentioned, on a given computer system there may be more than one program available to generate uniform deviates. Often a portable generator is slower than a nonportable one, so for portable generators of nonuniform distributions those that require a small number of uniform deviates may be better. If evaluation of elementary functions is a part of the algorithm for generating random deviates, then the speed of the overall algorithm depends on the speed of the evaluation of the functions. The relative speed of elementary function evaluation is different on different computer systems.

The algorithm for a given distribution is some specialized version of those methods discussed in the previous section. Often the algorithm uses some combination of these general techniques.

Many algorithms require some setup steps to compute various constants and to store tables; therefore, there are two considerations for the speed: the setup time and the generation time. In some applications many random numbers from the same distribution are required. In those cases the setup
time may not be too important. In other applications the random numbers come from different distributions, probably the same family of distributions, but with changing values of the parameters. In those cases the setup time may be very significant. If the best algorithm for a given distribution has a long setup time, it may be desirable to identify another algorithm for use when the parameters vary. Any computation that results in a quantity that is constant with respect to the parameters of the distribution should of course be performed as part of the setup computations, so as to avoid performing the computation in every pass through the main part of the algorithm.

The efficiency of an algorithm may depend on the values of the parameters of the distribution. Many of the best algorithms, therefore, switch from one method to another, depending on the values of the parameters. In some cases the speed of the algorithm is independent of the parameters of the distribution. Such an algorithm is called a uniform time algorithm. In many cases, the most efficient algorithm in one range of the distribution is not the most efficient in other regions. Many of the best algorithms, therefore, use mixtures of the distribution.

Sometimes it is necessary to generate random numbers from some sub-range of a given distribution, such as the tail region. In some cases, there are efficient algorithms for such truncated distributions. (If there is no specialized algorithm for a truncated distribution, acceptance/rejection applied to the full distribution will always work, of course.)

Methods for generating random variates from specific distributions is an area in which there have been literally hundreds of papers, each proposing some wrinkle (not always new or significant). Because the relative efficiencies (“efficiency” here means “speed”) of the individual operations in the algorithms vary from one computing system to another, and also because these individual operations can be programmed in various ways, it is very difficult to compare the relative efficiencies for the algorithms. This provides fertile ground for a proliferation of “research” papers. Two other things contribute to the large numbers of insignificant papers in this area. It is easy to look at some algorithm, modify some step, and then offer the new algorithm. Thus, the intellectual capitalization required to enter the field is small. (In business and economics, this is the same reason that so many restaurants are started; only a relatively small capitalization is required.)

Another reason for the large number of papers purporting to give new and better algorithms is the diversity of the substantive and application areas that constitute the backgrounds of the authors. Monte Carlo simulation is widely used throughout both the hard and the soft sciences. Research workers in one field often are not aware of the research published in another field.

Although, of course, it is important to seek efficient algorithms, it is also necessary to consider a problem in its proper context. In Monte Carlo simulation applications, literally millions of random numbers may be generated, but the time required to generate them is likely only to be a very small fraction of the total computing time. In fact, it is probably the case that the fraction
of time required for the generation of the random numbers is somehow negatively correlated with the importance of the problem. The importance of the time required to perform some task usually depends more on its proportion of the overall time of the job, rather than on its total time.

Another consideration is whether or not the algorithm is portable, that is, whether or not it yields the same stream on different computer systems. As we mention in Section 4.2.3, methods that accept or reject a candidate variate based on a floating-point comparison, may not yield the same streams on different systems.

4.3.1 Modifications of Standard Distributions

For many of the common distributions there are variations that are useful either for computational or other practical reasons or because they model some stochastic process well. A distribution can sometimes be simplified by transformations of the random variable that effectively remove certain parameters that characterize the distribution. In many cases the algorithms for generating random deviates address the simplified version of the distribution. An appropriate transformation is then applied to yield deviates from the distribution with the given parameters.

Standard Distributions

A linear transformation, \( Y = aX + b \), is simple to apply, and is one of the most useful. The multiplier affects the scale and the addend affects the location. For example, a “three-parameter” gamma distribution, with density,

\[
p(y) = \frac{1}{\Gamma(\alpha) \beta^\alpha} (y - \gamma)^{\alpha-1} e^{-(y-\gamma)/\beta}, \quad \text{for } \gamma \leq y \leq \infty,
\]

can be formed from the simpler distribution with density

\[
g(x) = \frac{1}{\Gamma(\alpha)} x^{\alpha-1} e^{-x}, \quad 0 \leq x \leq \infty
\]

using the transformation \( Y = \beta X + \gamma \). (Here, and elsewhere when we give an expression for a probability density function, we imply that the density is equal to 0 outside of the range specified.) The \( \beta \) parameter is a scale parameter, and \( \gamma \) is a location parameter. (The remaining \( \alpha \) parameter is called the “shape parameter”, and it is the essential parameter of the family of gamma distributions.) The simpler form is called the standard gamma distribution.

Other distributions have similar standard forms. Standard distributions (or standardized random variables) allow us to develop simpler algorithms and more compact tables of values that can be used for a range of parameter values.
Truncated Distributions

In many stochastic processes, the realizations of the random variable are constrained to be within a given region of the support of the random variable. Over the allowable region, the random variable has a probability density (or probability function) that is proportional to the density (or probability) of the unconstrained random variable. If the random variable $Y$ has probability density $p(y)$ over a domain $S$, and if $Y$ is constrained to $R \subset S$, the probability density of the constrained random variable is

$$p_c(x) = \frac{1}{\Pr(Y \in R)} p(x), \quad \text{for } x \in R;$$

$$= 0, \quad \text{elsewhere.}$$

The most common types of constraint are truncations, either left or right. In a left truncation at $\tau$, say, the random variable $Y$ is constrained by $\tau \leq Y$; and in a right truncation, it is constrained by $Y \leq \tau$.

Generation of random variates with constraints can be handled by the general methods discussed in the previous section. The use of acceptance/rejection is obvious; merely generate from the full distribution and reject any realizations outside of the acceptable region. For truncated distributions, modification to the inverse CDF method is simple. For a right truncation at $\tau$ of a distribution with CDF $P_Y$, instead of the basic transformation (4.23), page 171, we use

$$X = P_Y^{-1}(U P_Y(\tau)), \quad (4.29)$$

where $U$ is a random variable from $U(0, 1)$.

There are usually more efficient ways of generating variates from constrained distributions. We describe a few of the more common ones (which are invariably truncations) in the following sections.

“Inverse” Distributions

In Bayesian applications, joint probability densities of interest often involve a product of the density of some well-known random variable and what might be considered the density of the multiplicative inverse of another well-known random variable. Common examples of this are the statistics used in studentization, the chi-squared and the Wishart. Many authors refer to the distribution of such a random variable as the “inverse distribution”; for example, an “inverse chi-squared distribution” is the distribution of $X^{-1}$ where $X$ has a chi-squared distribution. Other distributions with this interpretation are the inverse gamma distribution and the inverse Wishart distribution. (This interpretation of “inverse” is not the same as for that word in the inverse Gaussian distribution. In the cases of the inverse gamma, chi-squared, and Wishart distributions, the method for generation of random variates is the obvious one: generate from the regular distribution and then obtain the inverse.
4.3 Simulating Random Numbers from Specific Distributions

Folded Symmetric Distributions

For symmetric distributions, a useful nonlinear transformation is the absolute value. The distribution of the absolute value is often called a “folded” distribution. The exponential distribution, for example, is the folded double exponential distribution (see page 202).

Probability-Skewed Symmetric Distributions

The (standard) skew-normal distribution has density

\[ g(x) = \frac{2}{\sqrt{2\pi}} e^{-x^2/2} \Phi(\lambda x), \quad \text{for } -\infty \leq x \leq \infty. \]

where \( \Phi(\cdot) \) is the standard normal CDF, and \( \lambda \) is a constant such that \( -\infty < \lambda < \infty \). For \( \lambda = 0 \), the skew-normal distribution is the normal distribution, and in general, if \( |\lambda| \) is relatively small, the distribution is close to the normal. For larger \( |\lambda| \) the distribution is more skewed, either positively or negatively. This distribution is an appropriate distribution for variables that would otherwise have a normal distribution, but have been screened on the basis of a correlated normal random variable. See Arnold et al. (1993) for discussions.

Other symmetric distributions may also be skewed by the probability of a smaller value in this manner. The motivation would be similar to the modeling application. The general form of the probability density is

\[ g(x) \propto p(x) P(\lambda x), \]

where \( p(\cdot) \) is the density of the underlying symmetric distribution and \( P(\cdot) \) is corresponding CDF. Arnold and Beaver (2000) discuss definitions and applications of such densities, specifically, a skew-Cauchy density.

In most cases, if \( |\lambda| \) is relatively small, generation of random variables from a probability-skewed symmetric distribution using an acceptance/rejection method with the underlying symmetric distribution as the majorizing density is entirely adequate. For larger values of \( |\lambda| \) it is necessary to divide the support into two or more intervals. It is still generally possible to use the same majorizing density, but the multiplicative constant can be different in different intervals.

4.3.2 Some Specific Univariate Distributions

In this section we consider several of the more common univariate distributions and indicate methods for simulating them. The methods discussed are generally among the better ones, at least according to some criteria, but the discussion is not exhaustive. We give the details for some simpler algorithms, but in many cases the best algorithm involves many lines of a program with
several constants that optimize a majorizing function or a squeeze function or
the break points of mixtures. We sometimes do not describe the best method
in detail, but rather refer the interested reader to the relevant literature.
Devroye (1986a) has given a comprehensive treatment of methods for generat-
ing deviates from various distributions, and more information on many of
the algorithms in this section can be found in that reference. Chapter 5 of
Gentle (2003) also describes methods for additional specific distributions.

The descriptions of the algorithms that we give indicate the computations,
but if the reader develops a program from the algorithm, issues of computa-
tional efficiency should be considered. For example, in the descriptions, we do
not identify the computations that should be removed from the main body of
the algorithm and made part of some setup computations.

Two variations of a distribution are often of interest. In one variation, the
distribution is truncated. In this case as we mentioned above, the range of the
original distribution is restricted to a subrange, and the probability measure
adjusted accordingly. In another variation, the role of the random variable
and the parameter of the distribution are interchanged. In some cases these
quantities have a natural association, and the corresponding distributions are
said to be conjugate. An example of two such distributions are the binomia-
and the beta. What is a realization of a random variable in one distribution is
a parameter in the other distribution. For many distributions, we may want
to generate samples of a parameter, given realizations of the random variable
(the data).

**Normal Distribution**

The normal distribution, which we denote by \( N(\mu, \sigma^2) \), has the probability
density

\[
p(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-(x-\mu)^2/(2\sigma^2)}, \quad \text{for } -\infty \leq x \leq \infty.
\]  

(4.30)

If \( Z \sim N(0,1) \) and \( X = \sigma Z + \mu \), then \( X \sim N(\mu, \sigma^2) \). Because of this simple
relationship, it is sufficient to develop methods to generate deviates from the
*standard* normal distribution, \( N(0,1) \), and so there is no setup involved. All
constants necessary in any algorithm can be precomputed and stored.

There are several methods for transforming uniform random variates into
normal random variates.

One transformation *not* to use:

if \( \{U_i, \text{ for } i = 1, \ldots, 12\} \text{ i.i.d. } \sim U(0,1) \), then

\[
X = \sum U_i - 6
\]

has an approximate \( N(0,1) \) distribution.

This method is the Central Limit Theorem applied to a sample of size 12. Not
only is the method approximate, it is also slower than better methods.
A simple and good method is the Box-Muller method arising from a polar transformation: If $U_1$ and $U_2$ are independently distributed as $U(0, 1)$, and

$$X_1 = \sqrt{-2 \log(U_1)} \cos(2\pi U_2)$$
$$X_2 = \sqrt{-2 \log(U_1)} \sin(2\pi U_2)$$

then $X_1$ and $X_2$ are independently distributed as $N(0, 1)$. This method is frequently maligned by people who analyze it with inappropriate uniform generators. For example, Neave (1973) shows that when the uniform generator is a congruential generator with a very small multiplier, the resulting normals are deficient in the tails; and Golder and Settle (1976) under similar conditions demonstrate that the density of the generated normal variates has a jagged shape, especially in the tails. Of course, if they had analyzed their small-multiplier congruential generator, they would have found that generator lacking. (See the discussion about Figure 4.2, page 126.) These studies emphasize the importance of using a good uniform generator for whatever distribution is to be simulated. It is especially important to be wary of the effects of a poor uniform generator in algorithms that require more than one uniform deviate (see the discussion beginning on page 174).

Bratley, Fox, and Schrage (1987) show that the generated normal variates lie pairwise on spirals, and say because of this, as “an approximation to a pair of independent variates [the method] is terrible”. The spirals, however, are of exactly the same origin as the lattice of the congruential generator itself.

To alleviate potential problems of patterns in the output of a polar method such as the Box-Muller transformation, some authors have advocated that for each pair of uniforms, only one of the resulting pair of normals be used. If there is any marginal gain in quality, it is generally not noticeable.

The Box-Muller transformation is one of several polar methods. They all have similar properties, but the Box-Muller transformation generally requires slower computations. Although most currently available computing systems can evaluate the necessary trigonometric functions extremely rapidly, the Box-Muller transformation can often be performed more efficiently using an acceptance/rejection algorithm, as we indicated in the general discussion of acceptance/rejection methods. The Box-Muller transformation is implemented via rejection in Algorithm 4.9.

**Algorithm 4.9 A Rejection Polar Method for Normal Variates**

1. Generate $v_1$ and $v_2$ independently from $U(-1, 1)$, and set $r^2 = v_1^2 + v_2^2$.
2. If $r^2 \geq 1$, then go to step 1;
   otherwise deliver
   $$x_1 = v_1 \sqrt{-2 \log(r^2/r^2)}$$
   $$x_2 = v_2 \sqrt{-2 \log(r^2/r^2)}.$$
Ahrens and Dieter (1988) describe fast polar methods for the Cauchy and exponential distributions in addition to the normal.

The fastest algorithms for generating normal deviates use either a ratio-of-uniforms method or a mixture with acceptance/rejection. One of the best algorithms, called the rectangle/wedge/tail method, is described by Marsaglia, MacLaren, and Bray (1964). In that method the normal density is decomposed into a mixture of densities with shapes as shown in Figure 4.19. It is easy to generate a variate from one of the rectangular densities, so the decomposition is done to give a high probability of being able to use a rectangular density. That, of course, means lots of rectangles, which brings some inefficiencies. The optimal decomposition must address those trade-offs. The wedges are nearly linear densities so generating from them is relatively fast. The tail region takes the longest time, so the decomposition is such as to give a small probability to the tail. Ahrens and Dieter (1972) give an implementation of the rectangle/wedge/tail method that can be optimized at the bit level.

Kinderman and Monahan (1977), when they first introduced the ratio-of-uniforms method, applied it to the normal distribution. Leva (1992a) gives an algorithm based on improved bounding curves in the ratio-of-uniforms method. (The 15-line Fortran program implementing Leva’s method is Algorithm 712 of CALGO; see Leva, 1992b.)

Kinderman and Ramage (1976) represent the normal density as a mixture, and apply a variety of acceptance/rejection and table look-up techniques.
for the components. The individual techniques for various regions have been developed by Marsaglia (1964), Marsaglia and Bray (1964), and Marsaglia, MacLaren, and Bray (1964). Marsaglia and Tsang (1984) also give a decomposition, resulting in what they call the “ziggurat method”.

Given the current speed of the standard methods of evaluating the inverse normal CDF, the inverse CDF method is often useful, especially if order statistics are of interest. Even with the speed of the standard algorithms for the inverse normal CDF, specialized versions, possibly to a slightly lower accuracy, have been suggested, for example by Marsaglia (1991) and Marsaglia, Zaman, and Marsaglia (1994). (The latter reference gives two algorithms for inverting the normal CDF; one very accurate, and one faster but slightly less accurate.)

### Truncated Normal Distribution

In Monte Carlo studies, the tail behavior is often of interest. Variates from the tail of a distribution can always be formed by selecting variates generated from the full distribution, of course; but this can be a very slow process. Marsaglia (1964), Geweke (1991a), Robert (1995), and Damien and Walker (2001) give methods for generating directly from a truncated normal distribution. The truncated normal with left truncation point $\tau$ has density

$$p(x) = \frac{e^{-(x-\mu)^2/(2\sigma^2)}}{\sqrt{2\pi}\sigma \left(1 - \Phi\left(\frac{\tau-\mu}{\sigma}\right)\right)}, \quad \text{for } \tau \leq x \leq \infty,$$

where $\Phi(\cdot)$ is the standard normal CDF.

The method of Robert uses an acceptance/rejection method with a translated exponential as the majorizing density, that is,

$$g(y) = \lambda^* e^{-\lambda^*(y-\tau)}, \quad \text{for } \tau \leq y \leq \infty,$$

where

$$\lambda^* = \frac{\tau + \sqrt{\tau^2 + 4}}{2}.$$  \hspace{1cm} (4.32)

(See the next section for methods to generate exponential random variates.)

The method of Damien and Walker uses conditional distributions. The range of the conditional uniform that yields the normal is taken as the intersection of the truncated range and the full conditional range.

### Lognormal and Halfnormal Distributions

Two distributions closely related to the normal are the lognormal and the halfnormal. The lognormal is the distribution of a random variable whose logarithm has a normal distribution. A very good way to generate lognormal variates is just to generate normal variates and exponentiate. The halfnormal is the folded normal distribution. The best way to generate deviates from the halfnormal is just to take the absolute value of normal deviates.
Exponential, Double Exponential, and Exponential Power Distributions

The exponential distribution with parameter $\lambda > 0$ has the probability density

$$p(x) = \lambda e^{-\lambda x}, \quad \text{for } 0 \leq x \leq \infty. \quad (4.33)$$

If $Z$ has the standard exponential distribution, that is, with parameter equal to 1, and $X = Z/\lambda$, then $X$ has the exponential distribution with parameter $\lambda$ (called the “rate”). Because of this simple relationship, it is sufficient to develop methods to generate deviates from the standard exponential distribution. The exponential distribution is a special case of the gamma distribution, whose density is given in equation (4.37). The parameters of the gamma distribution are $\alpha = 1$ and $\beta = \frac{1}{\lambda}$.

The inverse CDF method is very easy to implement and is generally satisfactory for the exponential distribution. The method is to generate $u$ from $U(0, 1)$ and then take

$$x = -\frac{\log(u)}{\lambda}. \quad (4.34)$$

(This and similar computations are why we require that the simulated uniform not include its endpoints.)

Many other algorithms for generating exponential random numbers have been proposed over the years. Marsaglia, MacLaren, and Bray (1964) apply the rectangle/wedge/tail method to the exponential distribution. Ahrens and Dieter (1972) give a method that can be highly optimized at the bit level.

Ahrens and Dieter also provide a catalog of other methods for generating exponentials. These other algorithms seek greater speed by avoiding the computation of the logarithm. Many simple algorithms for random number generation involve evaluation of elementary functions. As we have indicated, evaluation of an elementary function at a random point can often be performed equivalently by acceptance/rejection, and Ahrens and Dieter (1988) describe a method for the exponential that does that. (See Hamilton, 1998, for some corrections to their algorithm.) As the software for evaluating elementary functions has become faster, the need to avoid their evaluation has become less.

A common use of the exponential distribution is as the model of the interarrival times in a Poisson process. A (homogeneous) Poisson process,

$$T_1 < T_2 < \ldots,$$

with rate parameter $\lambda$ can be generated by taking the output of an exponential random number generator with parameter $\lambda$ as the times,

$$t_1, t_2 - t_1, \ldots.$$
4.3 Simulating Random Numbers from Specific Distributions

Truncated Exponential Distribution

The interarrival process is memoryless, and the tail of the exponential distribution has an exponential distribution, that is, if \( X \) has the density (4.33), and \( Y = X + \tau \), then \( Y \) has the density

\[
\lambda e^{\lambda \tau} e^{-\lambda y}, \quad \text{for } \tau \leq y \leq \infty.
\]

This fact provides a very simple process for generating from the tail of an exponential distribution.

Double Exponential Distribution

The double exponential distribution, also called the Laplace distribution, with parameter \( \lambda > 0 \) has the probability density

\[
p(x) = \frac{\lambda}{2} e^{-\lambda |x|}, \quad \text{for } -\infty \leq x \leq \infty.
\]  \hspace{1cm} (4.35)

The double exponential distribution is often used in Monte Carlo studies of robust procedures, because it has a heavier tail than the normal distribution, yet corresponds well with observed distributions.

If \( Z \) has the standard exponential distribution and \( X = SZ/\lambda \), where \( S \) is a random variable with probability mass \( \frac{1}{2} \) at \(-1\) and at \(+1\), then \( X \) has the double exponential distribution with parameter \( \lambda \). This fact is the basis for the method of generating double exponential variates; generate an exponential, and change the sign with probability \( \frac{1}{2} \). The method of bit stripping (see page 121) can be used to do this, so long as the lower-order bits are the ones used, and assuming the basic uniform generator is a very good one.

Exponential Power Distribution

A generalization of the double exponential distribution is the exponential power distribution, whose density is

\[
p(x) \propto e^{-\lambda |x|^\alpha}, \quad \text{for } -\infty \leq x \leq \infty.
\]  \hspace{1cm} (4.36)

For \( \alpha = 2 \), the exponential power distribution is the normal distribution. The members of this family with \( 1 \leq \alpha < 2 \) are often used to model distributions with slightly heavier tails than the normal distribution. Either the double exponential or the normal distribution, depending on the value of \( \alpha \), works well as a majorizing density to generate exponential power variates by acceptance/rejection (see Tadikamalla, 1980a).
Gamma Distribution

The gamma distribution with parameters $\alpha > 0$ and $\beta > 0$ has the probability density

$$p(x) = \frac{1}{\Gamma(\alpha)\beta^\alpha} x^{\alpha-1}e^{-x/\beta}, \quad \text{for } 0 \leq x \leq \infty,$$

where $\Gamma(\alpha)$ is the complete gamma function. The $\alpha$ parameter is called the shape parameter, and $\beta$ is called the scale parameter. If the random variable $Z$ has the standard gamma distribution with shape parameter $\alpha$ and scale parameter 1, and $X = \beta Z$, then $X$ has gamma distribution with parameters $\alpha$ and $\beta$. (Notice that the exponential is a gamma with $\alpha = 1$, and $\beta = 1/\lambda$.)

Of the special distributions we have considered so far, this is the first one that has a parameter that cannot be handled by simple translations and scalings. Hence, the best algorithms for the gamma may be different depending on the value of $\alpha$ and on how many deviates are to be generated for a given value of $\alpha$.

Cheng and Feast (1979) and Kinderman and Monahan (1980) use the ratio-of-uniforms method for a gamma distribution with $\alpha > 1$. The method of Cheng and Feast is shown in Algorithm 4.10. The mean time of this algorithm is $O(\alpha^{1.2})$, so for larger values of $\alpha$ it is less efficient. Cheng and Feast (1980) give an acceptance/rejection method that was better for large values of the shape parameter. Schmeiser and Lal (1980) use a composition of 10 densities, some of the rectangle/wedge/tail type, followed by the acceptance/rejection method. The Schmeiser/Lal method is the algorithm used in the IMSL Libraries for values of the shape parameter greater than 1. The speed of the Schmeiser/Lal method does not depend on the value of the shape parameter. Sarkar (1996) gives a modification of the Schmeiser/Lal method that has greater efficiency because of using more intervals, resulting in tighter majorizing and squeeze functions, and because of using an alias method to help speed the process.

Algorithm 4.10 The Cheng/Feast (1979) Algorithm for Generating Gamma Random Variates when the Shape Parameter is Greater than 1

1. Generate $u_1$ and $u_2$ independently from $U(0, 1)$, and set

$$v = \left(\frac{\alpha - 1}{6\alpha}\right)u_1.$$

2. If

$$\frac{2(u_2 - 1)}{\alpha - 1} + v + \frac{1}{v} \leq 2,$$

then deliver $x = (\alpha - 1)v$;

otherwise

if
4.3 Simulating Random Numbers from Specific Distributions

\[
\frac{2 \log u_2}{\alpha - 1} - \log v + v \leq 1
\]

then deliver \( x = (\alpha - 1)v \).

3. Go to step 1.

An efficient algorithm for values of the shape parameter less than 1 is the acceptance/rejection method described in Ahrens and Dieter (1974) and modified by Best (1983), as shown in Algorithm 4.11. That method is the algorithm used in the IMSL Libraries for values of the shape parameter less than 1.

Algorithm 4.11 The Best/Ahrens/Dieter Algorithm for Generating Gamma Random Variates when the Shape Parameter is Less than 1

0. Set \( t = 0.07 + 0.75\sqrt{1 - \alpha} \) and \( b = 1 + e^{-t_\alpha} \).

1. Generate \( u_1 \) and \( u_2 \) independently from \( U(0,1) \), and set \( v = bu_1 \).

2. If \( v \leq 1 \), then
   
   set \( x = tv \)

   if \( u_2 \leq \frac{2 - x}{2 + x} \), then deliver \( x \);
   
   otherwise
   
   if \( u_2 \leq e^{-x} \), deliver \( x \);
   
   otherwise
   
   set \( x = -\log\left( \frac{t(b - v)}{\alpha} \right) \) and \( y = x/t \);

   if \( u_2(\alpha + y(1 - \alpha)) \leq 1 \), then deliver \( x \);

   otherwise

   if \( u_2 \leq y^{\alpha-1} \), deliver \( x \).

3. Go to step 1.

There are two cases of the gamma distribution that are of particular interest. The shape parameter \( \alpha \) often is a positive integer. In that case, the distribution is sometimes called the Erlang distribution. If \( Y_1, Y_2, \ldots, Y_\alpha \) are independently distributed as exponentials with parameter \( 1/\beta \), the \( X = \sum Y_i \) has a gamma (Erlang) distribution with parameters \( \alpha \) and \( \beta \). Using the inverse CDF method (equation (4.34)) with the independent realizations \( u_1, u_2, \ldots, u_\alpha \), we generate an Erlang as

\[
x = -\beta \log \left( \prod_{i=1}^{\alpha} u_i \right).
\]

The general algorithms for gammas work better if \( \alpha \) is large.

The other special case of the gamma is the chi-squared distribution in which the scale parameter \( \beta \) is 2. Twice the shape parameter \( \alpha \) is called the degrees of freedom.


**Generalized Gamma Distributions**

There are a number of generalizations of the gamma distribution. The generalizations provide more flexibility in modeling because they have more parameters. Stacy (1962) defined a generalization that has two shape parameters. It is especially useful in failure-time models. The distribution has density

\[
p(x) = \frac{|\gamma|}{\Gamma(\alpha)\beta^\gamma} x^{\alpha\gamma-1} e^{(-x/\beta)^\gamma}, \quad \text{for } 0 \leq x \leq \infty.
\]

This distribution includes as special cases the ordinary gamma (with \(\gamma = 1\)), the halfnormal distribution (with \(\alpha = \frac{1}{2}\) and \(\gamma = 2\)), and the Weibull (with \(\alpha = 1\)). The best way to generate a generalized gamma distribution is to use the best method for the corresponding gamma and then exponentiate.

Everitt (1998) describes a generalized gamma distribution, which he calls the “Creedy and Martin generalized gamma”, whose density is

\[
p(x) = \theta_0 x^{\theta_1} e^{\theta_2 x + \theta_3 x^2 + \theta_4 x^3} \quad \text{for } 0 \leq x \leq \infty.
\]

This density can of course be scaled with a \(\beta\), as in the other gamma distributions we have discussed.

Ghitany (1998) and Agarwal and Al-Saleh (2001) have described a generalized gamma distribution based on a generalized gamma function, \(\Gamma(\alpha,\nu,\lambda)\), introduced by Kobayashi (1991). The distribution has density

\[
p(x) = \frac{1}{\Gamma(\alpha,\nu,\lambda)} x^{\alpha-1} e^{(-x/\lambda)^\nu}, \quad \text{for } 0 \leq x \leq \infty.
\]

This distribution is useful in reliability studies because of the shapes of the hazard function that are possible for various values of the parameters. It is the same as the ordinary gamma for \(\lambda = 0\).

**Beta Distribution**

The beta distribution with parameters \(\alpha > 0\) and \(\beta > 0\) has the probability density

\[
p(x) = \frac{1}{B(\alpha, \beta)} x^{\alpha-1} (1-x)^{\beta-1}, \quad \text{for } 0 \leq x \leq 1,
\]

(4.38)

where \(B(\alpha, \beta)\) is the complete beta function.

Efficient methods for the beta require different algorithms for different values of the parameters. If either parameter is equal to 1, it is very simple to generate beta variates using the inverse CDF method, which in this case would
just be a root of a uniform. If both values of the parameters are less than 1, the
simple acceptance/rejection method of Jöhnk (1964), given as Algorithm 4.12,
is one of the best. If one parameter is less than 1 and the other is greater than
1, the method of Atkinson (1979) is useful. If both parameters are greater
than 1, the method of Schmeiser and Babu (1980) is very efficient, except
that it requires a lot of setup time. For the case of both parameters greater
than 1, Cheng (1978) gives an algorithm that requires very little setup time.
The IMSL Library uses all five of these methods, depending on the values of
the parameters and how many deviates are to be generated for a given setting
of the parameters.

Algorithm 4.12 Jöhnk’s Algorithm for Generating Beta Random
Variates when Both Parameters are Less than 1

1. Generate \( u_1 \) and \( u_2 \) independently from \( U(0,1) \), and set \( v_1 = u_1^{1/\alpha} \) and \( v_2 = u_2^{1/\beta} \).
2. Set \( w = v_1 + v_2 \).
3. If \( w > 1 \), then go to step 1.
4. Set \( x = \frac{v_1}{w} \) and deliver \( x \).

Binomial Distribution

The probability function for the binomial distribution with parameters \( n \) and
\( \pi \) is

\[
p(x) = \frac{n!}{x!(n-x)!} \pi^x (1 - \pi)^{n-x}, \quad \text{for } x = 0, 1, \ldots, n, \tag{4.39}
\]

where \( n \) is a positive integer and \( 0 < \pi < 1 \).

To generate a binomial, a simple way is to sum Bernoullis (equation (4.25),
and Algorithm 4.2, page 172), which is equivalent to an inverse CDF technique.
If \( n \), the number of independent Bernoullis, is small, this method is adequate.
The time required for this kind of algorithm is obviously \( O(n) \). For larger val-
ues of \( n \), the median of a random sample of size \( n \) from a Bernoulli distribution
can be generated (it has an approximate beta distribution; see Relles, 1972),
and then the inverse CDF method can be applied from that point. Starting
at the median allows the time required to be halved. Kemp (1986) shows that
starting at the mode results in an even faster method and gives a method to
approximate the modal probability quickly. If this idea is applied recursively,
the time becomes \( O(\log n) \). The time required for any method based on the
CDF of the binomial is an increasing function of \( n \).

Several methods whose efficiency is not so dependent on \( n \) are available,
and for large values of \( n \) they are to be preferred to methods based on the CDF.
(The value of \( \pi \) also affects the speed; the inverse CDF methods are generally
competitive so long as \( n\pi < 500 \).) Stadlober (1991) described an algorithm
based on a ratio-of-uniforms method. Kachitvichyanukul (1982) gives an ef-
ficient method using acceptance/rejection over a composition of four regions
Poisson Distribution

The probability function for the Poisson distribution with parameter $\theta > 0$ is

$$p(x) = \frac{e^{-\theta} \theta^x}{x!}, \quad \text{for } x = 0, 1, 2, \ldots$$

(4.40)

A Poisson with a small mean, $\theta$, can be generated efficiently by the inverse CDF technique. Kemp and Kemp (1991) describe a method that begins at the mode of the distribution and proceeds in the appropriate direction to identify the inverse. They give a method for identifying the mode and computing the modal probability.

Many of the other methods that have been suggested for the Poisson also require longer times for distributions with larger means. Ahrens and Dieter (1980) and Schmeiser and Kachitvichyanukul give efficient methods whose times do not depend on the mean (see Schmeiser, 1983). The method of Schmeiser and Kachitvichyanukul uses acceptance/rejection over a composition of four regions. This is the method used in the IMSL Libraries.

4.3.3 Some Specific Multivariate Distributions

Multivariate distributions can be built from univariate distributions either by a direct transformation of a vector of i.i.d. scalar variates or by a sequence of conditional scalar variates.

The use of various Markov chain methods in Monte Carlo has made the conditional approaches more popular in generating multivariate deviates. The hit-and-run sampler (see page 192) is particularly useful in generating from multivariate distributions (see Belisle, Romeijn, and Smith, 1993, for example). A survey of methods and applications of multivariate simulation can be found in Johnson (1987). Gentle (2003) discusses additional methods for simulating multivariate distributions.

Elliptically contoured multivariate distributions are of special interest. The densities of these distributions have concentric ellipses with constant values. The density of an elliptically contoured distribution is of the form

$$p(x) = c \frac{1}{|\Sigma|^2} g((x - \mu)^T \Sigma^{-1} (x - \mu)),$$

where $c$ is a positive constant of proportionality and $g$ is a nonnegative real scalar valued function. The multivariate normal distribution is an elliptically contoured distribution.


4.3 Simulating Random Numbers from Specific Distributions

**Multivariate Normal Distribution**

The $d$-variate normal distribution with mean vector $\mu$ and nonsingular variance-covariance matrix $\Sigma$, which we denote by $N_d(\mu, \Sigma)$, has the probability density function

$$p(x) = \frac{1}{(2\pi)^{d/2} |\Sigma|^{1/2}} \exp \left( -\frac{(x - \mu)^T \Sigma^{-1}(x - \mu)}{2} \right).$$  \hfill (4.41)

A direct way of generating random vectors from the multivariate normal distribution is to generate a $d$-vector of i.i.d. standard normal deviates $z = (z_1, z_2, \ldots, z_d)$ and then to form the vector

$$x = T^T z + \mu,$$  \hfill (4.42)

where $T$ is a $d \times d$ matrix such that $T^T T = \Sigma$. ($T$ could be a Cholesky factor of $\Sigma$, for example, see page 285.) Then $x$ has a $N_d(\mu, \Sigma)$ distribution.

Another approach for generating the $d$-vector $x$ from $N_d(\mu, \Sigma)$ is to generate $x_1$ from $N_1(0, \sigma_{11})$, generate $x_2$ conditionally on $x_1$, generate $x_3$ conditionally on $x_1$ and $x_2$, and so on.

Multivariate distributions that are restricted to some subspace of the standard range may be difficult to simulate in general. Often we must resort to generation of variates from the full distribution followed by rejection of those that do not meet the restriction. A multivariate normal distribution that is truncated by linear restrictions, however, can be handled easily. If the variate $x$ from $N_d(\mu, \Sigma)$ must satisfy the restriction

$$a \leq Cx \leq b,$$

where $C$ is a full rank matrix, the restrictions can be applied to a vector of i.i.d. standard normal deviates $z$,

$$(T^T)^{-1}(C^{-1}a - \mu) \leq z \leq (T^T)^{-1}(C^{-1}b - \mu),$$

where $T^T$ is as in equation (4.42). The standard normal deviates with these restriction can be generated as described above, using an exponential majorizing density. Geweke (1991a) describes this method and gives some timing comparisons.

**Multinomial Distribution**

The probability function for the $d$-variate multinomial distribution is

$$p(x) = \frac{n!}{\prod x_j! \prod \pi_j^{x_j}}, \quad \text{for } x_j \geq 0, \text{ and } \Sigma x_j = n.$$  \hfill (4.43)

The parameters $\pi_j$ must be positive and sum to 1.
To generate a multinomial, a simple way is to work with the marginals; they are binomials. The generation is done sequentially. Each succeeding conditional marginal is binomial. For efficiency, the first marginal considered would be the one with the largest probability.

Another interesting algorithm for the multinomial, due to Brown and Bromberg (1984), is based on the fact that the conditional distribution of independent Poisson random variables, given their sum, is multinomial. The use of this relationship requires construction of extensive tables. Davis (1993) found the Brown/Bromberg method to be slightly faster than the sequential conditional marginal binomial method, once the setup operations are performed. If multinomials are to be generated from distributions with different parameters, however, the sequential conditional marginal method is more efficient.

4.4 Software for Random Number Generation

Random number generators are widely available in a variety of software packages. As Park and Miller (1988) state, however, “good ones are hard to find”.

Programming Issues

In addition to the quality of the algorithm for generation of random numbers, there are also many issues relating to the quality of the computer implementation of the random number generator. Because of the limitations of representing numbers exactly, a computer program rarely corresponds exactly to a mathematical expression. Sometimes a poor program completely negates the positive qualities of the algorithm it was intended to implement. The programming considerations relevant to implementing a random number generator are often subtle and somewhat different from those arising in other mathematical software (see Gentle, 1990).

To the extent that widely-used and well-tested software for random number generation is available, it should be used instead of software developed ad hoc.

Portable Random Number Generators

A major problem with random number generation on computers is that programs for the same generators on different computers often yield different sequences. For programs that make use of random number generators to be portable, the generator must produce the same sequence in all computer/compiler environments. It is desirable that the generators be portable to facilitate transfer of research and development efforts. Portability reduces the number of times the wheel is reinvented as well as the amount of the computer-knowledge overhead that burdens a researcher. The user can devote
attention to the research problem rather than to the extraneous details of the computer tools used to address the problem.

The heterogeneous computing environment in which most statisticians work has brought an increased importance to portability of software. Formerly, portability was a concern primarily for distributors of software, for users who may be switching jobs, or for computer installations changing or contemplating changing their hardware. With the widespread availability of personal computers, all computer users now are much more likely to use (or to attempt to use) the same program on more than one machine. There are both technical and tactical reasons for using a micro and a mainframe while working on the same problem. The technical reasons include the differences in resources (memory, CPU speed, software) available on micros and mainframes. These differences likely will continue. As new and better micros are introduced and more software is developed for them, new and better supercomputers will also be developed.

The availability of the different computers in different working environments such as home, lab, and office means that using multiple computers on a single problem can make more efficient use of one’s time. These tactical reasons for using multiple computers will persist, and it will become increasingly commonplace for a researcher to use more than one computer.

Many programming languages and systems come with built-in random number generators. The quality of the built-in generators varies widely (see Lewis and Orav, 1989, for analyses of some of these generators). Generators in the same software system, such as rand() in stdlib.h of the C programming language, may not generate the same sequence on different machines or even in different C compilers on the same machine. It is generally better to use a random number generator from a system such as the IMSL Library, which provides portability across different platforms.

The algorithms for random number generation are not always straightforward to implement, as we have discussed on page 134. Algorithms with relatively small operands, such as in Wichmann and Hill (1982) and the alternate generator of L’Ecuyer (1988), are likely to be portable; and, in fact, can even be implemented on computers with 16-bit words. The multipliers suggested by Wu (1997) that we discussed above can fairly easily be implemented portably.

We have mentioned several subtle problems for implementing congruential generators on page 134. Other generators have similar problems, such as the 0 and 1 problem, of which most people who have never built generators used in large-scale simulations are not aware. See Gentle (1981) for further discussion of the issue of portability and how it can be achieved in random number generators.
Basic Uniform Generators

Some programming languages such as C, Fortran 95, and Ada 95 provide built-in random number generators. In C the generator is the function `rand()` in `stdlib.h`. This function returns an integer in the range 0 through `RAND_MAX`, so the result must be normalized to the range (0, 1). (The scaling should be done with care. The seed for the C random number generator is set in `srand()`.

In Fortran 95 the generator is the subroutine `random_number`, which returns U(0, 1) numbers. (The user must be careful, however; the generator may yield either a 0 or a 1.) The seed can be set in the subroutine `random_seed`. The design of the Fortran 95 module as a subroutine yields a major advantage over the C function in terms of efficiency. (Of course, because Fortran 95 has the basic advantage of arrays, the module could have been designed as an array function and would still have had an advantage over the C function.)

A basic problem with the built-in generator of C, Fortran 95, and Ada 95 is lack of portability. The standards do not specify the algorithm. The bindings are portable, but neither generator will necessarily generate the same sequence on different platforms.

Other Distributions

Given a uniform random number generator, it is usually not too difficult to generate variates from other distributions. For example, in Fortran 95, the inverse CDF technique for generating a random deviate from a Bernoulli distribution with parameter $\pi$ can be implemented by the code in Figure 4.20.

```fortran
integer, parameter :: n = 100 ! INITIALIZE THIS
real, parameter :: pi = .5 ! INITIALIZE THIS
real, dimension (n) :: uniform
real, dimension (n) :: bernoulli
call random_number (uniform)
where (uniform .le. pi)
  bernoulli = 1.0
elsewhere
  bernoulli = 0.0
endwhere
```

**Figure 4.20.** A Fortran 95 Code Fragment to Generate $n$ Bernoulli Random Deviates with Parameter $\pi$

Efficiency

Implementing one of the simple methods to convert a uniform deviate to that of another distribution may not be as efficient as a special method for the
target distribution; and, as we have indicated, those special methods may be somewhat complicated. The IMSL Libraries and S-Plus and R have a number of modules that use efficient methods to generate variates from several of the more common distributions. Matlab has a basic uniform generator, \texttt{rand}, and a standard normal generator, \texttt{randn}. The Matlab Statistics Toolbox also contains generators for several other distributions.

In programming random number generators, the standard principles of efficiency apply: precompute and store constants; remove any constants within a loop from the loop; rearrange computations to their simplest form (in terms of computer operations); and so on.

It should be noted that the algorithms described in Sections 4.2 and 4.3 are written with an emphasis on clarity, rather than on computational efficiency; therefore, in some cases, the code should not correspond directly to the algorithm description.

**Choice of Software for Monte Carlo Studies**

Monte Carlo studies typically require many repetitive computations, which are usually implemented through looping program control structures. Some higher-level languages do not provide efficient looping structures. For this reason, it is usually desirable to conduct moderate- to large-scale Monte Carlo studies using a lower-level language such as C or Fortran. Some higher-level languages provide the capability to produce compiled code, which will execute faster. If Monte Carlo studies are to be conducted using an interpretive language, and if the production of compiled code is an option, that option should be chosen for the Monte Carlo work.

**Publicly Available Software**

A number of Fortran or C programs are available in collections published by *Applied Statistics* and by *ACM Transactions on Mathematical Software*. These collections are available online at \texttt{statlib} and \texttt{netlib}, respectively. See the bibliography for more information.

The R programming system, which contains several random number generators (see below) is freely available at

\[ \text{http://lib.stat.cmu.edu/R/CRAN/} \]

The freely-distributed GNU Scientific Library (GSL) contains several C functions for random number generation. There are several different basic uniform generators in the library. Utility functions in the library allow selection of a uniform generator for use by the functions that generate nonuniform numbers. In addition to a number of newer uniform generators, there are basic uniform generators that yield output sequences that correspond (or almost correspond) to legacy generators provided by various systems developers, such as the IBM \texttt{RANDU} and generators associated with various Unix distributions.
Information about the GNU Scientific Library, including links to sites from which source can be obtained, is available at

http://sources.redhat.com/gsl/

The Guide to Available Mathematical Software, or GAMS (see the bibliography) can be used to locate special software for various distributions.

**The User Interface for Random Number Generators**

Software for random number generation must provide a certain amount of control by the user, including the ability to

- set or retrieve the seed
- select seeds that yield separate streams
- possibly select the method from a limited number of choices.

Whenever the user invokes a random number generator for the first time in a program or a session, the software should not require the specification of a seed, but it should allow the user to set it if desired. If the user does not specify the seed, the software should use some mechanism, such as accessing the system clock, to form a “random” seed. On a subsequent invocation of the random number generator, unless the user specifies a seed, the software should use the last value of the seed from the previous invocation. This means that the routine for generating random numbers must produce a “side effect”; that is, it changes something other than the main result. It is a basic tenet of software engineering that careful note must be taken of side effects. At one time side effects were generally to be avoided. In object-oriented programming, however, objects may encapsulate many entities, and as the object is acted upon, any of the components may change; so in object-oriented software side effects are to be expected. In object-oriented software for random number generation, the state of the generator is an object.

Another issue to consider in the design of a user interface for a random number generator is whether the output is a single value (and an updated seed) or an array of values. Although a function that produces a single value as the C function `rand()` is convenient to use, it can carry quite a penalty in execution time because of the multiple invocations required to generate an array of random numbers. It is generally better to provide both single- and multivalued procedures for random number generation, especially for the basic uniform generator.

**Controlling the Seeds in Monte Carlo Studies**

There are three reasons the user must be able to control the seeds in Monte Carlo studies: for testing of the program, for use of blocks in Monte Carlo experiments, and for combining results of Monte Carlo studies.
In the early phases of programming for a Monte Carlo study it is very important to be able to test the output of the program. To do this it is necessary to use the same seed from one run of the program to another.

Controlling seeds in a parallel random number generator is much more complicated than in a serial generator. Performing Monte Carlo computations in parallel requires some way of insuring the independence of the parallel streams.

**Random Number Generation in IMSL Libraries**

For doing Monte Carlo studies, it is usually better to use a software system with a compilable programming language, such as Fortran or C. Not only do such systems provide more flexibility and control, but the programs built in the compiler languages execute faster. To do much work in such a system, however, a library or routines both to perform the numerical computations in the inner loop of the Monte Carlo study and to generate the random numbers driving the study are needed.

The IMSL Libraries contain a large number of routines for random number generation. The libraries are available in both Fortran and C, each providing the same capabilities and with essentially the same interface within the two languages. In Fortran the basic uniform generator is provided in both function and subroutine forms.

The uniform generator allows the user to choose among seven different algorithms: a linear congruential generator with modulus of $2^{31} - 1$ and with three choices of multiplier, each with or without shuffling; and the generalized feedback shift generator described by Fushimi (1990), which has a period of $2^{521} - 1$. The multipliers that the user can choose are the “minimal standard” one of Park and Miller (1988), which goes back to Lewis, Goodman, and Miller (1969) found by Fishman and Moore (1982, 1986).

The user chooses which of the basic uniform generators to use by means of the Fortran routine `rnopt` or the C function `imsls_random_option`. For whatever choice is in effect, that form of the uniform generator will be used for whatever type of pseudorandom events are to be generated. The states of the generators are maintained in a common block (for the simple congruential generators, the state is a single seed; for the shuffled generators and the GFSR generator, the state is maintained in a table). There are utility routines for setting and saving states of the generators and a utility routine for obtaining a seed to skip ahead a fixed amount.

There are routines to generate deviates from most of the common distributions. Most of the routines are subroutines but some are functions. The algorithms used often depend on the values of the parameters, so as to achieve greater efficiency. The routines are available in both single and double precision. (Double precision is more for the purpose of convenience for the user than it is for increasing accuracy of the algorithm.)
A single precision IMSL Fortran subroutine for generating from a specific distribution has the form

\[ \text{rnname} \ (\text{number, parameter}_1, \text{parameter}_2, ..., \text{output}_\text{array}) \]

where “\text{name}” is an identifier for the distribution, “\text{number}” is the number of random deviates to be generated, “\text{parameter}_i” are parameters of the distribution, and “\text{output}_\text{array}” is the output argument with the generated deviates. The Fortran subroutines generate variates from standard distributions, so location and scale parameters are not included in the argument list. The subroutine and formal arguments to generate gamma random deviates, for example, are

\[ \text{rngam} \ (\text{nr, a, r}) \]

where \text{a} is the shape parameter (\(\alpha\)) of the gamma distribution. The other parameter in the common two-parameter gamma distribution (usually called \(\beta\)) is a scale parameter. The deviates produced by the routine \text{rngam} have a scale parameter of 1; hence, for a scale parameter of \text{b}, the user would follow the call above with a call to a BLAS routine:

\[ \text{sscal} \ (\text{nr, b, r, 1}) \]

Identifiers of distributions include those shown in Table 4.1.

For general distributions, the IMSL Libraries provide routines for an alias method and for table lookup, for either discrete or continuous distributions. The user specifies a discrete distribution by providing a vector of the probabilities at the mass points, and specifies a continuous distribution by giving the values of the cumulative distribution function at a chosen set of points. In the case of a discrete distribution, the generation can be done either by an alias method or by an efficient table lookup method. For a continuous distribution, a cubic spline is first fit to the given values of the cumulative distribution function, and then an inverse CDF method is used to generate the random numbers from the target distribution. Another routine uses the Thompson-Taylor data-based scheme (Taylor and Thompson, 1986) to generate deviates from an unknown population from which only a sample is available.

Other routines in the IMSL Libraries generate various kinds of time series, random permutations, and random samples. The routine \text{rnuno}, which generates order statistics from a uniform distribution, can be used to generate order statistics from other distributions.

All of the IMSL routines for random number generation are available in both Fortran and C. The C functions have more descriptive names, such as \text{random_normal}. Also, the C functions may allow specification of additional arguments, such as location and scale parameters. For example, \text{random_normal} has optional arguments IMSLS\_MEAN and IMSLS\_VARIANCE.
4.4 Software for Random Number Generation

Table 4.1. Root Names for IMSL Random Number Generators

<table>
<thead>
<tr>
<th>Continuous Distributions</th>
<th>Discrete Distributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>un or unf</td>
<td>bin binomial</td>
</tr>
<tr>
<td>nor, noa, or nof</td>
<td>nbn negative binomial</td>
</tr>
<tr>
<td>mvn</td>
<td>poisson</td>
</tr>
<tr>
<td>chi</td>
<td>chi-squared</td>
</tr>
<tr>
<td>stt</td>
<td>geo geometric</td>
</tr>
<tr>
<td>tri</td>
<td>hyp hypergeometric</td>
</tr>
<tr>
<td>lnl</td>
<td>lgr logarithmic</td>
</tr>
<tr>
<td>exp</td>
<td>und discrete uniform</td>
</tr>
<tr>
<td>gam</td>
<td>mtn multinomial</td>
</tr>
<tr>
<td>wib</td>
<td>tab two-way tables</td>
</tr>
<tr>
<td>chy</td>
<td>Cauchy</td>
</tr>
<tr>
<td>beta</td>
<td>beta</td>
</tr>
<tr>
<td>vms</td>
<td>von Mises</td>
</tr>
<tr>
<td>stab</td>
<td>stable</td>
</tr>
<tr>
<td>ext</td>
<td>exponential mixture</td>
</tr>
<tr>
<td>cor</td>
<td>correlation matrices</td>
</tr>
<tr>
<td>sph</td>
<td>points on a circle or sphere</td>
</tr>
<tr>
<td>nos</td>
<td>order statistics from a normal</td>
</tr>
<tr>
<td>uno</td>
<td>order statistics from a uniform</td>
</tr>
<tr>
<td>arm</td>
<td>ARMA process</td>
</tr>
<tr>
<td>npp</td>
<td>nonhomogeneous Poisson process</td>
</tr>
</tbody>
</table>

Controlling the State of the Generators

Figure 4.21 illustrates the way to save the state of an IMSL generator and then restart it. The functions to save and to set the seed are `rnget` and `rnset`.

```fortran
  call rnget (iseed) ! save it
  call rnun (nr, y) ! get sample, analyze, etc.
  ...
  call rnset (iseed) ! restore seed
  call rnun (nr, yagain) ! will be the same as y
```

Figure 4.21. Fortran Code Fragment to Save and Restart a Random Sequence Using the IMSL Library

In a library of numerical routines such as the IMSL Libraries, it is likely that some of the routines will use random numbers in regular deterministic computations, such as an optimization routine generating random starting points. In a well-designed system, before a routine in the system uses a random number generator in the system, it will retrieve the current value of the seed if one has been set, use the generator, and then reset the seed to the former
Random Number Generation

S-Plus (software) R (software) R (software) DIEHARD tests for random number generators

value. IMSL subprograms are designed this way. This allows the user to control the seeds in the routines called directly.

Random Number Generation in R and S-Plus

The software system called S was developed at Bell Laboratories in the mid-1970s. Work on S has continued at Bell Labs and the system has evolved considerably since the early versions (see Becker, Chambers, and Wilks, 1988, and Chambers, 1997). S is both a data analysis system and an object-oriented programming language.

S-Plus is an enhancement of S, developed by StatSci, Inc. (now a part of Insightful Corporation). The enhancements include graphical interfaces, more statistical analysis functionality, and support.

There is a freely available package, called R, that provides generally the same functionality in the same language as S (see Gentleman and Ihaka, 1997).

S-Plus and R do not use the same random number generators. Monte Carlo studies conducted using built-in random number generators in one system cannot reliably be reproduced exactly in the other system with its built-in generators.

Random number generators in S-Plus are all based upon a single uniform random number generator that is a combination of a linear congruential generator and a Tausworthe generator. The original generator, called “Super-Duper”, is due to George Marsaglia in the 1970’s. It is described in Learmonth and Lewis (1973). McCullough (1999) reports results of the DIEHARD tests on the S-Plus generator. The tests raise some questions about the quality of the generator.

Several choices for the basic uniform generator are available in R. The function RNGkind can be used to choose the generator. One of the choices is Super-Duper, but the implementation is slightly different from the implementation in S-Plus. The user can also specify a user-defined and programmed generator. The chosen (or default) basic uniform generator is used in the generation of nonuniform variates.

In S-Plus and R there are some basic functions with the form

\[ r\text{name} \left( \text{number} \left[, \text{parameters} \right] \right) \]

where “name” is an identifier for the distribution, “number” is the number of random deviates to be generated, which can be specified by an array argument, in which case the number is the number of elements in the array; “parameters” are parameters of the distribution, which may or may not be required.

For distributions with standard forms, such as the normal, the parameters may be optional, in which case they take on default values if they are not specified. For other distributions, such as the gamma or the \( t \), there are required parameters. Optional parameters are both positional and keyword.

For example, the normal variate generation function is
4.4 Software for Random Number Generation

\[
\text{rnorm}(n, \text{mean}=0, \text{sd}=1) \]

so,
\[
\begin{align*}
\text{rnorm}(n) & \quad \text{yields } n \text{ normal (0,1) variates} \\
\text{rnorm}(n, 100, 10) & \quad \text{yields } n \text{ normal (100,100) variates} \\
\text{rnorm}(n, 100) & \quad \text{yields } n \text{ normal (100,1) variates} \\
\text{rnorm}(n, \text{sd}=10) & \quad \text{yields } n \text{ normal (0,100) variates}
\end{align*}
\]

(Note that S-Plus and R consider one of the natural parameters of the normal distribution to be the standard deviation or the scale, rather than the variance, as is more common.)

For the gamma distribution, at least one parameter (the shape parameter) is required. The function reference
\[
\text{rgamma}(100, 5)
\]
generates 100 random numbers from a gamma distribution with shape parameter of 5 and a scale parameter of 1 (a standard gamma distribution).

Identifiers of distributions include those shown in Table 4.2.

<table>
<thead>
<tr>
<th>Continuous Distributions</th>
<th>Discrete Distributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>unif uniform</td>
<td>binom binomial</td>
</tr>
<tr>
<td>norm normal</td>
<td>nbinom negative binomial</td>
</tr>
<tr>
<td>mvnorm multivariate normal</td>
<td>pois Poisson</td>
</tr>
<tr>
<td>chisq chi-squared</td>
<td>geom geometric</td>
</tr>
<tr>
<td>t t</td>
<td>hyper hypergeometric</td>
</tr>
<tr>
<td>f F</td>
<td>wilcox Wilcoxon rank sum statistic</td>
</tr>
<tr>
<td>lnorm lognormal</td>
<td></td>
</tr>
<tr>
<td>exp exponential</td>
<td></td>
</tr>
<tr>
<td>gamma gamma</td>
<td></td>
</tr>
<tr>
<td>weibull Weibull</td>
<td></td>
</tr>
<tr>
<td>cauchy Cauchy</td>
<td></td>
</tr>
<tr>
<td>beta beta</td>
<td></td>
</tr>
<tr>
<td>logis logistic</td>
<td></td>
</tr>
<tr>
<td>stab stable</td>
<td></td>
</tr>
</tbody>
</table>

The function \texttt{sample} generates a random sample with or without replacement. Sampling with replacement is equivalent to generating random numbers from a (finite) discrete distribution. The mass points and probabilities can be specified in optional arguments:
\[
xx \leftarrow \text{sample}(\text{massp}, n, \text{replace}=\text{T, probs})
\]
Order statistics in S-Plus and R can be generated using the beta distribution and the inverse distribution function. For example, 10 maximum order statistics from normal samples of size 30 can be generated by

\[ x \leftarrow \text{qnorm}(\text{rbeta}(10, 30, 1)) \]

**Controlling the State of the Generators**

Both S-Plus and R use an object called `.Random.seed` to maintain the state of the random number generators. In R, `.Random.seed` also maintains an indicator of which of the basic uniform random number generators is the current choice. Anytime random number generation is performed, if `.Random.seed` does not exist in the user’s working directory, it is created. If it exists, it is used to initiate the pseudorandom sequence, and then is updated after the sequence is generated. Setting a different working directory will change the state of the random number generator.

The function `set.seed(i)` provides a convenient way of setting the value of the `.Random.seed` object in the working directory to one of a fixed number of values. The argument `i` is an integer between 0 and 1023, and each value represents a state of the generator, which is “far away” from the other states that can be set in `set.seed`.

To save the state of the generator, just copy `.Random.seed` into a named object; and to restore, just copy the named object back into `.Random.seed`, as in Figure 4.22.

```r
oldseed <- .Random.seed  # save it
y <- runif(1000)         # get sample, analyze, etc.
...
.Random.seed <- oldseed   # restore seed
yagain <- rnorm(1000)    # will be the same as y
```

**Figure 4.22.** Code Fragment to Save and Restart a Random Sequence Using S-Plus or R

A common situation is one in which computations for a Monte Carlo study are performed intermittently, and are interspersed with other computations, perhaps broken over multiple sessions. In such a case, we may begin by setting the seed using the function `set.seed(i)`, save the state after each set of computations in the study, and then restore it prior to resuming the computations, similar to the code shown in Figure 4.23.

The built-in functions in S-Plus that use the random number generators have the side effect of changing the state of the generators, so the user must be careful in Monte Carlo studies where the computational nuclei, such as `ltsreg` for robust regression, for example, invoke an S-Plus random number generator. In this case, the user must retrieve the state of the generator prior
Figure 4.23. Starting and Restarting Monte Carlo Studies in S-Plus or R

In order to avoid the side effect of changing the state of the generator, when writing a function in S-Plus or R, the user can preserve the state upon entry to the function and restore it prior to exit. The assignment

```
.Random.seed <- oldseed
```

in Figure 4.22, however, does not work if it occurs within a user-written function in S-Plus or R. Within a function, the assignment must be performed by the `<<-` operator. A well-designed S-Plus or R function that invokes a random number generator would have code similar to that in Figure 4.24.

```
oldseed <- .Random.seed  # save seed on entry
...
.Random.seed <<- oldseed  # restore seed on exit
return(...)
```

Figure 4.24. Saving and Restoring the State of the Generator within an S-Plus or R Function

Monte Carlo in S-Plus and R

Explicit loops in S-Plus or R execute very slowly. For that reason, it is best to use array arguments for functions, rather than to loop over scalar values of the arguments. Consider, for example, the problem of evaluating the integral

\[
\int_0^2 \log(x+1)x^2(2-x)^3 \, dx.
\]

This could be estimated in a loop as follows:

```
# First, initialize n.
uu <- runif(n, 0, 2)
eu <- 0
for (i in 1:n) eu <- eu + log(uu[i]+1)*uu[i]^2*(2-uu[i])^3
eu <- 2*eu/n
```

A much more efficient way, without the for loop, but still using the uniform, is

\[
\begin{align*}
\text{uu} & \leftarrow \text{runif}(n, 0, 2) \\
\text{eu} & \leftarrow 2 \times \text{sum(} \log(\text{uu}+1) \times \text{uu}^2 \times (2-\text{uu})^3 \text{)/n}
\end{align*}
\]

Alternatively, using the beta density as a weight function, we have

\[
\text{eb} \leftarrow (16/15) \times \text{sum(} \log(2 \times \text{rbeta}(n,3,4)+1) \text{)/n}
\]

(Of course, if we recognize the relationship of the integral to the beta distribution, we would not use Monte Carlo as the method of integration.)

For large-scale Monte Carlo studies, an interpretive language like S-Plus or R may require an inordinate amount of running time. These systems are very useful for prototyping Monte Carlo studies, but it is often better to do the actual computations in a compiled language like Fortran or C.

**Exercises**

**4.1. Modular reduction and uniform distributions.**

a) Let \( R \) be a random variable with a \( U(0,1) \) distribution, let \( k \) be a nonzero integer constant, and let \( c \) be a real constant. Let

\[ S \equiv (kR + c) \mod 1, \text{ with } 0 \leq S \leq 1. \]

Show that \( S \) has a \( U(0,1) \) distribution. **Hint:** First let \( c = 0 \) and consider \( kR \); then consider \( T + c \), where \( T \) is from \( U(0,1) \).

b) Prove a generalization of Exercise 4.1a in which the constant \( c \) is replaced by a random variable with any distribution.

c) Let \( T \) be a random variable with a discrete uniform distribution with mass points \( 0, 1, \ldots, d - 1 \). Let \( W_1, W_2, \ldots, W_n \) be independently distributed as discrete uniform random variables with integers as mass points. Show that

\[ T + \sum_{i=1}^{n} W_i \mod d \]

has the same distribution as \( T \). (The reduced modulus is used in this expression, of course.) **Hint:** First consider \( T + W_1 \), and write \( (T + W_1) \mod d \) as \( j + kd \), where \( j \) and \( k \) are integers with \( 0 \leq j \leq d - 1 \). (See also L’Ecuyer, 1988.)

**4.2. Use Fortran, C, or some other programming system to write a program to implement a generator using a multiplicative congruential method with \( m = 2^{13} - 1 \) and \( a = 17 \). Generate 500 numbers \( x_i \). Compute the correlation of the pairs of successive numbers, \( x_{i+1} \) and \( x_i \). Plot the pairs. On how many lines do the points lie? Now let \( a = 85 \). Generate 500 numbers, compute the correlation of the pairs and plot them. Now, look at the pairs \( x_{i+2} \) and \( x_i \). Compute their correlation.
4.3. Now modify your program from Exercise 4.2 so as to implement a matrix congruential method

\[
\begin{bmatrix}
  x_{1i} \\
  x_{2i}
\end{bmatrix} \equiv
\begin{bmatrix}
  a_{11} & a_{12} \\
  a_{21} & a_{22}
\end{bmatrix}
\begin{bmatrix}
  x_{1,i-1} \\
  x_{2,i-1}
\end{bmatrix} \mod m,
\]

with \( m = 2^{13} - 1 \), \( a_{11} = 17 \), \( a_{22} = 85 \), and \( a_{12} \) and \( a_{21} \) variable. Letting \( a_{12} \) and \( a_{21} \) vary between 0 and 17, generate 500 vectors and compute their sample variance-covariances. Are the variances of the two elements in your vectors constant? Explain. What about the covariances? Can you see any relationship between the covariances and \( a_{12} \) and \( a_{21} \)? Is there any reason to vary \( a_{12} \) and \( a_{21} \) separately? Can a lower triangular matrix, that is, one with \( a_{21} = 0 \), provide all of the flexibility of matrices with varying values of \( a_{21} \)?

4.4. Write a Fortran or C function to implement the multiplicative congruential generator (4.3) (RANDU) on page 128.

a) Generate a sequence \( x_i \) of length 20,002. For all triplets in your sequence, \( (x_i, x_{i+1}, x_{i+2}) \), in which \( 0.5 \leq x_{i+1} \leq 0.51 \), plot \( x_i \) versus \( x_{i+2} \). Comment on the pattern of your scatterplot. (This is similar to the graphical analysis performed by Lewis and Orav, 1989.)

b) Generate a sequence of length 1002. Use a program that plots points in three dimensions and rotates the axes to rotate the points until the 15 planes are clearly visible. (A program that could be used for this is the S-Plus function \texttt{spin}, for example.)

4.5. Using an analysis similar to that leading to equation (4.4) on page 129, determine the maximum number of different planes on which triplets from the generator (4.3) would lie if instead of 65,539, the multiplier were 65,541. Determine the number of different planes if the multiplier were 65,533. (Notice both of these multipliers are congruent to 5 mod 8, as James, 1990, suggested.)

4.6. Write a Fortran or C function to use a multiplicative congruential method with \( m = 2^{31} - 1 \) and \( a = 16,807 \) (the “minimal standard”).

4.7. Write a Fortran or C function to use a multiplicative congruential method with \( m = 2^{31} - 1 \) and \( a = 950,706,376 \). Test for correctness (not for statistical quality) by using a seed of 1 and generating 10 numbers.

4.8. Suppose a sequence is generated using a linear congruential generator with modulus \( m \), beginning with the seed \( x_0 \). Show that this sequence and the sequence generated with the seed \( m - x_0 \) are antithetic. \textit{Hint: Use induction.}

4.9. Suppose a sequence is generated by \( x_{i+1} \equiv ax_i \mod m \), and a second sequence is generated by \( y_{i+1} \equiv by_i \mod m \), where

\[
b \equiv a^{c-1} \mod m,
\]

and \( c \) is the period. Prove that the sequences are in reverse order.
4.10. For the generator \( x_i \equiv 16807x_{i-1} \mod(2^{31} - 1) \), determine the value \( x_0 \) that will yield the largest possible value for \( x_1 \). (This seed can be used as a test that the largest value yielded by the generator is less than 1. It is desirable to scale all numbers into the open interval \((0, 1)\), because the numbers from the uniform generator may be used to an inverse CDF method for a distribution with an infinite range. To insure that this is the case, the value used for scaling must be greater than \(2^{31} - 1\). See Gentle, 1990.)

4.11. In this exercise use a package that supports computations for number theory, such as Maple.
   a) Check that the multipliers in Exercises 4.2, 4.5, and 4.10 are all primitive roots of the respective moduli.
   b) Determine all of the primitive roots of the modulus \(2^{13} - 1\) of Exercise 4.2.

4.12. Suppose for one stream from a given linear congruential generator the seed is 500, and for another stream from the same generator the seed is 1000. What is the approximate correlation between the two streams?

4.13. Consider the Wichmann/Hill random number generator (page 148). Because the moduli are relatively prime, the generator cannot yield an exact zero. Could a computer implementation of this generator yield a zero? First, consider a computer that has only two digits of precision. The answer is obvious. Now, consider a computer with a more realistic number system (such as whatever computer you use most often). How likely is the generator to yield a 0 on this computer? Perform some computations to explore the possibilities. Can you make a simple adjustment to the generator to prevent a 0 from occurring?

4.14. Write pseudocode for a random number generator for a parallel processing computer. Be sure that your generator preserves independence of the separate streams. Also, consider the issues of the user interface (for example, what input does the user provide?)

4.15. The inverse CDF method.
   a) Prove that if \( X \) is a random variable with an absolutely continuous distribution function \( P_X \), the random variable \( P_X(X) \) has a \( U(0, 1) \) distribution.
   b) Prove that the inverse CDF method for discrete random variables as specified in the relationship in expression (4.24) on page 172 is correct.

4.16. Formally prove that the random variable delivered in Algorithm 4.3 on page 176 has the density \( p_X \). \textit{Hint:} For the delivered variable, \( Z \), determine the distribution function \( \Pr(Z \leq x) \) and differentiate.

4.17. Write a Fortran or C function to implement the acceptance/rejection method for generating a beta(3,2) random deviate. Use the majorizing function shown in Figure 4.11 on page 177. The value of \( c \) is 1.2. Use the inverse CDF method to generate a deviate from \( g \). (This will involve taking a square root.)

4.18. Acceptance/rejection methods.
4.4 Software for Random Number Generation  

a) Give an algorithm to generate a normal random deviate using the acceptance/rejection method with the double exponential density as the majorizing density. After you have obtained the acceptance/rejection test, try to simplify it.

b) Write a program to generate bivariate normal deviates with mean \((0, 0)\), variance \((1, 1)\), and correlation \(\rho\). Use a bivariate product double exponential density as the majorizing density. Now set \(\rho = 0.5\) and generate a sample of 1,000 bivariate normals. Compare the sample statistics with the parameters of the simulated distribution.

4.19. What would be the problem with using a normal density to make a majorizing function for the double exponential distribution (or using a half-normal for an exponential)?

4.20. Consider the acceptance/rejection method given in Algorithm 4.3 to generate a realization of a random variable \(X\) with density function \(p_X\), using a density function \(g_Y\).

a) Let \(T\) be the number of passes through the three steps until the desired variate is delivered. Determine the mean and variance of \(T\) (in terms of \(p_X\) and \(g_Y\)).

b) Now consider a modification of the rejection method in which steps 1 and 2 are reversed, and the branch in step 3 is back to the new step 2, that is:
   1. Generate \(u\) from a uniform \((0, 1)\) distribution.
   2. Generate \(y\) from the distribution with density function \(g_Y\).
   3. If \(u \leq p_X(y)/cg_Y(y)\), then take \(y\) as the desired realization; otherwise return to step 2.

Is this a better method? Let \(Q\) be the number of passes through these three steps until the desired variate is delivered. Determine the mean and variance of \(Q\). (This method was suggested by Sibuya, 1961, and analyzed by Greenwood, 1976c.)
Vectors and matrices are useful in representing multivariate data, and they occur naturally in working with linear equations or when expressing linear relationships among objects. Numerical algorithms for a variety of tasks involve matrix and vector arithmetic. An optimization algorithm to find the minimum of a function, for example, may use a vector of approximate first derivatives and a matrix of second derivatives; and a method to solve a differential equation may use a matrix with a few diagonals for computing differences. There are various precise ways of defining vectors and matrices, but we will generally think of them merely as arrays of numbers, or scalars, on which an algebra is defined. Occasionally we will take a geometrical perspective for vectors and will consider matrices to define geometrical transformations.

5.1 Basic Vector/Matrix Computations

We assume the reader has a working knowledge of linear algebra, but in this first section, we give several definitions and state many useful facts about vectors and matrices. We also discuss vectors from a geometrical perspective.

The presentation in this chapter is informal; neither definitions nor facts are highlighted by such words as “Definition”, “Theorem”, “Lemma”, and so forth. The facts generally have simple proofs, some of which occur naturally in the text (no “Proof” and “Q.E.D.” or “□” appear to indicate beginning and end). In many cases formal proofs are not given — although sometimes they appear as exercises!

In Section 5.1.2, beginning on page 274, we discuss some of the basic issues of vector/matrix storage and computations on a computer. After consideration of numerical methods for solving linear systems and for eigenanalysis in Sections ?? and 5.3, we resume the discussion of computer manipulations and software in Section 5.4 on page 321.
We occasionally refer to two standard software packages for linear algebra, LINPACK (Dongarra et al., 1979) and LAPACK. (Anderson et al., 1999). We discuss these further in Section 5.4.

5.1.1 Notation, Definitions, and Basic Properties

A vector (or \(n\)-vector) is an \(n\)-tuple, or ordered (multi)set, or array, of \(n\) numbers, called elements. The number of elements is sometimes called the order, or sometimes the “length”, of the vector. An \(n\)-vector can be thought of as representing a point in \(n\)-dimensional space. In this setting, the length of the vector may also mean the Euclidean distance from the origin to the point represented by the vector, that is, the square root of the sum of the squares of the elements of the vector. This Euclidean distance will generally be what we mean when we refer to the length of a vector.

The first element of an \(n\)-vector is the first (1\(^{st}\)) element and the last is the \(n^{th}\) element. (This statement is not a tautology; in some computer systems, the first element of an object used to represent a vector is the 0\(^{th}\) element of the object. This sometimes makes it difficult to preserve the relationship between the computer entity and the object that is of interest.) We will use paradigms and notation that maintain the priority of the object of interest, rather than the computer entity representing it.

We may write the \(n\)-vector \(x\) as

\[
x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix},
\]

or as

\[
x = (x_1, x_2, \ldots, x_n).
\]

We make no distinction between these two notations, although in some contexts we think of a vector as a “column”, so the first notation may be more natural. We use the notation \(\mathbb{R}^n\) to denote the set of \(n\)-vectors with real elements.

Linear Combinations

The elements of a vector are elements of a field, and most vector operations are defined in terms of operations in the field. The elements of the vectors we will use in this book are real numbers, that is, elements of \(\mathbb{R}\).

Two vectors can be added if they are of the same length (that is, have the same number of elements); the sum of two vectors is the vector whose elements are the sums of the corresponding elements of the addends. Vectors with the same number of elements are said to be conformable for addition. A scalar
5.1 Basic Vector/Matrix Computations

multiple of a vector, that is, the product of an element from the field and a vector, is the vector whose elements are the multiples of the corresponding elements of the original vector.

We overload the usual symbols for the operations on the reals for the corresponding operations on vectors or matrices when the operations are defined, so “+”, for example, can mean addition of scalars or addition of conformable vectors.

A very common operation in working with vectors is the addition of a scalar multiple of one vector to another vector:

\[ ax + y, \quad (5.1) \]

where \( a \) is a scalar and \( x \) and \( y \) are vectors of equal length. Viewed as a single operation with three operands, this is called an “axpy” for obvious reasons. (Because the Fortran versions of BLAS to perform this operation were called saxpy and daxpy, the operation is also sometimes called “saxpy” or “daxpy”. See Section 5.4.1, page 324, for a description of the BLAS.) Such linear combinations of vectors are important operations.

**Linear Independence**

If a given vector can be formed by a linear combination of one or more vectors, the set of vectors (including the given one) is said to be linearly dependent; conversely, if in a set of vectors no one vector can be represented as a linear combination of any of the others, the set of vectors is said to be linearly independent. It is easy to see that the maximum number of \( n \)-vectors that can form a set that is linearly independent is \( n \). Linear independence is one of the most important concepts in linear algebra.

**Vector Spaces**

Let \( V \) be a set of \( n \)-vectors such that for any vectors in \( V \), any linear combination of those vectors is also in \( V \). Then the set \( V \) together with the usual vector algebra is called a vector space. (Technically, the “usual algebra” is for the operations of vector addition and scalar times vector multiplication. It has closure of the space under axpy, commutativity and associativity of addition, an additive identity and inverses, a multiplicative identity, distribution of multiplication over both vector addition and scalar addition, and associativity of scalar multiplication and scalar times vector multiplication.

The length or order of the vectors is the order of the vector space, and the maximum number of linearly independent vectors in the space is the dimension of the vector space.

We generally use a calligraphic font to denote a vector space; \( \mathcal{V} \), for example. We denote the additive identity in a vector space of order \( n \) by \( 0_n \), or sometimes by 0. This is the vector consisting of all zeros. Likewise, we denote
the vector consisting of all ones, by $1_n$, or sometimes by 1. Whether 0 and 1 represent vectors or scalars is usually clear from the context. The vector space consisting of all $n$-vectors with real elements is denoted $\mathbb{R}^n$. (As mentioned earlier, the notation $\mathbb{R}^n$ also refers to just the set of $n$-vectors with real elements; that is, to the set over which the vector space is defined.)

Although a vector space is a set together with operations, we often speak of a vector space as if it were a set; and we use some of the same notation to refer to vector spaces as the notation used to refer to sets. For example, if $V$ is a vector space, the notation $W \subseteq V$ indicates that $W$ is a vector space, that the set of vectors in the vector space $W$ is a subset of the vectors in $V$, and that the operations in the two objects are the same. A subset of a vector space $V$ that is itself a vector space is called a subspace of $V$.

The intersection of two vector spaces is a vector space, but their union is not necessarily a vector space (because for $v_1 \in V_1$ and $v_2 \in V_2$, $v_1 + v_2$ may not be in $V_1 \cup V_2$). If $V_1$ and $V_2$ are vector spaces, the space of vectors

$$V = \{ v, \text{ s.t. } v = v_1 + v_2, \ v_1 \in V_1, \ v_2 \in V_2 \}$$

is called the sum (or direct sum) of the vector spaces $V_1$ and $V_2$. The relation is denoted by

$$V = V_1 \oplus V_2.$$  \hspace{1cm} (5.2)

**Basis Sets**

If each vector in the vector space $V$ can be expressed as a linear combination of the vectors in the set $G$, then $G$ is said to be a generating set or spanning set of $V$, and this construction of the vector space may be denoted by $V(G)$. This vector space is also denoted by "span($G$)". A set of linearly independent vectors that span a space is said to be a basis for the space.

**Inner Products**

The elements of a vector often represent coefficients of scalar variables; for example, given the variables $x_1, x_2, \ldots, x_n$, we may be interested in the linear combination

$$c_1x_1 + c_2x_2 + \ldots + c_nx_n.$$  

The vector $c = (c_1, c_2, \ldots, c_n)$ is the coefficient vector and the sum $\sum_i c_i x_i$ is the dot product, the inner product, or the scalar product of the vectors $c$ and $x$. (The dot product is actually a special type of inner product, but it is the most commonly used inner product.) We denote the dot product of $c$ and $x$ by $\langle c, x \rangle$:

$$\langle c, x \rangle = \sum_i c_i x_i.$$  \hspace{1cm} (5.3)
The dot product is also sometimes written as $c \cdot x$, hence the name. Yet another notation for the dot product is $c^T x$, and we see later that this notation is natural in the context of matrix multiplication.

The dot product is a mapping from a vector space $V$ into $\mathbb{R}$ that has the following properties:

1. Nonnegativity and mapping of the identity:
   if $x \neq 0$, then $\langle x, x \rangle > 0$ and $\langle 0, 0 \rangle = 0$.

2. Commutativity:
   $\langle x, y \rangle = \langle y, x \rangle$.

3. Factoring of scalar multiplication in dot products:
   $\langle ax, y \rangle = a \langle x, y \rangle$ for real $a$.

4. Relation of vector addition to addition of dot products:
   $\langle x + y, z \rangle = \langle x, z \rangle + \langle y, z \rangle$.

These properties in fact define the more general inner product. A vector space together with such an operator is called an inner product space.

We also denote the dot product by $c^T x$, as we do with matrix multiplication. The dot product is not the same as the result of a matrix multiplication, however. The dot product is a scalar, and the result of a matrix multiplication is a matrix. Nevertheless, throughout this book we will treat a one by one matrix or a vector of length one as a scalar whenever it is convenient to do so. For example, we may write an expression such as $1/c^T x$.

A useful property of inner products is the Cauchy-Schwarz inequality:

$$\langle x, y \rangle \leq \langle x, x \rangle^{1/2} \langle y, y \rangle^{1/2}.$$  \hspace{1cm} (5.4)

This is easy to see, by first observing for every real number $t$,

\begin{align*}
0 &\leq (\langle tx + y, tx + y \rangle)^2 \\
&= \langle x, x \rangle t^2 + 2 \langle x, y \rangle t + \langle y, y \rangle \\
&= at^2 + bt + c,
\end{align*}

where the constants $a$, $b$, and $c$ correspond to the dot products in the preceding equation. This quadratic in $t$ cannot have two distinct real roots, hence the discriminant, $b^2 - 4ac$, must be less than or equal to zero; that is,

$$\left( \frac{1}{2} b \right)^2 \leq ac.$$

By substituting and taking square roots, we get the Cauchy-Schwarz inequality. It is also clear from this proof that equality holds only if $x = 0$ or if $y = rx$, for some scalar $r$.

**Geometrical Properties of Vectors**

Points in a cartesian geometry can be identified with vectors. Several definitions and properties can be motivated by the geometric interpretation of the vectors.
A cartesian coordinate system in $d$ dimensions is defined by $d$ unit vectors, each with $d$ elements. The $i^{\text{th}}$ unit vector, denoted by $e_i$, has a 1 in the $i^{\text{th}}$ position and 0’s in all other positions:

$$e_i = (0, \ldots, 0, 1, 0, \ldots, 0). \quad (5.5)$$

A unit vector is also called a principal axis of the coordinate system. (There is an implied number of elements of a unit vector that is inferred from the context. Also parenthetically, we remark that the phrase “unit vector” is sometimes used to refer to a vector the sum of whose squared elements is 1, that is, whose length, in the Euclidean distance sense described below, is 1. We refer this latter type of vectors as “normalized vectors”.)

A point, $x$, with cartesian coordinates $(x_1, \ldots, x_d)$ is associated with a vector from the origin to the point, that is, the vector $(x_1, \ldots, x_d)$. The vector can be written as the linear combination

$$x = x_1 e_1 + x_2 e_2 + \ldots + x_d e_d,$$

or, equivalently, as the sum of dot products,

$$x = \langle x, e_1 \rangle + \langle x, e_2 \rangle + \ldots + \langle x, e_d \rangle.$$

The length of the vector $x$ is $\sqrt{\langle x, x \rangle}$. The length is also called the norm of the vector, although as we see on page 257, it is just one of many norms.

The angle $\theta$ between the vectors $x$ and $y$ is defined by

$$\cos(\theta) = \frac{\langle x, y \rangle}{\sqrt{\langle x, x \rangle \langle y, y \rangle}}. \quad (5.6)$$

A given vector can be defined in terms of its length and the angles $\theta_i$ that it makes with the unit vectors. The cosines of these angles are just the scaled coordinates of the vector:

$$\cos(\theta_i) = \frac{\langle x, e_i \rangle}{\sqrt{\langle x, x \rangle \langle y, y \rangle}} = \frac{1}{\sqrt{\langle x, x \rangle}} x_i. \quad (5.7)$$

These quantities are called the direction cosines of the vector.

The properties of vectors defined in terms of a cartesian geometry have analogs in Euclidean geometry without a coordinate system. In such a system only length and direction are defined, and two vectors are considered to be the same vector if they have the same length and direction. There is no “position” associated with a vector.

The geometric property of the angle between vectors has important implications for certain operations, both because it may indicate that rounding
will have deleterious effects and because it may indicate a deficiency in the understanding of the application.

Two vectors, \( v_1 \) and \( v_2 \), whose dot product is 0 are said to be *orthogonal*, written \( v_1 \perp v_2 \) because this is equivalent to the corresponding geometric property. (Sometimes we exclude the zero vector from this definition, but it is not important to do so.) A vector whose dot product with itself is 1, is said to be *normalized*. (The word “normal” is also used to denote this property, but because this word is used to mean several other things, “normalized” is preferred.) Normalized vectors that are all orthogonal to each other are called *orthonormal* vectors. (If the elements of the vectors are from the field of complex numbers, orthogonality and normality are defined in terms of the dot products of a vector with a complex conjugate of a vector.)

A set of vectors that are mutually orthogonal are necessarily linearly independent. A basis for a vector space is often chosen to be an orthonormal set. The set of unit vectors is orthonormal.

The projection of the vector \( x \) onto the vector \( y \) is the vector

\[
\frac{\langle x, y \rangle}{\langle y, y \rangle} y.
\] (5.8)

Subsets of points defined by linear equations are called *flats*. In a \( d \)-dimensional cartesian system (or a vector space of order \( n \)), the flat consisting of the points that satisfy an equation

\[
c_1x_1 + c_2x_2 + \ldots + c_dx_d = b
\] (5.9)

is called a *hyperplane*. Lines and other flat geometric objects can be defined by systems of linear equations. A hyperplane through the origin, that is the set of points satisfying the equation

\[
c_1x_1 + c_2x_2 + \ldots + c_nx_n = 0.
\]

is a vector space.

For the special case of the vector space \( \mathbb{R}^3 \), another useful vector product is the *cross product*. For the vectors

\[
x = (x_1, x_2, x_3) \quad y = (y_1, y_2, y_3)
\]

the cross product, written \( x \times y \), is defined as

\[
x \times y = (x_2y_3 - x_3y_2, x_3y_1 - x_1y_3, x_1y_2 - x_2y_1). \quad (5.10)
\]

The cross product has the following properties, which are immediately obvious from the definition:

1. Self-nilpotency:
   \[ x \times x = 0, \text{ for all } x. \]
2. Anti-commutativity:
\[ x \times y = -y \times x. \]
3. Factoring of scalar multiplication:
\[ ax \times y = a(x \times y) \text{ for real } a. \]
4. Relation of vector addition to addition of cross products:
\[ (x + y) \times z = (x \times z) + (y \times z). \]

The cross product is useful in modeling phenomena in nature, which are often represented as vectors in \( \mathbb{R}^3 \). It is also useful in computer graphics to simulate the behavior of light.

**Matrices**

A matrix is a rectangular array. We speak of the *rows* and *columns* of a matrix. An \( n \times m \) matrix is one with \( n \) rows and \( m \) columns. The number of rows and the number of columns determine the *shape* of the matrix. If the number of rows is the same as the number of columns, the matrix is said to be square; otherwise, it is called nonsquare.

The number of dimensions of an array is often called the *rank* of the array. Thus, a vector is an array of rank 1 and a matrix is an array of rank 2. A scalar has rank 0. When referring to computer software objects, “rank” is generally used in this sense. (This term comes from its use in describing a *tensor*. A rank 0 tensor is a scalar, a rank 1 tensor is a vector, a rank 2 tensor is a *square* matrix, and so on. In our usage referring to arrays, we do not require that the dimensions be equal.) On page 242 we discuss a different meaning of the work “rank”, and one that is more often used in linear algebra.

The elements or components of either a vector or a matrix are elements of a field. We generally assume the elements are real numbers, although sometimes we have occasion to work with matrices whose elements are complex numbers.

We usually use a lower-case letter to represent a vector, and we use the same letter with a single subscript to represent an element of the vector. We usually use an upper-case letter to represent a matrix. To represent an element of the matrix, we use the corresponding lower-case letter with a subscript to denote the row and a second subscript to represent the column. If a nontrivial expression is used to denote the row or the column, we separate the row and column subscripts with a comma.

We also use the notation \( a_j \) to correspond to the \( j^{\text{th}} \) column of the matrix \( A \), and \( a^i_j \) to represent the vector that corresponds to the \( i^{\text{th}} \) row. The objects are vectors, but this notation does not uniquely identify the type of object, because we use the same notation for an element of a vector. The context, however, almost always makes the meaning clear.

The first row is the 1\(^{\text{st}}\) (first) row, and the first column is the 1\(^{\text{st}}\) (first) column. (Again, we remark that computer entities used in some systems to represent matrices and to store elements of matrices as computer data sometimes index the elements beginning with 0. Further, some systems use the
first index to represent the column and the second index to indicate the row. We are not speaking here of the storage order — “row major” versus “column major” — we address that later. Rather, we are speaking of the mechanism of referring to the abstract entities. In image processing, for example, it is common practice to reverse use the first index to represent the column and the second index to represent the row. In the software package PV-Wave, for example, there are two different kinds of two-dimensional objects: arrays, in which the indexing is done as in image processing, and matrices, in which the indexing is done as we have described.)

The \( n \times m \) matrix \( A \) can be written

\[
A = \begin{bmatrix}
a_{11} & a_{12} & \ldots & a_{1m} \\
a_{21} & a_{22} & \ldots & a_{2m} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n1} & a_{n2} & \ldots & a_{nm}
\end{bmatrix},
\]

We also write the matrix \( A \) above as

\[
(a_{ij}),
\]

with the indices \( i \) and \( j \) ranging over \( \{1, 2, \ldots, n\} \) and \( \{1, 2, \ldots, m\} \), respectively. We will sometimes use the notation \( \mathbb{R}^{n \times m} \) to refer to the set of all \( n \times m \) matrices with real elements.

The vector space generated by the columns of the \( n \times m \) matrix \( A \) is of order \( n \) and of dimension \( m \) or less, and is called the column space of \( A \), the range of \( A \), or the manifold of \( A \). This vector space is often denoted by \( \mathcal{V}(A) \) or by \( \text{span}(A) \). (Recall that if \( G \) is a set of vectors, the symbol \( \text{span}(G) \) denotes the vector space generated by the vectors in \( G \).)

The \( a_{ii} \) elements of a matrix are called diagonal elements; an element, \( a_{ij} \), with \( i < j \) is said to be “above the diagonal”, and one with \( i > j \) is said to be “below the diagonal”. The vector consisting of all of the \( a_{ii} \)'s is called the principal diagonal, or just the diagonal. The elements \( a_{i,i+c} \) are called “codiagonals” or “minor diagonals”. These phrases are used with both square and nonsquare matrices.

If all except the principal diagonal elements of matrix are 0, the matrix is called a diagonal matrix. If all elements below the diagonal are 0, the matrix is called an upper triangular matrix; and a lower triangular matrix is defined similarly. If all elements are 0 except \( a_{i,i+c} \) for some small number of integers, \( c_k \), the matrix is called a band matrix (or banded matrix). In many applications \( c_k \in \{-w_l, -w_l + 1, \ldots, -1, 0, 1, \ldots, w_u - 1, w_u\} \). In such a case, \( w_l \) is called the lower band width and \( w_u \) is called the upper band width. These patterned matrices arise in solutions of differential equations and so are very important in applications of linear algebra. Although it is often the case that band matrices are symmetric, or at least have the same number of codiagonals that are nonzero, neither of these conditions always occurs in applications of band matrices. Notice that the terms defined here also apply to nonsquare matrices.
A band matrix with lower and upper band width of 1, and such that all elements $a_{i,i\pm 1}$ are nonzero, is called a “matrix of type 2”. It can be shown that the inverses of certain matrices arising in statistical applications are matrices of type 2.

If $a_{i,i+c_k} = d_{ck}$, where $d_{ck}$ is constant for fixed $c_k$, the matrix is called a Toeplitz matrix:

$$
\begin{bmatrix}
  d_0 & d_1 & d_2 & \cdots & d_{n-1} \\
  d_{-1} & d_0 & d_1 & \cdots & d_{n-2} \\
  & \ddots & \ddots & \ddots & \ddots \\
  d_{-n+1} & d_{-n+2} & d_{-n+3} & \cdots & d_0 \\
\end{bmatrix}
$$

that is, a Toeplitz matrix is a matrix with constant codiagonals. A Toeplitz matrix may or may not be a band matrix (have many 0 codiagonals) and it may or may not be symmetric.

Because the matrices with special patterns are usually characterized by the locations of zeros and nonzeros, we often use an intuitive notation with $X$ and $0$ to indicate the pattern. Thus, a band matrix may be written as

$$
\begin{bmatrix}
  X & X & 0 & \cdots & 0 & 0 \\
  X & X & X & \cdots & 0 & 0 \\
  0 & X & X & \cdots & 0 & 0 \\
  & \ddots & \ddots & \ddots & \ddots & \ddots \\
  0 & 0 & 0 & \cdots & X & X \\
\end{bmatrix}
$$

In this notation $X$ is not the same object each place it occurs. The $X$ and $0$ may also indicate “submatrices”, which we discuss in the section on partitioned matrices.

Computational methods for banded matrices may be of order $n$ or $n^2$, whereas the corresponding computations over full matrices may of order $n^2$ or $n^3$. Such efficiency increases are extremely important. Banded matrices occur in numerical solutions of partial differential equations. Banded Toeplitz matrices occur frequently in time series studies. The covariance matrix in an ARMA($p, q$) process, for example, is a symmetric Toeplitz matrix with $2 \max(p, q)$ nonzero off-diagonal bands.

Matrix Shaping Operators: Transpose

The *transpose* of a matrix is the matrix whose $i^{\text{th}}$ row is the $i^{\text{th}}$ column of the original matrix, and whose $j^{\text{th}}$ column is the $j^{\text{th}}$ row of the original matrix. We use a superscript “$T$” to denote the transpose of a matrix; thus, if $A = (a_{ij})$, then $A^T = (a_{ji})$. (In other literature, the transpose is often denoted by a prime, as in $A' = (a_{ji})$.) If the elements of the matrix are from the field of complex numbers, the *conjugate transpose* is a useful concept. We use a superscript “$H$” to denote the conjugate transpose of a matrix; thus, if $A = (a_{ij})$, then $A^H = (\bar{a}_{ji})$, where $\bar{a}$ represents the conjugate of the complex
number \( a \). (The conjugate transpose is often denoted by an asterisk, as in \( A^* = (\bar{a}_{ji}) \). This notation is more common when a prime is used to denote transpose.)

If \( A = A^T \), \( A \) is said to be **symmetric**. A symmetric matrix is necessarily square. If \( A = A^H \), \( A \) is said to be **Hermitian**. A Hermitian matrix is also necessarily square.

### Matrix Shaping Operators: \( \text{diag} \)

A square diagonal matrix can be specified by listing the diagonal elements with the “\( \text{diag} \)” constructor function that operates on a vector:

\[
\text{diag}(d_1, d_2, \ldots, d_n) = \begin{bmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_n \end{bmatrix}.
\]

(Notice that the argument of \( \text{diag} \) is a vector; that is why there are two sets of parentheses in the expression above.)

### Matrix Shaping Operators: \( \text{vec} \) and \( \text{vech} \)

It is sometimes useful to consider the elements of a matrix to be elements of a single vector. The most common way this is done is to string the columns of the matrix end-to-end into a vector. The “\( \text{vec} \)” function does this:

\[
\text{vec}(A) = (a_{11}^T, a_{21}^T, \ldots, a_{m1}^T),
\]

where \((a_1, a_2, \ldots, a_m)\) are the column vectors of the matrix \( A \). For a symmetric matrix \( A \), with elements \( a_{ij} \), the “\( \text{vech} \)” function stacks the unique elements into a vector:

\[
\text{vech}(A) = (a_{11}, a_{21}, a_{22}, a_{31}, \ldots, a_{m1}, \ldots, a_{mm}).
\]

Henderson and Searle (1979) derive several interesting properties of \( \text{vec} \) and \( \text{vech} \).

### Matrix Operators: \( \text{trace} \)

The sum of the diagonal elements of a square matrix is called the **trace** of the matrix. We use the notation “\( \text{trace}(A) \)” to denote the trace of the matrix \( A \):

\[
\text{trace}(A) = \sum_i a_{ii}.
\]

From the definition we see

\[
\text{trace}(A) = \text{trace}(A^T)
\]
Matrix Operators: det

For an $n \times n$ (square) matrix $A$, consider the product $a_{1j_1}a_{2j_2}\cdots a_{nj_n}$, where $j_1, j_2, \ldots, j_n$ is some permutation of the integers from 1 to $n$. Define a permutation to be even if the number of times that consecutive pairs have a larger first element is an even number, and define the permutation to be odd otherwise. (For example, 1,3,2 is an odd permutation; and 3,2,1 is an even permutation.) Let $\sigma(j_1, j_2, \ldots, j_n) = 1$ if $j_1, j_2, \ldots, j_n$ is an even permutation, and let $\sigma(j_1, j_2, \ldots, j_n) = -1$ otherwise. Then the determinant of $A$, denoted by “det($A$)” is defined by:

$$\det(A) = \sum_{\text{all permutations}} \sigma(j_1, j_2, \ldots, j_n)a_{1j_1}a_{2j_2}\cdots a_{nj_n}. \quad (5.15)$$

The determinant is also sometimes written as $|A|$. The determinant of a triangular matrix is just the product of the diagonal elements.

For an arbitrary matrix, the determinant is rather difficult to compute. The method for computing a determinant is not the one that would arise directly from the definition given above; rather, it involves first decomposing the matrix, as we discuss in later sections. Neither the trace nor the determinant is very often useful in computations; but, although it may not be obvious from their definitions, both objects are very useful in establishing properties of matrices.

From the definition we see

$$\det(A) = \det(A^T) \quad (5.16)$$

Addition and Multiplication of Vectors and Matrices

The elements of a vector or matrix are elements of a field; and, as we have seen, most matrix and vector operations are defined in terms of operations in the field.

A scalar multiple of a matrix is the matrix whose elements are the multiples of the corresponding elements of the original matrix.

Matrix Addition

The sum of two matrices of the same shape is the matrix whose elements are the sums of the corresponding elements of the addends. Addition of matrices is also indicated by “$+$”, as with scalar and vector addition. We assume throughout that writing a sum of matrices, $A + B$, implies that they are of the same shape, that is, that they are conformable for addition.

The matrix additive identity is a matrix with all elements zero. We sometimes denote such a matrix with $n$ rows and $m$ columns as $0_{nm}$, or just as 0.

A useful (and obvious) property of the trace is:

$$\text{trace}(A + B) = \text{trace}(A) + \text{trace}(B) \quad (5.17)$$
Matrix Multiplication (Cayley)

There are various kinds of multiplication of matrices that may be useful. If the number of columns of the matrix $A$, with elements $a_{ij}$, and the number of rows of the matrix $B$, with elements $b_{ij}$, are equal, then the (Cayley) product of $A$ and $B$, is defined as the matrix $C$ with elements

$$c_{ij} = \sum_k a_{ik}b_{kj}.$$  \hspace{1cm} (5.18)

This is the most common type of product, and it is what we refer to by the unqualified phrase “matrix multiplication”.

Matrix multiplication is indicated by juxtaposition, with no intervening symbol for the operation: $C = AB$.

If the matrix $A$ is $n \times m$ and the matrix $B$ is $m \times p$, the product $C = AB$ is $n \times p$:

$$C = A \cdot B$$

Cayley matrix multiplication is a mapping from $\mathbb{R}^{n \times m} \times \mathbb{R}^{m \times p}$ to $\mathbb{R}^{n \times p}$.

We assume throughout that writing a product of matrices $AB$ implies that the number of columns of the first matrix is the same as the number of rows of the second, that is, they are conformable for multiplication in the order given.

It is obvious that while the product $C = AB$ may be well defined, the product $BA$ is defined only if $n = p$, that is, if the matrices $AB$ and $BA$ are square. It is easy to see from the definition of matrix multiplication (5.18) that in general, even for square matrices, $AB \neq BA$. It is also obvious that if $C = AB$, then $B^T A^T$ exists and, in fact, $C^T = B^T A^T$. The product of symmetric matrices is not, in general, symmetric. If (but not only if) $A$ and $B$ are symmetric, then $AB = (BA)^T$.

For a square matrix, its product with itself is defined; and so for a positive integer $k$, we write $A^k$ to mean $k - 1$ multiplications: $AA \cdots A$.

Here, as throughout the field of numerical analysis, we must remember that the definition of an operation, such as matrix multiplication, does not necessarily define a good algorithm for evaluating the operation.

Because matrix multiplication is not commutative, we often use the terms “premultiply” and “postmultiply”, and the corresponding noun forms of these terms. Thus in the product $AB$, we may say $B$ is premultiplied by $A$, or, equivalently, $A$ is postmultiplied by $B$.

Although matrix multiplication is not commutative, it is associative; that is, if the matrices are conformable,

$$A(BC) = (AB)C;$$

and it is distributive over addition; that is,
These properties are obvious from the definition of matrix multiplication.

An $n \times n$ matrix consisting of 1’s along the diagonal and 0’s everywhere else is a multiplicative identity for the set of $n \times n$ matrices and Cayley multiplication. Such a matrix is called the identity matrix of order $n$, and is denoted by $I_n$, or just by $I$. The columns of the identity matrix are unit vectors.

The identity matrix is a multiplicative identity for any matrix so long as the matrices are conformable for the multiplication. If $A$ is $n \times m$, then $I_n A = AI_m = A$. A useful (and obvious) property of the determinant is:

$$\det(AB) = \det(A) \det(B),$$

(5.19)

if $A$ and $B$ are square matrices conformable for multiplication.

Two useful properties of the trace, for the matrices $A$, $B$, and $C$ that are conformable for the multiplications indicated (which means that the products are square), are

$$\text{trace}(AB) = \text{trace}(BA)$$

(5.20)

$$\text{trace}(ABC) = \text{trace}(BCA) = \text{trace}(CAB)$$

(5.21)

These properties are obvious from the definitions of matrix multiplication and the trace.

Matrix Hadamard Multiplication

Three other types of matrix multiplication that are useful are Hadamard multiplication, Kronecker multiplication, and dot product multiplication. Hadamard multiplication is defined for matrices of the same shape as the multiplication of each element of one matrix by the corresponding element of the other matrix. Hadamard multiplication immediately inherits the commutativity, associativity, and distribution over addition of the ordinary multiplication of the underlying field of scalars. Hadamard multiplication is also called array multiplication and element-wise multiplication.

The identity for Hadamard multiplication is the matrix of appropriate shape whose elements are all 1’s.

Matrix Kronecker Multiplication

Kronecker multiplication, denoted by $\otimes$, is defined for any two matrices $A_{n \times m}$ and $B_{p \times q}$ as

$$A \otimes B = \begin{bmatrix}
a_{11}B & a_{12}B & \ldots & a_{1m}B \\
\vdots & \vdots & \ddots & \vdots \\
a_{n1}B & a_{n2}B & \ldots & a_{nm}B
\end{bmatrix}.$$
The Kronecker product of $A$ and $B$ is $np \times mq$. Kronecker multiplication is also called “direct multiplication”. Kronecker multiplication is associative and distributive over addition, but it is not commutative.

The identity for Kronecker multiplication is the $1 \times 1$ matrix with the element 1; that is, it is the same as the scalar 1.

A relationship between the vec function and Kronecker multiplication is

$$\text{vec}(ABC) = (C^T \otimes A)\text{vec}(B),$$

for matrices $A$, $B$, and $C$ that are conformable for the multiplication indicated.

**Matrix Inner Products**

The dot product or inner product of matrices is defined for matrices of the same shape as the sum of the dot products of the vectors formed from the columns of one matrix with vectors formed from the corresponding columns of the other matrix. The dot product of real matrices is a real number, as is the dot product of real vectors. The dot product of the matrices $A$ and $B$ with the same shape is denoted by $A \cdot B$, or $\langle A, B \rangle$, just as the dot product of vectors. For conformable matrices $A$, $B$, and $C$, the following properties of the dot product of matrices are straightforward:

- $\langle A, B \rangle = \langle B, A \rangle$
- $\langle A, B \rangle = \text{trace}(A^T B)$
- $\langle A, A \rangle \geq 0$, with equality only if $A = 0$
- $\langle sA, B \rangle = s\langle A, B \rangle$, for a scalar $s$
- $\langle (A + B), C \rangle = \langle A, C \rangle + \langle B, C \rangle$

Dot products of matrices also obey the Cauchy-Schwarz inequality (compare (5.4), page 231):

$$\langle A, B \rangle \leq (A, A)^{\frac{1}{2}}(B, B)^{\frac{1}{2}},$$

with equality holding only if $A = 0$ or $B = sA$, for some scalar $s$. This is easy to prove by the same argument as used for inequality (5.4) on page 231. (You are asked to write out the details in Exercise 5.3.)

**Multiplication of Matrices and Vectors; Quadratic Forms**

It is often convenient to think of a vector as a matrix with the length of one dimension being 1. This provides for an immediate extension of the definition of matrix multiplication to include vectors as either or both factors. In this scheme, we adopt the convention that a vector corresponds to a column, that is, if $x$ is a vector and $A$ is a matrix, $Ax$ or $x^T A$ may be well-defined; but $Ax^T$ would not represent anything, except in the case when all dimensions are 1. The dot product or inner product, $\langle c, x \rangle$, of the vectors $x$ and $y$ can be
The outer product of the vectors \( x \) and \( y \) is the matrix \( xy^T \). Note that the definition of the outer product does not require the vectors to be of equal length.

A variation of the vector dot product, \( x^TAy \), is called a bilinear form, and the special bilinear form \( x^TAx \) is called a quadratic form. Although in the definition of quadratic form we do not require \( A \) to be symmetric — because for a given value of \( x \) and a given value of the quadratic form, \( x^TAx \), there is a unique symmetric matrix \( A_s \) such that \( x^TA_sx = x^TAx \) — we generally work only with symmetric matrices in dealing with quadratic forms. (The matrix \( A_s \) is \( \frac{1}{2}(A + A^T) \). See Exercise 5.4.) Quadratic forms correspond to sums of squares, and, hence, play an important role in statistical applications.

The invariance of the trace to cyclic permutations (equation (5.21)) is particularly useful in working with quadratic forms. Because the quadratic form itself is a scalar (or a \( 1 \times 1 \) matrix), and because of the invariance, we have

\[
\begin{align*}
    x^TAx &= \text{trace}(x^TAx) \\
           &= \text{trace}(Ax^T).
\end{align*}
\] (5.23) (5.24)

Matrix Rank

The linear dependence or independence of the vectors forming the rows or columns of a matrix is an important characteristic of the matrix. The maximum number of linearly independent vectors (either those forming the rows or the columns) is called the rank of the matrix. (We have used the term “rank” before to denote dimensionality of an array. “Rank” as we have just defined it applies only to a matrix or to a set of vectors. The meaning is clear from the context.) Although some people use the terms “row rank” or “column rank”, the single word “rank” is sufficient because they are the same. It is obvious that the rank of a matrix can never exceed its smaller dimension. Whether or not a matrix has more rows than columns, the rank of the matrix is the same as the dimension of the column space of the matrix.

We use the notation “\( \text{rank}(A) \)” to denote the rank of the matrix \( A \).

If the rank of a matrix is the same as its smaller dimension, we say the matrix is of full rank. In this case we may say the matrix is of full row rank or full column rank. A full rank matrix is also called nonsingular, and one that is not nonsingular is called singular. These words are often restricted to square matrices, and the phrase “full row rank” or “full column rank”, as appropriate, is used to indicate that a nonsquare matrix is of full rank.

In practice, it is not always clear whether a matrix is nonsingular. Because of rounding on the computer, a matrix that is mathematically nonsingular may appear to be singular. We sometimes use the phrase “nearly singular” or “algorithmically singular” to describe such a matrix. In general, the numerical determination of the rank of a matrix is not an easy task.

The rank of the product of two matrices is less than or equal to the lesser of the ranks of the two:
5.1 Basic Vector/Matrix Computations

\[ \text{rank}(AB) \leq \min\{\text{rank}(A), \text{rank}(B)\}. \quad (5.25) \]

From this we see that the rank of an outer product matrix is 1.

A matrix \( A \) such that \( AA = A \) is called an idempotent matrix. An idempotent matrix is either singular or it is the identity matrix.

Inverses

The elements in a set that has an identity with respect to some operation may have inverses with respect to that operation. The only type of matrix multiplication for which an inverse is of common interest is Cayley multiplication of square matrices. The inverse of the \( n \times n \) matrix \( A \) is the matrix \( A^{-1} \) such that

\[ A^{-1}A = AA^{-1} = I_n. \]

A matrix has an inverse if and only if the matrix is square and of full rank.

For scalars, the combined operations of inversion and multiplication is equivalent to the single operation of division. From the analogy with scalar operations, we sometimes denote \( AB^{-1} \) by \( A/B \). Because matrix multiplication is not commutative, we often use the notation "\( \backslash \)" to indication the combined operations of inversion and multiplication on the left; that is, \( B\backslash A \) is the same as \( B^{-1}A \).

As we have indicated, important applications of vectors and matrices involve systems of linear equations:

\[
\begin{align*}
& a_{11}x_1 + a_{12}x_2 + \cdots + a_{1m}x_m = b_1 \\
& a_{21}x_1 + a_{22}x_2 + \cdots + a_{2m}x_m = b_2 \\
& \vdots + \vdots + \vdots \\
& a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nm}x_m = b_n
\end{align*}
\]

An objective with such a system is to determine \( x \)'s that satisfy these equations for given \( a \)'s and \( b \)'s. In vector/matrix notation, these equations are written as

\[ Ax = b, \]

and if \( n = m \) and \( A \) is nonsingular, the solution is

\[ x = A^{-1}b. \]

The solution could also be represented by \( A\backslash b \). We discuss the solution of systems of equations in Section ??, but here we will point out that when we write an expression that involves computations to evaluate it, such as \( A^{-1}b \) or \( A\backslash b \), the form of the expression does not specify how to do the computations.

Often in linear regression analysis we need inverses of various sums of matrices. This is often because we wish to update regression estimates based on additional data or because we wish to delete some observations. If \( A \) and
Let $B$ be full rank matrices of the same size, the following relationships are easy to show. (They are easily proven if taken in the order given.)

\[
\begin{align*}
(I + A^{-1})^{-1} &= A(A + I)^{-1} \\
(A + BB^T)^{-1}B &= A^{-1}B(I + B^TA^{-1}B)^{-1} \\
(A^{-1} + B^{-1})^{-1} &= A(A + B)^{-1}B \\
A - A(A + B)^{-1}A &= B - B(A + B)^{-1}B \\
A^{-1} + B^{-1} &= A^{-1}(A + B)B^{-1} \\
(I + AB)^{-1} &= I - A(I + BA)^{-1}B \\
(I + AB)^{-1}A &= A(I + BA)^{-1}
\end{align*}
\]  

(5.27)

From the relationship $\det(AB) = \det(A)\det(B)$ for square matrices mentioned earlier, it is easy to see that for nonsingular $A$,

\[
\det(A) = 1/\det(A^{-1}).
\]  

(5.28)

For a square matrix $A$, $\det(A) = 0$ if and only if $A$ is singular.

**Partitioned Matrices**

We often find it useful to partition a matrix into submatrices, and we usually denote those submatrices with capital letters with subscripts indicating the relative positions of the submatrices. Hence, we may write

\[
A = \begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix},
\]  

(5.29)

where the matrices $A_{11}$ and $A_{12}$ have the same number of rows, $A_{21}$ and $A_{22}$ have the same number of rows, $A_{11}$ and $A_{21}$ have the same number of columns, and $A_{12}$ and $A_{22}$ have the same number of columns.

The term “submatrix” is also sometimes used to refer to a matrix formed from another one by deleting various rows and columns of the given matrix. In this terminology, $B$ is a submatrix of $A$ if for each element $b_{ij}$ there is an $a_{kl}$ with $k \geq i$ and $l \geq j$, such that $b_{ij} = a_{kl}$; that is, the rows and/or columns of the submatrix are not contiguous in the original matrix.

A submatrix whose principal diagonal elements are elements of the principal diagonal of the given matrix is called a **principal submatrix**. $A_{11}$ is a principal submatrix in the example above, and if $A_{22}$ is square it is also a principal submatrix. Sometimes the term “principal submatrix” is restricted to square submatrices.

A principal submatrix that contains the $(1,1)$ and whose rows and columns are contiguous in the original matrix is called a **leading principal submatrix**. $A_{11}$ is a principal submatrix in the example above.

Multiplication and other operations with matrices, such as transposition, are carried out with their submatrices in the obvious way. Thus,
5.1 Basic Vector/Matrix Computations 245

\[
\begin{bmatrix}
A_{11} & A_{12} & A_{13} \\
A_{21} & A_{22} & A_{23}
\end{bmatrix}^T =
\begin{bmatrix}
A_{11}^T & A_{12}^T \\
A_{12}^T & A_{22}^T \\
A_{13}^T & A_{23}^T
\end{bmatrix},
\]

and, assuming the submatrices are conformable for multiplication,

\[
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix} =
\begin{bmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{bmatrix}.
\]

Sometimes a matrix may be partitioned such that one partition is just a single column or row, that is, a vector or the transpose of a vector. In that case, we may use a notation such as

\[
[X \ y]
\]
or

\[
[X | \ y],
\]

where \(X\) is a matrix and \(y\) is a vector. We develop the notation in the obvious fashion; for example,

\[
[X \ y]^T [X \ y] =
\begin{bmatrix}
X^T X & X^T y \\
y^T X & y^T y
\end{bmatrix}.
\]

(5.30)

Partitioned matrices may also have useful patterns. A “block diagonal” matrix is one of the form

\[
\begin{bmatrix}
X & 0 & \cdots & 0 \\
0 & X & \cdots & 0 \\
\vdots & \ddots & \ddots & \vdots \\
0 & 0 & \cdots & X
\end{bmatrix},
\]

where 0 represents a submatrix with all zeros, and \(X\) represents a general submatrix, with at least some nonzeros. The \(\text{diag}(\cdot)\) function previously introduced for a vector is also defined for a list of matrices:

\[
\text{diag}(A_1, A_2, \ldots, A_k)
\]
denotes the block diagonal matrix with submatrices \(A_1, A_2, \ldots, A_k\) along the diagonal and zeros elsewhere.

**Inverses of Partitioned Matrices**

If \(A\) is nonsingular, and can be partitioned as

\[
A =
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix},
\]

where both \(A_{11}\) and \(A_{22}\) are nonsingular, it is easy to see (Exercise 5.5, page 342) that the inverse of \(A\) is given by
Numerical Linear Algebra

A partitioned matrix, inverse Schur complement partitioned matrix overdetermined linear system consistent system of equations

\[
A^{-1} = \begin{bmatrix}
A_{11}^{-1} + A_{11}^{-1}A_{12}Z^{-1}A_{21}A_{11}^{-1} - A_{11}^{-1}A_{12}Z^{-1} \\
-Z^{-1}A_{21}A_{11}^{-1} & Z^{-1}
\end{bmatrix},
\]  
(5.31)

where \(Z = A_{22} - A_{21}A_{11}^{-1}A_{12}\). In this partitioning \(Z\) is called the **Schur complement** of \(A_{11}\) in \(A\).

If

\[
A = [Xy]^T [Xy]
\]

and is partitioned as in (5.30) on page 245 and \(X\) is of full column rank, then the Schur complement of \(X^T X\) in \([Xy]^T [Xy]\) is

\[
y^T y - y^T X(X^T X)^{-1}X^T y.
\]

This particular partitioning is useful in linear regression analysis, where this Schur complement is the residual sum of squares.

**Determinants of Partitioned Matrices**

If \(A\) is partitioned as

\[
A = \begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix},
\]

and \(A_{11}\) is nonsingular, then

\[
\det(A) = \det(A_{11})\det(A_{22} - A_{21}A_{11}^{-1}A_{12}).
\]  
(5.32)

This result is obtained by first observing that

\[
\det \begin{bmatrix}
A_{11} & 0 \\
A_{21} & A_{22}
\end{bmatrix} = \det \begin{bmatrix}
A_{11} & A_{12} \\
0 & A_{22}
\end{bmatrix} = \det(A_{11})\det(A_{22}).
\]

**Linear Systems**

Often in mathematical modeling applications, the number of equations in the system (5.26) is not equal to the number of variables. If \(n > m\) and \(\text{rank}([A \mid b]) > \text{rank}(A)\), the system is said to be **overdetermined**. There is no \(x\) that satisfies such a system, but approximate solutions are useful. We discuss approximate solutions of such systems in Section 5.2.7 and in Chapters 7 and ??.

A system (5.26) for which

\[
\text{rank}([A \mid b]) = \text{rank}(A)
\]

is said to be **consistent**. A consistent system has a solution. Furthermore, any system admitting a solution is consistent. The square system in which \(A\) is nonsingular, for example, is clearly consistent.
A consistent system in which \( n < m \) is said to be \textit{underdetermined}. For such a system there will be more than one solution. In fact, there will be infinitely many solutions, because if the vectors \( x_1 \) and \( x_2 \) are solutions, the vector \( wx_1 + (1-w)x_2 \) is likewise a solution for any scalar \( w \).

Underdetermined systems arise in analysis of variance in statistics, and it is useful to have a compact method of representing the solution to the system. It is also desirable to identify a unique solution that has some kind of optimal properties.

\textbf{Generalized Inverses}

Suppose the system \( Ax = b \) is consistent, and \( A^\perp \) is any matrix such that \( AA^\perp A = A \). Then \( x = A^\perp b \) is a solution to the system. Furthermore, if \( Gb \) is any solution, then \( AGA = A \). The former statement is true because if \( AA^\perp A = A \), then \( AA^\perp Ax = Ax \) and since \( Ax = b \), \( AA^\perp b = b \). The latter statement can be seen by the following argument. Let \( a_j \) be the \( j \)-th column of \( A \). The \( m \) systems of \( n \) equations, \( Ax = a_j, j = 1, \ldots, m \), all have solutions (a vector with 0’s in all positions except the \( j \)-th position, in which is a 1). Now, if \( Gb \) is a solution to the original system, then \( Ga_j \) is a solution to the system \( Ax = a_j \). So \( AGa_j = a_j \) for all \( j \); hence \( AGA = A \).

A matrix \( A^\perp \) such that \( AA^\perp A = A \) is called a \textit{generalized inverse}, an \textit{inner pseudoinverse}, or a \( g_1 \) inverse of \( A \). A \( g_1 \) inverse is not unique, but if we impose three more conditions we arrive at a unique matrix, denoted by \( A^+ \), that yields a solution that has some desirable properties. (For example, the length of \( A^+ b \), in the sense of the Euclidean distance, is the smallest of any solution to \( Ax = b \). See Section 5.2.7.)

For matrix \( A \), the conditions that yield a unique generalized inverse, called the \textit{Moore-Penrose inverse}, and denoted by \( A^+ \), are

1. \( AA^+ A = A \) (that is, it is an inner pseudoinverse or a \( g_1 \) inverse).
2. \( A^+ A A^+ = A^+ \). (A matrix \( A^+ \) that satisfies this condition is called an \textit{outer pseudoinverse}. A \( g_1 \) inverse that satisfies this condition is called a \( g_2 \) inverse, and is denoted by \( A^* \).)
3. \( A^+ A \) is symmetric.
4. \( AA^+ \) is symmetric.


For any matrix \( A \), the Moore-Penrose inverse exists and is unique. If \( A \) is nonsingular, obviously \( A^+ = A^{-1} \).

If \( A \) is partitioned as
A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},

then, similarly to equation (5.31), a generalized inverse of $A$ is given by

$$A^{-} = \begin{bmatrix} A_{11}^{-} + A_{11}^{-}A_{12}Z^{-}A_{21}^{-}A_{11}^{-} - A_{11}^{-}A_{12}Z^{-} \\ -Z^{-}A_{21}^{-}A_{11}^{-} Z^{-} \end{bmatrix},$$

(5.33)

where $Z = A_{22} - A_{21}A_{11}^{-}A_{12}$ (see Exercise 5.6, page 342).

Null Spaces and Orthogonal Complements

If $A$ is an $n \times m$ matrix of rank $r$, with $r < \min(n,m)$, the vector space generated by all solutions, $x$, of the system

$$Ax = 0$$

is called the null space of $A$. The dimension of the null space is $n - r$. We denote the null space of $A$ by $\mathcal{N}(A)$. It is the orthogonal complement of $\mathcal{V}(A)$. For any matrix $B$ whose columns are in $\mathcal{N}(A)$, $A^T B = 0$, and $B^T A = 0$.

All vectors in the null space of the matrix $A$ are orthogonal to all vectors in the column space of $A$. In general, two vector spaces $\mathcal{V}_1$ and $\mathcal{V}_2$ are said to be orthogonal, written $\mathcal{V}_1 \perp \mathcal{V}_2$, if each vector in one is orthogonal to every vector in the other. The intersection of two orthogonal vector spaces consists only of the zero vector. If $\mathcal{V}_1 \perp \mathcal{V}_2$ and $\mathcal{V}_1 \oplus \mathcal{V}_2 = \mathbb{R}^n$, then $\mathcal{V}_2$ is called the orthogonal complement of $\mathcal{V}_1$, and this is written as $\mathcal{V}_2 = \mathcal{V}_1^\perp$. The null space of the matrix $A$ is the orthogonal complement of $\mathcal{V}(A)$:

$$\mathcal{N}(A) \oplus \mathcal{V}(A) = \mathbb{R}^n.$$  

(5.34)

Instead of defining orthogonality in terms of dot products, we can define it more generally in terms of a bilinear form. If the bilinear form $x^T A y = 0$, we say $x$ and $y$ are orthogonal with respect to the matrix $A$. In this case we often use a different term, and say that the vectors are conjugate with respect to $A$. The usual definition of orthogonality in terms of a dot product is equivalent to the definition in terms of a bilinear form in the identity matrix.

We will encounter properties similar to orthogonality of vectors when we consider more general orthogonal systems in Section 8.3.

A matrix whose rows or columns constitute a set of orthonormal vectors is said to be an orthogonal matrix. If $Q$ is an $n \times m$ matrix, then $QQ^T = I_n$ if $n \leq m$, and $Q^T Q = I_m$ if $n \geq m$. Such a matrix is also called a unitary matrix. (For matrices whose elements are complex numbers, a matrix is said to be unitary if the matrix times its conjugate transpose is the identity, that is, if $QQ^H = I$. Both of these definitions are in terms of orthogonality of the rows or columns of the matrix.)

The determinant of a square orthogonal matrix is 1.
The definition given above for orthogonal matrices is sometimes relaxed to require only that the columns or rows be orthogonal (rather than also normal). If normality is not required, the determinant is not necessarily 1. If \( Q \) is a matrix that is “orthogonal” in this weaker sense of the definition, and \( Q \) has more rows than columns, then

\[
Q^TQ = \begin{bmatrix}
X & 0 & \cdots & 0 \\
0 & X & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & X
\end{bmatrix}.
\]

The definition of orthogonality is also sometimes made more restrictive to require that the matrix be square.

In this text we use the term orthogonal matrix only to refer to a matrix whose columns are orthonormal.

**Permutation Matrices**

In the course of performing computations on a matrix, it is often desirable to interchange the rows or columns of the matrix. Interchange of two rows of a matrix can be accomplished by premultiplying the matrix by a matrix that is the identity, with its same two rows interchanged. For example,

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
a_{11} & a_{12} & a_{13} \\
a_{21} & a_{22} & a_{23} \\
a_{31} & a_{32} & a_{33} \\
a_{41} & a_{42} & a_{43}
\end{bmatrix}
= 
\begin{bmatrix}
a_{11} & a_{12} & a_{13} \\
a_{31} & a_{32} & a_{33} \\
a_{21} & a_{22} & a_{23} \\
a_{41} & a_{42} & a_{43}
\end{bmatrix}.
\]

The first matrix in the expression above is called an *elementary permutation matrix*. It is the identity matrix with its second and third rows (or columns) interchanged. An elementary permutation matrix that is the identity with the \( j \)-th and \( k \)-th rows interchanged is denoted by \( E_{jk} \). That is, \( E_{jk} \) is the identity, except the \( j \)-th row is \( e_k^T \) and the \( k \)-th row is \( e_j^T \). Note \( E_{jk} = E_{kj} \). Thus, for example,

\[
E_{23} = E_{32} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}.
\]

Premultiplying a matrix \( A \) by a (conformable) \( E_{jk} \) results in an interchange of the \( j \)-th and \( k \)-th rows of \( A \) as we see above. Postmultiplying a matrix \( A \) by a (conformable) \( E_{jk} \) results in an interchange of the \( j \)-th and \( k \)-th columns of \( A \):

\[
\begin{bmatrix}
a_{11} & a_{12} & a_{13} \\
a_{21} & a_{22} & a_{23} \\
a_{31} & a_{32} & a_{33} \\
a_{41} & a_{42} & a_{43}
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 \\
0 & 0 & 1 \\
0 & 1 & 0
\end{bmatrix}
= 
\begin{bmatrix}
a_{11} & a_{13} & a_{12} \\
a_{21} & a_{23} & a_{22} \\
a_{31} & a_{33} & a_{32} \\
a_{41} & a_{43} & a_{42}
\end{bmatrix}.
\]
It is easy to see from the definition that an elementary permutation matrix is symmetric and orthogonal. A more general permutation matrix can be built as the product of elementary permutation matrices. Such a matrix is not necessarily symmetric, but its transpose is also a permutation matrix. A permutation matrix is orthogonal.

**Projection Matrices**

For a given vector space \( \mathcal{V} \), a symmetric idempotent matrix \( A \) whose columns span \( \mathcal{V} \) is said to be an orthogonal projection matrix onto \( \mathcal{V} \). (Note that an orthogonal projection matrix is not an orthogonal matrix, unless it is the identity matrix.) An orthogonal projection matrix is also just called a “projection matrix”. A matrix is a projection matrix if and only if it is symmetric and idempotent.

It is easy to see that for any vector \( x \), if \( A \) is a projection matrix, the vector \( Ax \) is in \( \mathcal{V} \) and vector \( x - Ax \) is in \( \mathcal{V}^\perp \) (the vectors \( Ax \) and \( x - Ax \) are orthogonal). Stating this in alternate notation, if \( A \) is an orthogonal projection matrix and \( A \in \mathbb{R}^{n \times n} \), then \( A \) is a mapping from \( \mathbb{R}^n \) onto \( \mathcal{V}(A) \), and \( I - A \) is a projection matrix that is a mapping from \( \mathbb{R}^n \) onto the orthogonal complement, \( \mathcal{N}(A) \). (These spaces are such that \( \mathcal{V}(A) \oplus \mathcal{N}(A) = \mathbb{R}^n \).)

Because a projection matrix is idempotent, the matrix projects any of its columns onto itself, and of course it projects the full matrix onto itself: \( AA = A \). More generally if \( x \) and \( y \) are vectors in \( \text{span}(A) \) and \( a \) is a scalar, then

\[
A(ax + y) = ax + y.
\]

(To see this, we merely represent \( x \) and \( y \) as linear combinations of columns (or rows) of \( A \) and substitute in the equation.)

An important matrix that occurs in linear regression analysis,

\[
X(X^T X)^{-1}X^T,
\]

is a projection matrix. This matrix is called the “hat matrix” because it projects the observed response vector, often denoted by \( y \), onto a predicted response vector, often denoted by \( \hat{y} \):

\[
\hat{y} = X(X^T X)^{-1}X^T y
\]

The trace of the matrix \( X(X^T X)^{-1}X^T \) has an interesting property:

\[
\text{trace}(X(X^T X)^{-1}X^T) = \text{number of columns of } X.
\]

(5.35)

(This can be see by use of the invariance of the trace to cyclic permutations (equation (5.21)). In linear regression analysis \( \text{trace}(X(X^T X)^{-1}X^T) \) is the regression degrees of freedom. The “regression sum of squares” is just \( y^T X(X^T X)^{-1}X^T y \).

The complementary projection matrix,
\[ I - X(X^TX)^{-1}X^T, \]

also has interesting properties that relate to linear regression analysis. In geometrical terms, this matrix projects a vector from the space spanned by the columns of \( X \) onto a set of vectors that constitute what is called the residual vector space. Its trace is the number of rows in \( X \) (that is, the number of observations) minus the regression degrees of freedom. This quantity is the “residual degrees of freedom” (unadjusted).

These two projection matrices partition the total sums of squares:

\[
y^T y = y^T (X(X^TX)^{-1}X^T) y + y^T (I - X(X^TX)^{-1}X^T) y.
\]

**Sign Vector**

An often-useful vector is the *sign vector*, which is formed from signs of the elements of a given vector. It is denoted by \( \text{sign}(\cdot) \), and defined by

\[
\text{sign}(x)_i = \begin{cases} 
1 & \text{if } x_i > 0, \\
0 & \text{if } x_i = 0, \\
-1 & \text{if } x_i < 0.
\end{cases}
\]

**Vandermonde Matrix**

The *Vandermonde matrix* is an \( n \times m \) matrix with orthogonal columns that are defined by monomials:

\[
V_{n \times m} = \begin{bmatrix} 
1 & x_1 & x_1^2 & \cdots & x_1^{m-1} \\
1 & x_2 & x_2^2 & \cdots & x_2^{m-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & x_n & x_n^2 & \cdots & x_n^{m-1} 
\end{bmatrix},
\]

where \( x_i \neq x_j \) if \( i \neq j \).

The Vandermonde matrix is useful in forming orthogonal vectors that correspond to orthogonal polynomials.

**Eigenanalysis**

If \( A \) is an \( n \times n \) (square) matrix, \( v \) is a vector not equal to 0, and \( \lambda \) is a scalar, such that

\[ Av = \lambda v, \]

then \( v \) is called an *eigenvector* of the matrix \( A \) and \( \lambda \) is called an *eigenvalue* of the matrix \( A \). Some authors restrict the definition of an eigenvalue to real values that satisfy (5.36). An eigenvalue is also called a *latent root*, a *characteristic value*, or a *proper value*, and similar synonyms exist for an eigenvector.

An eigenvalue is also sometimes called a *singular value*, but the latter term
has a different meaning that we will use in this book. (The absolute value of an eigenvalue is a singular value, but singular values are also defined for non-square matrices.)

A real matrix may have complex eigenvalues (and, hence, eigenvectors). If \( \lambda \) is an eigenvalue and \( v \) is a corresponding eigenvector for a real matrix \( A \), we see immediately from the definition the following properties.

- \( cv \) is an eigenvector of \( A \), where \( c \) is any nonzero scalar.
- \( c\lambda \) is an eigenvalue of \( cA \), where \( c \) is any nonzero scalar.
- \( \lambda \) is an eigenvalue of \( A^T \) (the eigenvectors of \( A^T \), however, are not necessarily the same as those of \( A \)).
- \( 1/\lambda \) is an eigenvalue of \( A^{-1} \), if \( A \) is nonsingular.
- \( \lambda^2 \) is an eigenvalue of \( A^2 \).
- \( \lambda \) is real if \( A \) is symmetric.
- \( \bar{\lambda} \) is an eigenvalue of \( A \) (where \( \bar{\lambda} \) is the complex conjugate of \( \lambda \)).
- \( \lambda\bar{\lambda} \) is an eigenvalue of \( A^T A \).

It is often desirable to scale an eigenvector \( v \) so that \( v^Tv = 1 \). Such a normalized eigenvector is also called a unit eigenvector.

The set of all the eigenvalues of a matrix is called the spectrum of the matrix. It is possible that a given eigenvalue may have more than one associated eigenvector that are linearly independent of each other. (For example, the identity matrix has only one unique eigenvalue, namely 1; but any vector is an eigenvector, and so the number of linearly independent eigenvectors is equal to the number of rows or columns of the identity.) If a single eigenvalue has \( m \) associated linearly independent eigenvectors, we say the eigenvalue has multiplicity \( m \). (There is another type of multiplicity, so we sometimes refer to the type defined here as algebraic multiplicity.) We often use the phrase “number of nonzero eigenvalues” to mean the sum of the multiplicities, instead of the cardinality of the spectrum, the number of unique values.

Rewriting the definition (5.36) as \( (A - \lambda I)v = 0 \), we see that in order for \( v \) to be nonnull, \( (A - \lambda I) \) must be singular, and hence

\[
\det(A - \lambda I) = 0, \tag{5.37}
\]

which is a polynomial of degree \( n \) in \( \lambda \). This polynomial is called the characteristic equation, and an eigenvalue of \( A \) is a root of the polynomial. An eigenvalue with multiplicity \( m \) occurs as a root of the characteristic equation \( m \) times.

Because most of the matrices in statistical applications are real, in the following we will generally restrict our attention to real matrices. It is important to note that the eigenvalues and eigenvectors of a real matrix are not necessarily real. The eigenvalues of a symmetric real matrix are real, however. This is easily seen by noting that if \( A \) is symmetric, the eigenvalues of \( A^TA \) are the eigenvalues of \( A^2 \), which from the definition, are obviously nonnegative.

There are some important facts about the numbers of eigenvalues and the properties of eigenvectors that are rather more difficult to see. One fact is that
the number of nonzero eigenvalues is equal to the rank of the matrix. (That is, the sum of the multiplicities is equal to the rank of the matrix.)

If \( V \) is a matrix whose columns correspond to the eigenvectors of \( A \) and \( A \) is a diagonal matrix whose entries are the eigenvalues corresponding to the columns of \( V \), then from equation (5.36) we have

\[
AV = VA
\]

or

\[
A = VΛV^{-1}.
\] (5.38)

Expression (5.38) represents a diagonal factorization of the matrix \( A \). This representation is sometimes called the similar canonical form of \( A \).

Not all matrices can be factored as in (5.38). If a matrix can be factored as in (5.38), it is called a simple matrix or a regular matrix; a matrix that cannot be factored in that way is called a deficient matrix or a defective matrix. A simple matrix is said to be diagonalizable.

Any symmetric matrix or any matrix all of whose eigenvalues are unique is simple. For a matrix to be simple, however, it is not necessary that it either be symmetric or have all unique eigenvalues.

A necessary and sufficient condition for a matrix to be simple can be stated in terms of the unique eigenvalues and their multiplicities. This is called the “diagonalizability theorem”. Suppose for the \( n \times n \) matrix \( A \), the distinct eigenvalues \( λ_1, λ_2, \ldots, λ_k \) have multiplicities \( m_1, m_2, \ldots, m_k \). If, for \( i = 1, \ldots, k \),

\[
\text{rank}(A - λ_iI) = n - m_i
\]

then \( A \) is simple, and this condition is also necessary for \( A \) to be simple. Most matrices encountered in statistics applications are simple, that is, diagonalizable.

The factorization (5.38) implies that the eigenvectors of a simple matrix are linearly independent.

From this factorization (5.38) we also see that the determinant of a simple matrix is equal to the product of the eigenvalues.

The eigenvectors corresponding to an eigenvalue with multiplicity \( m \) can be chosen to be orthogonal to each other. To see this, we first note that the \( m \) eigenvectors are linearly independent of each other (else the eigenvalue is not of multiplicity \( m \)), and so any vector in the space spanned by these vectors is an eigenvector. A set of \( m \) linearly independent vectors can be transformed into a set of \( m \) orthogonal vectors that span the same space (for example, by Gram-Schmidt transformations, as in equation (5.56), page 262).

**Eigenanalysis of Symmetric Matrices**

In the case of a symmetric matrix \( A \), any eigenvectors corresponding to distinct eigenvalues are also orthogonal. This is easily seen by assuming that \( λ_1 \)
and $\lambda_2$ are unequal eigenvalues with corresponding eigenvectors $v_1$ and $v_2$. Now consider $v_1^T v_2$. Multiplying this by $\lambda_2$, we get
\[
\lambda_2 v_1^T v_2 = v_1^T A v_2 = v_2^T A v_1 = \lambda_1 v_1^T v_2 = \lambda_1 v_1^T v_2.
\]
Because $\lambda_1 \neq \lambda_2$, we have $v_1^T v_2 = 0$. For a symmetric matrix, therefore, the $V$ in (5.38) can be chosen to be orthogonal, and the similar canonical form (5.38) for the symmetric nonsingular matrix $A$ can be written as
\[
A = VAV^T, \tag{5.39}
\]
where $VV^T = V^TV = I$.

When $A$ is symmetric, and the eigenvectors $v_i$ are chosen to be orthonormal,
\[
I = \sum_i v_i v_i^T,
\]
so
\[
A = A \sum_i v_i v_i^T = \sum_i Av_i v_i^T = \sum_i \lambda_i v_i v_i^T. \tag{5.40}
\]

This representation is called the spectral decomposition of the symmetric matrix $A$. The $\lambda_i$ and $v_i$ can be labeled so that $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$.

The spectral decomposition also applies to powers of $A$:
\[
A^k = \sum_i \lambda_i^k v_i v_i^T,
\]
where $k$ is an integer. If $A$ is nonsingular, $k$ can be negative in the expression above.

Equation (5.40) yields important facts about quadratic forms in $A$. Because $V$ is of full rank an arbitrary vector $x$ can be written as $Vc$ for some vector $c$. Therefore, for the quadratic form $x^T Ax$ we have
\[
x^T Ax = x^T \sum_i \lambda_i v_i v_i^T x = \sum_i c^T V^T v_i v_i^T V c \lambda_i = \sum_i c_i^2 \lambda_i.
\]

This immediately gives, for the maximum eigenvalue of $A$, $\lambda_n$, the inequality

\[
x^T Ax \leq n \lambda_n.
\]
5.1 Basic Vector/Matrix Computations

\[ x^T A x \leq \lambda_n c^T c. \]

Further, if \( x \neq 0 \), using \( x^T x = c^T c \), we have the important inequality,

\[ \frac{x^T A x}{x^T x} \leq \lambda_n. \]

Equality is achieved if \( x \) is the eigenvector corresponding to \( \lambda_n \), so we have

\[ \max_{x \neq 0} \frac{x^T A x}{x^T x} = \lambda_n. \]  (5.41)

**Singular Values and the Singular Value Decomposition**

An additional factorization applicable to nonsquare matrices is the *singular value decomposition* (SVD). For the \( n \times m \) matrix \( A \), this factorization is

\[ A = U \Sigma V^T, \]  (5.42)

where \( U \) is an \( n \times n \) orthogonal matrix, \( V \) is an \( m \times m \) orthogonal matrix, and \( \Sigma \) is an \( n \times m \) diagonal matrix with nonnegative entries. The elements on the diagonal of \( \Sigma \), \( \sigma_i \), are called the *singular values* of \( A \).

For a matrix with more rows than columns, in an alternate definition of the singular value decomposition, the matrix \( U \) is \( n \times m \) with orthogonal columns, and \( \Sigma \) is an \( m \times m \) diagonal matrix with nonnegative entries. Likewise, for a matrix with more columns than rows, the singular value decomposition can be defined as above but with the matrix \( V \) being \( m \times n \) with orthogonal columns, and \( \Sigma \) being \( m \times m \) and diagonal with nonnegative entries.

The general matrix \( A \) also has a spectral decomposition in terms of the singular values and outer products of the columns of the factor matrices:

\[ A = \sum_i \sigma_i u_i v_i^T, \]  (5.43)

with \( \sigma_1 \leq \sigma_2 \leq \cdots \leq \sigma_{\max(n,m)} \). The spectral decomposition follows immediately from equation (5.42).

All matrices have a factorization of the form (5.42). Forming the diagonal matrix \( \Sigma^T \Sigma \) or \( \Sigma \Sigma^T \), and using the factorization in equation (5.38), it is easy to see that the nonzero singular values of \( A \) are the square roots of the nonzero eigenvalues of symmetric matrix \( A^T A \) (or \( AA^T \)). If \( A \) is square, the singular values are the absolute values of the eigenvalues. We discuss this decomposition and other factorizations in Section 5.2.2, beginning on page 283.

**Similarity Transformations**

Two \( n \times n \) matrices, \( A \) and \( B \), are said to be *similar* if there exists a nonsingular matrix \( P \) such that
The transformation in (5.44) is called a similarity transformation. It is clear from this definition that the similarity relationship is both commutative and transitive. We see from equation (5.38) that a matrix $A$ with eigenvalues $\lambda_1, \ldots, \lambda_n$ is similar to the matrix $\text{diag}(\lambda_1, \ldots, \lambda_n)$.

If $A$ and $B$ are similar, as in (5.44), then

\[ B - \lambda I = P^{-1}BP - \lambda P^{-1}P = A - \lambda I, \]

and, hence, $A$ and $B$ have the same eigenvalues. This fact also follows immediately from the transitivity of the similarity relationship and the fact that a matrix is similar to the diagonal matrix formed from its eigenvalues.

An important type of similarity transformation is based on an orthogonal matrix. If $Q$ is orthogonal and

\[ A = Q^TBQ, \]

$A$ and $B$ are said to be orthogonally similar.

**Positive Definiteness**

A symmetric matrix $A$ such that for any (conformable) vector $x \neq 0$, the quadratic form

\[ x^T Ax > 0, \]

is called a positive definite matrix. A positive definite matrix is necessarily nonsingular. There are two related terms, positive semidefinite matrix and nonnegative definite matrix, which are not used consistently in the literature. In this text, we use the term nonnegative definite matrix for any symmetric matrix $A$ for which for any (conformable) vector $x$, the quadratic form $x^T Ax$ is nonnegative, that is,

\[ x^T Ax \geq 0. \quad (5.45) \]

(Some authors call this “positive semidefinite”, but other authors use the term “positive semidefinite” to refer to a nonnegative definite matrix that is not positive definite. We will generally avoid the term “semidefinite”.)

From the definition (5.45), it is easy to see that for any matrix $B$, $A = B^TB$ is nonnegative definite.

There are several other properties that follow immediately from the definition. It is obvious that all diagonal elements of a positive definite matrix are positive. Furthermore, any square submatrix whose principal diagonal is a subset of the principal diagonal of a positive definite matrix is positive definite, and similarly for nonnegative definite matrices. In particular, any square principal submatrix of a positive definite matrix is positive definite. It is easy to show that these facts follow from the definition by considering a vector
x with zeros in all positions except those corresponding to the submatrix in question. For example, to see that all diagonal elements of a positive definite matrix are positive, assume the \((i, i)\) element is not positive, and then consider the vector \(x\) to consist of all zeros except for a 1 in the \(i\)th position. It is easy to see that the quadratic form is not quadratic, so the assumption that the \((i, i)\) element is not positive leads to a contradiction.

It is easy to see that a diagonal matrix is positive (nonnegative) definite if and only if all of the diagonal elements are positive (nonnegative).

Furthermore, any symmetric (real) matrix is positive (nonnegative) definite if and only if all of its eigenvalues are positive (nonnegative). We can see this using the factorization (5.39) of a symmetric matrix. One factor is the diagonal matrix \(A\) of the eigenvalues and the other factors are orthogonal. Hence, for any \(x\), we have \(x^T Ax = x^T V \Lambda V^T x = y^T Ay\), where \(y = V^T x\); and so

\[x^T Ax > (\geq) 0\]

if and only if

\[y^T Ay > (\geq) 0.\]

The determinant of a positive definite matrix is positive.

A sufficient condition for a symmetric matrix to be positive definite is that the determinant of each of the leading principal submatrices be positive. To see this, first let the \(n \times n\) symmetric matrix \(A\) be partitioned as

\[
A = \begin{bmatrix}
A_{n-1} & a \\
a^T & a_{nn}
\end{bmatrix},
\]

and assume that \(A_{n-1}\) is positive definite and that \(\det(A) > 0\). From equation 5.32,

\[
\det(A) = \det(A_{n-1})(a_{nn} - a^T A_{n-1}^{-1} a)
\]

Because \(A_{n-1}\) is positive definite, \(\det(A_{n-1}) > 0\), and so \((a_{nn} - a^T A_{n-1}^{-1} a) > 0\); hence, the \(1 \times 1\) matrix \((a_{nn} - a^T A_{n-1}^{-1} a)\) is positive definite. That any matrix whose leading principal submatrices have positive determinants follows from this by induction, beginning with a 2 \(\times\) 2 matrix.

Another important fact that can be shown by induction is that for any symmetric positive definite \(n \times n\) matrix \(A\), there exists a nonsingular upper triangular matrix \(T\), such that \(A = T^T T\).

**Norms**

For a set of objects \(S\) that has an addition-type operator, \(+_S\), a corresponding additive identity, \(0_S\), and a scalar multiplication, that is, a multiplication of the objects by a real (or complex) number, a norm is a function, \(\|\cdot\|\), from \(S\) to \(\mathbb{R}\) that satisfies the following three conditions.

1. Nonnegativity and mapping of the identity:
   - if \(x \neq 0_S\), then \(\|x\| > 0\), and \(\|0_S\| = 0\)
2. Relation of scalar multiplication to real multiplication:
\[ \|ax\| = |a| \|x\| \text{ for real } a \]

3. Triangle inequality:
\[ \|x + y\| \leq \|x\| + \|y\| \]

(If property 1 is relaxed to require only \( \|x\| \geq 0 \), the function is called a seminorm.) Sets of various types of objects (functions, for example) can have norms, but our interest in the present context is in norms for vectors and matrices. For vectors, \( 0_S \) is the zero vector (of the appropriate length) and \( +_S \) is vector addition (which implies that the vectors are of the same length).

The triangle inequality suggests the origin of the concept of a norm. It clearly has its roots in vector spaces. For some types of objects the norm of an object may be called its “length” or its “size”. (Recall the ambiguity of “length” of a vector that we mentioned at the beginning of this chapter.)

There are many norms that could be defined for vectors. One type of norm is called an \( L_p \) norm, often denoted as \( \| \cdot \|_p \). For \( p \geq 1 \), it is defined as
\[
\|x\|_p = \left( \sum_i |x_i|^p \right)^{1/p}
\]

This is also sometimes called the Minkowski norm.

It is easy to see that the \( L_p \) norm satisfies the first two conditions above. For general \( p \geq 1 \) it is somewhat more difficult to prove the triangular inequality (which for the \( L_p \) norms is also called the Minkowski inequality), but for some special cases it is straightforward, as we see below.

The most common \( L_p \) norms, and in fact, the most commonly used vector norms, are:

- \( \|x\|_1 = \sum_i |x_i| \), also called the Manhattan norm because it corresponds to sums of distances along coordinate axes, as one would travel along the rectangular street plan of Manhattan.
- \( \|x\|_2 = \sqrt{\sum_i x_i^2} = \sqrt{\langle x, x \rangle} \), also called the Euclidean norm, or the vector length. This is the square root of the inner product of the vector with itself.
- \( \|x\|_\infty = \max_i |x_i| \), also called the max norm or the Chebyshev norm.

The \( L_\infty \) norm is defined by taking the limit in an \( L_p \) norm. An \( L_p \) norm is also called a \( p \)-norm, or 1-norm, 2-norm, or \( \infty \)-norm in those special cases.

The triangle inequality is obvious for the \( L_1 \) and \( L_\infty \) norms. For the \( L_2 \) norm it can be shown using the Cauchy-Schwarz inequality (5.4), page 231.

The triangle inequality for the \( L_2 \) norm on vectors is
\[
\sqrt{\sum (x_i + y_i)^2} \leq \sqrt{\sum x_i^2} + \sqrt{\sum y_i^2}
\]

or
\[
\sum (x_i + y_i)^2 \leq \sum x_i^2 + 2\sqrt{\sum x_i^2 \sum y_i^2} + \sum y_i^2.
\]
Now,
\[ \sum (x_i + y_i)^2 = \sum x_i^2 + 2 \sum x_i y_i + \sum y_i^2 , \]
and by the Cauchy-Schwarz inequality,
\[ \sum x_i y_i \leq \sqrt{\sum x_i^2} \sqrt{\sum y_i^2} , \]
so the triangle inequality follows.

It is easy to see that the \( L_p \) norms have the relationship
\[ \| x \|_\infty \leq \| x \|_2 \leq \| x \|_1, \]
(5.48)
for any vector \( x \). More generally, for given \( x \) and \( p \geq 1 \), \( \| x \|_p \) is a nonincreasing function of \( p \).

A generalization of the \( L_p \) vector norm is the \textit{weighted} \( L_p \) vector norm defined by
\[ \| x \|_p = \left( \sum_i |x_i|^p w_i \right)^{\frac{1}{p}} , \]
(5.49)
where \( w_i \geq 0 \).

The \( L_2 \) norm of a vector is the square root of the quadratic form of the vector with respect to the identity matrix. A generalization of the \( L_2 \) vector norm, called an \textit{elliptic norm}, is defined for the vector \( x \) as the square root of the quadratic form \( x^T Ax \), for any symmetric positive-definite matrix \( A \). It is easy to see that \( \sqrt{x^T Ax} \) satisfies the definition of a norm given earlier. If \( A \) is a diagonal matrix with elements \( w_i \geq 0 \), the elliptic norm is the weighted \( L_2 \) norm of equation 5.49.

**Matrix Norms**

A matrix norm is required to have another property in addition to the three general properties on page 257 that define a norm in general. A matrix norm must also satisfy the \textit{consistency property}:

4. \( \| AB \| \leq \| A \| \| B \| \),

where \( AB \) represents the usual Cayley product of the conformable matrices \( A \) and \( B \).

A matrix norm is often defined in terms of a vector norm. Given the vector norm \( \| \cdot \|_v \), the matrix norm \( \| \cdot \|_M \) \textit{induced} by \( \| \cdot \|_v \) is defined by
\[ \| A \|_M = \max_{x \neq 0} \frac{\| Ax \|_v}{\| x \|_v} . \]
(5.50)
It is easy to see that an induced norm is indeed a matrix norm (i.e., that it satisfies the consistency property). We usually drop the \( v \) or \( M \) subscript and the notation \( \| \cdot \| \) is overloaded to mean either a vector or matrix norm.
Matrix norms are somewhat more complicated than vector norms because, for matrices that are not square, there is a dependence of the definition of the norm on the shape of the matrix.

The induced norm of $A$ given in equation (5.50) is sometimes called the *maximum magnification* by $A$. The expression looks very similar to the maximum eigenvalue, and indeed it is in some cases.

For any vector norm and its induced matrix norm it is easy to see that

$$
\|Ax\| \leq \|A\| \|x\|. \tag{5.51}
$$

The matrix norms that correspond to the $L^p$ vector norms are defined for the matrix $A$ as

$$
\|A\|_p = \max_{\|x\|_p=1} \|Ax\|_p. \tag{5.52}
$$

(Notice that the restriction on $\|x\|_p$ makes this an induced norm as defined in equation (5.50). Notice also the overloading of the symbols; the norm on the left that is being defined is a matrix norm, whereas those on the right of the equation are vector norms.) It is clear that the $L^p$ norms satisfy the consistency property, because they are induced norms.

The $L_1$ and $L_\infty$ norms have interesting simplifications:

- $\|A\|_1 = \max_j \sum_i |a_{ij}|$, also called the “column-sum norm”, and
- $\|A\|_\infty = \max_i \sum_j |a_{ij}|$, also called the “row-sum norm”.

Alternative formulations of the $L^2$ norm of a matrix are not so obvious from (5.52). It is related to the eigenvalues (or the singular values) of the matrix. For a square matrix $A$, the squared $L^2$ norm is the maximum eigenvalue of $A^T A$.

The $L_p$ matrix norms do not satisfy inequalities (5.48) for the $L_p$ vector norms.

For the $n \times n$ matrix $A$, with eigenvalues, $\lambda_1, \lambda_2, \ldots, \lambda_n$, the maximum, $\max |\lambda_i|$, is called the *spectral radius*, and is denoted by $\rho(A)$:

$$
\rho(A) = \max |\lambda_i|.
$$

It can be shown (see Exercise 5.11, page 343) that

$$
\|A\|_2 = \sqrt{\rho(A^T A)}.
$$

If $A$ is symmetric

$$
\|A\|_2 = \rho(A).
$$

The spectral radius is a measure of the condition of a matrix for certain iterative algorithms. The $L^2$ matrix norm is also called the spectral norm.

For $Q$ orthogonal, the $L_2$ norm has the important property,

$$
\|Qx\|_2 = \|x\|_2 \tag{5.53}
$$
For this reason, an orthogonal matrix is sometimes called an isometric matrix. By proper choice of \( x \), it is easy to see from (5.53) that
\[
\|Q\|_2 = 1.
\]
These properties do not in general hold for other norms.

The \( L_2 \) matrix norm is a Euclidean-type norm since it is based on the Euclidean vector norm, but a different matrix norm is often called the Euclidean matrix norm. This is the Frobenius norm:
\[
\|A\|_F = \sqrt{\sum_{i,j} a_{ij}^2}.
\]
It is easy to see that the Frobenius norm has the consistency property and that for any square matrix \( A \) with real elements
\[
\|A\|_2 \leq \|A\|_F.
\]
(See Exercises 5.13 and 5.14, page 343.) A useful property of the Frobenius norm, which is obvious from the definition above, is
\[
\|A\|_F = \sqrt{\text{trace}(A^T A)} = \sqrt{\langle A, A \rangle}.
\]
If \( A \) and \( B \) are orthogonally similar, then
\[
\|A\|_F = \|B\|_F.
\]
To see this, let \( A = Q^T BQ \), where \( Q \) is an orthogonal matrix. Then
\[
\|A\|_F^2 = \text{trace}(A^T A)
= \text{trace}(Q^T B^T QQ^T BQ)
= \text{trace}(B^T BQQ^T)
= \text{trace}(B^T B)
= \|B\|_F^2.
\]
(The norms are nonnegative, of course, and so equality of the squares is sufficient.)

**Orthogonal Transformations**

In the previous section we observed some interesting properties of orthogonal matrices. From equation (5.53), we see that orthogonal transformations preserve lengths.

If \( Q \) is orthogonal, for vectors \( x \) and \( y \), we have
angle between vectors
orthogonally similar
orthogonalization
Gram-Schmidt
transformation
ill-conditioned data

\[ \langle Qx, Qy \rangle = (xQ)^T(Qy) = x^TQ^TQy = x^Ty = \langle x, y \rangle, \]

hence,

\[ \arccos \left( \frac{\langle Qx, Qy \rangle}{\|Qx\|_2 \|Qy\|_2} \right) = \arccos \left( \frac{\langle x, y \rangle}{\|x\|_2 \|y\|_2} \right). \]  

(5.55)

Thus we see that orthogonal transformations preserve angles.

From equation (5.54) we see \|Q^{-1}\|_2 = 1, and thus \(\kappa_2(Q) = 1\) for the orthogonal matrix \(Q\). This means use of computations with orthogonal matrices will not make problems more ill-conditioned.

It is easy to see from (5.53) that if \(A\) and \(B\) are orthogonally similar,

\[ \kappa_2(A) = \kappa_2(B). \]

Later we use orthogonal transformations that preserve lengths and angles while reflecting regions of \(\mathbb{R}^n\), and others that rotate \(\mathbb{R}^n\). The transformations are appropriately called reflectors and rotators, respectively.

**Orthogonalization Transformations**

Given two nonnull, linearly independent vectors, \(x_1\) and \(x_2\), it is easy to form two orthonormal vectors, \(\tilde{x}_1\) and \(\tilde{x}_2\), that span the same space:

\[ \tilde{x}_1 = \frac{x_1}{\|x_1\|_2} \]

\[ \tilde{x}_2 = \frac{(x_2 - \tilde{x}_1^T x_2 \tilde{x}_1)}{\|x_2 - \tilde{x}_1^T x_2 \tilde{x}_1\|_2}. \]  

(5.56)

These are called \textit{Gram-Schmidt transformations}. They can easily be extended to more than two vectors. The Gram-Schmidt transformations are the basis for other computations we will discuss in Section 5.2.2, on page 294.

**Condition of Matrices**

Data are said to be “ill-conditioned” for a particular computation if the data are likely to cause problems in the computations, such as severe loss of precision. More generally, the term “ill-conditioned” is applied to a problem in which small changes to the input result in large changes in the output. In the case of a linear system

\[ Ax = b \]

the problem of solving the system is ill-conditioned if small changes to some elements of \(A\) or of \(b\) will cause large changes in the solution \(x\).

Consider, for example, the system of equations

\[ 1.000x_1 + 0.500x_2 = 1.500 \]

\[ 0.667x_1 + 0.333x_2 = 1.000 \] 

(5.57)
The solution is easily seen to be \( x_1 = 1.000 \) and \( x_2 = 1.000 \).

Now consider a small change in the right-hand side:

\[
\begin{align*}
1.000x_1 + 0.500x_2 &= 1.500 \\
0.667x_1 + 0.333x_2 &= 0.999
\end{align*}
\] (5.58)

This system has solution \( x_1 = 0.000 \) and \( x_2 = 3.000 \).

Alternatively, consider a small change in one of the elements of the coefficient matrix:

\[
\begin{align*}
1.000x_1 + 0.500x_2 &= 1.500 \\
0.667x_1 + 0.334x_2 &= 1.000
\end{align*}
\] (5.59)

The solution now is \( x_1 = 2.000 \) and \( x_2 = -1.000 \).

In both cases, small changes of the order of \( 10^{-3} \) in the input (the elements of the coefficient matrix or the right-hand side) result in relatively large changes (of the order of 1) in the output (the solution). Solving the system (either one of them) is an ill-conditioned problem.

The nature of the data that causes ill-conditioning depends on the type of problem. In this case, the problem is that the lines represented by the equations are almost parallel, as seen in Figure 5.1, and so their point of intersection is very sensitive to slight changes in the coefficients defining the lines.

Figure 5.1. Almost Parallel Lines: Ill-Conditioned Coefficient Matrices, Equations (5.57) and (5.58) grl010
For a specific problem such as solving a system of equations, we may quantify the condition of the matrix by a \textit{condition number}. To develop this quantification for the problem of solving linear equations, consider a linear system $Ax = b$, with $A$ nonsingular and $b \neq 0$, as above. Now perturb the system slightly by adding a small amount, $\delta b$, to $b$, and let $\tilde{b} = b + \delta b$. The system

$A\tilde{x} = \tilde{b}$

has a solution $\tilde{x} = \delta x + x = A^{-1}\delta b$. (Notice that $\delta b$ and $\delta x$ do not necessarily represent scalar multiples of the respective vectors.) If the system is well-conditioned, for any reasonable norm, if $\|\delta b\|/\|b\|$ is small, then $\|\delta x\|/\|x\|$ is likewise small.

From $\delta x = A^{-1}\delta b$ and the inequality in (5.51) (page 260), for the induced norm on $A$, we have

\[
\|\delta x\| \leq \|A^{-1}\| \|\delta b\|. \tag{5.60}
\]

Likewise, because $b = Ax$, we have

\[
\frac{1}{\|x\|} \leq \|A\| \frac{1}{\|b\|}; \tag{5.61}
\]

and (5.60) and (5.61) together imply

\[
\frac{\|\delta x\|}{\|x\|} \leq \|A\| \|A^{-1}\| \frac{\|\delta b\|}{\|b\|}. \tag{5.62}
\]

This provides a bound on the change in the solution $\|\delta x\|/\|x\|$ in terms of the perturbation $\|\delta b\|/\|b\|$.

The bound in (5.62) motivates us to define the \textit{condition number with respect to inversion}, $\kappa(A)$, by

\[
\kappa(A) = \|A\| \|A^{-1}\|, \tag{5.63}
\]

for nonsingular $A$. In the context of linear algebra the condition number with respect to inversion is so dominant in importance that we generally just refer to it as the “condition number”. A condition number is a useful measure of the condition of $A$ for the problem of solving a linear system of equations. There are other condition numbers useful in numerical analysis, however, such as the condition number for computing the sample variance in equation (3.8) on page 60, or the condition number for a root of a function.

We can write (5.62) as

\[
\frac{\|\delta x\|}{\|x\|} \leq \kappa(A) \frac{\|\delta b\|}{\|b\|}, \tag{5.64}
\]

and, following a similar development as above, write

\[
\frac{\|\delta b\|}{\|b\|} \leq \kappa(A) \frac{\|\delta x\|}{\|x\|}. \tag{5.65}
\]
These inequalities, as well as the other ones we write in this section, are sharp, as we can see by letting $A = I$.

Because the condition number is an upper bound on a quantity that we would not want to be large, a large condition number is “bad”.

Notice our definition of the condition number does not specify the norm; it only required that the norm be an induced norm. (An equivalent definition does not rely on the norm being an induced norm.) We sometimes specify a condition number with regard to a particular norm, and just as we sometimes denote a specific norm by a special symbol, we may use a special symbol to denote a specific condition number. For example, $\kappa_p(A)$ may denote the condition number of $A$ in terms of an $L_p$ norm. Most of the properties of condition numbers are independent of the norm used.

The coefficient matrix in equations (5.57) and (5.58) is

$$A = \begin{bmatrix} 1.000 & 0.500 \\ 0.667 & 0.333 \end{bmatrix},$$

and its inverse is

$$A^{-1} = \begin{bmatrix} -666 & 1000 \\ 1344 & -2000 \end{bmatrix}.$$  

It is easy to see that

$$\|A\|_1 = 1.667,$$

and

$$\|A^{-1}\|_1 = 3000,$$

hence,

$$\kappa_1(A) = 5001.$$

Likewise,

$$\|A\|_\infty = 1.500,$$

and

$$\|A^{-1}\|_\infty = 3344,$$

hence,

$$\kappa_\infty(A) = 5016.$$

Notice that the condition numbers are not exactly the same, but they are close. Although we used this matrix in an example of ill-conditioning, these condition numbers, although large, are not so large as to cause undue concern for numerical computations. Indeed, the systems of equations in (5.57), (5.58), and (5.59) would not cause problems for a computer program to solve them. Notice also that the condition numbers are of the order of magnitude of the ratio of the output perturbation to the input perturbation in those equations.

An interesting relationship for the condition number is

$$\kappa(A) = \frac{\max_{x \neq 0} \frac{\|Ax\|}{\|x\|}}{\min_{x \neq 0} \frac{\|Ax\|}{\|x\|}}$$  \hspace{1cm} (5.66)
orthogonally similar
artificial
ill-conditioning
scaling of a vector or
matrix

(see Exercise 5.17, page 343).

The numerator and denominator in (5.66) look somewhat like the maximum and minimum eigenvalues, as we have suggested. Indeed, the \( L_2 \) condition number is just the ratio of the largest eigenvalue in absolute value to the smallest (see page 260).

The eigenvalues of the coefficient matrix in equations (5.57) and (5.58) are 

\[ 1.333375 \quad \text{and} \quad -0.0003750, \]

and so 

\[ \kappa_2(A) = 3555.67, \]

which is the same order of magnitude as \( \kappa_\infty(A) \) and \( \kappa_1(A) \) computed above.

Some useful facts about condition numbers are:

- \( \kappa(A) = \kappa(A^{-1}) \)
- \( \kappa(cA) = \kappa(A) \), for \( c \neq 0 \)
- \( \kappa(A) \geq 1 \)
- \( \kappa_1(A) = \kappa_\infty(A^T) \)
- \( \kappa_2(A^T) = \kappa_2(A) \)
- \( \kappa_2(A^T A) = \kappa_2^2(A) \)
- \( \kappa_2(A) \geq \kappa_2(A) \)
- if \( A \) and \( B \) are orthogonally similar then, \( \|A\|_2 = \|B\|_2 \) (see equation (5.53))

Even though the condition number provides a very useful indication of the condition of the problem of solving a linear system of equations, it can be misleading at times. Consider, for example, the coefficient matrix

\[ A = \begin{bmatrix} 1 & 0 \\ 0 & \epsilon \end{bmatrix}, \]

where \( \epsilon < 1 \). The condition numbers are

\[ \kappa_1(A) = \kappa_2(A) = \kappa_\infty(A) = \frac{1}{\epsilon}, \]

and so if \( \epsilon \) is small, the condition number is large. It is easy to see, however, that small changes to the elements of \( A \) or of \( b \) in the system \( Ax = b \) do not cause undue changes in the solution (our heuristic definition of ill-conditioning). In fact, the simple expedient of multiplying the second row of \( A \) by \( 1/\epsilon \) (that is, multiplying the second equation, \( a_{21}x_1 + a_{22}x_2 = b_2 \), by \( 1/\epsilon \)) yields a linear system that is very well-conditioned.

This kind of apparent ill-conditioning is called *artificial ill-conditioning*. It is due to the different rows (or columns) of the matrix having a very different *scale*; the condition number can be changed just by scaling the rows or columns. This usually does not make a linear system any better or any worse conditioned.

In Section 5.2.4 we relate the condition number to bounds on the numerical accuracy of the solution of a linear system of equations.
5.1 Basic Vector/Matrix Computations 267

The relationship between the size of the matrix and its condition number is interesting. In general, we would expect the condition number to increase as the size increases. This is the case, but the nature of the increase depends on the type of elements in the matrix. If the elements are randomly and independently distributed as normal or uniform with mean of zero and variance of one, the increase in the condition number is approximately linear in the size of the matrix (see Exercise 5.23, page 344).

Our definition of condition number given above is for nonsingular matrices. We can formulate a useful alternate definition that extends to singular matrices and to nonsquare matrices: the condition number of a matrix is the ratio of the largest singular value in absolute value to the smallest nonzero singular value in absolute value.

The condition number, like the determinant, is not easy to compute (see page 309 in Section 5.2.8).

Vector/Matrix Derivatives and Antiderivatives

The operations of differentiation and integration of vectors and matrices are logical extensions of the corresponding operations of scalars. The specific meaning of the operation depends on the type of the operand object and the type of the variable of the operator.

Derivative of an Array with Respect to a Scalar

The derivative of a vector or matrix with respect to a scalar variable is just the array with the same shape (vector or matrix) whose elements are the ordinary derivative with respect to the scalar. Thus, the derivative of the vector $y = (y_1, \ldots, y_n)$ with respect to the scalar $x$ is the vector $dy/dx = (dy_1/dx, \ldots, dy_n/dx)$; and the derivative of the matrix $A = (a_{ij})$ with respect to the scalar $x$ is the matrix $dA/dx = (da_{ij}/dx)$.

Derivative of a Scalar with Respect to a Vector

The derivative of a scalar-valued function with respect to a vector is a vector of the partial derivatives of the function with respect to the elements of the vector. If $f$ is a function, and $x = (x_1, \ldots, x_n)$ is a vector,

$$\frac{df}{dx} = \left( \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right).$$

(5.67)

This vector is called the gradient, and is sometimes denoted by $\mathbf{g}_f$ or by $\nabla f$. (The gradient is useful in finding the maximum or minimum of a function. Some methods of solving linear systems of equations formulate the problem as a minimization problem. We discuss one such method beginning on page 297.)

The expression
Jacobian

\[ \frac{df}{dx} \]

may also be written as

\[ \frac{d}{dx}f. \]

When we write such an expression, it is interpreted in the sense of equation (5.67) if \( x \) is a vector; that is, the expression represents a vector. If \( x \) is a scalar, the expression itself represents a scalar.

**Derivative of a Vector with Respect to a Vector**

For a vector-valued function \( f \), the matrix whose rows are the transposes of the gradients is called the *Jacobian*. We denote the Jacobian of the function \( f \) by \( J_f \). The transpose of the Jacobian, that is, the matrix whose columns are the gradients, is denoted by \( \nabla f \) for the vector-valued function \( f \). (Note that the \( \nabla \) symbol can denote either a vector or a matrix.) Thus, the Jacobian for the system above is

\[
J_f = \begin{bmatrix}
\frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_m} \\
\frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_m} \\
\cdots & \cdots & \cdots & \cdots \\
\frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_m}
\end{bmatrix} = (\nabla f)^T.
\]

**Derivative of a Scalar with Respect to a Matrix**

The derivative of a function with respect to a matrix is a matrix with the same shape consisting of the partial derivatives of the function with respect to the elements of the matrix. The derivative of a matrix \( Y \) with respect to the matrix \( X \) is thus

\[
\frac{dY}{dX} = Y \otimes \frac{d}{dX}.
\]

Rogers (1980) and Magnus and Neudecker (1999) provide extensive discussions of matrix derivatives.

**Formulas for Derivatives**

Derivatives of vector/matrix expressions with respect to a vector are similar to derivatives of similar expressions with respect to a scalar. For example, if \( x \) is a vector and \( A \) is a conformable matrix that is constant with respect to \( x \), we have:
\[ \frac{dx^T A}{dx} = A \]
\[ \frac{dA x}{dx} = A^T \]
\[ \frac{dx^T A x}{dx} = Ax + A^T x \]

Note in these formulas the kinds of objects that result from the basic definitions:

- the derivative of a vector with respect to a vector is a matrix,
- the derivative of a scalar (a quadratic form) with respect to a vector is a vector.

The normal equations that determine a least squares fit of a linear regression model are obtained by taking the derivative of \((y - X b)^T (y - X b)\) and equating it to zero. Doing this, using the rules above we get

\[
\frac{d(y - X b)^T (y - X b)}{db} = \frac{d(y^T y - 2b^T X^Ty + b^T X^T X b)}{db} \\
= -2X^T y + 2X^T X b \\
= 0.
\]

**Derivative of a Matrix with Respect to a Vector**

The derivative of a matrix with respect to a vector is a three-dimensional object resulting from applying equation (5.67) to each of the elements of the matrix. For this reason, it is simpler to consider only the partial derivatives of the matrix, \(A\), with respect to the individual elements of the vector, \(x\), \(\frac{\partial A}{\partial x_i}\). The expressions involving the partial derivatives can be thought of as giving the relationship of one two-dimensional layer of a three-dimensional object.

Using the rules for differentiation of powers that result directly from the definitions, we can write the partial derivatives of the inverse of the matrix \(A\) as

\[
\frac{\partial}{\partial x_i} A^{-1} = -A^{-1} \left( \frac{\partial}{\partial x_i} A \right) A^{-1}.
\]

Expressions such as this in terms of \(\partial/\partial x_i\), can also be written in terms of \(d/dx\), as in Exercise 5.18.

Beyond the basics of differentiation of constant multiples or powers of a variable, the two most important properties of derivatives of expressions are the linearity of the operation and the chaining of the operation. These yield rules that correspond to the familiar rules of the differential calculus. A simple result of the linearity of the operation is the rule for differentiation of the trace:

\[
\frac{\partial}{\partial x_i} \text{trace}(A) = \text{trace} \left( \frac{\partial}{\partial x_i} A \right).
\]
Multidimensional Integrals and Integrals Involving Vectors and Matrices

The definition of multiple integrals is a straightforward extension of the definition of a single integral. In general, the rank of the integral is the same as the rank of the integrand. (“Rank” is used here in the sense of “number of dimensions”.) Some of the most familiar integrals are those used in multivariate probability distributions; for example, the fundamental integral that relates to the \( d \)-variate normal distribution:

\[
\int_{\mathbb{R}^d} e^{-(x-\mu)^T \Sigma^{-1} (x-\mu)/2} \, dx = (2\pi)^{d/2} (\det(\Sigma))^{1/2}.
\]

This expression is a simple result that follows from the evaluation of the individual single integrals. In this case the integrand and the integral are scalars. An example when the integrand and the integral are vectors is the normal mean:

\[
E(X) = (2\pi)^{-d/2} (\det(\Sigma))^{-1/2} \int_{\mathbb{R}^d} xe^{-(x-\mu)^T \Sigma^{-1} (x-\mu)/2} \, dx = \mu.
\]

An example when the integrand and the integral are matrices is the normal variance:

\[
V(X) = E \left( (X - E(X))(X - E(X))^T \right) = (2\pi)^{-d/2} (\det(\Sigma))^{-1/2} \int_{\mathbb{R}^d} \left( (x-\mu)(x-\mu)^T \right) e^{-(x-\mu)^T \Sigma^{-1} (x-\mu)/2} \, dx = \Sigma.
\]

Because they are linear operators, integration and the trace can be performed in either order:

\[
\int \text{trace}(A) \, dx = \text{trace} \left( \int A \, dx \right).
\]

Geometric Transformations

In many important applications of linear algebra, a vector represents a point in space, with each element of the vector corresponding to an element of a coordinate system, usually a cartesian system. A set of vectors describes a geometric object. Algebraic operations are geometric transformations that rotate, deform, or translate the object. While these transformations are often used in the two or three dimensions that correspond to the easily-perceived physical space, they have similar applications in higher dimensions.

Important characteristics of these transformations are what they leave unchanged; that is, their invariance properties. We have seen, for example, that
an orthogonal transformation preserves lengths of vectors (equation (5.53)) and angles between vectors (equation (5.55)). A transformation that preserves lengths and angles is called an isometric transformation. Such a transformation also preserves areas and volumes.

Another isometric transformation is a translation, which for a vector \( x \) is just the addition of another vector:

\[
\tilde{x} = x + t.
\]

A transformation that preserves angles is called an isotropic transformation. An example of an isotropic transformation that is not isometric is a uniform scaling or dilation transformation, \( \tilde{x} = ax \), where \( a \) is a scalar.

The transformation \( \tilde{x} = Ax \), where \( A \) is a diagonal matrix with not all elements the same, does not preserve angles; it is an anisotropic scaling.

Another anisotropic transformation is a shearing transformation, \( \tilde{x} = Ax \), where \( A \) is the same as an identity matrix, except for a single row or column that has a one on the diagonal, but possibly nonzero elements in the other positions; for example,

\[
\begin{bmatrix}
1 & 0 & a_1 \\
0 & 1 & a_1 \\
0 & 0 & 1
\end{bmatrix}.
\]

Although they do not preserve angles, both anisotropic scaling and shearing transformations preserve parallel lines. A transformation that preserves parallel lines is called an affine transformation. Preservation of parallel lines is equivalent to preservation of collinearity, and so an alternative characterization of an affine transformation is one that preserves collinearity. More generally, we can combine nontrivial scaling and shearing transformations to see that the transformation \( Ax \) for any nonsingular matrix \( A \) is affine. It is easy to see that addition of a constant vector to all vectors in a set preserves collinearity within the set, so a more general affine transformation is \( \tilde{x} = Ax + t \), for a nonsingular matrix \( A \) and a vector \( t \).

All of these transformations are linear transformations, because they preserve straight lines. A projective transformation, which uses the homogeneous coordinate system of the projective plane, preserves straight lines, but does not preserve parallel lines. These transformations are very useful in computer graphics (see Riesenfeld, 1981). Mortenson (1995) describes all of these types of transformations and discusses their applications in geometric modeling.

**Rotations**

The simplest rotation of a vector can be thought of as the rotation of a plane defined by two coordinates about the other principal axes. Such a rotation changes two elements of all vectors in the in that plane and leaves all the other elements, representing the other coordinates, unchanged. This rotation
can be described in a two-dimensional space defined by the coordinates being changed, without reference to the other coordinates.

Consider the rotation of the vector $x$ through the angle $\theta$ into $\tilde{x}$. The length is preserved, so we have $\|\tilde{x}\| = \|x\|$. Referring to Figure 5.2, we can write

$$\tilde{x}_1 = \|x\| \cos(\phi + \theta)$$
$$\tilde{x}_2 = \|x\| \sin(\phi + \theta).$$

Figure 5.2. Rotation of $x$ grl014

Now, from elementary trigonometry, we know

$$\cos(\phi + \theta) = \cos \phi \cos \theta - \sin \phi \sin \theta$$
$$\sin(\phi + \theta) = \sin \phi \cos \theta + \cos \phi \sin \theta.$$

Because $\cos \phi = x_1/\|x\|$ and $\sin \phi = x_2/\|x\|$, we can combine these equations to get

$$\tilde{x}_1 = x_1 \cos \theta - x_2 \sin \theta$$
$$\tilde{x}_2 = x_1 \sin \theta + x_2 \cos \theta. \quad (5.68)$$

Hence, multiplying $x$ by the orthogonal matrix

$$\begin{bmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta
\end{bmatrix}$$

performs the rotation of $x$. 
This idea easily extends to the rotation of a plane formed by two coordinates about all of the other (orthogonal) principal axes. By convention, we assume clockwise rotations for axes that increase in the direction from which the system is viewed. For example, if there were an \( x_3 \) axis in Figure 5.2, it would point toward the viewer. (This is called a “right-hand” coordinate system.)

The rotation matrix about principal axes is the same as an identity matrix with two diagonal elements changed to \( \cos \theta \) and the corresponding off-diagonal elements changed to \( \sin \theta \) and \(-\sin \theta\).

To rotate a 3-vector, \( x \), about the \( x_2 \) axis in a right-hand coordinate system, we would use the rotation matrix

\[
\begin{pmatrix}
\cos \theta & 0 & \sin \theta \\
0 & 1 & 0 \\
-\sin \theta & 0 & \cos \theta
\end{pmatrix}.
\]

A rotation of any plane can be formed by successive rotations of planes formed by two principal axes. A rotation of an arbitrary plane can be defined in terms of the direction cosines of a vector in the plane before and after the rotation. In a coordinate geometry, rotation of a plane can be viewed equivalently as a rotation of the coordinate system in the opposite direction. This is accomplished by rotation of the unit vectors \( e_i \) into \( \tilde{e}_i \).

A special type of transformation that rotates a vector to be perpendicular to a principal axis is called a Givens rotation. We discuss the use of this type of transformation in Section 5.2.2, on page 291.

**Translations**

Translations are relatively simple transformations involving the addition of vectors. Rotations, as we have seen, and other geometric transformations such as shearing, as we have indicated, involve multiplication by an appropriate matrix. In applications where several geometric transformations are to be made, it would be convenient if translations could also be performed by matrix multiplication. This can be done by use of homogeneous coordinates.

Homogeneous coordinates, which form the natural coordinate system for projective geometry, have a very simple relationship to cartesian coordinates. The point with cartesian coordinates \((x_1, x_2, \ldots, x_d)\) is represented in homogeneous coordinates as \((x^h_0, x^h_1, x^h_2, \ldots, x^h_d)\), where, for arbitrary \( x^h_0 \) not equal to zero, \( x^h_1 = x^h_0 x_1 \), \( x^h_2 = x^h_0 x_2 \), and so on. Each value of \( x^h_0 \) corresponds to a hyperplane in the ordinary cartesian coordinate system. The special plane \( x^h_0 = 0 \) does not have a meaning in the cartesian system. It corresponds to a hyperplane at infinity in the projective geometry. (Alternatively, of course, the homogeneous coordinate defining the hyperplane could be written at the end of the set of coordinates; that is, it could be taken to be \( x^h_{d+1} \).

We can easily effect the translation \( \tilde{x} = x + t \) by first representing the point \( x \) as \((1, x_1, x_2, \ldots, x_d)\) and then multiplying by the matrix.
We will use the symbol $x^h$ to represent the vector of corresponding homogeneous coordinates:

$$x^h = (1, x_1, x_2, \ldots, x_d).$$

We must be careful to distinguish the point $x$ from the vector that represents the point. In cartesian coordinates, there is a natural correspondence and the symbol $x$ representing a point may also represent the vector $(x_1, x_2, \ldots, x_d)$. The vector of homogeneous coordinates of the result $Tx^h$ correspond to the cartesian coordinates of $\tilde{x}$, $(x_1 + t_1, x_2 + t_2, \ldots, x_d + t_d)$, which is the desired result.

Homogeneous coordinates are used extensively in computer graphics, not only for the ordinary geometric transformations, but also for projective transformations, which model visual properties. Riesenfeld (1981) and Mortenson (1997) describe many of these applications.

### 5.1.2 Computer Representations and Basic Operations

Most scientific computational problems involve vectors and matrices. It is necessary to work with either the elements of vectors and matrices individually or with the arrays themselves. Programming languages such as Fortran 77 and C provide the capabilities for working with the individual elements, but not directly with the arrays. Fortran 95 and higher-level languages such as Matlab allow direct manipulation with vectors and matrices.

As we discuss in Chapter 3, page 32, the distinction between the set of real numbers, $\mathbb{R}$, and the set of floating-point numbers, $\mathbb{R}^f$, that we use in the computer has important implications for numerical computations. As we discuss in Section 3.1.2, beginning on page 44, an element $x$ of a vector or matrix is approximated by $[x]_c$, and a mathematical operation $\circ$ is simulated by a computer operation $[\circ]_c$. The familiar laws of algebra for the field of the reals do not hold in $\mathbb{R}^f$, especially if uncontrolled parallel operations are allowed. These distinctions, of course, carry over to arrays of floating-point numbers that represent real numbers, and the properties of vectors and matrices that we have discussed in the preceding section may not hold for their computer counterparts. For example, the dot product of a nonzero vector with itself is positive (page 231), but $\langle x_c, x_c \rangle_c = 0$ does not imply $x_c = 0$.

### Errors in Vectors and Matrices

As we discuss in Section 3.1.2, we measure error in a scalar quantity either as absolute error, $|\tilde{r} - r|$, where $r$ is the true value and $\tilde{r}$ is the computed or
rounded value, or as relative error, $|\tilde{r} - r|/r$ (as long as $r \neq 0$). The errors in vectors or matrices are generally expressed in terms of norms. The relative error in the representation of the vector $v$, or as a result of computing $v$, may be expressed as $\|\tilde{v} - v\|/\|v\|$ (as long as $\|v\| \neq 0$), where $\tilde{v}$ is the computed vector. We often use the notation $\tilde{v} = v + \delta v$, and so $\|\delta v\|/\|v\|$ is the relative error. The vector norm used may depend on practical considerations about the errors in the individual elements.

**Computer Representation of Vectors and Matrices**

The elements of vectors and matrices are represented as ordinary numeric data as we described in Section 3.1.1, in either fixed-point or floating-point representation. The elements are generally stored in a logically contiguous area of the computer memory. What is logically contiguous may not be physically contiguous, however. There are no convenient mappings of computer memory that would allow matrices to be stored in a logical rectangular grid, so matrices are usually stored either as columns strung end-to-end (a “column-major” storage) or as rows strung end-to-end (a “row-major” storage). In using a computer language or a software package, sometimes it is necessary to know which way the matrix is stored. For some software to deal with matrices of varying sizes, the user must specify the length of one dimension of the array containing the matrix. (In general, the user must specify the lengths of all dimensions of the array except one.) In Fortran subroutines it is common to have an argument specifying the leading dimension (number of rows), and in C functions it is common to have an argument specifying the column dimension.

Sometimes in accessing a partition of a given matrix, the elements occur at fixed distances from each other. If the storage is row-major for an $n \times m$ matrix, for example, the elements of a given column occur at a fixed distance of $m$ from each other. This distance is called the “stride”, and it is often more efficient to access elements that occur with a fixed stride than it is to access elements randomly scattered. Just accessing data from computer memory contributes significantly to the time it takes to perform computations.

If a matrix has many elements that are zeros, and if the positions of those zeros are easily identified, many operations on the matrix can be speeded up. Matrices with many zero elements are called sparse matrices; they occur often in certain types of problems, for example in the solution of differential equations and in statistical designs of experiments. The first consideration is how to represent the matrix and to store the matrix and the location information. Different software systems may use different schemes to store sparse matrices. The method used in the IMSL Libraries, for example, is described on page 328. Another important consideration is how to preserve the sparsity during intermediate computations. Pissanetzky (1984) considers these and other issues in detail.
Multiplication of Vectors and Matrices

Arithmetic on vectors and matrices involves arithmetic on the individual elements. The arithmetic on the elements is performed as we have discussed in Section 3.1.2.

The way the storage of the individual elements is organized is very important for the efficiency of computations. Also, the way the computer memory is organized and the nature of the numerical processors affect the efficiency and may be an important consideration in the design of algorithms for working with vectors and matrices.

The best methods for performing operations on vectors and matrices in the computer may not be the methods that are suggested by the definitions of the operations.

In most numerical computations with vectors and matrices there is more than one way of performing the operations on the scalar elements. Consider the problem of evaluating the matrix times vector product, \( c = Ab \), where \( A \) is \( n \times m \). There are two obvious ways of doing this:

- compute each of the \( n \) elements of \( c \), one at a time, as an inner product of \( m \)-vectors, \( c_i = a_i^T b = \sum_j a_{ij} b_j \), or
- update the computation of all of the elements of \( c \) simultaneously as

1. For \( i = 1, \ldots, n \), let \( c_i^{(0)} = 0 \).
2. For \( j = 1, \ldots, m \),
   \[
   \begin{align*}
   & \text{for } i = 1, \ldots, n, \\
   & \quad \{ \\
   & \quad \quad \text{let } c_i^{(i)} = c_i^{(i-1)} + a_{ij} b_j.
   \}
   \]

If there are \( p \) processors available for parallel processing, we could use a fan-in algorithm (see page 48) to evaluate \( Ax \) as a set of inner products:

\[
\begin{array}{cccc}
 \quad c_1^{(1)} = \quad & a_{11} b_1 + a_{12} b_2 \\
 \quad = \quad & a_{13} b_3 + a_{14} b_4 \\
 \quad c_1^{(2)} = \quad & c_1^{(1)} + c_2^{(1)} \\
 & \vdots \\
 \quad c_i = c_i^{(i)} + c_{i+1}^{(i)} \\
 \end{array}
\]

The order of the computations is \( nm \) (or \( n^2 \)).

Multiplying two matrices \( A \) and \( B \) can be considered as a problem of multiplying several vectors \( b_i \) by a matrix \( A \), as described above. In the following
we will assume $A$ is $n \times m$ and $B$ is $m \times p$, and we will use the notation $a_i$ to represent the $i^{th}$ column of $A$, $a_i^T$ to represent the $i^{th}$ row of $A$, $b_i$ to represent the $i^{th}$ column of $B$, $c_i$ to represent the $i^{th}$ column of $C = AB$, and so on. (This notation is somewhat confusing because here we are not using $a_i^T$ to represent the transpose of $a_i$, as we normally do. The notation should be clear in the context of the diagrams below, however.) Using the inner product method above results in the first step of the matrix multiplication forming,

$$
\begin{bmatrix}
a_1^T \\
\vdots \\
\vdots \\
\vdots \\
\end{bmatrix}
\begin{bmatrix}
b_1 \\
\vdots \\
\vdots \\
\vdots \\
\end{bmatrix}
\rightarrow
\begin{bmatrix}
c_{11} = a_1^T b_1 & \cdots \\
\vdots & \ddots & \vdots \\
\vdots & \vdots & \ddots \\
\end{bmatrix}.
$$

Using the second method above in which the elements of the product vector are updated all at once results in the first step of the matrix multiplication forming,

$$
\begin{bmatrix}
a_1 \\
\vdots \\
\vdots \\
\vdots \\
\end{bmatrix}
\begin{bmatrix}
b_{11} \\
\vdots \\
\vdots \\
\vdots \\
\end{bmatrix}
\rightarrow
\begin{bmatrix}
c_{(1)11} = a_{11} b_{11} & \cdots \\
\vdots & \ddots & \vdots \\
\vdots & \vdots & \ddots \\
\end{bmatrix}.
$$

The next and each successive step in this method is a saxpy operation:

$$
c_{ij}^{(k+1)} = b_{(k+1)i} a_1 + c_{ij}^{(k)},
$$

for $k$ going to $m - 1$.

Another method for matrix multiplication is to perform saxpy operations using all of the elements of $b_1^T$, before completing the computations for any of the columns of $C$. This method is to form the outer products $a_i b_i^T$. In the notation used above for the other methods, we have

$$
\begin{bmatrix}
a_1 \\
\vdots \\
\vdots \\
\vdots \\
\end{bmatrix}
\begin{bmatrix}
b_1^T \\
\vdots \\
\vdots \\
\vdots \\
\end{bmatrix}
\rightarrow
\begin{bmatrix}
c_{ij}^{(1)} = a_1 b_1^T \\
\vdots \\
\vdots \\
\vdots \\
\end{bmatrix},
$$

and the update is

$$
c_{ij}^{(k+1)} = a_{k+1} b_{k+1}^T + c_{ij}^{(k)}.
$$

The order of computations for any of these methods is $O(nmp)$, or just $O(n^3)$, if the dimensions are all approximately the same. Strassen’s method discussed next reduces the order of the computations.
Another method for multiplying matrices that can be faster for large matrices is the so-called Strassen algorithm (from Strassen, 1969). Suppose $A$ and $B$ are square matrices with equal and even dimensions. Partition them into submatrices of equal size, and consider the block representation of the product:

\[
\begin{bmatrix}
    C_{11} & C_{12} \\
    C_{21} & C_{22}
\end{bmatrix} = \begin{bmatrix}
    A_{11} & A_{12} \\
    A_{21} & A_{22}
\end{bmatrix} \begin{bmatrix}
    B_{11} & B_{12} \\
    B_{21} & B_{22}
\end{bmatrix},
\]

where all blocks are of equal size. Form

\[
P_1 = (A_{11} + A_{22})(B_{11} + B_{22})
\]
\[
P_2 = (A_{21} + A_{22})B_{11}
\]
\[
P_3 = A_{11}(B_{12} - B_{22})
\]
\[
P_4 = A_{22}(B_{21} - B_{11})
\]
\[
P_5 = (A_{11} + A_{12})B_{22}
\]
\[
P_6 = (A_{21} - A_{11})(B_{11} + B_{12})
\]
\[
P_7 = (A_{12} - A_{22})(B_{21} + B_{22}).
\]

Then we have (see the discussion on partitioned matrices in Section 5.1.1):

\[
C_{11} = P_1 + P_4 - P_5 + P_7
\]
\[
C_{12} = P_3 + P_5
\]
\[
C_{21} = P_2 + P_4
\]
\[
C_{22} = P_1 + P_3 - P_2 + P_6.
\]

Notice that the total number of multiplications of matrices is seven, instead of eight as it would be in forming

\[
\begin{bmatrix}
    A_{11} & A_{12} \\
    A_{21} & A_{22}
\end{bmatrix} \begin{bmatrix}
    B_{11} & B_{12} \\
    B_{21} & B_{22}
\end{bmatrix},
\]

directly. Whether the blocks are matrices or scalars, the same analysis holds. Of course, in either case there are more additions. Addition of two $k \times k$ matrices is $O(k^2)$, so for a large enough value of $n$ the total number of operations using the Strassen algorithm is less than the number required for performing the multiplication in the usual way.

This idea can also be used recursively. (If the dimension, $n$, contains a factor $2^e$, the algorithm can be used directly $e$ times and then use conventional matrix multiplication on any submatrix of dimension $\leq n/2^e$.)

If the dimension of the matrices is not even, or if the matrices are not square, it is a simple matter to pad the matrices with zeros, and use this same idea.
The order of computations of the Strassen algorithm is $O(n^{\log_2 7})$, instead of $O(n^3)$ as in the ordinary method ($\log_2 7 = 2.81$). The algorithm can be implemented in parallel (see Bailey, Lee, and Simon, 1990).

5.2 Solution of Linear Systems

One of the most common problems in numerical computing is to solve the linear system

$$Ax = b,$$

that is, for given $A$ and $b$, to find $x$ such that the equation holds. The system is said to be consistent if there exists such an $x$, and in that case a solution $x$ may be written as $A^{-1}b$, where $A^{-1}$ is some inverse of $A$. If $A$ is square and of full rank, we can write the solution as $A^{-1}b$.

It is important to distinguish the expression $A^{-1}b$ or $A^+b$, which represents the solution, from the method of computing the solution. We would never compute $A^{-1}$ just so we could multiply it by $b$ to form the solution $A^{-1}b$.

There are two general methods of solving a system of linear equations: direct methods and iterative methods. A direct method uses a fixed number of computations that would in exact arithmetic lead to the solution; an iterative method generates a sequence of approximations to the solution. Iterative methods often work well for very large sparse matrices.

5.2.1 Gaussian Elimination

The most common direct method for the solution of linear systems is Gaussian elimination. The basic idea in this method is to form equivalent sets of equations, beginning with the system to be solved, $Ax = b$, or

$$a_1^T x = b_1$$
$$a_2^T x = b_2$$
$$\ldots = \ldots$$
$$a_n^T x = b_n,$$

where $a_j^T$ is the $j^{th}$ row of $A$. An equivalent set of equations can be formed by a sequence of elementary operations on the equations in the given set. There are two kinds of elementary operations: an interchange of two equations,

$$a_j^T x = b_j \leftarrow a_k^T x = b_k$$
$$a_k^T x = b_k \leftarrow a_j^T x = b_j,$$

which affects two equations simultaneously, or the replacement of a single equation with a linear combination of it and another equation:
Numerical Linear Algebra

\[ a_j^T x = b_j \quad \leftarrow \quad c_j a_j^T x + c_k a_k^T x = c_j b_j + c_k b_k, \]

where \( c_j \neq 0 \). If \( c_k = 0 \) in this operation, it is the simple elementary operation of scalar multiplication of a single equation.

The interchange operation can be accomplished by premultiplication by an elementary permutation matrix (see page 249):

\[ E_{jk} Ax = E_{jk} b. \]

Likewise, the linear combination elementary operation can be effected by premultiplication by a matrix formed from the identity matrix by replacing its \( j^{th} \) row by a row with all zeros except for \( c_j \) in the \( j^{th} \) column and \( c_k \) in the \( k^{th} \) column. Such a matrix is denoted by \( E_{jk}(c_j, c_k) \), for example,

\[
E_{23}(c_2, c_3) = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & c_2 & 0 & 0 \\
0 & 0 & c_3 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}.
\]

Both \( E_{jk} \) and \( E_{jk}(c_j, c_k) \) are called elementary operator matrices.

The elementary operation on the equation

\[ a_1^T x = b_1 \]

in which the first equation is combined with it using \( c_1 = -a_{21}/a_{11} \) and \( c_2 = 1 \) will yield an equation with a zero coefficient for \( x_1 \). Generalizing this, we perform elementary operations on the second through the \( n^{th} \) equations to yield a set of equivalent equations in which all but the first have zero coefficients for \( x_1 \).

Next, we perform elementary operations using the second equation with the third through the \( n^{th} \) equations, so that the new third through the \( n^{th} \) equations have zero coefficients for \( x_2 \).

The sequence of equivalent equations is

\[
\begin{array}{c}
\begin{align*}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
\vdots & \vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n
\end{align*}
\end{array}
\]

(1)

\[
\begin{array}{c}
\begin{align*}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1^{(1)} \\
a_{22}x_2 + \cdots + a_{2n}x_n &= b_2^{(1)} \\
\vdots & \vdots \\
a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n^{(1)}
\end{align*}
\end{array}
\]

(2)

\[
\begin{array}{c}
\begin{align*}
\vdots & \vdots
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n^{(1)}
\end{align*}
\end{array}
\]
This last system is easy to solve. It is upper triangular. The last equation in
the system yields

\[ x_n = \frac{b_{n-1}^{(n-1)}}{a_{nn}}. \]

By back substitution we get

\[ x_{n-1} = \frac{b_{n-1}^{(n-2)} - a_{n-1,n}^{(n-2)}x_n}{a_{n-1,n-1}^{(n-2)}}, \]

and the rest of the \( x \)'s in a similar manner.

Thus, Gaussian elimination consists of two steps, the forward reduction,
which is order \( O(n^3) \), and the back substitution, which is order \( O(n^2) \).

The only obvious problem with this method arises if some of the \( a_{kk}^{(k-1)} \)'s
used as divisors are zero (or very small in magnitude). These divisors are
called “pivot elements”.

Suppose, for example, we have the equations

\[
\begin{align*}
0.0001x_1 + x_2 & = 1 \\
x_1 + x_2 & = 2
\end{align*}
\]

The solution is \( x_1 = 1.0001 \) and \( x_2 = 0.9999 \). Suppose we are working with 3
digits of precision (so our solution is \( x_1 = 1.00 \) and \( x_2 = 1.00 \)). After the first
step in Gaussian elimination we have

\[
\begin{align*}
0.0001x_1 + x_2 & = 1 \\
-10,000x_2 & = -10,000
\end{align*}
\]

and so the solution by back substitution is \( x_2 = 1.00 \) and \( x_1 = 0.000 \). The
\( L_2 \) condition number of the coefficient matrix is 2.618, so even though the
coefficients do vary greatly in magnitude, we certainly would not expect any
difficulty in solving these equations.

A simple solution to this potential problem is to interchange the equation
having the small leading coefficient with an equation below it. Thus, in our
example, we first form

\[
\begin{align*}
x_1 + x_2 & = 2 \\
0.0001x_1 + x_2 & = 1
\end{align*}
\]
so that after the first step we have

\[ x_1 + x_2 = 2 \\
 2x_2 = 1 \]

and the solution is \( x_2 = 1.00 \) and \( x_1 = 1.00 \).

Another strategy would be to interchange the column having the small
leading coefficient with a column to its right. Both the row interchange and the
column interchange strategies could be used simultaneously, of course. These
processes, which obviously do not change the solution, are called pivoting. The
equation or column to move into the active position may be chosen in such a
way that the magnitude of the new diagonal element is the largest possible.

Performing only row interchanges, so that at the \( k \)th stage the equation
with

\[
\max_{i=k}^{n} |a_{ik}^{(k-1)}|
\]

is moved into the \( k \)th row, is called partial pivoting. Performing both row
interchanges and column interchanges, so that

\[
\max_{i=k, j=k}^{nl} |a_{ij}^{(k-1)}|
\]

is moved into the \( k \)th diagonal position, is called complete pivoting. See Exercises ?? and ??.

It is always important to distinguish descriptions of effects of actions from
the actions that are actually carried out in the computer. Pivoting is “inter-
changing” rows or columns. We would usually do something like that in the
computer only when we are finished and want to produce some output. In the
computer, a row or a column is determined by the index identifying the row
or column. All we do for pivoting is to keep track of the indices that we have
permuted.

There are many more computations required in order to perform complete
pivoting than are required to perform partial pivoting. Gaussian elimination
with complete pivoting can be shown to be stable that is, the algorithm yields
an exact solution to a slightly perturbed system, \((A + \delta A)x = b\). For Gaussian
elimination with partial pivoting there exist examples to show that it is not
stable. These examples are somewhat contrived, however, and experience over
many years has indicated that Gaussian elimination with partial pivoting is
stable for most problems occurring in practice. For this reason together with
the computational savings, Gaussian elimination with partial pivoting is one
of the most commonly used methods for solving linear systems. See Golub
and Van Loan (1996) for a further discussion of these issues.

There are two modifications of partial pivoting that result in stable al-
gorithms. One is to add one step of iterative refinement (see Section 5.2.5,
page 301) following each pivot. It can be shown that Gaussian elimination
with partial pivoting together with one step of iterative refinement is un-
conditionally stable (Skeel, 1980). Another modification is to consider two
columns for possible interchange in addition to the rows to be interchanged. This does not require nearly as many computations as complete pivoting does. Higham (1997) shows that this method, suggested by Bunch and Kaufman (1977) and used in LINPACK and LAPACK, is stable.

Each step in Gaussian elimination is equivalent to multiplication of the current coefficient matrix, \( A^{(k)} \), by some matrix \( L_k \). If we ignore pivoting (i.e., assume it is handled by permutation vectors), the \( L_k \) matrix has a particularly simple form:

\[
L_k = \begin{bmatrix}
    1 & \cdots & 0 & 0 & \cdots & 0 \\
    & \ddots & & & & \\
    & & 1 & 0 & \cdots & 0 \\
    & & & \frac{a_{k+1,k}^{(k)}}{a_{kk}^{(k)}} & 1 & \cdots & 0 \\
    & & & & \ddots & & \\
    & & & & & \frac{a_{nk}^{(k)}}{a_{kk}^{(k)}} & 0 & \cdots & 1
\end{bmatrix}
\]

Each \( L_k \) is nonsingular, with a determinant of 1. The whole process of forward reduction can be expressed as a matrix product,

\[
U = L_{n-1}L_{n-2}\ldots L_2L_1A,
\]

and by the way we have performed the forward reduction, \( U \) is an upper triangular matrix. The matrix \( L_{n-1}L_{n-2}\ldots L_2L_1 \) is nonsingular and is unit lower triangular (all 1’s on the diagonal). Its inverse is also, therefore, unit lower triangular. Call its inverse \( L \). The forward reduction is equivalent to expressing \( A \) as \( LU \),

\[
A = LU;
\]

hence this process is called an \textit{LU factorization} or an \textit{LU decomposition}. (We use the terms “matrix factorization” and “matrix decomposition” interchangeably.)

Notice, of course, that we do not necessarily store the two matrix factors in the computer.

### 5.2.2 Matrix Factorizations

Direct methods of solution of linear systems all use some form of matrix factorization, similar to the \( LU \) factorization in the last section. Matrix factorizations are also performed for reasons other than to solve a linear system. The important matrix factorizations are:

- \( LU \) factorization and \( LDU \) factorization (primarily, but not necessarily, for square matrices)
- Cholesky factorization (for nonnegative definite matrices)
In this section we discuss each of these factorizations.

**LU Factorization**

The $LU$ factorization is the most commonly used method to solve a linear system. For any matrix (whether square or not) that is expressed as $LU$, where $L$ is unit lower triangular and $U$ is upper triangular, the product $LU$ is called the $LU$ factorization. If an $LU$ factorization exists, it is clear that the upper triangular matrix, $U$, can be made unit upper triangular (all 1’s on the diagonal), by putting the diagonal elements of the original $U$ into a diagonal matrix $D$, and then writing the factorization as $LDU$, where $U$ is now a unit upper triangular matrix.

The computations leading up to equation (5.69) provide a method of computing an $LU$ factorization. This method, based on Gaussian elimination over rows, consists of a sequence of outer products. Another way of arriving at the $LU$ factorization is by use of the inner product. From equation (5.69), we see

$$a_{ij} = \sum_{k=1}^{i-1} l_{ik} u_{kj} + u_{ij},$$

so

$$l_{ij} = \frac{a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}}{u_{jj}}, \quad \text{for } i = j + 1, j + 2, \ldots, n. \quad (5.70)$$

The use of computations implied by equation (5.70) is called the Doolittle method or the Crout method. (There is a slight difference in the Doolittle method and the Crout method: the Crout method yields a decomposition in which the 1’s are on the diagonal of the $U$ matrix, rather than the $L$ matrix.)

Whichever method is used to compute the $LU$ decomposition, $n^3/3$ multiplications and additions are required.

It is neither necessary nor sufficient that a matrix be nonsingular for it to have an $LU$ factorization. An example of a singular matrix that has an $LU$ factorization is any upper triangular matrix with all zeros on the diagonal. In this case, $U$ can be chosen as the matrix itself, and $L$ chosen as the identity:

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}.$$ 

An example of a nonsingular matrix that does not have an $LU$ factorization is an identity matrix with permuted rows or columns:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$
If a nonsingular matrix has an $LU$ factorization, $L$ and $U$ are unique.

A sufficient condition for an $n \times m$ matrix $A$ to have an $LU$ factorization is that for $k = 1, 2, \ldots, \min(n-1, m)$, each $k \times k$ principal submatrix of $A$, $A_k$, be nonsingular. Note this fact also provides a way of constructing a singular matrix that has an $LU$ factorization. Furthermore, for $k = 1, 2, \ldots, \min(n, m)$,

$$\det(A_k) = u_{11} u_{22} \cdots u_{kk}.$$  

### Cholesky Factorization

If the coefficient matrix $A$ is symmetric and positive definite, that is, if $x^T Ax > 0$, for all $x \neq 0$, another important factorization is the Cholesky decomposition. In this factorization,

$$A = T^T T,$$

where $T$ is an upper triangular matrix with positive diagonal elements. In Section 5.1.1 we show that such a matrix exists.

The factor $T$ in the Cholesky decomposition is sometimes called the square root for obvious reasons. It is also sometimes denoted by $A^{1/2}$. A factor of this form is unique up to the sign, just as a square root is. To make the Cholesky factor unique, we require that the diagonal elements be positive. The elements along the diagonal of $T$ will be square roots. Notice, for example, $t_{11}$ is \(\sqrt{a_{11}}\).

The Cholesky decomposition can also be formed as $\tilde{T}^T D \tilde{T}$, where $D$ is a diagonal matrix that allows the diagonal elements of $\tilde{T}$ to be computed without taking square roots. This modification is sometimes called a Banachiewicz factorization or root-free Cholesky. The Banachiewicz factorization can be computed in essentially the same way as the Cholesky factorization shown in Algorithm 5.1: just put 1’s along the diagonal of $T$, and store the unsquared quantities in a vector $d$.

In Exercise ?? you are asked to use the algorithm for computing the Cholesky factorization serves as a constructive proof of the existence and uniqueness. (The uniqueness is seen by factoring the principal square submatrices.)

### Algorithm 5.1 Cholesky Factorization

1. Let $t_{11} = \sqrt{a_{11}}$.  
2. For $j = 2, \ldots, n$, let $t_{1j} = a_{1j}/t_{11}$.  
3. For $i = 2, \ldots, n$,  
   
   $\{$
   
   let $t_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} t_{ki}^2}$, and
   
   for $j = i + 1, \ldots, n$,  
   
   $\{$
   
   let $t_{ij} = (a_{ij} - \sum_{k=1}^{i-1} t_{ki} t_{kj})/t_{ii}$.
   
   $\} $

   $\}$
There are other algorithms for computing the Cholesky decomposition. The method given in Algorithm 5.1 is sometimes called the inner-product formulation because the sums in step 3 are inner products. The algorithm for computing the Cholesky decomposition is numerically stable. Although the order of the number of computations is the same, there are only about half as many computations in the Cholesky factorization as in the LU factorization. Another advantage of the Cholesky factorization is that there are only $n(n + 1)/2$ unique elements, as opposed to $n^2 + n$ in the LU decomposition. An important difference, however, is that Cholesky applies only to symmetric, positive definite matrices.

For a symmetric matrix, the $LDU$ factorization is $U^TDU$; hence we have for the Cholesky factor,

$$T = D^\frac{1}{2}U,$$

where $D^\frac{1}{2}$ is the matrix whose elements are the square roots of the corresponding elements of $D$. (This is consistent with our notation above for Cholesky factors; $D^\frac{1}{2}$ is the Cholesky factor of $D$, and it is symmetric.)

Any symmetric nonnegative definite matrix has a decomposition similar to the Cholesky for a positive definite matrix. If $A$ is $n \times n$ with rank $r$, there exists a unique matrix $T$, such that $A = TT^T$, where $T$ is an upper triangular matrix with $r$ positive diagonal elements and $n - r$ rows containing all zeros. The algorithm is the same as Algorithm 5.1, except in step 3 if $t_{ii} = 0$, the entire row is set to zero. The algorithm serves as a constructive proof of the existence and uniqueness.

The LU and Cholesky decompositions generally are applied to square matrices. However, many of the linear systems that occur in scientific applications are overdetermined; that is, there are more equations than there are variables, resulting in a nonsquare coefficient matrix.

An overdetermined system may be written as

$$Ax \approx b,$$

where $A$ is $n \times m$ ($n \geq m$), or it may be written as

$$Ax = b + e,$$

where $e$ is an $n$-vector of possibly arbitrary “errors”. Because all equations cannot be satisfied simultaneously, we must define a meaningful “solution”. A useful solution is an $x$ such that $e$ has a small norm. The most common definition is an $x$ such that $e$ has the least Euclidean norm, that is, such that the sum of squares of the $e_i$’s is minimized.

It is easy to show that such an $x$ satisfies the square system $A^T Ax = A^T b$, the so-called “normal equations”. This expression is important and allows us to analyze the overdetermined system (not just to solve for the $x$, but to gain some better understanding of the system). It is easy to show that if $A$ is full rank (i.e., of rank $m$, or all of its columns are linearly independent, or,
5.2 Solution of Linear Systems

If \( A \) has full column rank (i.e., is of full column rank), then \( A^T A \) is positive definite. Therefore, we could apply either Gaussian elimination or the Cholesky decomposition to obtain the solution.

As we have emphasized many times before, however, \textit{useful conceptual expressions are not necessarily useful as computational formulations}. That is sometimes true in this case also. Among other indications that it may be better to work directly on \( A \) is the fact that the condition number of \( A^T A \) is the square of the condition number of \( A \). We discuss solution of overdetermined systems in Section 5.2.7, beginning on page 303.

\textbf{QR Factorization}

A very useful factorization is

\[ A = QR, \]  
(5.72)

where \( Q \) is orthogonal and \( R \) is upper triangular. This is called the QR factorization.

If \( A \) is nonsquare in (5.72), then \( R \) is such that its leading square matrix is upper triangular; for example if \( A \) is \( n \times m \), and \( n \geq m \), then

\[ R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}, \]  
(5.73)

where \( R_1 \) is upper triangular.

For the \( n \times m \) matrix \( A \), with \( n \geq m \), we can write

\[ A^T A = R^T Q^T Q R \]

so we see that the matrix \( R \) in the QR factorization is (or at least can be) the same as the matrix \( T \) in the Cholesky factorization of \( A^T A \).

There is some ambiguity in the \( Q \) and \( R \) matrices, but if the diagonal entries of \( R \) are required to be nonnegative, the ambiguity disappears, and the matrices in the QR decomposition are unique.

It is interesting to note that the Moore-Penrose inverse of \( A \) is immediately available from the QR factorization:

\[ A^+ = \begin{bmatrix} R_1^{-1} \\ 0 \end{bmatrix} Q^T. \]  
(5.74)

If \( A \) is not of full rank, we apply permutations to the columns of \( A \) by multiplying on the right by a permutation matrix. The permutations can be taken out by a second multiplication on the right. If \( A \) is of rank \( r \) (\( \leq m \)), the resulting decomposition consists of three matrices, an orthogonal \( Q \), a \( T \) with an \( r \times r \) upper triangular submatrix, and a permutation matrix \( P^T \):

\[ A = QTP^T. \]  
(5.75)
The matrix $T$ has the form
\[
T = \begin{bmatrix} T_1 & T_2 \\ 0 & 0 \end{bmatrix},
\]
where $T_1$ is upper triangular and is $r \times r$. The decomposition in (5.75) is not unique because of the permutation matrix. Choice of the permutation matrix is the same as the pivoting that we discussed in connection with Gaussian elimination. A generalized inverse of $A$ is immediately available from (5.75):
\[
A^{-} = P \begin{bmatrix} T_1^{-1} & 0 \\ 0 & 0 \end{bmatrix} Q^T.
\]

Additional orthogonal transformations can be applied from the right side of $A$ in the form (5.75) to yield
\[
A = QRU^T,
\]
where $R$ has the form
\[
R = \begin{bmatrix} R_1 & 0 \\ 0 & 0 \end{bmatrix},
\]
where $R_1$ is $r \times r$ upper triangular, $Q$ is as in (5.75), and $U^T$ is orthogonal. (The permutation matrix in (5.75) is also orthogonal, of course.) The decomposition (5.78) is unique. This decomposition provides the Moore-Penrose generalized inverse of $A$:
\[
A^+ = U \begin{bmatrix} R_1^{-1} & 0 \\ 0 & 0 \end{bmatrix} Q^T.
\]

It is often of interest to know the rank of a matrix. Given a decomposition of the form (5.75), the rank is obvious, and in practice, this $QR$ decomposition with pivoting is a good way to determine the rank of a matrix. The $QR$ decomposition is said to be “rank-revealing”. The computations are quite sensitive to rounding, however, and the pivoting must be done with some care (see Hong and Pan, Section 2.7.3 of Björck, 1996, and Bischof and Quintana-Ortí, 1998a,b.)

There are three good methods for obtaining the $QR$ factorization: Householder transformations, or reflections, Givens transformations, or rotations, and the (modified) Gram-Schmidt procedure. Different situations may make one or the other of these procedures better than the other two. For example, if the data are available only one row at a time, the Givens transformations are very convenient.

Whichever method is used to compute the $QR$ decomposition, at least $2n^3/3$ multiplications and additions are required (and this is possible only when clever formulations are used). The operation count is therefore about twice as great as that for an $LU$ decomposition.

The $QR$ factorization is particularly useful in computations for overdetermined systems, as we see in Section 5.2.7, page 303, and in other computations involving nonsquare matrices.
5.2 Solution of Linear Systems 289

**Householder Transformations (Reflections)**

Let \( u \) and \( v \) be orthonormal vectors, and let \( x \) be a vector in the space spanned by \( u \) and \( v \), so

\[
x = c_1 u + c_2 v
\]

for some scalars \( c_1 \) and \( c_2 \). The vector

\[
\tilde{x} = -c_1 u + c_2 v
\]

is a *reflection* of \( x \) through the line defined by the vector \( u \).

Now consider the matrix

\[
Q = I - 2uu^T,
\]

and note that

\[
Qx = c_1 u + c_2 v - 2c_1 uu^T c_2 v + 2c_2 v
\]

\[
= c_1 u + c_2 v - 2c_1 uu^T c_2 u + 2c_2 u^T v
\]

\[
= -c_1 u + c_2 v
\]

\[
= \tilde{x}.
\]

The matrix \( Q \) is a reflector. A reflection is also called a Householder reflection or a Householder transformation, and the matrix \( Q \) is called a Householder matrix. The following properties of \( Q \) are obvious.

- \( Qu = -u \)
- \( Qv = v \) for any \( v \) orthogonal to \( u \)
- \( Q = Q^T \) (symmetric)
- \( Q^T = Q^{-1} \) (orthogonal)

The matrix \( uu^T \) is symmetric, idempotent, and of rank 1. (A transformation by a matrix of the form \( A - uv^T \) is often called a “rank-one” update, because \( uv^T \) is of rank 1. Thus, a Householder reflection is a special rank-one update.)

The usefulness of Householder reflections results from the fact that it is easy to construct a reflection that will transform a vector

\[
x = (x_1, x_2, \ldots, x_n)
\]

into a vector

\[
\tilde{x} = (\tilde{x}_1, 0, \ldots, 0).
\]

Now, if \( Qx = \tilde{x} \), then \( \|x\|_2 = \|\tilde{x}\|_2 \) (see equation 5.53), so \( \tilde{x}_1 = \pm \|x\|_2 \). To construct the reflector, form the normalized vector \( (x - \tilde{x}) \), that is, let

\[
v = (x_1 + \text{sign}(x_1)\|x\|, x_2, \ldots, x_n),
\]

and \( u = v/\|v\| \), where all norms are the L2 norm. Notice that we could have chosen \( \|x\| \) or \( -\|x\| \) for the first element in \( \tilde{x} \). We *choose the sign so as not*
to add quantities of different signs and possibly similar magnitudes. (See the discussions of catastrophic cancellation in Chapter 3.)

Consider, for example, the vector

\[ x = (3, 1, 2, 1, 1). \]

We have

\[ \|x\| = 4, \]

so we form the vector

\[ u = \frac{1}{\sqrt{56}} (7, 1, 2, 1, 1), \]

and the reflector,

\[ Q = I - 2uu^T \]

\[ = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} - \frac{1}{28} \begin{bmatrix} 49 & 7 & 14 & 7 & 7 \\ 7 & 1 & 2 & 1 & 1 \\ 14 & 2 & 4 & 2 & 2 \\ 7 & 1 & 2 & 1 & 1 \\ 7 & 1 & 2 & 1 & 1 \end{bmatrix} \]

\[ = \frac{1}{28} \begin{bmatrix} -21 & -7 & -14 & -7 & -7 \\ -7 & 27 & -2 & -1 & -1 \\ -14 & -2 & 24 & -2 & -2 \\ -7 & -1 & -2 & 27 & -1 \\ -7 & -1 & -2 & -1 & 27 \end{bmatrix}, \]

to yield \( Qx = (-4, 0, 0, 0, 0) \).

To use reflectors to compute a QR factorization, we form in sequence the reflector for the \( i \)th column that will produce 0’s below the \( (i, i) \) element. For a convenient example, consider the matrix

\[ A = \begin{bmatrix} 3 & \frac{9}{28} & X & X & X \\ 1 & \frac{122}{28} & X & X \\ 2 & \frac{63}{28} & X & X \\ 1 & \frac{92}{28} & X & X \\ 1 & \frac{145}{28} & X & X \end{bmatrix}. \]

The first transformation applied would be \( P_1 \), given as \( Q \) above, yielding
5.2 Solution of Linear Systems

\[
P_1 A = \begin{bmatrix}
-4 & 1 & \text{X} & \text{X} & \text{X} \\
0 & 5 & \text{X} & \text{X} & \text{X} \\
0 & 1 & \text{X} & \text{X} & \text{X} \\
0 & 3 & \text{X} & \text{X} & \text{X} \\
0 & 1 & \text{X} & \text{X} & \text{X}
\end{bmatrix}.
\]

We now choose a reflector to transform \((5, 1, 3, 1)\) to \((-6, 0, 0, 0)\). Forming the vector \((11, 1, 3, 1)/\sqrt{132}\), and proceeding as before, we get the reflector

\[
Q_2 = I - \frac{1}{66} (11, 1, 3, 1)(11, 1, 3, 1)^T = \frac{1}{66}
\begin{bmatrix}
-55 & -11 & -33 & -11 \\
-11 & 65 & -3 & -1 \\
-33 & -3 & 57 & -3 \\
-11 & -1 & -3 & 65
\end{bmatrix}.
\]

We do not want to disturb the first column in \(P_1 A\) shown above, so we form \(P_2\) as

\[
P_2 = \begin{bmatrix}
1 & 0 & \cdots & 0 \\
0 & & & \\
\vdots & & & Q_2 \\
0 & & & 
\end{bmatrix}.
\]

Now we have

\[
P_2 P_1 A = \begin{bmatrix}
-4 & \text{X} & \text{X} & \text{X} \\
0 & -6 & \text{X} & \text{X} \\
0 & 0 & \text{X} & \text{X} \\
0 & 0 & \text{X} & \text{X}
\end{bmatrix}.
\]

Continuing in this way for three more steps we would have the QR decomposition of \(A\), with \(Q = P_5 P_4 P_3 P_2 P_1\).

The number of computations for the QR factorization of an \(n \times n\) matrix using Householder reflectors is \(2n^3/3\) multiplications and \(2n^3/3\) additions. Carrig and Meyer (1997) describe two variants of the Householder transformations that take advantage of computer architectures that have a cache memory or that have a bank of floating-point registers whose contents are immediately available to the computational unit.

**Givens Transformations (Rotations)**

Another way of forming the QR decomposition is by use of orthogonal transformations that rotate a vector in such a way that a specified element becomes 0 and only one other element in the vector is changed. Such a method may be particularly useful if only part of the matrix to be transformed is available. These transformations are called Givens transformations, or Givens rotations, or sometimes Jacobi transformations.
The basic idea of the rotation, which is a special case of the rotations discussed on page 271, can be seen in the case of a vector of length 2. Given the vector \( x = (x_1, x_2) \), we wish to rotate it to \( \tilde{x} = (\tilde{x}_1, 0) \). As with a reflector, \( \tilde{x}_1 = \|x\| \). Geometrically, we have the picture shown in Figure 5.3.

\[
\begin{align*}
\tilde{x}_1 &= \frac{x_1^2}{r} + \frac{x_2^2}{r} \\
&= \|x\| \\
\tilde{x}_2 &= -\frac{x_2x_1}{r} + \frac{x_1x_2}{r} \\
&= 0.
\end{align*}
\]

As with the Householder reflection that transforms a vector
\[
x = (x_1, x_2, x_3, \ldots, x_n)
\]
into a vector
\[
\tilde{x}_H = (\tilde{x}_{H1}, 0, 0, \ldots, 0),
\]
it is easy to construct a Givens rotation that transforms \( x \) into
\[
\tilde{x}_G = (\tilde{x}_{G1}, 0, x_3, \ldots, x_n).
\]

More generally, we can construct an orthogonal matrix, \( G_{pq} \), similar to that shown in (5.81), that will transform the vector.
5.2 Solution of Linear Systems

\[ x = (x_1, \ldots, x_p, \ldots, x_q, \ldots, x_n) \]  

fast Givens rotation

to

\[ \tilde{x} = (x_1, \ldots, \tilde{x}_p, \ldots, 0, \ldots, x_n). \]

The orthogonal matrix that will do this is

\[
Q_{pq}(\theta) = \begin{bmatrix}
1 & 0 & \cdots & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\
0 & 1 & \cdots & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\
& & \ddots & & & & & & & \\
0 & 0 & \cdots & 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\
0 & 0 & \cdots & 0 & \cos \theta & \cdots & 0 & \sin \theta & 0 & \cdots & 0 \\
0 & 0 & \cdots & 0 & 0 & \cdots & 1 & 0 & \cdots & 0 \\
0 & 0 & \cdots & 0 & -\sin \theta & \cdots & 0 & \cos \theta & 0 & \cdots & 0 \\
0 & 0 & \cdots & 0 & 0 & \cdots & 0 & 0 & \cdots & 1 \\
\end{bmatrix},
\]

(5.82)

where the entries in the \(p^{th}\) and \(q^{th}\) rows and columns are

\[ \cos \theta = \frac{x_p}{r} \]

and

\[ \sin \theta = \frac{x_q}{r}, \]

where \(r = \sqrt{x_p^2 + x_q^2}\). A rotation matrix is the same as an identity matrix with four elements changed.

Considering \(x\) to be the \(p^{th}\) column in a matrix \(X\), we can easily see how to zero out the \(q^{th}\) element of that column while affecting only the \(p^{th}\) and \(q^{th}\) rows and columns of \(X\).

Just as we built the \(QR\) factorization by applying a succession of Householder reflections, we can also apply a succession of Givens rotations to achieve the factorization. If the Givens rotations are applied directly, however, the number of computations is about twice as many as for the Householder reflections. A succession of “fast Givens rotations” can be constructed, however, that will reduce the total number of computations by about one half. To see how this is done, first write the matrix \(Q\) in (5.81) as \(CT\),

\[
\begin{bmatrix}
\cos \theta & \sin \theta \\
-\sin \theta & \cos \theta
\end{bmatrix} =
\begin{bmatrix}
\cos \theta & 0 \\
0 & \cos \theta
\end{bmatrix}
\begin{bmatrix}
1 & \tan \theta \\
-\tan \theta & 1
\end{bmatrix},
\]

(5.83)

Instead of working with matrices such as \(Q\), which require 4 multiplications and 2 additions, we work with matrices such as \(T\), involving the tangents,
which require only 2 multiplications and 2 additions. The diagonal matrices such as $C$ must be accumulated and multiplied in at some point. If this is done cleverly, the number of computations for Givens rotations is not much greater than that for Householder reflections. The diagonal elements in the accumulate $C$ matrices can become widely different in absolute values, so to avoid excessive loss of accuracy, it is usually necessary to rescale the elements periodically. Anda and Park (1994, 1996) describe methods of doing the rescaling dynamically. Their methods involve adjusting the first diagonal element by multiplication by the square of the cosine and adjusting the second diagonal element by division by the square of the cosine. See Golub and Van Loan (1996) for further discussion of the fast Givens transformations. Bindel et al. (2002) discuss in detail techniques for performing Givens rotations efficiently while still maintaining accuracy. The BLAS routines (see page 324) `rotmg` and `rotm` respectively set up and apply fast Givens rotations.

**Gram-Schmidt Transformations**

Gram-Schmidt transformations yield a set of orthonormal vectors that span the same space as a given set of linearly independent vectors, \{$x_1, x_2, \ldots, x_m$\}. Application of these transformations is called Gram-Schmidt orthogonalization. If the given linearly independent vectors are the columns of a matrix $A$, the Gram-Schmidt transformations ultimately yield the $QR$ factorization of $A$. The basic Gram-Schmidt transformation is shown in equation (5.56), page 262.

At the $k$th stage of the Gram-Schmidt method, the vector $x_k^{(k)}$ is taken as $x_k^{(k-1)}$ and the vectors $x_{k+1}^{(k)}, x_{k+2}^{(k)}, \ldots, x_m^{(k)}$ are all made orthogonal to $x_k^{(k)}$. After the first stage all vectors have been transformed. (This method is sometimes called “modified Gram-Schmidt”, because some people have performed the basic transformations in a different way, so that at the $k$th iteration, starting at $k = 2$, the first $k - 1$ vectors are unchanged, i.e., $x_i^{(k)} = x_i^{(k-1)}$ for $i = 1, 2, \ldots, k - 1$, and $x_k^{(k)}$ is made orthogonal to the $k - 1$ previously orthogonalized vectors $x_1^{(k)}, x_2^{(k)}, \ldots, x_{k-1}^{(k)}$. This method is called “classical Gram-Schmidt”, for no particular reason. The “classical” method is not as stable, and should not be used. See Rice, 1966, and Björck, 1967, for discussions.) In the following, “Gram-Schmidt” is the same as what is sometimes called “modified Gram-Schmidt”.

The Gram-Schmidt algorithm for forming the $QR$ factorization is just a simple extension of equation (5.56); see Exercise ?? on page ??.

**Singular Value Factorization**

Another factorization useful in solving linear systems is the singular value decomposition, or SVD, shown in (5.42), page 255. For the $n \times m$ matrix $A$, this is
5.2 Solution of Linear Systems

\[ A = U \Sigma V^T, \]

where \( U \) is an \( n \times n \) orthogonal matrix, \( V \) is an \( m \times m \) orthogonal matrix, and \( \Sigma \) is a diagonal matrix of the singular values. Golub and Kahan (1965) showed how to use a \( QR \)-type factorization to compute a singular value decomposition. This method, with refinements as presented in Golub and Reinsch (1970), is the best algorithm for singular value decomposition.

nonnegative matrix factorization, \( X = WH \), where \( W \) is a matrix with nonnegative elements. See Lee and Seung (1999)

Choice of Direct Methods

An important consideration for the various direct methods is the efficiency of the method for certain patterned matrices. If a matrix initially has a large number of zeros, it is important to preserve zeros as the matrix is operated on. This helps to avoid unnecessary computations. Pissanetzky (1984) discusses some of the ways of doing this. The iterative methods discussed in the next section are often more useful for sparse matrices.

Another important consideration is how easily an algorithm lends itself to implementation on advanced computer architectures. Many of the algorithms for linear algebra can be vectorized easily. It is now becoming more important to be able to parallelize the algorithms. The iterative methods discussed in the next section can often be parallelized more easily.

5.2.3 Iterative Methods

An iterative method for solving the linear system \( Ax = b \) obtains the solution by a sequence of successive approximations.

The Gauss-Seidel Method with Successive Overrelaxation

One of the simplest iterative procedures is the Gauss-Seidel method. In this method, we begin with an initial approximation to the solution, \( x^{(0)} \). We then compute an update for the first element of \( x \):

\[ x_1^{(1)} = \frac{1}{a_{11}} \left( b_1 - \sum_{j=2}^{n} a_{1j} x_j^{(0)} \right). \]

Continuing in this way for the other elements of \( x \), we have for \( i = 1, \ldots, n \)

\[ x_i^{(1)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(0)} \right), \]
where no sums are performed if the upper limit is smaller than the lower limit.

After getting the approximation \( x^{(1)} \), we then continue this same kind of iteration for \( x^{(2)} \), \( x^{(3)} \), etc. We continue the iterations until a convergence criterion is satisfied. As we discussed on page 64, this criterion may be of the form

\[
\Delta(x^{(k)}, x^{(k-1)}) \leq \epsilon,
\]

where \( \Delta(x^{(k)}, x^{(k-1)}) \) is a measure of the difference of \( x^{(k)} \) and \( x^{(k-1)} \), such as \( \|x^{(k)} - x^{(k-1)}\| \). We may also base the convergence criterion on \( \|r^{(k)} - r^{(k-1)}\| \), where \( r^{(k)} = b - Ax^{(k)} \).

The Gauss-Seidel iterations can be thought of as beginning with a rearrangement of the original system of equations as

\[
\begin{align*}
  a_{11}x_1 & = b_1 - a_{12}x_2 - \cdots - a_{1n}x_n \\
  a_{21}x_1 + a_{22}x_2 & = b_2 - a_{23}x_3 - \cdots - a_{2n}x_n \\
  \vdots & + \vdots & \vdots & + \vdots \\
  a_{(n-1)1}x_1 + a_{(n-1)2}x_2 + \cdots & = b_{n-1} - a_{nn}x_n \\
  a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n & = b_n
\end{align*}
\]

In this form, we identify three matrices – a diagonal matrix \( D \), a lower triangular \( L \) with 0’s on the diagonal, and an upper triangular \( U \) with 0’s on the diagonal:

\[(D + L)x = b - Ux.\]

We can write this entire sequence of Gauss-Seidel iterations in terms of these three fixed matrices,

\[x^{(k+1)} = (D + L)^{-1}(-Ux^{(k)} + b).\]  \hspace{1cm} \text{(5.84)}

This method will converge for any arbitrary starting value \( x^{(0)} \) if and only if the spectral radius of \( (D + L)^{-1}U \) is less than 1. (See Golub and Van Loan, 1996, for a proof of this.) Moreover, the rate of convergence increases with decreasing spectral radius.

Gauss-Seidel may be unacceptably slow, so it may be modified so that the update is a weighted average of the regular Gauss-Seidel update and the previous value. This kind of modification is called successive overrelaxation, or SOR. The update is given by

\[
\frac{1}{\omega}(D + L)x^{(k+1)} = \frac{1}{\omega}((1 - \omega)D - \omega U)x^{(k)} + b,
\]

where the relaxation parameter \( \omega \) is usually chosen between 0 and 1. For \( \omega = 1 \) the method is the ordinary Gauss-Seidel method. See Exercises ??, ??, and ??.
Solution of Linear Systems as an Optimization Problem; Conjugate Gradient Methods

The problem of solving the linear system $Ax = b$ is equivalent to finding the minimum of the function

$$f(x) = \frac{1}{2} x^T Ax - x^T b.$$  \hfill (5.85)

By setting the derivative of $f$ to 0, we see that a stationary point of $f$ occurs at the point $x$ where $Ax = b$. If $A$ is nonsingular, the minimum of $f$ is at $x = A^{-1}b$, and the value of $f$ at the minimum is $-\frac{1}{2} b^T A b$.

The minimum point can be approached iteratively by starting at a point $x^{(0)}$, moving to a point $x^{(1)}$ that yields a smaller value of the function, and continuing to move to points yielding smaller values of the function. The $k$th point is $x^{(k-1)} + \alpha_k d_k$, where $\alpha_k$ is a scalar and $d_k$ is a vector giving the direction of the movement. Hence, for the $k$th point we have the linear combination,

$$x^{(k)} = x^{(0)} + \alpha_1 d_1 + \cdots + \alpha_k d_k.$$

The convergence criterion is based on $\|x^{(k)} - x^{(k-1)}\|$ or on $\|r^{(k)} - r^{(k-1)}\|$, where $r^{(k)} = b - Ax^{(k)}$.

At the point $x^{(k)}$, the function $f$ decreases most rapidly in the direction of the negative gradient, $-\nabla f(x^{(k)})$. The negative gradient is just the residual,

$$r^{(k)} = b - Ax^{(k)}.$$

If this residual is 0, no movement is indicated, because we are at the solution. Moving in the direction of steepest descent may cause a slow convergence to the minimum. (The curve that leads to the minimum on the quadratic surface is obviously not a straight line.)

A good choice for the sequence of directions $p_1, p_2, \ldots$ is such that

$$p_i^T A p_i = 0, \quad \text{for } i = 1, \ldots, k - 1.$$

Such a vector $p_k$ is said to be $A$-conjugate to $p_1, p_2, \ldots p_{k-1}$. The path defined by the directions $p_1, p_2, \ldots$ and the distances $\alpha_1, \alpha_2, \ldots$ is called the conjugate gradient. A conjugate gradient method for solving the linear system is shown in Algorithm 5.2.

**Algorithm 5.2 The Conjugate Gradient Method for Solving $Ax = b$, Starting with $x^{(0)}$**

0. Set $k = 0$; $r^{(k)} = b - Ax^{(k)}$; $s^{(k)} = A^T r^{(k)}$; $p^{(k)} = s^{(k)}$; and $\gamma^{(k)} = \|s^{(k)}\|^2$.
1. If $\gamma^{(k)} \leq \epsilon$, set $x = x^{(k)}$ and terminate.
2. Set $q^{(k)} = A p^{(k)}$.
3. Set $\alpha^{(k)} = \frac{\gamma^{(k)}}{q^{(k)} p^{(k)}}$.
4. Set $x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)}$. 

conjugate gradient method
5. Numerical Linear Algebra

Lanczos method

Krylov space

5. Set \( r(k+1) = r(k) - \alpha(k)q(k) \).
6. Set \( s(k+1) = A^T r(k+1) \).
7. Set \( \gamma(k+1) = \| s(k+1) \|_2^2 \).
8. Set \( p(k+1) = s(k+1) + \frac{\gamma(k+1)}{\gamma(k)} p(k) \).
9. If \( k < k_{\text{max}} \),
   set \( k = k + 1 \) and go to 1;
   otherwise,
   issue message that
   ‘algorithm did not converge in \( k_{\text{max}} \) iterations’.  

For example, the function (5.85) arising from the system

\[
\begin{bmatrix}
5 & 2 \\
2 & 3 \\
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
\end{bmatrix}
= \begin{bmatrix}
18 \\
16 \\
\end{bmatrix}
\]

has level contours as shown in Figure 5.4, and the conjugate gradient method
would move along the line shown, toward the solution at \( x = (2, 4) \).

Figure 5.4. Solution of a Linear System Using a Conjugate Gradient Method grl020

The conjugate gradient method and related procedures, called Lanczos methods,
moves through a Krylov space in the progression to the solution (see Freund, Golub, and Nachtigal, 1992). A Krylov space is the \( k \)-dimensional vector space of order \( n \) generated by the \( n \times n \) matrix \( A \) and the vector \( v \) by forming the basis \( \{ v, Av, A^2v, \ldots, A^{k-1}v \} \). We often denote this space as \( K_k(A, v) \), or just as \( K_k \).
The generalized minimal residual (GMRES) method of Saad and Schultz (1986) for solving \( Ax = b \) begins with an approximate solution \( x^{(0)} \) and takes \( x^{(k)} \) as \( x^{(k-1)} + z^{(k)} \), where \( z^{(k)} \) is the solution to the minimization problem,

\[
\min_{z \in K_k(A,r^{(k-1)})} \| r^{(k-1)} - Az \|,
\]

where, as before, \( r^{(k)} = b - Ax^{(k)} \). This minimization problem is a constrained least squares problem. In the original implementations, the convergence of GMRES could be very slow, but modifications have speeded it up considerably. See Walker (1988) and Walker and Zhou (1994) for details of the methods. Brown and Walker (1997) consider the behavior of GMRES when the coefficient matrix is singular, and give conditions for GMRES to converge to a solution of minimum length (the solution corresponding to the Moore-Penrose inverse, see page 305).

Iterative methods have important applications in solving differential equations. The solution of differential equations by a finite difference discretization involves the formation of a grid. The solution process may begin with a fairly coarse grid, on which a solution is obtained. Then a finer grid is formed, and the solution is interpolated from the coarser grid to the finer grid to be used as a starting point for a solution over the finer grid. The process is then continued through finer and finer grids. If all of the coarser grids are used throughout the process, the technique is a multigrid method. There are many variations of exactly how to do this. Multigrid methods are useful solution techniques for differential equations.

Iterative methods are particularly useful for large, sparse systems. Another advantage of many of the iterative methods is that they can be parallelized more readily (see Heath, Ng, and Peyton, 1991). An extensive discussion of iterative methods is given in Axelsson (1994).

### 5.2.4 Numerical Accuracy

The condition numbers we defined in Section 5.1.1 are useful indicators of the accuracy we may expect when solving a linear system, \( Ax = b \). Suppose the entries of the matrix \( A \) and the vector \( b \) are accurate to approximately \( p \) decimal digits, so we have the system

\[
(A + \delta A)(x + \delta x) = b + \delta b,
\]

with

\[
\frac{\| \delta A \|}{\| A \|} \approx 10^{-p}
\]

and

\[
\frac{\| \delta b \|}{\| b \|} \approx 10^{-p}.
\]
Assume $A$ is nonsingular, and suppose that the condition number with respect to inversion, $\kappa(A)$, is approximately $10^t$, so

$$
\kappa(A) \frac{\|\delta A\|}{\|A\|} \approx 10^{t-p}.
$$

Ignoring the approximation of $b$, that is, assuming $\delta b = 0$, we can write

$$
\delta x = -A^{-1} \delta A (x + \delta x),
$$

which, together with the triangular inequality and inequality (5.51), page 260, yields the bound

$$
\|\delta x\| \leq \|A^{-1}\| \|\delta A\| (\|x\| + \|\delta x\|).
$$

Using equation (5.63) with this we have

$$
\|\delta x\| \leq \kappa(A) \frac{\|\delta A\|}{\|A\|} (\|x\| + \|\delta x\|),
$$

or

$$
\left(1 - \kappa(A) \frac{\|\delta A\|}{\|A\|}\right) \|\delta x\| \leq \kappa(A) \frac{\|\delta A\|}{\|A\|} \|x\|.
$$

If the condition number is not too large relative to the precision, that is, if $10^{t-p} \ll 1$, then we have

$$
\frac{\|\delta x\|}{\|x\|} \approx \kappa(A) \frac{\|\delta A\|}{\|A\|} \approx 10^{t-p}.
$$

Expression (5.86) provides a rough bound on the accuracy of the solution in terms of the precision of the data and the condition number of the coefficient matrix. This result must be used with some care, however. Rust (1994), among others, points out failures of the condition number for setting bounds on the accuracy of the solution.

Another consideration in the practical use of (5.86) is the fact that the condition number is usually not known, and methods for computing it suffer from the same rounding problems as the solution of the linear system itself. In Section 5.2.8 we describe ways of estimating the condition number, but as the discussion there indicates, these estimates are often not very reliable.

We would expect the norms in the expression (5.86) to be larger for larger size problems. The approach taken above addresses a type of “total” error. It may be appropriate to scale the norms to take into account the number of elements. Chaitin-Chatelin and Frayssé (1996) discuss error bounds for individual elements of the solution vector and condition measures for elementwise error.

Another approach to determining the accuracy of a solution is to use random perturbations of $A$ and/or $b$ and then to estimate the effects of the


Another way of improving the accuracy is by use of iterative refinement, which we now discuss.

### 5.2.5 Iterative Refinement

Once an approximate solution, $x^{(0)}$, to the linear system $Ax = b$ is available, iterative refinement can yield a solution that is closer to the true solution. The residual

$$r = b - Ax^{(0)}$$

is used for iterative refinement. Clearly, if $h = A^+ r$, then $x^{(0)} + h$ is a solution to the original system.

The problem considered here is not just an iterative solution to the linear system, as we discussed in Section 5.2.3. Here, we assume $x^{(0)}$ was computed accurately given the finite precision of the computer. In this case it is likely that $r$ cannot be computed accurately enough to be of any help. If, however, $r$ can be computed using a higher precision, then a useful value of $h$ can be computed. This process can then be iterated as shown in Algorithm 5.3.

#### Algorithm 5.3 Iterative Refinement of the Solution to $Ax = b$,

Starting with $x^{(0)}$

0. Set $k = 0$.
1. Compute $r^{(k)} = b - Ax^{(k)}$ in higher precision.
2. Compute $h^{(k)} = A^+ r^{(k)}$.
3. Set $x^{(k+1)} = x^{(k)} + h^{(k)}$.
4. If $\|h^{(k)}\| \leq \epsilon \|x^{(k+1)}\|$, then set $x = x^{(k+1)}$ and terminate; otherwise,
   - if $k < k_{\text{max}}$,
     - set $k = k + 1$ and go to step 1;
   - otherwise,
     - issue message that ‘algorithm did not converge in $k_{\text{max}}$ iterations’.

In step 2, if $A$ is full rank then $A^+$ is $A^{-1}$. Also, as we have emphasized already, because we write an expression such as $A^+ r$ does not mean that we
compute $A^+$. The norm in step 4 is usually chosen to be the $\infty$-norm. The algorithm may not converge, so it is necessary to have an alternative exit criterion, such as a maximum number of iterations.

Use of iterative refinement as a general-purpose method is severely limited by the need for higher precision in step 1. On the other hand, if computations in higher precision can be performed, they can be applied to step 2 — or just in the original computations for $x^{(0)}$. In terms of both accuracy and computational efficiency, use of higher precision throughout is usually better.

### 5.2.6 Updating a Solution

In applications of linear systems, it is often the case that after the system $Ax = b$ has been solved, the right-hand side is changed, and the system $Ax = c$ must be solved. If the linear system $Ax = b$ has been solved by a direct method using one of the factorizations discussed in Section 5.2.2, the factors of $A$ can be used to solve the new system $Ax = c$. If the right-hand side is a small perturbation of $b$, say $c = b + \delta b$, an iterative method can be used to solve the new system quickly, starting from the solution to the original problem.

If the coefficient matrix in a linear system $Ax = b$ is perturbed to result in the system $(A + \delta A)x = b$, it may be possible to use the solution $x_0$ to the original system to arrive efficiently at the solution to the perturbed system. One way, of course, is to use $x_0$ as the starting point in an iterative procedure. Often in applications, the perturbations are of a special type, such as

$$A = A - uv^T,$$

where $u$ and $v$ are vectors. (This is a “rank-one” perturbation of $A$, and when the perturbed matrix is used as a transformation, it is called a “rank-one” update. As we have seen, a Householder reflection is a special rank-one update.) Assuming $A$ is an $n \times n$ matrix of full rank, it is easy to write $A^{-1}$ in terms of $A^{-1}$:

$$A^{-1} = A^{-1} + \alpha(A^{-1}u)(v^TA^{-1}),$$

with

$$\alpha = \frac{1}{1 - v^TA^{-1}u}.$$

These are called the Sherman-Morrison formulas (from J. Sherman and W. J. Morrison, 1950, “Adjustment of an inverse matrix corresponding to a change in one element of a given matrix,” *Annals of Mathematical Statistics* 21, 124–127). $A^{-1}$ exists so long as $v^TA^{-1}u \neq 1$. Because $x_0 = A^{-1}b$, the solution to the perturbed system is

$$\hat{x}_0 = x_0 + \frac{(A^{-1}u)(v^Tx_0)}{(1 - v^TA^{-1}u)}.$$

If the perturbation is more than rank one, that is, if the perturbation is
where $U$ and $V$ are $n \times m$ matrices with $n \geq m$, a generalization of the Sherman-Morrison formula, sometimes called the Woodbury formula, is

$$\tilde{A}^{-1} = A^{-1} + A^{-1}U(I_m - V^T A^{-1} U)^{-1} V^T A^{-1}$$

(5.89)

(from M. A. Woodbury, 1950, “Inverting Modified Matrices”, Memorandum Report 42, Statistical Research Group, Princeton University). The solution to the perturbed system is easily seen to be

$$\tilde{x}_0 = x_0 + A^{-1}U(I_m - V^T A^{-1} U)^{-1} V^T x_0.$$  

As we have emphasized many times, we rarely compute the inverse of a matrix, and so the Sherman-Morrison-Woodbury formulas are not used directly. Because of having already solved $Ax = b$, it should be easy to solve another system, say $Ay = u_i$ where $u_i$ is a column of $U$. If $m$ is relatively small, as it is in most applications of this kind of update, there are not many systems $Ay = u_i$ to solve. Solving these systems, of course, yields $A^{-1}U$, the most formidable component of the Sherman-Morrison-Woodbury formula. The system to solve is of order $m$ also.

Occasionally the updating matrices in equation (5.88) may be used with a weighting matrix, so we have $\tilde{A} = A - UWV^T$. An extension of the Sherman-Morrison-Woodbury formula is

$$(A - UWV^T)^{-1} = A^{-1} + A^{-1}U(W^{-1} - V^T A^{-1} U)^{-1} V^T A^{-1}.$$  

(5.90)

This is sometimes called the Hemes formula. (The attributions of discovery are somewhat murky.)

Another situation that requires an update of a solution occurs when the system is augmented with additional equations and more variables:

$$\begin{bmatrix} A & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x \\ x_+ \end{bmatrix} = \begin{bmatrix} b \\ b_+ \end{bmatrix}$$

A simple way of obtaining the solution to the augmented system is to use the solution $x_0$ to the original system in an iterative method. The starting point for a method based on Gauss-Seidel or a conjugate gradient method can be taken as $(x_0, 0)$, or as $(x_0, x_+^{(0)})$ if a better value of $x_+^{(0)}$ is known.

In many statistical applications the systems are overdetermined, with $A$ being $n \times m$ and $n > m$. In the next section we consider the problem of updating a least squares solution to an overdetermined system.

### 5.2.7 Overdetermined Systems; Least Squares

An overdetermined system may be written as
Ax \approx b, \quad (5.91)

where \( A \) is \( n \times m \) and \( \text{rank}(A|b) > m \). The problem is to determine a value of \( x \) that makes the approximation close, in some sense. We sometimes refer to this as “fitting” the system, which is referred to as a “model”. Although there is no \( x \) that will make the system an equation, the system can be written as the equation

\[
Ax = b + e,
\]

where \( e \) is an \( n \)-vector of possibly arbitrary “errors”.

A least squares solution \( \hat{x} \) to the system in (5.91) is one such that the Euclidean norm of the vector \( b - Ax \) is minimized. By differentiating, we see that the minimum of the square of this norm,

\[
(b - Ax)^T(b - Ax), \quad (5.92)
\]

occurs at \( \hat{x} \) that satisfies the square system

\[
A^T A \hat{x} = A^T b. \quad (5.93)
\]

The system (5.93) is called the normal equations. The matrix \( A^T A \) is called the Gram matrix, or the Gramian. Its condition determines the expected accuracy of a solution to the least squares problem. As we mentioned in Section 5.2.2, however, because the condition number of \( A^T A \) is the square of the condition number of \( A \), it may be better to work directly on \( A \) in (5.91) rather than to use the normal equations. In fact, this was the reason that we introduced the QR factorization for nonsquare matrices, because the LU and Cholesky factorizations had been described only for square matrices. The normal equations are useful expressions, however, whether or not they are used in the computations. This is another case where a formula does not define an algorithm, as with other cases we have encountered many times. We should note, of course, that any information about the stability of the problem that the Gramian may provide, can be obtained from \( A \) directly.

It is interesting to note from equation (5.93) that the residual vector, \( b - A\hat{x} \), is orthogonal to each column in \( A \):

\[
A^T(b - A\hat{x}) = 0. \quad (5.94)
\]

Overdetermined systems abound in fitting equations to data. The linear regression model is an overdetermined system. Sometimes instead of the norm (5.92), we use the weighted norm,

\[
(b - Ax)^T W (b - Ax). \quad (5.95)
\]

**Full Rank Coefficient Matrix**

If \( A \) is of full rank, the least squares solution, from (5.93), is \( \hat{x} = (A^T A)^{-1} A^T b \) and is obviously unique. A good way to compute this is to form the QR factorization of \( A \).
First we write $A = QR$, as in (5.72) on page 287, where $R$ is as in (5.73):

$$R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix},$$

with $R_1$ an $m \times m$ upper triangular matrix. The residual norm (5.92) can be written as

$$(b - Ax)^T(b - Ax) = (b - QRx)^T(b - QRx)$$

$$= (Q^Tb - Rx)^T(Q^Tb - Rx)$$

$$= (c_1 - R_1x)^T(c_1 - R_1x) + c_2^Tc_2.$$  (5.96)

where $c_1$ is a vector of length $m$ and $c_2$ is a vector of length $n - m$, such that

$$Q^Tb = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}.$$

Because quadratic forms are nonnegative, the minimum of the residual norm in (5.96) occurs when $(c_1 - R_1x)^T(c_1 - R_1x) = 0$, that is, when $(c_1 - R_1x) = 0$, or

$$R_1x = c_1.$$  (5.97)

We could also use the same technique of differentiation to find the minimum of (5.96) that we did to find the minimum of (5.92).

Because $R_1$ is triangular, the system is easy to solve: $\hat{x} = R_1^{-1}c_1$.

We also see from (5.96) that the minimum of the residual norm is $c_2^Tc_2$.

This is called the residual sum of squares in the least squares fit.

In passing, we note from (5.74) that $\hat{x} = A^+b$.

The use of the $QR$ factorization for the overdetermined system in which the weighted norm (5.95) is minimized is similar to the development above. Gulliksson and Wedin (1992) and Gulliksson (1995) discuss some of additional computational issues for the weighted case.

**Coefficient Matrix Not of Full Rank**

If $A$ is not of full rank, that is, if $A$ has rank $r < m$, the least squares solution is not unique, and in fact, a solution is any vector $\hat{x} = A^-b$, where $A^-$ is any generalized inverse. For any generalized inverse $A^-$, the set of all solutions is

$$A^-b + (I - A^-A)z,$$

for an arbitrary vector $z$.

The solution whose $L_2$-norm $\|x\|_2$ is minimum is unique, however. That solution is the one corresponding to the Moore-Penrose inverse.

To see that this solution has minimum norm, first factor $A$, as in equation (5.78), page 288,

$$A = QRU^T,$$
and form the Moore-Penrose inverse, as in equation (5.80):

\[ A^+ = U \begin{bmatrix} R_1^{-1} & 0 \\ 0 & 0 \end{bmatrix} Q^T. \]

Then

\[ \hat{x} = A^+ b \]

is a least squares solution, just as in the full rank case. Now, let

\[ Q^T b = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}, \]

as above, except \( c_1 \) is of length \( r \) and \( c_2 \) is of length \( n - r \), and let

\[ U^T x = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}, \]

where \( z_1 \) is of length \( r \). We proceed as in the equations (5.96) (except here we use the \( L_2 \) norm notation). We seek to minimize \( \| b - Ax \|_2 \); and because multiplication by an orthogonal matrix does not change the norm, we have

\[ \| b - Ax \|_2 = \| Q^T (b - AU U^T x) \|_2 \]

\[ = \left\| \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} - \begin{bmatrix} R_1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \right\|_2 \]

\[ = \left\| \begin{bmatrix} c_1 - R_1 z_1 \\ c_2 \end{bmatrix} \right\|_2. \]  

(5.99)

The residual norm is minimized for \( z_1 = R_1^{-1} c_1 \) and \( z_2 \) arbitrary. However, if \( z_2 = 0 \), then \( \| z \|_2 \) is also minimized. Because \( U^T x = z \) and \( U \) is orthogonal, \( \| \hat{x} \|_2 = \| z \|_2 \), and so \( \| \hat{x} \|_2 \) is minimized.

**Updating a Solution to an Overdetermined System**

In the last section we considered the problem of updating a given solution to be a solution to a perturbed consistent system. An overdetermined system is often perturbed by adding either some rows or some columns to the coefficient matrix \( A \). This corresponds to including additional equations in the system,

\[ \begin{bmatrix} A \\ A^+ \end{bmatrix} x \approx \begin{bmatrix} b \\ b^+ \end{bmatrix}, \]

or to adding variables,

\[ \begin{bmatrix} A \\ A^+ \end{bmatrix} \begin{bmatrix} x \\ x^+ \end{bmatrix} \approx b. \]
In either case, if the $QR$ decomposition of $A$ is available, the decomposition of the augmented system can be computed readily. Consider, for example, the addition of $k$ equations to the original system $Ax \approx b$, which has $n$ approximate equations. With the $QR$ decomposition, for the original full rank system, putting $Q^T A$ and $Q^T b$ as partitions in a matrix, we have

$$
\begin{bmatrix}
R_1 & c_1 \\
0 & c_2
\end{bmatrix} = Q^T \begin{bmatrix}
A & b
\end{bmatrix}.
$$

Augmenting this with the additional rows yields

$$
\begin{bmatrix}
R & c_1 \\
0 & c_2
\end{bmatrix} = \begin{bmatrix}
Q^T & 0 \\
0 & I
\end{bmatrix} \begin{bmatrix}
A & b \\
A_+ & b_+
\end{bmatrix}.
$$

(5.100)

All that is required now is to apply orthogonal transformations, such as Givens rotations, to the system (5.100) to produce

$$
\begin{bmatrix}
R_* & c_{1*} \\
0 & c_{2*}
\end{bmatrix},
$$

where $R_*$ is an $m \times m$ upper triangular matrix and $c_{1*}$ is an $m$-vector as before, but $c_{2*}$ is an $(n-m+k)$-vector.

The updating is accomplished by applying $m$ rotations to (5.100) so as to zero out the $(n+q)^{th}$ row, for $q = 1, 2, \ldots, k$. These operations go through an outer loop with $p = 1, 2, \ldots, n$, and an inner loop with $q = 1, 2, \ldots, k$. The operations rotate $R$ through a sequence $R^{(p,q)}$ into $R_*$, and they rotate $A_+$ through a sequence $A_+^{(p,q)}$ into 0. At the $p, q$ step, the rotation matrix $Q_{pq}$ corresponding to (5.82), page 293, has

$$
\cos \theta = \frac{R^{(p,q)}_{pp}}{r}
$$

and

$$
\sin \theta = \frac{(A_+^{(p,q)})_{q,p}}{r},
$$

where

$$
r = \sqrt{(R^{(p,q)}_{pp})^2 + (A_+^{(p,q)})_{q,p}^2}.
$$

Gentleman (1974) and Miller (1992) give Fortran programs that implement this kind of updating. The software from Applied Statistics is available in statlib (see page ??).

5.2.8 Other Computations for Linear Systems
**Rank Determination**

It is often easy to determine that a matrix is of full rank. If the matrix is not of full rank, however, or if it is very ill-conditioned, it is difficult to determine its rank. This is because the computations to determine the rank eventually approximate 0. It is difficult to approximate 0; the relative error (if defined) would be either 0 or infinite. The rank-revealing QR factorization (equation (5.78), page 288) is the preferred method to estimate the rank. When this decomposition is used to estimate the rank, it is recommended that complete pivoting be used in computing the decomposition. The LDU decomposition, described on page 284, can be modified the same way we used the modified QR to estimate the rank of a matrix. Again, it is recommended that complete pivoting be used in computing the decomposition.

The singular value decomposition (SVD) shown in (5.42), page 255, also provides an indication of the rank of the matrix. For the $n \times m$ matrix $A$, the SVD is

$$A = U \Sigma V^T,$$

where $U$ is an $n \times n$ orthogonal matrix, $V$ is an $m \times m$ orthogonal matrix, and $\Sigma$ is a diagonal matrix of the singular values. The number of nonzero singular values is the rank of the matrix. Of course, again, the question is whether or not the singular values are zero. It is unlikely that the values computed are exactly zero.

A problem related to rank determination is to approximate the matrix $A$ with a matrix $A_r$ of rank $r \leq \text{rank}(A)$. The singular value decomposition provides an easy way to do this:

$$A_r = U \Sigma_r V^T,$$

where $\Sigma_r$ is the same as $\Sigma$, except with zeros replacing all but the $r$ largest singular values. A result of Eckart and Young (1936) guarantees $A_r$ is the rank $r$ matrix closest to $A$ as measured by the Frobenius norm,

$$\|A - A_r\|_F.$$

This kind of matrix approximation is the basis for dimension reduction by principal components, which we discuss in Section 13.4.3, beginning on page 589.

**Computing the Determinant**

The determinant of a square matrix can be obtained easily as the product of the diagonal elements of the triangular matrix in any factorization that yields an orthogonal matrix times a triangular matrix. As we have stated before, it is not often that the determinant need be computed, however.
Computing the Condition Number

The computation of a condition number of a matrix can be quite involved. Various methods have been proposed to estimate the condition number using relatively simple computations. Cline et al. (1979) suggest a method that is easy to perform and is widely used. For a given matrix $A$ and some vector $v$, solve

$$A^T x = v,$$

and then

$$Ay = x.$$

By tracking the computations in the solution of these systems, Cline et al. conclude that

$$\frac{\|y\|}{\|x\|}$$

is approximately equal to, but less than, $\|A^{-1}\|$. This estimate is used with respect to the $L_1$ norm in LINPACK, but the approximation is valid for any norm. Solving the two systems above probably does not require much additional work because the original problem was likely to solve $Ax = b$, and solving a system with multiple right-hand sides can be done efficiently using the solution to one of the right-hand sides. The approximation is better if $v$ is chosen so that $\|x\|$ is as large as possible relative to $\|v\|$.

Stewart (1980) and Cline and Rew (1983) investigated the validity of the approximation. The LINPACK estimator can underestimate the true condition number considerably, although generally not by an order of magnitude. Cline, Conn, and Van Loan (1982) give a method of estimating the $L_2$ condition number of a matrix that is a modification of the $L_1$ condition number used in LINPACK. This estimate generally performs better than the $L_1$ estimate, but the Cline/Conn/Van-Loan estimator still can have problems (see Bischof, 1990).

Hager (1984) gives another method for an $L_1$ condition number. Higham (1988) provides an improvement of Hager’s method, given as Algorithm 5.4 below, which is used in LAPACK.

Algorithm 5.4 The Hager/Higham LAPACK Condition Number Estimator $\gamma$ of the $n \times n$ Matrix $A$

Assume $n > 1$; else $\gamma = |A1|$. (All norms are $L_1$ unless specified otherwise.)

0. Set $k = 1$; $v^{(k)} = \frac{1}{n} A1$; $\gamma^{(k)} = \|v^{(k)}\|$; and $x^{(k)} = A^T \text{sign}(v^{(k)})$.
1. Set $j = \min\{i, \text{s.t. } |x_i^{(k)}| = \|x^{(k)}\|_\infty\}$.
2. Set $k = k + 1$.
3. Set $v^{(k)} = Ae_j$.
4. Set $\gamma^{(k)} = \|v^{(k)}\|$.
5. If $\text{sign}(v^{(k)}) = \text{sign}(v^{(k-1)})$ or $\gamma^{(k)} \leq \gamma^{(k-1)}$, then go to step 8.
6. Set $x^{(k)} = A^T \text{sign}(v^{(k)})$. 
7. If \( \|x^{(k)}\|_{\infty} \neq x_j^{(k)} \) and \( k \leq k_{\text{max}} \) then go to step 1.

8. For \( i = 1, 2, \ldots, n \), set \( x_i = (-1)^{i+1} \left( 1 + \frac{i-1}{n} \right) \).

9. Set \( x = Ax \).

10. If \( \frac{2 \|x\|}{(3n)} > \gamma^{(k)} \), set \( \gamma^{(k)} = \frac{2 \|x\|}{(3n)} \).

11. Set \( \gamma = \gamma^{(k)} \).

Higham (1987) compares Hager’s condition number estimator with that of Cline et al. (1979) and finds that the Hager LAPACK estimator is generally more useful. Higham (1990) gives a survey and comparison of the various ways of estimating and computing condition numbers. You are asked to study the performance of the LAPACK estimate using Monte Carlo in Exercise ??, page ??.

### 5.3 Computation of Eigenvectors and Eigenvalues and the Singular Value Decomposition

Before we discuss methods for computing eigenvalues, we mention an interesting observation. Consider the polynomial, \( f(\lambda) \),

\[
\lambda^p + a_{p-1}\lambda^{p-1} + \cdots + a_1\lambda + a_0.
\]

Now form the matrix, \( A \),

\[
\begin{bmatrix}
0 & 1 & 0 & \cdots & 0 \\
0 & 0 & 1 & \cdots & 0 \\
& & \ddots & & \\
0 & 0 & 0 & \cdots & 1 \\
-a_0 & -a_1 & -a_2 & \cdots & -a_{p-1}
\end{bmatrix}
\]

The matrix \( A \) is called the *companion matrix* of the polynomial \( f(\lambda) \). It is easy to see that the characteristic equation of \( A \), equation (5.38) on page 253, is the polynomial \( f(\lambda) \):

\[
\det(A - \lambda I) = f(\lambda).
\]

Thus, given a general polynomial \( f \), we can form a matrix \( A \) whose eigenvalues are the roots of the polynomial. It is a well-known fact in the theory of equations that there is no general formula for the roots of a polynomial of degree greater than 4. This means that we cannot expect to have a direct method for calculating eigenvalues; rather, we will have to use an iterative method.

In statistical applications, the matrices whose eigenvalues are of interest are almost always symmetric. Because the eigenvalues of a symmetric (real) matrix are real, the problem of determining the eigenvalues of a symmetric matrix is simpler than the corresponding problem for a general matrix.
5.3 Computation of Eigenvectors and Eigenvalues and the Singular Value Decomposition

We describe three methods for computing eigenvalues — the power method, the Jacobi method, and the QR method. Each method has some desirable property for particular applications. A QR-type method can also be used effectively to evaluate singular values.

If \( v \) is an eigenvector of \( A \), the corresponding eigenvalue is easy to determine; it is the common ratio \((Av)_i/v_i\). Likewise, if the eigenvalue \( \lambda \) is known, the corresponding eigenvector is the solution to the system

\[
(A - \lambda I)v = 0.
\]

5.3.1 Power Method

Let \( A \) be a real \( n \times n \) symmetric matrix with eigenvalues \( \lambda_i \) indexed so that

\[
|\lambda_1| \leq |\lambda_2| \leq \cdots \leq |\lambda_n|,\]

with corresponding unit eigenvectors \( v_i \). We restrict our attention to simple matrices (see page 253), and assume that \( \lambda_{n-1} < \lambda_n \) (i.e., \( \lambda_n \) and \( v_n \) are unique). In this case \( \lambda_n \) is called the dominant eigenvalue

and \( v_n \) is called the dominant eigenvector. Now let \( x(0) \) be an \( n \)-vector that is not orthogonal to \( v_n \).

Because \( A \) is assumed to be simple, \( x(0) \) can be represented as a linear combination of the eigenvectors:

\[
x(0) = c_1 v_1 + c_2 v_2 + \cdots + c_n v_n,
\]

and because \( x(0) \) is not orthogonal to \( v_n \), \( c_n \neq 0 \). The power method is based on a sequence that continues the finite Krylov space generating set:

\[
x(0), Ax(0), A^2 x(0), \ldots
\]

From the relationships above and the definition of eigenvalues and eigenvectors, we have

\[
A x(0) = c_1 A v_1 + c_2 A v_2 + \cdots + c_n A v_n
= c_1 \lambda_1 v_1 + c_2 \lambda_2 v_2 + \cdots + c_n \lambda_n v_n
\]

\[
A^2 x(0) = c_1 \lambda_1^2 v_1 + c_2 \lambda_2^2 v_2 + \cdots + c_n \lambda_n^2 v_n
\]

\[
\vdots
\]

\[
A^j x(0) = c_1 \lambda_1^j v_1 + c_2 \lambda_2^j v_2 + \cdots + c_n \lambda_n^j v_n
= \lambda_n^j \left( c_1 \left( \frac{\lambda_1}{\lambda_n} \right)^j v_1 + c_2 \left( \frac{\lambda_2}{\lambda_n} \right)^j v_2 + \cdots + c_n v_n \right).
\] (5.101)

To simplify the notation, let \( u^{(j)} = A^j x(0)/\lambda_n^j \) (or, equivalently, \( u^{(j)} = A u^{(j-1)}/\lambda_n \)). From (5.101) and the fact that \( |\lambda_i| < |\lambda_n| \) for \( i < n \), we see that \( u^{(j)} \to c_n v_n \), which is the unnormalized dominant eigenvector.

We have the bound
\[ \left\| u^{(j)} - c_n v_n \right\| = \left\| c_1 \left( \frac{\lambda_1}{\lambda_n} \right)^j v_1 + c_2 \left( \frac{\lambda_2}{\lambda_n} \right)^j v_2 + \cdots + c_{n-1} \left( \frac{\lambda_{n-1}}{\lambda_n} \right)^j v_{n-1} \right\| \]

\[ \leq |c_1| \left| \frac{\lambda_1}{\lambda_n} \right|^j \| v_1 \| + |c_2| \left| \frac{\lambda_2}{\lambda_n} \right|^j \| v_2 \| + \cdots + |c_{n-1}| \left| \frac{\lambda_{n-1}}{\lambda_n} \right|^j \| v_{n-1} \| \]

\[ \leq (|c_1| + |c_2| + \cdots + |c_{n-1}|) \left| \frac{\lambda_{n-1}}{\lambda_n} \right|^j. \quad (5.102) \]

The last expression results from the facts that \(|\lambda_i| \leq |\lambda_{n-1}|\) for \(i < n - 1\) and that the \(v_i\) are unit vectors.

From (5.102), we see that the norm of the difference of \(u^{(j)}\) and \(c_n v_n\) decreases by a factor of approximately \(|\lambda_{n-1}/\lambda_n|\) with each iteration; hence, this ratio is an important indicator of the rate of convergence of \(u^{(j)}\) to the dominant eigenvector.

If \(|\lambda_{n-2}| < |\lambda_{n-1}| < |\lambda_n|\), \(c_{n-1} \neq 0\), and \(c_n \neq 0\) the power method converges linearly (see page 64); that is,

\[ 0 < \lim_{j \to \infty} \frac{\left\| x^{(j+1)} - c_n v_n \right\|}{\left\| x^{(j)} - c_n v_n \right\|} < 1 \quad (5.103) \]

(see Exercise 5.24c, page 344).

If an approximate value of the eigenvector \(v_n\) is available and \(x^{(0)}\) is taken to be that approximate value, the convergence will be faster. If an approximate value of the dominant eigenvalue, \(\hat{\lambda}_n\), is available, starting with any \(y^{(0)}\), a few iterations on

\[ (A - \hat{\lambda}_n I)y^{(k)} = y^{(k-1)} \]

may yield a better starting value for \(x^{(0)}\).

Once the dominant eigenvector is determined, the dominant eigenvalue \(\lambda_n\) can be easily determined.

In some applications, only the dominant eigenvalue is of interest. If other eigenvalues are needed, however, we can use the known dominant eigenvalue and eigenvector to compute the others. Whenever one eigenvalue, \(\lambda_i\), and corresponding eigenvector, \(v_i\), of a matrix \(A\) are available, another matrix can be formed that has all the same nonzero eigenvalues and corresponding eigenvectors as \(A\), except for the \(i^{th}\) one. To see how to do this, let \(\lambda_j\) be an eigenvalue of \(A\) such that \(\lambda_j \neq \lambda_i\). Now, \(\lambda_j\) is also an eigenvalue of \(A^T\) (see the properties listed on page 252). Let \(w_j\) be the corresponding eigenvector of \(A^T\). Now,

\[ \langle Av_j, w_j \rangle = \langle \lambda_j v_j, w_j \rangle = \lambda_j \langle v_j, w_j \rangle \]
5.3 Computation of Eigenvectors and Eigenvalues and the Singular Value Decomposition

But also,

$$\langle Av_i, w_j \rangle = \langle v_i, A^T w_j \rangle = \langle v_i, \lambda_j w_j \rangle = \lambda_j \langle v_i, w_j \rangle.$$  

But if

$$\lambda_i \langle v_i, w_j \rangle = \lambda_j \langle v_i, w_j \rangle,$$

and $\lambda_j \neq \lambda_i$, then $\langle v_i, w_j \rangle = 0$. Now let $w_i$ be the eigenvector of $A^T$ corresponding to $\lambda_i$, and consider the matrix

$$B = A - \lambda_i w_i w_i^T. \quad (5.104)$$

We see that

$$Bw_j = Aw_j - \lambda_i w_i w_i^T w_j = Aw_j = \lambda_j w_j,$$

so $\lambda_j$ and $w_j$ are respectively an eigenvalue and an eigenvector of $B$. This gives us a way to use the power method to find the second largest eigenvalue once the largest one is found.

If $A$ is nonsingular, we can also use the power method on $A^{-1}$ to determine the smallest eigenvalue of $A$.

### 5.3.2 Jacobi Method

The Jacobi method for determining the eigenvalues of a simple symmetric matrix $A$ uses a sequence of orthogonal similarity transformations that eventually result in the transformation

$$A = P \Lambda P^{-1},$$

(see equation (5.38), page 253) or

$$A = P^{-1} A P,$$

where $A$ is diagonal. Recall that similar matrices have the same eigenvalues.

The matrices for the similarity transforms are the Givens rotation or Jacobi rotation matrices discussed on page 291. The general form of one of these orthogonal matrices, $Q_{pq}(\theta)$, given in (5.82) on page 293, is the identity matrix with $\cos \theta$ in the $(p, p)^{th}$ and $(q, q)^{th}$ positions, $\sin \theta$ in the $(p, q)^{th}$ position, and $-\sin \theta$ in the $(q, p)^{th}$ position:

$$Q_{pq}(\theta) = P \begin{bmatrix} 1 & p & q \\ 0 & \cos \theta & 0 & 0 \\ 0 & 0 & I & 0 \\ q & 0 & -\sin \theta \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$
The Jacobi iteration is
\[ A^{(k)} = Q_{p_k q_k}^T (\theta_k) A^{(k-1)} Q_{p_k q_k} (\theta_k), \]
where \( p_k, q_k, \) and \( \theta_k \) are chosen so that the \( A^{(k)} \) is “more diagonal” than \( A^{(k-1)} \). Specifically, the iterations will be chosen so as to reduce the sum of the squares of the off-diagonal elements, which for any square matrix \( A \) is
\[ \| A \|_F^2 - \sum_i a_{ii}^2. \]

The orthogonal similarity transformations preserve the Frobenius norm
\[ \| A^{(k)} \|_F = \| A^{(k-1)} \|_F. \]

Because the rotation matrices change only the elements in the \((p, p)^{th}\), \((q, q)^{th}\), and \((p, q)^{th}\) positions (and also the \((q, p)^{th}\) position since both matrices are symmetric), we have
\[ (a_{pp}^{(k)})^2 + (a_{qq}^{(k)})^2 + 2 (a_{pq}^{(k)})^2 = (a_{pp}^{(k-1)})^2 + (a_{qq}^{(k-1)})^2 + 2 (a_{pq}^{(k-1)})^2. \]

The off-diagonal sum of squares at the \( k^{th} \) stage in terms of that at the \((k-1)^{th}\) stage is
\[ \| A^{(k)} \|_F^2 - \sum_i (a_{ii}^{(k)})^2 = \| A^{(k)} \|_F^2 - \sum_{i \neq p,q} (a_{ii}^{(k)})^2 - \left( (a_{pp}^{(k)})^2 + (a_{qq}^{(k)})^2 \right) \]
\[ = \| A^{(k-1)} \|_F^2 - \sum_i (a_{ii}^{(k-1)})^2 - 2 (a_{pq}^{(k-1)})^2 + 2 (a_{pq}^{(k)})^2. \]

Hence, for a given index pair, \((p, q)\), at the \( k^{th} \) iteration the sum of the squares of the off-diagonal elements is minimized by choosing the rotation matrix so that
\[ a_{pq}^{(k)} = 0. \]

As we saw on page 293, it is easy to determine the angle \( \theta \) so as to introduce a zero in a single Givens rotation. Here, we are using the rotations in a similarity transformation, so it is a little more complicated.

The requirement that \( a_{pq}^{(k)} = 0 \) implies
\[ a_{pq}^{(k-1)} (\cos^2 \theta - \sin^2 \theta) + \left( a_{pp}^{(k-1)} - a_{qq}^{(k-1)} \right) \cos \theta \sin \theta = 0. \]

Using the trigonometric identities
\[ \cos(2\theta) = \cos^2 \theta - \sin^2 \theta \]
\[ \sin(2\theta) = 2 \cos \theta \sin \theta, \]
in (5.107), we have
\[ \tan(2\theta) = \frac{2a^{(k-1)}_{pq}}{a^{(k-1)}_{pp} - a^{(k-1)}_{qq}}, \]
which yields a unique angle in \([-\pi/4, \pi/4]\). Of course, the quantities we need are \( \cos \theta \) and \( \sin \theta \), not the angle itself. First, using the identity
\[ \tan \theta = \frac{\tan(2\theta)}{1 + \sqrt{1 + \tan^2(2\theta)}}, \]
we get \( \tan \theta \) from \( \tan(2\theta) \); and then from \( \tan \theta \), we can compute the quantities required for the rotation matrix \( Q_{pq}(\theta) \):
\[ \cos \theta = \frac{1}{\sqrt{1 + \tan^2 \theta}}, \]
\[ \sin \theta = \cos \theta \tan \theta. \]

Convergence occurs when the off-diagonal elements are sufficiently small. The quantity (5.105) using the Frobenius norm is the usual value to compare with a convergence criterion, \( \epsilon \).

From (5.106) we see that the best index pair, \((p, q)\), is such that
\[ \left| a^{(k-1)}_{pq} \right| = \max_{i \neq j} \left| a^{(k-1)}_{ij} \right|. \]
If this choice is made, the Jacobi method can be shown to converge. The method with this choice is called the classical Jacobi method.

For an \( n \times n \) matrix, the number of operations to identify the maximum off-diagonal is \( O(n^2) \). The computations for the similarity transform itself are only \( O(n) \) because of the sparsity of the rotators. Of course the computations for the similarity transformations are more involved than those to identify the maximum off-diagonal, so for small \( n \), the classical Jacobi method should be used. If \( n \) is large, however, it may be better not to spend time looking for the maximum off-diagonal. Various cyclic Jacobi methods have been proposed, in which the pairs \((p, q)\) are chosen systematically without regard to the magnitude of the off-diagonal being zeroed. Depending on the nature of the cyclic Jacobi method, it may or may not be guaranteed to converge. For certain schemes, quadratic convergence has been proven; for at least one other scheme, an example showing failure of convergence has been given. See Watkins (2002) for a discussion of the convergence issues.

The Jacobi method is one of the oldest algorithms for computing eigenvalues, and has recently become important again because it lends itself to easy implementation on parallel processors.

Notice that at the \( k^{th} \) iteration, only two rows and two columns of \( A^{(k)} \) are modified. This is what allows the Jacobi method to be performed in parallel. We can form \([n/2]\) pairs, and do \([n/2]\) rotations simultaneously. Thus, each parallel iteration consists of a choice of a set of index pairs and then a batch
of rotations. Although, as we have indicated, the convergence may depend on which rows are chosen for the rotations, if we are to achieve much efficiency by performing the operations in parallel, we cannot spend much time in deciding how to form the pairs for the rotations. Various schemes have been suggested for forming the pairs for a parallel iteration. Luk and Park (1989) have analyzed some of the proposed schemes. A simple scheme, called “mobile Jacobi” (see Watkins, 2002), is

1. perform $\lfloor n/2 \rfloor$ rotations using the pairs 
   $$(1, 2), \ (3, 4), \ (5, 6), \ldots$$

2. interchange all rows and columns that were rotated

3. perform $\lfloor (n − 1)/2 \rfloor$ rotations using the pairs 
   $$(2, 3), \ (4, 5), \ (6, 7), \ldots$$

4. interchange all rows and columns that were rotated

5. if convergence has not been achieved, go to 1.

The notation above that specifies the pairs refers to the rows and columns at the current state; that is, after the interchanges up to that point. The interchange operation is a similarity transformation using an elementary permutation matrix (see page 249), hence the eigenvalues are left unchanged by this operation.

### 5.3.3 QR Method for Eigenanalysis

The most common algorithm for extracting eigenvalues is the QR method. The method devised by Francis (1961a, 1961b) can be used for nonsymmetric matrices. It is simpler for symmetric matrices, of course, because the eigenvalues are real. Also for symmetric matrices the computer storage is less, the computations are fewer, and some transformations are particularly simple.

The QR method requires that the matrix first be transformed into upper Hessenberg form. A matrix is in upper Hessenberg form, and is called a Hessenberg matrix, if it is upper triangular except for the first subdiagonal, which may be nonzero. That is, $a_{ij} = 0$ for $i > j + 1$:

\[
\begin{bmatrix}
X & X & \cdots & X \\
X & X & \cdots & X \\
0 & X & \cdots & X \\
0 & 0 & \cdots & X \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & X
\end{bmatrix}
\]

A matrix can be reduced to Hessenberg form in a finite number of similarity transformations, using either Householder reflections or Givens rotations.
The Hessenberg form for a symmetric matrix is tridiagonal. The Hessenberg form allows a large savings in the subsequent computations, even for nonsymmetric matrices.

The $QR$ method for determining the eigenvalues is iterative and produces a sequence of Hessenberg matrices $A^{(0)}, A^{(1)}, A^{(2)}, \ldots$, which converges to a triangular matrix.

The eigenvalues of a matrix in upper Hessenberg form are extracted by a process called “chasing”, which consists of steps that alternate between creating nonzero entries in positions $(i+2, i)$, $(i+3, i)$, and $(i+3, i+1)$ and restoring these entries to zero, as the nonzero entries are moved farther down the matrix. For example,

\[
\begin{bmatrix}
X & X & X & X & X & X & X \\
X & X & X & X & X & X & X \\
0 & X & X & X & X & X & X \\
0 & Y & X & X & X & X & X \\
0 & Y & Y & X & X & X & X \\
0 & 0 & 0 & 0 & X & X & X \\
0 & 0 & 0 & 0 & 0 & X & X \\
\end{bmatrix}
\rightarrow
\begin{bmatrix}
X & X & X & X & X & X & X \\
X & X & X & X & X & X & X \\
0 & X & X & X & X & X & X \\
0 & Y & X & X & X & X & X \\
0 & Y & Y & X & X & X & X \\
0 & 0 & 0 & 0 & X & X & X \\
0 & 0 & 0 & 0 & 0 & X & X \\
\end{bmatrix}.
\]

In the $j^{th}$ step of the $QR$ method, a bulge is created and is chased down the matrix by similarity transformations, usually Givens transformations,

\[G_k^{-1}A^{(j-1,k)}G_k.\]

The transformations are based on the eigenvalues of $2 \times 2$ matrices in the lower right-hand part of the matrix.

There are some variations on the way the chasing occurs. Haag and Watkins (1993) describe an efficient modified $QR$ algorithm that uses both Givens transformations and Gaussian elimination transformations, with or without pivoting. For the $n \times n$ Hessenberg matrix $A^{(0,0)}$, the first step of the Haag-Watkins procedure begins with a $3 \times 3$ Householder reflection matrix, $\tilde{G}_0$, whose first column is

\[(A^{(0,0)} - \sigma_1 I)(A^{(0,0)} - \sigma_2 I)e_1,
\]

where $\sigma_1$ and $\sigma_2$ are the eigenvalues of the $2 \times 2$ matrix

\[
\begin{bmatrix}
a_{n-1,n-1} & a_{n-1,n} \\
a_{n-1,n} & a_{n,n}
\end{bmatrix},
\]

and $e_1$ is the first unit vector of length $n$. The $n \times n$ matrix $G_0$ is $\text{diag}(\tilde{G}_0, I)$.

The initial transformation $G_0^{-1}A^{(0,0)}G_0$ creates a bulge with nonzero elements $a_{31}^{(0,1)}$, $a_{41}^{(0,1)}$, and $a_{52}^{(0,1)}$.

After the initial transformation, the Haag-Watkins procedure makes $n-3$ transformations.
for \( k = 1, 2, \ldots, n-3 \), that chase the bulge diagonally down the matrix, so that
\( A^{(0,k+1)} \) differs from Hessenberg form only by the nonzero elements
\( a_{k+3,k+1}^{(0,k+1)} \),
\( a_{k+4,k+1}^{(0,k+1)} \), and \( a_{k+4,k+2}^{(0,k+1)} \). To accomplish this, the matrix \( G_k \) differs from the
identity only in rows and columns \( k+1 \), \( k+2 \), and \( k+3 \). The transformation
\[
G_k^{-1} A^{(0,k)}
\]
annihilates the entries \( a_{k+2,k}^{(0,k)} \) and \( a_{k+3,k}^{(0,k)} \), and the transformation
\[
(G_k^{-1} A^{(0,k)}) G_k
\]
produces \( A^{(0,k+1)} \) with two new nonzero elements \( a_{k+1,k+1}^{(0,k+1)} \) and \( a_{k+1,k+2}^{(0,k+1)} \). The
final transformation in the first step, for \( k = n-2 \), annihilates \( a_{n,n-2}^{(0,k)} \). The
transformation matrix \( G_{n-2} \) differs from the identity only in rows and columns
\( n-1 \) and \( n \). These steps are iterated until the matrix becomes triangular.
As the subdiagonal elements converge to zero, the shifts for use in the first
transformation of a step (corresponding to \( \sigma_1 \) and \( \sigma_2 \)) are determined by
2 \( \times \) 2 submatrices higher on the diagonal. Special consideration must be given
to situations in which these submatrices contain zero elements. For this, the

This description has just indicated the general flavor of the \( QR \) method.
There are different variations on the overall procedure, and then many computa-
tional details that must be observed. In the Haag-Watkins procedure, for
example, the \( G_k \)'s are not unique; and their form can affect the efficiency and
the stability of the algorithm. Haag and Watkins (1993) describe criteria for
the selection of the \( G_k \)'s. They also discuss some of the details of programming
the algorithm.

### 5.3.4 Singular Value Decomposition

The standard algorithm for computing the singular value decomposition
\[
A = U \Sigma V^T
\]
is due to Golub and Reinsch (1970), and is built on ideas of Golub and Ka-
han (1965). The first step in the Golub-Reinsch algorithm for the singular
value decomposition of the \( n \times m \) matrix \( A \) is to reduce \( A \) to upper bidiagonal
form:
5.3 Computation of Eigenvectors and Eigenvalues and the Singular Value Decomposition

\[ A^{(0)} = \begin{bmatrix} X & X & 0 & \cdots & 0 & 0 \\ 0 & X & X & \cdots & 0 & 0 \\ 0 & 0 & X & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & X & X \\ 0 & 0 & 0 & \cdots & 0 & X \\ 0 & 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 0 \end{bmatrix}. \]

We assume \( n \geq m \). (If this is not the case, we merely use \( A^T \).) This algorithm is basically a factored form of the QR algorithm for the eigenvalues of \( A^{(0)T}A^{(0)} \), which would be symmetric and tridiagonal.

The Golub-Reinsch method produces a sequence of upper bidiagonal matrices, \( A^{(0)}, A^{(1)}, A^{(2)}, \ldots \), which converges to the diagonal matrix \( \Sigma \). (Each of these has a zero submatrix below the square submatrix.) Similarly to the QR method for eigenvalues, the transformation from \( A^{(j)} \) to \( A^{(j+1)} \) is effected by a sequence of orthogonal transformations,

\[
A^{(j+1)} = R^T_{m-2} R^T_{m-3} \cdots R^T_0 A^{(j)} T_0 T_1 \cdots T_{m-2} = R^T A^{(j)} T,
\]

which first introduces a nonzero entry below the diagonal (\( T_0 \) does this) and then chases it down the diagonal. After \( T_0 \) introduces a nonzero entry in the (2,1) position, \( R^T_0 \) annihilates it and produces a nonzero entry in the (1,3) position; \( T_1 \) annihilates the (1,3) entry and produces a nonzero entry in the (3,2) position, which \( R^T_1 \) annihilates, and so on. Each of the \( R_k \)'s and \( T_k \)'s are Givens transformations, and, except for \( T_0 \), it should be clear how to form them.

If none of the elements along the main diagonal or the diagonal above the main diagonal is zero, then \( T_0 \) is chosen as the Givens transformation such that \( T_0^T \) will annihilate the second element in the vector

\[(a_1^2 - \sigma_1, a_{11}a_{12}, 0, \cdots, 0),\]

where \( \sigma_1 \) is the eigenvalue of the lower right-hand \( 2 \times 2 \) submatrix of \( A^{(0)T}A^{(0)} \) that is closest in value to the \((m,m)\) element of \( A^{(0)T}A^{(0)} \). This is easy to compute (see Exercise 5.29).

If an element along the main diagonal or the diagonal above the main diagonal is zero, we must proceed slightly differently. (Remember that for purposes of computations, “zero” generally means “near zero”, that is, to within some set tolerance.)

If an element above the main diagonal is zero, the bidiagonal matrix is separated at that value into a block diagonal matrix, and each block (which is bidiagonal) is treated separately.
If an element on the main diagonal, say $a_{kk}$, is zero, then a singular value is zero. In this case we apply a set of Givens transformations from the left. We first use $G_1$, which differs from the identity only in rows and columns $k$ and $k + 1$, to annihilate the $(k, k + 1)$ entry and introduce a nonzero in the $(k, k + 2)$ position. We then use $G_2$, which differs from the identity only in rows and columns $k$ and $k + 2$, to annihilate the $(k, k + 2)$ entry and introduce a nonzero in the $(k, k + 3)$ position. Continuing this process, we form a matrix of the form

$$
\begin{bmatrix}
XX0 & 0 & 0 & 0 & 0 \\
0XX & 0 & 0 & 0 & 0 \\
00X & Y & 0 & 0 & 0 \\
000 & 0 & 0 & 0 & 0 \\
000 & 0 & XX0 & 0 & 0 \\
000 & 0 & 0XX0 & 0 & 0 \\
000 & 0 & 00XX & 0 & 0 \\
000 & 0 & 000X & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
000 & 0 & 0000 & 0 & 0
\end{bmatrix}.
$$

The $Y$ in this matrix (in position $(k - 1, k)$) is then chased up the upper block consisting of the first $k$ rows and columns of the original matrix by using Givens transformations applied from the right. This then yields two block bidiagonal matrices (and a $1 \times 1$ $0$ matrix). We operate on the individual blocks as before.

After the steps have converged to yield a diagonal matrix, $\tilde{\Sigma}$, all of the Givens matrices applied from the left are accumulated into a single matrix, and all from the right are accumulated into a single matrix, to yield a decomposition

$$A = \tilde{U} \tilde{\Sigma} \tilde{V}^T.$$

There is one last thing to do. The elements of $\tilde{\Sigma}$ may not be nonnegative. This is easily remedied by postmultiplying by a diagonal matrix $D$ that is the same as the identity except for having a $-1$ in any position corresponding to a negative value in $\Sigma$. In addition, we generally form the singular value decomposition is such a way that the elements in $\Sigma$ are nonincreasing. The entries in $\tilde{\Sigma}$ can be rearranged by a permutation matrix $P$ so they are in nonincreasing order. So we have

$$\Sigma = P^T \tilde{\Sigma} DP,$$

and the final decomposition is

$$A = \tilde{U} PD \Sigma P^T \tilde{V}^T$$

$$= U \Sigma V^T.$$

If $n \geq \frac{2}{3}m$, a modification of this algorithm by Chan (1982a, 1982b) is more efficient than the standard Golub-Reinsch method.
5.4 Software for Numerical Linear Algebra

Because of the importance of linear algebraic computations, there is a wide range of software for these computations. The Guide to Available Mathematical Software (GAMS) (see the bibliography) is a good source of information about software.

For some types of software, it is important to be aware of the way the data are stored in the computer, as we discussed in Section 5.1.2, beginning on page 274. This may include such things as whether the storage is row-major or column-major, which will determine the stride, and may determine the details of an algorithm so as to enhance the efficiency. Software written in a language such as Fortran or C often requires the specification of the number of rows (in Fortran) or columns (in C) that have been allocated for the storage of a matrix. As we have indicated before, the amount of space allocated for the storage of a matrix may not correspond exactly to the size of the matrix.

There are many issues to consider in evaluating software or to be aware of when developing software. The portability of the software is an important consideration because a user’s programs are often moved from one computing environment to another.

Some situations require special software that is more efficient than general-purpose software would be. Software for sparse matrices, for example, is specialized to take advantage of the zero entries. For sparse matrices it is necessary to have a scheme for identifying the locations of the nonzeros, and for specifying their values. The nature of storage schemes varies from one software package to another. The reader is referred to GAMS as a resource for information about software for sparse matrices.

Occasionally we need to operate on vectors or matrices whose elements are variables. Software for symbolic manipulation, such as Maple, can perform vector/matrix operations on variables. See Exercise 5.35, page 346.

5.4.1 Fortran and C

Fortran and C are the most commonly used procedural languages for scientific computation. Fortran has evolved over many years of usage by scientists and engineers.

C began as a low-level language that provided many of the capabilities of a higher-level language together with more direct access to the operating system. It lacks some of the facilities that are very useful in scientific computation, such as complex data types, an exponentiation operator, and direct manipulation of arrays as vectors or matrices. An advantage of C, however, is that it provides for easier communication between program units, so it is often used when larger program systems are being put together.
Several libraries of program modules for numerical linear algebra are available both in Fortran and in C.

Newer versions of Fortran, Fortran 95 and 95 (see Kerrigan, 1993, Metcalf and Reid, 1999, or Press et al., 1996), and Fortran 2000, provide additional facilities for working directly with arrays.

Class libraries can be built in C++ to provide similar capabilities as are available in Fortran 9x and Fortran 2000.

**Indexing Arrays**

Neither Fortran 77 nor C allows vectors and matrices to be treated as atomic units. Numerical operations on vectors and matrices are performed either within loops of operations on the individual elements or by invocation of a separate program module.

The natural way of representing vectors and matrices in Fortran and C is as array variables with indexes. Fortran handles arrays as multiply indexed memory locations, consistent with the nature of the object. Indexes start at 1, just as in the mathematical notation used throughout this book. The storage of two-dimensional arrays in Fortran is column-major, that is, the array $A$ is stored as vec($A$). To reference the contiguous memory locations, the first subscript varies fastest. In general-purpose software consisting of Fortran subprograms, it is often necessary to specify the lengths of all dimensions of a Fortran array except the last one.

An array in C is an ordered set of memory locations referenced by a pointer or by a name and an index. Indexes start at 0. The indexes are enclosed in rectangular brackets following the variable name. An element of a multidimensional array in C is indexed by multiple indexes, each within rectangular brackets. If the $3 \times 4$ matrix $A$ is as stored in the C array $A$, the (2, 3) element, $A_{2,3}$ is referenced as $A[1][2]$.

Multidimensional arrays in C are arrays of arrays, in which the array constructors operate from right to left. This results in two-dimensional C arrays being stored in row-major order, that is, the array $A$ is stored as vec($A^T$). To reference the contiguous memory locations, the last subscript varies fastest. In general-purpose software consisting of C functions, it is often necessary to specify the lengths of all dimensions of a C array except the first one.

**Computational Efficiency**

Two seemingly trivial things can have major effects on computational efficiency. One is movement of data from the computer’s memory into the computational unit. How quickly this movement occurs depends, among other things, on the organization of the data in the computer. Multiple elements of an array can be retrieved from memory more quickly if they are in contiguous memory locations. (Location in computer memory does not necessarily refer to a physical place; in fact, memory is often divided into banks, and adjacent
“locations” are in alternate banks. Memory is organized to optimize access.) The main reason that storage of data in contiguous memory locations affects efficiency involves the different levels of computer memory. A computer often has three levels of randomly accessible memory, ranging from “cache” memory, which is very fast, to “disk” memory, which is relatively slower. When data are used in computations they may be moved in blocks, or pages, from contiguous locations in one level of memory to a higher level. This allows faster subsequent access to other data in the same page. When one block of data is moved into the higher level of memory, another block is moved out. The movement of data (or program segments, which are also data) from one level of memory to another is called “paging”.

In Fortran a column of a matrix occupies contiguous locations, so when paging occurs, elements in the same column are moved. Hence, a column of a matrix can often be operated on more quickly in Fortran than a row of a matrix. In C a row can be operated on more quickly.

Some computers have array processors that provide basic arithmetic operations for vectors. The processing units are called vector registers, and typically hold 128 or 256 full-precision floating-point numbers (see Section 3.1.1). For software to achieve high levels of efficiency, computations must be organized to match the length of the vector processors as often as possible. See Dongarra et al. (1991 and 1998) for descriptions of methods for matrix computations on array processors.

Another thing that affects the performance of software is the execution of loops. In the simple loop

\begin{verbatim}
  do i = 1, n
    sx(i) = sin(x(i))
  end do
\end{verbatim}

it may appear that the only computing is just the evaluation of the sine of the elements in the vector \( x \). In fact, a nonnegligible amount of time may be spent in keeping track of the loop index and in accessing memory. A compiler on a vector computer may organize the computations so that they are done in groups corresponding to the length of the vector registers. On a computer that does not have vector processors, a technique called “unrolling do-loops” is sometimes used. For the code segment above, unrolling the do-loop to a depth of 7, for example, would yield the following code:

\begin{verbatim}
  do i = 1, n, 7
    sx(i) = sin(x(i))
    sx(i+1) = sin(x(i+1))
    sx(i+2) = sin(x(i+2))
    sx(i+3) = sin(x(i+3))
    sx(i+4) = sin(x(i+4))
    sx(i+5) = sin(x(i+5))
    sx(i+6) = sin(x(i+6))
  end do
\end{verbatim}
plus a short loop for any additional elements in \( x \) beyond \( 7\lfloor n/7 \rfloor \). Obviously, this kind of programming effort is warranted only when \( n \) is large and when the code segment is expected to be executed many times. For widely distributed programs, such as the BLAS discussed in the next section, the extra programming is worthwhile.

**BLAS**

There are several basic computations for vectors and matrices that are very common across a wide range of scientific applications. Computing the dot product of two vectors, for example, is a task that may occur in such diverse areas as fitting a linear model to data or determining the maximum value of a function. The sets of routines called “basic linear algebra subprograms” (BLAS) implement many of the standard operations for vectors and matrices. The BLAS represent a very significant step toward software standardization, because the definitions of the tasks and the user interface are the same on all computing platforms. The actual coding, however, may be quite different, to take advantage of special features of the hardware or underlying software, such as compilers.

The level 1 BLAS or BLAS-1, the original set of the BLAS, are for vector operations. They were defined by Lawson et al. (1979). Matrix operations, such as multiplying two matrices were built using the BLAS-1. Later, a set of the BLAS, called level 2 or the BLAS-2, for operations involving a matrix and a vector, was defined by Dongarra et al. (1988), a set called the level 3 BLAS or the BLAS-3, for operations involving two dense matrices, was defined by Dongarra et al. (1990), and a set of the level 3 BLAS for sparse matrices was proposed by Duff et al. (1997). An updated set of BLAS is described by Blackford et al. (2002).

The operations performed by the BLAS often cause an input variable to be updated. For example, in a Givens rotation, two input vectors are rotated into two new vectors. In this case, it is natural and efficient just to replace the input values with the output values (see below). A natural implementation of such an operation is to use an argument that is both input and output. In some programming paradigms, such a “side effect” can be somewhat confusing, but the value of this implementation outweighs the undesirable properties.

There is a consistency of the interface among the BLAS routines. The nature of the arguments and their order in the reference are similar from one routine to the next. The general order of the arguments is:

1. the size or shape of the vector or matrix,
2. the array itself, which may be either input or output,
3. the stride, and
4. other input arguments.

The first and second types of arguments are repeated as necessary for each of the operand arrays and the resultant array.
A BLAS routine is identified by a root character string that indicates the operation, for example, \texttt{dot} or \texttt{axpy}. The name of the BLAS program module may depend on the programming language. In Fortran, the root may be prefixed by \texttt{s} to indicate single precision, by \texttt{d} to indicate double precision, or by \texttt{c} to indicate complex, for example. If the language allows generic function and subroutine references, just the root of the name is used.

The \texttt{axpy} operation we referred to on page 229 multiplies one vector by a constant and then adds another vector \((ax + y)\). The BLAS routine \texttt{axpy} performs this operation. The interface is

\[
\text{axpy}(n, a, x, \text{incx}, y, \text{incy})
\]

where

\begin{itemize}
\item \texttt{n} – the number of elements in each vector
\item \texttt{a} – the scalar constant
\item \texttt{x} – the input/output one-dimensional array that contains the elements of the vector \texttt{x}
\item \texttt{incx} – the stride in the array \texttt{x} that defines the vector
\item \texttt{y} – the input/output one-dimensional array that contains the elements of the vector \texttt{y}
\item \texttt{incy} – the stride in the array \texttt{y} that defines the vector
\end{itemize}

Another example, the routine \texttt{rot} to apply a Givens rotation (similar to the routine \texttt{rotm} that we referred to in Section 5.2.2), has the interface:

\[
\text{rot}(n, x, \text{incx}, y, \text{incy}, c, s)
\]

where

\begin{itemize}
\item \texttt{n} – the number of elements in each vector
\item \texttt{x} – the input/output one-dimensional array that contains the elements of the vector \texttt{x}
\item \texttt{incx} – the stride in the array \texttt{x} that defines the vector
\item \texttt{y} – the input/output one-dimensional array that contains the elements of the vector \texttt{y}
\item \texttt{incy} – the stride in the array \texttt{y} that defines the vector
\item \texttt{c} – the cosine of the rotation
\item \texttt{s} – the sine of the rotation
\end{itemize}

This routine is invoked after \texttt{rotg} has been called to determine the cosine and the sine of the rotation. (See Exercise ??, page ??.)

Source programs and additional information about the BLAS can be obtained at
ATLAS is a software package called ATLAS will automatically generate BLAS and other software for linear algebra for various processors. Information about the ATLAS can be obtained at

http://www.netlib.org/atlas/

Fortran and C Libraries

When work was being done on the BLAS-1 in the 1970s, those lower-level routines were being incorporated into a higher-level set of Fortran routines for matrix eigensystem analysis, called EISPACK (Smith et al., 1976), and into a higher-level set of Fortran routines for solutions of linear systems, called LINPACK (Dongarra et al., 1979). As work progressed on the BLAS-2 and BLAS-3 in the 1980s and later, a unified set of Fortran routines for both eigenvalue problems and solutions of linear systems was developed, called LAPACK (Anderson et al., 1999). A Fortran 95 version of LAPACK is described by Barker et al. (2001). Information about LAPACK is available at

http://www.netlib.org/lapack/

There is a graphical user interface to help the user in navigating the LAPACK site and downloading LAPACK routines.

Another standard set of routines, called the BLACS (Basic Linear Algebra Communication Subroutines), provides a portable message-passing interface primarily for linear algebra computations with a user interface similar to that of the BLAS. A slightly higher-level set of routines, the PBLAS, combine both the data communication and computation into one routine, also with a user interface similar to that of the BLAS. Filippone and Colajanni (2000) provide a set of parallel BLAS for sparse matrices. Their system, called PSBLAS, shares the general design of the PBLAS for dense matrices and the design of the level 3 BLAS for sparse matrices proposed by Duff et al. (1997).

A distributed memory version of LAPACK, called ScaLAPACK, (see Blackford et al., 1997) has been built on the BLACS and the PBLAS modules.

Standards for message passing in a distributed-memory parallel processing environment are evolving. The MPI (message passing interface) standard is being developed, primarily at Argonne National Labs. This allows for standardized message passing across languages and systems. See Gropp, Lusk, and Skjellum (1999) and Gropp, Lusk, and Thakur (1999). IBM has built the Message Passing Library (MPL) in both Fortran and C that provides message passing kernels. PLAPACK is a package for linear algebra built on MPI. (See Van de Geijn, 1997.)
All of these packages are available on a range of platforms, especially on high-performance computers.

Two of the most widely used Fortran and C libraries are the IMSL Libraries and the Nag Library. They provide a large number of routines for numerical linear algebra, ranging from very basic computations as provided in the BLAS through complete routines for solving various types of systems of equations and for performing eigenanalysis. Both libraries are available in both Fortran and C versions, and in both single and double precision.

Matrix Storage Modes

Matrices that have multiple elements with the same value can often be stored in the computer in such a way that the individual elements do not all have separate locations. Symmetric matrices and matrices with many zeros, such as the upper or lower triangular matrices of the various factorizations we have discussed, are examples of matrices that do not require full rectangular arrays for their storage.

A special storage mode for symmetric matrices uses a linear array to store only the unique elements. The symmetric matrix $A$ is stored as $\text{vech}(A)$. For example, the symmetric matrix

$$
\begin{bmatrix}
1 & 2 & 4 & \cdots \\
2 & 3 & 5 & \cdots \\
4 & 5 & 6 & \cdots \\
\vdots & \vdots & \vdots & \ddots
\end{bmatrix}
$$

is represented by the array

$$(1, 2, 3, 4, 5, 6, \cdots).$$

For an $n \times n$ symmetric matrix $A$, the correspondence with the $n(n+1)/2$-vector $v$ is $v_{i(i-1)/2+j} = a_{i,j}$, for $i \geq j$. Notice that the relationship does not involve $n$. For $i \geq j$, in Fortran, it is

$$v(i*(i-1)/2+j) = a(i,j)$$

and in C it is

$$v[i*(i+1)/2+j] = a[i][j]$$

Although the amount of space saved by not storing the full symmetric matrix is only about one half of the amount of space required, the use of rank 1 arrays rather than rank 2 arrays can yield some reference efficiencies. (Recall that in discussions of computer software objects, “rank” usually means the number of dimensions.) For band matrices and for other sparse matrices, the savings in storage can be much larger.

The BLAS and the IMSL Libraries implement a wide range of matrix storage modes:
Symmetric mode. A full matrix is used for storage, but only the upper or lower triangular portion of the matrix is used. Some library routines allow the user to specify which portion is to be used, and others require that it be the upper portion.

Hermitian mode. This is the same as the symmetric mode, except for the obvious changes for the Hermitian transpose.

Triangular mode. This is the same as the symmetric mode (with the obvious changes in the meanings).

Band mode. For the \( n \times m \) band matrix \( A \) with lower band width \( w_l \) and upper band width \( w_u \), an \( w_l + w_u \times m \) array is used to store the elements. The elements are stored in the same column of the array, say \( \text{aa} \), as they are in the matrix; that is,

\[
\text{aa}(i - j + w_u + 1, j) = a_{i,j},
\]

for \( i = 1, 2, \ldots, w_l + w_u + 1 \).

Band symmetric, band Hermitian, and band triangular modes are all defined similarly. In each case, only the upper or lower bands are referenced.

Sparse storage mode. There are several different schemes for representing sparse matrices. The IMSL Libraries use three arrays, each of rank 1 and with length equal to the number of nonzero elements. The integer array \( i \) contains the row indicator, the integer array \( j \) contains the column indicator, and the floating-point array \( a \) contains the corresponding values; that is, the \((i(k), j(k))\) element of the matrix is stored in \( a(k) \).

The level 3 BLAS for sparse matrices proposed by Duff et al. (1997) have an argument to allow the user to specify the type of storage mode.

**Fortran 95 and Fortran 2000**

One of the most useful of the features of Fortran 95 and later versions of Fortran for the scientific programmer is the provision of primitive constructs for vectors and matrices. Whereas all of the Fortran 77 intrinsics are scalar-valued functions, Fortran 95 provides array-valued functions. For example, if \( \text{aa} \) and \( \text{bb} \) represent matrices conformable for multiplication, the statement

\[
\text{cc} = \text{matmul}(\text{aa}, \text{bb})
\]

yields the Cayley product in \( \text{cc} \). The \text{matmul} function also allows multiplication of vectors and matrices.

Indexing of arrays starts at 1 by default (any starting value can be specified, however), and storage is column-major.

Space must be allocated for arrays in Fortran 95, but this can be done at run time. An array can be initialized either in the statement allocating the space or in a regular assignment statement. A vector can be initialized by listing the elements between "(/" and "/)". This list can be generated in various ways. The \text{reshape} function can be used to initialize matrices.
5.4 Software for Numerical Linear Algebra

For example, a Fortran 95 statement to declare that the variable \( aa \) is to be used as a \( 3 \times 4 \) array, and to allocate the necessary space is

\[
\text{real, dimension(3,4)} :: aa
\]

A Fortran 95 statement to initialize \( aa \) with the matrix

\[
\begin{bmatrix}
1 & 4 & 7 & 10 \\
2 & 5 & 8 & 11 \\
3 & 6 & 9 & 12
\end{bmatrix}
\]

is

\[
aa = \text{reshape}( (/ 1., 2., 3., \\
4., 5., 6., \\
7., 8., 9., \\
10.,11.,12./), & \\
/3,4/) )
\]

Fortran 95 has an intuitive syntax for referencing subarrays, shown in Table 5.1.

<table>
<thead>
<tr>
<th>Subarray expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( aa(2:3,1:3) )</td>
<td>the ( 2 \times 3 ) submatrix in rows 2 and 3 and columns 1 to 3 of ( aa )</td>
</tr>
<tr>
<td>( aa(:,1:4:2) )</td>
<td>refers to the submatrix with all 3 rows and the 1\textsuperscript{st} and 3\textsuperscript{rd} columns of ( aa )</td>
</tr>
<tr>
<td>( aa(:,4) )</td>
<td>refers to the column vector that is the 4\textsuperscript{th} column of ( aa )</td>
</tr>
</tbody>
</table>

Notice that because the indexing starts with 1 (instead of with 0) the correspondence between the computer objects and the mathematical objects is a natural one. The subarrays can be used directly in functions. For example, if \( bb \) is the matrix

\[
\begin{bmatrix}
1 & 5 \\
2 & 6 \\
3 & 7 \\
4 & 8
\end{bmatrix}
\]

the Fortran 95 function reference

\[
\text{matmul}(aa(1:2,2:3), bb(3:4,:))
\]

yields the Cayley product

\[
\begin{bmatrix}
4 & 7 \\
5 & 8
\end{bmatrix}
\begin{bmatrix}
3 & 7 \\
4 & 8
\end{bmatrix}.
\]

(5.108)
Libraries built on Fortran 95 allow some of the basic operations of linear algebra to be implemented as operators whose operands are vectors or matrices.

Fortran 95 also contains some of the constructs, such as `forall`, that have evolved to support parallel processing.

A more extensive revision called Fortran 2000 includes such features as exception handling, interoperability with C, allocatable components, parameterized derived types, and object-oriented programming.

**Matrix and Vector Classes in C++**

In an object-oriented language such as C++ it is useful to define classes corresponding to matrices and vectors. Operators and/or functions corresponding to the usual operations in linear algebra can be defined so as to allow use of simple expressions to perform these operations.

A class library in C++ can be defined in such a way that the computer code corresponds more closely to mathematical code. The indexes to the arrays can be defined to start at 1, and the double index of a matrix can be written within a single pair of parentheses. For example, in a C++ class defined for use in scientific computations, the $(10, 10)$ element of the matrix $A$, that is, $a_{10,10}$, could be referenced as

$$ \texttt{aa(10,10)} $$

instead of as

$$ \texttt{aa[9][9]} $$

as it would be in ordinary C. Many computer engineers prefer the latter notation, however.

There are various C++ class libraries or templates for matrix and vector computations, for example, those of *Numerical Recipes* (Press, et al., 2000). The Template Numerical Toolkit,

http://math.nist.gov/tnt/

and the Matrix Template Library,

http://www.osl.iu.edu/research/mtl/

are templates based on the design approach of the C++ Standard Template Library,

http://www.sgi.com/tech/stl/

The class library in *Numerical Recipes* comes with wrapper classes for use with Template Numerical Toolkit or the Matrix Template Library.

Use of a C++ class library for linear algebra computations may carry a computational overhead that is unacceptable for large arrays. Both the Template Numerical Toolkit and the Matrix Template Library are designed to be very efficient (see Siek and Lumsdaine, 2000).
5.4.2 Interactive Systems for Array Manipulation

Many of the computations for linear algebra are implemented as simple operators on arrays in some interactive systems. Some of the more common interactive systems that provide for direct array manipulation are Matlab, R, S-Plus, SAS IML, APL, Lisp-Stat, Gauss, IDL, and PV-Wave. There is no need to allocate space for the arrays in these systems, as there is for arrays in Fortran and C.

In the next two sections we briefly describe the facilities for linear algebra in Matlab, R, and S-Plus. The purpose is to give a very quick comparative introduction.

Matlab

Matlab is an interactive, interpretive, expression language that works with only one type of object: a rectangular array of numbers (possibly complex). The array is a matrix. Scalars (even indices) are $1 \times 1$ matrices; equivalently, a $1 \times 1$ matrix can be treated as a scalar.

The indexing of arrays starts with 1.

If an assignment statement in Matlab is not terminated with a semicolon, the matrix on the left-hand side of the assignment is printed. If a statement consists only of the name of a matrix, the object is printed.

A comment statement in Matlab begins with a percent sign, “%”.

In Matlab a matrix is initialized by listing the elements row-wise within brackets and with semicolons marking the end of rows. (Matlab also has a reshape function similar to that of Fortran 95 that treats the matrix in a column-major fashion.)

In general, the operators in Matlab refer to the common vector/matrix operations. For example, Cayley multiplication is indicated by the usual multiplication symbol, “*”. The meaning of an operator can often be changed to become the corresponding element-by-element operation by preceding the operator with a period; for example, the symbol “.*” indicates the Hadamard product of two matrices. The expression

$$aa \ast bb$$

indicates the Cayley product of the matrices, where the number of columns of $aa$ must be the same as the number of rows of $bb$; and the expression

$$aa \ast \ast bb$$

indicates the Hadamard product of the matrices, where the numbers of rows and columns of $aa$ must be the same as the numbers of rows and columns of $bb$. The transpose of a vector or matrix is obtained by use of a postfix operator \texttt{''}, which is the same ASCII character as the apostrophe:

$$aa$$
As many other software systems for array manipulation, Matlab does not
distinguish between scalars and arrays of size 1. For example, if

\[
\begin{align*}
xx &= [1, 2]; \\
yy &= [1; 2]; \\
zz &= [1; 2; 3];
\end{align*}
\]

the expression \(xx \ast yy \ast zz\) yields the same value as \(5 \ast zz\) because the ex-
pression \(xx \ast yy \ast zz\) is interpreted as \((xx \ast yy) \ast zz\) and \((xx \ast yy)\) is a scalar. The expression \(xx \ast (yy \ast zz)\) is invalid because \(yy\) and \(zz\) are not
conformable for multiplication; thus

\[
(xx \ast yy) \ast zz \neq xx \ast (yy \ast zz).
\]

Figure 5.5 below shows Matlab code that initializes the same matrix \(aa\)
that we used as an example for Fortran 95 above. The code in Figure 5.5 also
initializes a vector \(xx\) and a \(4 \times 2\) matrix \(bb\), and then forms and prints some
products.

```matlab
% Matlab program fragment
xx = [1 2 3 4];  
% Storage is by rows; continuation is indicated by '...
aa = [1 4 7 10; ... 2 5 8 11; ... 3 6 9 12];  
bb = [1 5; 2 6; 3 7; 4 8];  
% Printing occurs automatically unless ';' is used to suppress it.
yy = a*xx' 
yy = xx(1:3)*aa  
cc = aa*bb
```

**Figure 5.5.** Matlab Code to Define and Initialize Two Matrices and a Vector, and
Then Form and Print Their Product

Matlab distinguishes between row vectors and column vectors. A row vec-
tor is a matrix whose first dimension is 1, and a column vector is a matrix
whose second dimension is 1. In either case, an element of the vector is refer-
cenced by a single index.

Subarrays in Matlab are defined in much the same way as in Fortran 95,
except for one major difference: the upper limit and the stride are reversed
in the triplet used in identifying the row or column indices. Examples of
subarray references in Matlab are shown in Table 5.2. Compare these with
the Fortran 95 references shown in Table 5.1.

The subarrays can be used directly in expressions. For example, the ex-
pression

\[
aa(1:2,2:3) \ast bb(3:4,:)\]

yields the product
\[
\begin{bmatrix}
4 & 7 \\
5 & 8
\end{bmatrix}
\begin{bmatrix}
3 & 7 \\
4 & 8
\end{bmatrix}
\]
as on page 329.

Matlab has functions for many of the basic operations we have discussed, some of which are shown in Table 5.3.

In addition to these functions, Matlab has special operators “\" and “/” for solving linear systems or for multiplying one matrix by the inverse of another. While the statement
\[aa\bb\]
refers to a quantity that has the same value as the quantity indicated by
\[\text{inv}(aa)\times bb\]
the computations performed are different (and, hence, the values produced may be different). The second expression is evaluated by performing the two operations indicated: \[aa\] is inverted, and the inverse used as the left factor in matrix or matrix/vector multiplication. The first expression, \[aa\bb\], indicates that the appropriate computations to evaluate \[x\] in \[Ax = b\] should be performed to evaluate the expression. (Here, \(x\) and \(b\) may be matrices or vectors.) Another difference between the two expressions is that \[\text{inv}(aa)\] requires \(aa\) to be square algorithmically nonsingular, whereas \[aa\bb\] produces a value that simulates \(A^{-1}b\).

Matlab is a proprietary package distributed by The Mathworks, Inc. There is a freely available package, called Octave, that provides essentially the same core functionality in the same language. There are a number of books on Matlab, including, for example, Hanselman and Littlefield (1996) and Etter (1996). The book by Coleman and Van Loan (1988) is not specifically on Matlab, but shows how to perform matrix computations in Matlab.

\section*{R and S-Plus}

The software system called S was developed at Bell Laboratories in the mid-1970s. Work on S has continued at Bell Labs and the system has evolved
### Table 5.3. Some Matlab Functions for Vector/Matrix Computations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>norm</td>
<td>Matrix or vector norm. For vectors all $L_p$ norms are available. For matrices the $L_1$, $L_2$, $L_\infty$, and Frobenius norms are available.</td>
</tr>
<tr>
<td>rank</td>
<td>Number of linearly independent rows or columns.</td>
</tr>
<tr>
<td>det</td>
<td>Determinant.</td>
</tr>
<tr>
<td>trace</td>
<td>Trace.</td>
</tr>
<tr>
<td>cond</td>
<td>Matrix condition number.</td>
</tr>
<tr>
<td>null</td>
<td>Null space.</td>
</tr>
<tr>
<td>orth</td>
<td>Orthogonalization.</td>
</tr>
<tr>
<td>inv</td>
<td>Matrix inverse.</td>
</tr>
<tr>
<td>pinv</td>
<td>Pseudoinverse.</td>
</tr>
<tr>
<td>lu</td>
<td>LU decomposition.</td>
</tr>
<tr>
<td>qr</td>
<td>QR decomposition.</td>
</tr>
<tr>
<td>chol</td>
<td>Cholesky factorization.</td>
</tr>
<tr>
<td>svd</td>
<td>Singular value decomposition.</td>
</tr>
<tr>
<td>linsolve</td>
<td>Solve system of linear equations.</td>
</tr>
<tr>
<td>lscov</td>
<td>Weighted least squares. The operator “&quot; can be used for ordinary least squares.</td>
</tr>
<tr>
<td>nnls</td>
<td>Nonnegative least-squares.</td>
</tr>
<tr>
<td>eig</td>
<td>Eigenvalues and eigenvectors.</td>
</tr>
<tr>
<td>poly</td>
<td>Characteristic polynomial.</td>
</tr>
<tr>
<td>hess</td>
<td>Hessenberg form.</td>
</tr>
<tr>
<td>schur</td>
<td>Schur decomposition.</td>
</tr>
<tr>
<td>balance</td>
<td>Diagonal scaling to improve eigenvalue accuracy.</td>
</tr>
<tr>
<td>expm</td>
<td>Matrix exponential.</td>
</tr>
<tr>
<td>logm</td>
<td>Matrix logarithm.</td>
</tr>
<tr>
<td>sqrtm</td>
<td>Matrix square root. (More general than \texttt{chol}.)</td>
</tr>
<tr>
<td>funm</td>
<td>Evaluate general matrix function.</td>
</tr>
</tbody>
</table>

considerably since the early versions (see Becker, Chambers, and Wilk, 1988, and Chambers, 1997).

S-Plus is an enhancement of S, developed by StatSci, Inc. (now a part of Insightful Corporation) The enhancements include graphical interfaces, more statistical analysis functionality, and support. S and S-Plus are both data analysis systems and object-oriented programming languages.

There is a freely available package, called R, that provides generally the same functionality in the same language as S-Plus (see Gentleman and Ihaka, 1997).

In the following, rather than continuing to refer to both S and S-Plus, as well as R, I will generally refer only to R, but most of the discussion applies to both systems. There are some functions that are available in S-Plus and not in R, and some available in R and not in S-Plus.

The most important R entity is the function. In R all actions are “functions”, and R has an extensive set of functions, that is, verbs. Assignment is
made by “<-” or by “_”. (The symbol “_” should never be used for assignment, in my opinion. It is not mnemonic, and it is often used as a connective. I have seen students use a variable L_p, with “_” being used as a connective, and then use a statement like norm_L_p, in which the first “_” is an assignment. Use of this symbol instead of <- saves exactly one unshifted keystroke!)

A comment statement in R begins with a pound sign, “#”.

R has a natural syntax and powerful functions for dealing with vectors and matrices, which are objects in the base language. R has functions for printing, but if a statement consists of just the name of an object, the object is printed.

Indexing of arrays starts at 1, and storage is column-major. Indexes are indicated by “[ ]”; for example, xx[1] refers to the first element of the one-dimensional array xx.

A list is constructed by the c function. A list can be treated as a vector without modification. A matrix is constructed from a list by the matrix function. A matrix can also be constructed by binding vectors as the columns of the matrix (the cbind function) or by binding vectors as the rows of the matrix (the rbind function).

Cayley multiplication is indicated by the symbol, “%*%”. Most operators with array operands are applied elementwise; for example, the symbol “*” indicates the Hadamard product of two matrices. The expression

\[
\text{aa} \%*\% \text{bb}
\]

indicates the Cayley product of the matrices, where the number of columns of aa must be the same as the number of rows of bb; and the expression

\[
\text{aa} \times \text{bb}
\]

indicates the Hadamard product of the matrices, where the numbers of rows and columns of aa must be the same as the numbers of rows and columns of bb. The transpose of a vector or matrix is obtained by use of the function “t”:

\[
\text{t(aa)}
\]

An important comment about S-Plus and R usage is in order here. The two functions c and t essentially preempt these two letters as names of variables. Although it is possible to redefine functions, it is probably better just to remember never to name a variable “c” or “t”.

Figure 5.6 below shows R or S-Plus code that does the same thing as the Matlab code in Figure 5.5, that is, initialize two matrices and a vector, and then form and print their products.

To the extent that R or S-Plus distinguishes between row vectors and column vectors, a vector is considered to be a column vector. In many cases, however, it does not distinguish. For example, if

\[
\text{xx} \gets \text{c}(1,2) \\
\text{yy} \gets \text{c}(1,2)
\]
# R / S-Plus program fragment
xx <- c(1 2 3 4)
# Storage is by column, but a matrix can be constructed by rows;
# the form of a statement indicates when it is complete, so
# statements continue automatically.
aa <- matrix(c( 1, 4, 7, 10,
               2, 5, 8, 11,
               3, 6, 9, 12),
              nrow=3, byrow=T)
bb <- matrix(seq(1,8), nrow=4)
yy <- aa %*% xx
# Printing is performed by entering the name of the object
yy
yy <- xx[c(1,2,3)] %*% aa
yy
c <- aa %*% bb
c

Figure 5.6. R or S-Plus Code to Define and Initialize Two Matrices and a Vector, and Then Form and Print Their Product

the expression xx %*% yy is the dot product; that is xx %*% yy is the same as t(xx) %*% yy. The outer product is xx %*% t(yy).

In the expressions

yy <- aa %*% xx

and

yy <- xx[c(1,2,3)] %*% aa

in Figure 5.6, the vector is interpreted as a row or column as appropriate for the multiplication to be defined. Compare the similar expressions in the Matlab code in Figure 5.5.

As many other software systems for array manipulation, R and S-Plus do not distinguish between scalars and arrays of size 1. For example, if

xx <- c(1,2)
yy <- c(1,2)
zz <- c(1,2,3)

the expression xx %*% yy %*% zz yields the same value as 5*zz because the expression xx %*% yy %*% zz is interpreted as (xx %*% yy) %*% zz and (xx %*% yy) is a scalar. The expression xx %*% (yy %*% zz) is invalid because yy and zz are not conformable for multiplication; thus

(xx%*%yy)*%zz ≠ xx%*%(yy%*%zz).

Subarrays in S-Plus or R are defined in a similar way as in Fortran 95. A missing index indicates that the entire corresponding dimension is to be used.
Groups of indices can be formed by the \texttt{c} function or the \texttt{seq} function, which is similar to the \texttt{i:j:k} notation of Fortran 95.

**Table 5.4.** Subarrays in R or S-Plus

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{aa[c(2,3),c(1,3)]}</td>
<td>the $2 \times 3$ submatrix in rows 2 and 3 and columns 1 to 3 of \texttt{aa}</td>
</tr>
<tr>
<td>\texttt{aa[,seq(1,4,2)]}</td>
<td>the submatrix with all 3 rows and the 1\textsuperscript{st} and 3\textsuperscript{rd} columns of \texttt{aa}</td>
</tr>
<tr>
<td>\texttt{aa[,4]}</td>
<td>the column vector that is the 4\textsuperscript{th} column of \texttt{aa}</td>
</tr>
</tbody>
</table>

The subarrays can be used directly in expressions. For example, the expression

\[
\texttt{aa[c(1,2),c(2,3)] %*% bb[c(3,4),]}
\]

yields the product

\[
\begin{bmatrix}
4 & 7 \\
5 & 8 \\
\end{bmatrix}
\begin{bmatrix}
3 & 7 \\
4 & 8 \\
\end{bmatrix}
\]

as on page 329.

R has functions for many of the basic operations on vectors and matrices. Some of the R functions are shown in Table 5.5.

**Table 5.5.** Some R Functions for Vector/Matrix Computations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{norm}</td>
<td>Matrix norm.</td>
</tr>
<tr>
<td>\texttt{vecnorm}</td>
<td>Vector $L_p$ norm.</td>
</tr>
<tr>
<td>\texttt{det}</td>
<td>Determinant.</td>
</tr>
<tr>
<td>\texttt{rcond}</td>
<td>Matrix condition number.</td>
</tr>
<tr>
<td>\texttt{solve}</td>
<td>Matrix inverse or pseudoinverse.</td>
</tr>
<tr>
<td>\texttt{lu}</td>
<td>$LU$ decomposition.</td>
</tr>
<tr>
<td>\texttt{qr}</td>
<td>$QR$ decomposition.</td>
</tr>
<tr>
<td>\texttt{chol}</td>
<td>Cholesky factorization.</td>
</tr>
<tr>
<td>\texttt{svd}</td>
<td>Singular value decomposition.</td>
</tr>
<tr>
<td>\texttt{solve}</td>
<td>Solve system of linear equations.</td>
</tr>
<tr>
<td>\texttt{lsfit}</td>
<td>Ordinary or weighted least squares.</td>
</tr>
<tr>
<td>\texttt{nnls.fit}</td>
<td>Nonnegative least-squares.</td>
</tr>
<tr>
<td>\texttt{eigen}</td>
<td>Eigenvalues and eigenvectors.</td>
</tr>
</tbody>
</table>

S-Plus is a proprietary system, but it have a very broad community of users and supporters. There are independently developed guides to the software, such as Everitt (1994), Spector (1994), and Krause and Olson (1997).
There are also texts that address the statistical methods in the context of S-Plus system, such as Chambers and Hastie (1992) (John Chambers was the principal designer of S), Härdle (1991), Venables and Ripley (1999), and Bruce and Gao (1996), who describe wavelet analysis in S-Plus.

5.4.3 Software for High-Performance Computers and Other Specialized Software

Because computations for linear algebra are so pervasive in scientific applications, it is important to have very efficient software for carrying out these computations. Surveys of specialized software for vector architectures and parallel processors are available in Dongarra et al. (1991 and 1998); Dongarra and Walker (1995); Gallivan et al. (1990); and Gallivan, Plemons, and Sameh (1990).

ScaLAPACK, described by Blackford et al. (1997), is a distributed memory version of LAPACK that uses the BLACS and the PBLAS modules. The computations in ScaLAPACK are organized as if performed in a “distributied linear algebra machine” (DLAM), which is constructed by interconnecting BLAS with a BLACS network. The BLAS perform the usual basic computations and the BLACS network exchanges data using primitive message-passing operations. The DLAM can be constructed either with or without a host process. If a host process is present, it would act like a server in receiving a user request, creating the BLACS network, distributing the data, starting the BLAS processes, and collecting the results. ScaLAPACK has a routines for LU, Cholesky, and QR decompositions and for computing eigenvalues of a symmetric matrix. The routines are similar to the corresponding routines in LAPACK; even the names are similar, for example, in Fortran:

<table>
<thead>
<tr>
<th>LAPACK</th>
<th>ScaLAPACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>dgetrf</td>
<td>pdgetrf</td>
</tr>
<tr>
<td>dpotrf</td>
<td>pdpotrf</td>
</tr>
<tr>
<td>dgeqrf</td>
<td>pdgeqrf</td>
</tr>
<tr>
<td>dsyevx</td>
<td>pdsyevx</td>
</tr>
</tbody>
</table>

Quinn (1994, Chapters 7 and 9) describes various algorithms for basic vector/matrix operations, including solution of linear systems, that perform parallel processing.

The constructs of Fortran 95 are helpful in thinking of operations in such a way that they are naturally parallelized. While addition of arrays in Fortran 77 or C is an operation that leads to loops of sequential scalar operations, in Fortran 95 it is thought of as a single higher-level operation. How to perform operations in parallel efficiently is still not a natural activity, however. For example, the two Fortran 95 statements to add the arrays aa and bb and then to add aa and cc:

```
dd = aa + bb
ee = aa + cc
```
may be less efficient than loops because the array \( aa \) may be accessed twice.


Freund, Golub, and Nachtigal (1992) describe algorithms for iterative matrix computations and identify some available software that implements them.

The software package PVM, or Parallel Virtual Machine, which was developed at Oak Ridge National Lab, University of Tennessee, and Emory University, provides a set of C functions or Fortran subroutines that allow a heterogeneous collection of Unix computers to operate smoothly as a multicomputer. See Geist et al. (1994). Likewise, the libraries built on the MPI (Message Passing Interface) standard provide functions that effectively build a multicomputer from a heterogeneous collection of Unix computers.

5.4.4 Test Data

As we discussed in Chapter 3, testbeds for software consist of test datasets that vary in condition, but that have known solutions or for which there is an easy way of verifying the solution. For testing software for matrix computations, a very common matrix is the Hilbert matrix, which has elements

\[
h_{ij} = \frac{1}{i + j - 1}.
\]

Hilbert matrices have large condition numbers; for example, the \( 10 \times 10 \) Hilbert matrix has condition number of order \( 10^{13} \). The Matlab function `hilb(n)` generates an \( n \times n \) Hilbert matrix.

Ericksen (1985) describes how to generate matrices with known inverses in such a way that the condition numbers vary widely. To generate an \( n \times n \) matrix \( A \), choose \( x_1, x_2, \ldots, x_n \) arbitrarily, except such that \( x_1 \neq 0 \), and take

\[
a_{1j} = x_1 \quad \text{for } j = 1, \ldots, n, \\
a_{i1} = x_i \quad \text{for } i = 2, \ldots, n, \\
a_{ij} = a_{i,j-1} + a_{i-1,j-1} \quad \text{for } i, j = 2, \ldots, n.
\]

To represent the elements of the inverse, first define \( y_1 = x_1^{-1} \), and for \( i = 2, \ldots, n \),

\[
y_i = -y_1 \sum_{k=0}^{i-1} x_{i-k} y_k.
\]

Then the elements of the inverse of \( A \), \( B = (b_{ij}) \), are given by

\[
b_{in} = (-1)^{i+k} \binom{n-1}{i-1} y_1 \quad \text{for } i = 1, \ldots, n,
\]

\[
b_{nj} = y_{n+1-j} \quad \text{for } j = 1, \ldots, n - 1,
\]

\[
b_{ij} = x_1 b_{in} b_{nj} + \sum_{k=i+1}^{n} b_{k,j+1} \quad \text{for } i, j = 1, \ldots, n - 1,
\]
where the binomial coefficient, \( \binom{k}{m} \), is defined to be 0 if \( k < m \) or \( m < 0 \).

The nonzero elements of \( L \) and \( U \) in the \( LU \) decomposition of \( A \) are easily seen to be \( l_{ij} = x_{i+1-j} \) and \( u_{ij} = \binom{j-1}{i-1} \). The nonzero elements of the inverses of \( L \) and \( U \) are then seen to have \( (i,j) \) elements \( y_{i+1-j} \) and \( (-1)^{i-j} \binom{j-1}{i-1} \). The determinant of \( A \) is \( x_n \). For some choices of \( x_1, \ldots, x_n \), it is easy to determine the condition numbers, especially with respect to the \( L_1 \) norm, of the matrices \( A \) generated in this way. Ericksen (1985) suggests that the \( x \)'s be chosen as

\[
x_1 = 2^m \quad \text{for } m \leq 0
\]

and

\[
x_i = \binom{k}{i-1}, \quad \text{for } i = 2, \ldots, n \quad \text{and } k \geq 2,
\]

in which case the \( L_1 \) condition number of \( 10 \times 10 \) matrices will range from about \( 10^7 \) to \( 10^{17} \) as \( n \) ranges from 2 to 20 for \( m = 0 \), and will range from about \( 10^{11} \) to \( 10^{23} \) as \( n \) ranges from 2 to 20 for \( m = -1 \).

For testing algorithms for computing eigenvalues, a useful matrix is a Wilkinson matrix, which is a symmetric, tridiagonal matrix with 1's on the off-diagonals. For an \( n \times n \) Wilkinson matrix, the diagonal elements are

\[
\frac{n-1}{2}, \quad \frac{n-3}{2}, \quad \frac{n-5}{2}, \ldots, \quad \frac{n-5}{2}, \quad \frac{n-3}{2}, \quad \frac{n-1}{2}.
\]

If \( n \) is odd, the diagonal includes 0, otherwise all of the diagonal elements are positive. The \( 5 \times 5 \) Wilkinson matrix, for example, is

\[
\begin{bmatrix}
2 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 2 \\
\end{bmatrix}
\]

The two largest eigenvalues of a Wilkinson matrix are very nearly equal. Other pairs are likewise almost equal to each other: the third and fourth largest eigenvalues are also close in size, the fifth and sixth largest, and so on. The largest pair is closest in size, and each smaller pair is less close in size.

The Matlab function \( \text{wilkinson}(n) \) generates an \( n \times n \) Wilkinson matrix. Another test matrix available in Matlab is the Rosser test matrix, which is an \( 8 \times 8 \) matrix with an eigenvalue of multiplicity two and three nearly equal eigenvalues. It is constructed by the Matlab function \( \text{rosser} \).

A well-known, large, and wide-ranging set of test matrices for computational algorithms for various problems in linear algebra was compiled and

Another set of test matrices is available through the “Matrix Market”, designed and developed by R. Boisvert, R. Pozo, and K. Remington of the National Institute of Standards and Technology, with contributions by various other people. The test matrices are accessed over the World Wide Web at

http://math.nist.gov/MatrixMarket

The database can be searched by specifying characteristics of the test matrix, such as size, symmetry and so on. Once a particular matrix is found, its sparsity pattern can be viewed at various levels of detail, and other pertinent data can be reviewed. If the matrix seems to be what the user wants, it can be downloaded.

The initial database for the Matrix Market is the approximately 300 problems from the Harwell-Boeing Sparse Matrix Collection.

Assessing the Accuracy of a Computed Result

In real-life applications, the correct solution is not known. (This is also often the case for randomly generated test datasets.) We would like to have some way of assessing the accuracy using the data themselves.

Sometimes a convenient way of assessing the accuracy of the computations in a given problem is to perform internal consistency tests. An internal consistency test may be an assessment of the agreement of various parts of the output. Relationships among the output are exploited to insure that the individually computed quantities satisfy these relationships. Other internal consistency tests may be performed by comparing the results of the solutions of two problems with a known relationship.

The solution to the linear system \( Ax = b \) has a simple relationship to the solution to the linear system \( Ax = b + ca_j \), where \( a_j \) is the \( j \)th column of \( A \) and \( c \) is a constant. A useful check on accuracy of a computed solution to \( Ax = b \) is to compare it with a computed solution to the modified system. Of course, if the expected relationship does not hold, we do not know which solution is incorrect, but it is probably not a good idea to trust either. Mullet and Murray (1971) describe this kind of consistency test for regression software.

To test the accuracy of the computed regression coefficients for regressing \( y \) on \( x_1, \ldots, x_m \), they suggest comparing them to the computed regression coefficients for regressing \( y + dx_j \) on \( x_1, \ldots, x_m \). If the expected relationships do not obtain, the analyst has strong reason to doubt the accuracy of the computations.

Another simple modification of the problem of solving a linear system with a known exact effect is the permutation of the rows or columns.

Another simple internal consistency test that is applicable to many problems is to use two levels of precision in the computations. In using this test,
one must be careful to make sure that the input data are the same. Rounding of the input data may cause incorrect output to result, but that is not the fault of the computational algorithm.

Internal consistency tests cannot confirm that the results are correct; they can only give an indication that the results are incorrect.

Exercises

5.1. Give an example of two vector spaces whose union is not a vector space.

5.2. Let \{v_i, for i = 1, 2, \ldots, n\} be an orthonormal basis for the n-dimensional vector space \(V\). Let \(x \in V\) have the representation

\[x = \sum c_i v_i.\]

Show that the coefficients \(c_i\) can be computed as

\[c_i = \langle x, v_i \rangle.\]

5.3. Prove the Cauchy-Schwarz inequality for the dot product of matrices, (5.22), page 241.

5.4. Show that for any quadratic form, \(x^T A x\), there is a symmetric matrix \(A_s\), such that \(x^T A_s x = x^T A x\). (The proof is by construction, with \(A_s = \frac{1}{2}(A + A^T)\), first showing \(A_s\) is symmetric, and then that \(x^T A_s x = x^T A x\).)

5.5. By writing \(AA^{-1} = I\), derive the expression for the inverse of a partitioned matrix given in equation (5.31).

5.6. Show that the expression given for the generalized inverse in equation (5.33) on page 248 is correct.

5.7. Prove that the eigenvalues of a symmetric matrix are real. Hint: \(A^T A = A^2\).

5.8. Let \(A\) be a matrix with an eigenvalue \(\lambda\) and corresponding eigenvector \(v\).

Consider the matrix polynomial in \(A\),

\[f(A) = c_p A^p + \cdots + c_1 A + c_0 I.\]

Show that \(f(\lambda)\), that is,

\[c_p \lambda^p + \cdots + c_1 \lambda + c_0,\]

is an eigenvalue of \(f(A)\) with corresponding eigenvector \(v\).

5.9. Give conditions on \(a, b,\) and \(c\) for the matrix below to be positive definite.

\[
\begin{bmatrix}
a & b \\
b & c
\end{bmatrix}
\]
5.10. Prove that the induced norm (page 259) is a matrix norm; that is, prove that it satisfies the consistency property.

5.11. Prove that, for the square matrix \( A \),

\[
\| A \|_2^2 = \rho(A^T A).
\]

*Hint:* Show that \( \| A \|_2^2 = \max x^T A^T A x \) for any normalized vector \( x \).

5.12. The triangle inequality for matrix norms: \( \| A + B \| \leq \| A \| + \| B \| \).

   a) Prove the triangle inequality for the matrix \( L_1 \) norm.
   b) Prove the triangle inequality for the matrix \( L_\infty \) norm.
   c) Prove the triangle inequality for the matrix Frobenius norm. (See the proof of inequality 5.47, on page 258.)

5.13. Prove that the Frobenius norm satisfies the consistency property.

5.14. Prove for any square matrix \( A \) with real elements,

\[
\| A \|_2 \leq \| A \|_F.
\]

*Hint:* Use the Cauchy-Schwarz inequality.

5.15. Prove the inequality (5.51) on page 260:

\[
\| A x \| \leq \| A \| \| x \|.
\]

*Hint:* Obtain the inequality from the definition of the induced matrix norm.

5.16. Let \( Q \) be an \( n \times n \) orthogonal matrix and let \( x \) be an \( n \)-vector.

   a) Prove equation (5.53):

\[
\| Q x \|_2 = \| x \|_2.
\]

   *Hint:* Write \( \| Q x \|_2 \) as \( \sqrt{(Q x)^T Q x} \).

   b) Give examples to show that this does not hold for other norms.

5.17. Let \( A \) be nonsingular, and let \( \kappa(A) = \| A \| \| A^{-1} \| \).

   a) Prove equation (5.66):

\[
\kappa(A) = \max_{x \neq 0} \frac{\| A x \|}{\| x \|} \min_{x \neq 0} \frac{\| A^{-1} x \|}{\| x \|}.
\]

   b) Using the relationship above, explain heuristically why \( \kappa(A) \) is called the “condition number” of \( A \).

5.18. For the square, nonsingular matrix \( A \), show that

\[
\frac{dA^{-1}}{dx} = -A^{-1} \frac{dA}{dx} A^{-1}.
\]

*Hint:* Differentiate \( AA^{-1} = I \).

5.19. Consider the transformation of the 3-vector \( x \), that first rotates the vector 30° about the \( x_1 \) axis, then rotates the vector 45° about the \( x_2 \) axis, then translates the vector by adding the 3-vector \( y \). Find the matrix \( A \) that effects these transformations by a single multiplication. Use the vector \( x^h \) of homogeneous coordinates that corresponds to the vector \( x \). (Thus, \( A \) is \( 4 \times 4 \).)
5.20. Determine the rotation matrix that rotates 3-vectors through an angle of 30° in the plane \( x_1 + x_2 + x_3 = 0 \).

5.21. Consider the four properties of a dot product listed on page 231. For each one, state whether the property holds in computer arithmetic. Give examples to support your answers.

5.22. Assuming the model (3.1) on page 32 for the floating-point number system, give an example of a nonsingular \( 2 \times 2 \) matrix that is algorithmically singular.

5.23. A Monte Carlo study of condition number and size of the matrix.
   For \( n = 5, 10, \ldots, 30 \), generate 100 \( n \times n \) matrices whose elements have independent \( N(0,1) \) distributions. For each, compute the \( L_2 \) condition number and plot the mean condition number versus the size of the matrix. At each point, plot error bars representing the sample “standard error” (the standard deviation of the sample mean at that point). How would you describe the relationship between the condition number and the size?

   a) Let \( A \) be an \( n \times n \) matrix whose elements are generated independently (but not necessarily identically) from real-valued continuous distributions. What is the probability that \( A \) is simple?
   b) Under the same conditions as in Exercise 5.24a, and with \( n \geq 3 \), what is the probability that \( |\lambda_{n-2}| < |\lambda_{n-1}| < |\lambda_n| \), where \( \lambda_{n-2}, \lambda_{n-1}, \) and \( \lambda_n \) are the three eigenvalues with largest absolute values?
   c) Prove that the power method converges linearly if \( |\lambda_{n-2}| < |\lambda_{n-1}| < |\lambda_n| \), \( c_{n-1} \neq 0 \), and \( c_n \neq 0 \). (The \( c \)'s are the coefficients in the expansion of \( x^{(0)} \).) Hint: Substitute the expansion in equation (5.102), page 312, into the expression for the convergence ratio in (5.103).
   d) Suppose \( A \) is simple, and the elements of \( x^{(0)} \) are generated independently (but not necessarily identically) from continuous distributions. What is the probability that the power method will converge linearly?

5.25. Consider the matrix
   \[
   \begin{bmatrix}
   4 & 1 & 2 & 3 \\
   1 & 5 & 3 & 2 \\
   2 & 3 & 6 & 1 \\
   3 & 2 & 1 & 7 
   \end{bmatrix}
   \]
   a) Use the power method to determine the largest eigenvalue of this matrix.
   b) Using Givens transformations, reduce the matrix to upper Hessenberg form.

5.26. In the matrix
   \[
   \begin{bmatrix}
   2 & 1 & 0 & 0 \\
   1 & 5 & 2 & 0 \\
   3 & 2 & 6 & 1 \\
   0 & 0 & 1 & 8 
   \end{bmatrix}
   \]
5.4 Software for Numerical Linear Algebra

5.27. In the matrix
\[
\begin{bmatrix}
2 & 1 & 0 & 0 \\
3 & 5 & 2 & 0 \\
0 & 0 & 6 & 1 \\
0 & 0 & 0 & 8 \\
0 & 0 & 0 & 0
\end{bmatrix}
\]
determine the Givens transformations to chase the 3 in the (2, 1) position out of the matrix.

5.28. In the QR methods for eigenvectors and singular values, why can we not just use additional orthogonal transformations to triangularize the given matrix (instead of just forming a similar Hessenberg matrix, as in Section 5.3.3), or to diagonalize the given matrix (instead of just forming the bidiagonal matrix, as in Section 5.3.4)?

5.29. Determine the eigenvalue $\sigma_1$ (on page 319) used in forming the matrix $T_0$ for initiating the chase in the algorithm for the singular value decomposition. Express it in terms of $a_{m,m}$, $a_{m-1,m-1}$, $a_{m-1,m}$, and $a_{m-1,m-2}$.

5.30. Write a recursive function in Fortran, C, Matlab, R, S-Plus, or PV-Wave to multiply two square matrices using the Strassen algorithm (page 278). Write the function so that it uses an ordinary multiplication method if the size of the matrices is below a threshold that is supplied by the user.

5.31. There are various ways to evaluate the efficiency of a program: counting operations, checking the “wall time”, using a shell level timer, and using a call within the program. In C the timing routine is `ctime`, and in Fortran 95 it is the subroutine `system_clock`. Fortran 77 does not have a built-in timing routine, but the IMSL Fortran Library provides one. For this assignment you are to write six short C programs and six short Fortran programs. The programs in all cases are to initialize an $n \times m$ matrix so that the entries are equal to the column numbers, that is, all elements in the first column are 1’s, all in the second column are 2’s, etc. The six programs arise from three matrices of different sizes: $10000 \times 10000$, $100 \times 1000000$, and $1000000 \times 100$; and from two different ways of nesting the loops: for each size matrix, first nest the row loop within the column loop and then reverse the loops. The number of operations is the same for all programs. For each program, use both a shell level timer (e.g., in Unix, use `time`) and a timer called from within your program. Make a table of the times:

<table>
<thead>
<tr>
<th></th>
<th>$10000 \times 10000$</th>
<th>$100 \times 1000000$</th>
<th>$1000000 \times 100$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran column-in-row</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>row-in-column</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>C column-in-row</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>row-in-column</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
5.32. Obtain the BLAS routines \texttt{rotg} and \texttt{rot} for constructing and applying a Givens rotation. These routines exist in both Fortran and C; they are available in the IMSL Libraries or from CALGO (\textit{Collected Algorithms of the ACM}; see the bibliography).

a) Using these two routines, apply a Givens rotation to the matrix used in Exercise ??,

\[
A = \begin{bmatrix}
3 & 5 & 6 \\
6 & 1 & 2 \\
8 & 6 & 7 \\
2 & 3 & 1 \\
\end{bmatrix},
\]

so that the second column becomes \((5, \tilde{a}_{22}, 6, 0)\).

b) Write a routine in Fortran or C that accepts as input a matrix and its dimensions, and uses the BLAS routines \texttt{rotg} and \texttt{rot} to produce its \(QR\) decomposition. There are several design issues you should address: how the output is returned (for purposes of this exercise, just return two arrays or pointers to the arrays in full storage mode); how to handle non-full rank matrices (for this exercise, assume that the matrix is full rank, so return an error message in this case); how to handle other input errors (what do you do if the user inputs a negative number for a dimension?); etc.

5.33. Using the BLAS routines \texttt{rotg} and \texttt{rot} for constructing and applying a Givens rotation and the program you wrote in Exercise ??, write a Fortran or C routine that accepts a simple symmetric matrix and computes its eigenvalues using the mobile Jacobi scheme. The outer loop of your routine consists of the steps shown on page 316, and the multiple actions of each of those steps can be implemented in a loop in serial mode. The importance of this algorithm, however, is realized when the actions in the individual steps on page 316 are performed in parallel.

5.34. Compute the two largest eigenvalues of the \(21 \times 21\) Wilkinson matrix to 15 digits.

5.35. Use a symbolic manipulation software package such as Maple to determine the inverse of the matrix:

\[
\begin{bmatrix}
a & b & c \\
d & e & f \\
g & h & i \\
\end{bmatrix}.
\]

Determine conditions for which the matrix would be singular. (You can use the \texttt{solve()} function in Maple on certain expressions in the symbolic solution you obtained.)

5.36. Consider the \(3 \times 3\) symmetric Toeplitz matrix with elements \(a, b,\) and \(c,\) that is, the matrix that looks like this:

\[
\begin{bmatrix}
a & b & c \\
b & a & b \\
c & b & a \\
\end{bmatrix}.
\]
5.4 Software for Numerical Linear Algebra

Invert this matrix.
Determine conditions for which the matrix would be singular.

5.37. Develop a class library in C++ for matrix and vector operations. Discuss carefully the issues you consider in designing the class constructors. Design them in such a way that the references

\[ xx(1) \]
\[ YY(1,1) \]

refer to the implied mathematical entities.
Design the operators “+” and “*” so that the references

\[ aa + bb \]
\[ aa * bb \]

will determine whether \( a \) and \( b \) are matrices and/or vectors conformable for the implied mathematical operations, and if so, will produce the object corresponding to the implied mathematical entity represented by the expression.
6

Nonlinear Systems

We begin this chapter with a discussion of some fundamental properties of functions. In the second section we discuss the solution of nonlinear systems of equations.

6.1 Basic Definitions and Properties

The goal in an optimization problem is to find the point at which the minimum (or maximum) of a real, scalar function $f$ occurs and, possibly, to find the value of the function at that point. In some cases the value of the function at its optimum is known, so all that is to be done is to determine the location of the point. Finding the value at a given location is usually trivial compared to other aspects of the problem; however, if the optimal function value is known, that knowledge can sometimes be used to increase the efficiency in finding the location of the optimum.

We use the term “optimum” or “extremum” to refer to the minimum or maximum. We commonly consider the minimization problem only. This is without loss of generality, because to maximize $f(x)$ is equivalent to minimizing its negative, $-f(x)$.

The general optimization problem can be stated as the problem of finding the vector $x_*$ such that

$$\min_x f(x) = f(x_*) \tag{6.1}$$

or

$$\arg\min f(x) = x_* \text{.}$$

The function $f$ is called the objective function. The elements of $x$ are often called decision variables. In a statistical estimation or modeling problem, the decisions variables are the parameters to be estimated.

The domain of $f$ may be countable, or discrete, in one or more dimensions. If the domain is countable, we say the optimization problem is “discrete”. If the domain is continuous in all variables, we say the optimization problem is
“continuous”. Many optimization problems have both discrete variables and continuous variables.

In notation such as $f(x)$, we implicitly assume $x$ is in the domain of $f$, and so the elements of $x$ are discrete or continuous as required by the definition of the function.

### 6.1.1 Shapes of Functions

A set of points, $S$, is convex if for all $x_1$ and $x_2$ in $S$, and for $0 < \alpha < 1$, the point

$$\alpha x_1 + (1 - \alpha)x_2$$

is in $S$. An extreme point is a point $x$ that cannot be represented as

$$x = \alpha x_1 + (1 - \alpha)x_2,$$

for some points $x_1$ and $x_2$ in $S$ and $0 < \alpha < 1$.

For any $d$, $\mathbb{R}^d$ is convex with no extreme points. A convex set in $\mathbb{R}^d$ with exactly $d + 1$ extreme points is called a simplex.

A convex function on a convex set $S$ is a function $f$ such that for $0 < \alpha < 1$, and for all $x_1$ and $x_2$ in $S$,

$$f(\alpha x_1 + (1 - \alpha)x_2) \leq \alpha f(x_1) + (1 - \alpha)f(x_2).$$

Strict convexity of a function is the condition in which the inequality above is strict. Concavity and strict concavity are the conditions in which the inequality is reversed. Functions of these four types are illustrated in Figure 6.2.
If $f$ is convex, $-f$ is concave. A concave function is sometimes said to be “concave down”, and a convex function is said to be “concave up”.

For a convex function $f$ of a scalar variable, if its first derivative exists, the derivative is nondecreasing. If its second derivative $f''$ exists, then

$$f''(x) \geq 0 \quad \text{for all } x.$$ 

Strict convexity implies that the second derivative is positive. Likewise, the second derivative of a concave function is nonpositive, the second derivative is negative if the function is strictly concave.

For a convex function of a vector variable, if the Hessian exists, it is positive semidefinite. Strict convexity implies that the Hessian is positive definite.

Generally, if the domain of a convex function is not specified, the function is assumed to be convex over the reals. Although the definitions of convexity and concavity as stated above apply to functions with continuous domains, analogous concepts can be defined for discrete problems.

A function $f(x)$ that is positive on an interval is said to be log convex if $\log f(x)$ is convex. Similar definitions apply for concavity. The normal probability density function shown in Figure 6.3, for example, is log concave. More generally, $f(x)$ is said to be $T$ convex (or $T$ concave) if $T(f(x))$ is convex (or concave).

A useful fact about convex functions that follows immediately by induction on the definition (6.2) is Jensen’s inequality: if $f$ is convex,

$$f\left(\sum_{i=1}^{n}\alpha_i x_i\right) \leq \sum_{i=1}^{n}\alpha_i f(x_i), \quad (6.3)$$
for all nonnegative $\alpha_i$ such that $\sum_{i=1}^{n} \alpha_i = 1$. In the case of strict convexity, equality holds only if $x_1 = x_2 = \cdots = x_n$.

6.1.2 Minima and Maxima of Functions

Methods for finding the minimum of a function of continuous variables may be quite different from methods for finding the minimum over a countable set of choices. Discrete optimization problems involve inspection of combinations. For example, the problem of determining the best way to travel between a set of cities so as to visit each one at least once involves selection of an optimal permutation of the list of cities. The objective function for this problem is the total distance, or some other measure of the cost in traveling between all the cities in the order specified. Although the set of possibilities is countable, optimization problems of this type often require extensive computations. Sometimes the objective function is very complicated and expensive or time-consuming to evaluate; the evaluation of the objective function may even involve making physical observations.

The objective function may have an optimum at only one point or may have optima at many points. If

$$f(x_*) \leq f(x) \quad \text{for all } x,$$

$f(x_*)$ is called a global minimum and $x_*$ is called a global minimizer. If $f(x_*) < f(x)$, at all points, $f(x_*)$ and $x_*$ are called a strict global minimum and a strict global minimizer respectively. For a function $f$ with continuous domain, if
6.1 Basic Definitions and Properties

\[ f(x_*) < f(x) \text{ for all } x \ni \|x - x_*\| < \delta \]

for some \( \delta > 0 \). \( f(x_*) \) is called a local minimum and \( x_* \) is called a local minimizer. Notice that a global minimum is also a local minimum. The qualifier "strict" is also applied to local minima in the same way as with global minima.

The terms "global", "local", and "strict" are also used in discrete problems. A local minimum in a discrete problem is a point that has a smaller function value than the function values of nearby points, for some definition of "nearby points".

**Stationary Points of Differentiable Functions**

If \( f \) is a differentiable function of a scalar variable, for a local or global minimizer \( x_* \), then

\[ f'(x_*) = 0. \]

This is trivially obvious by considering the definition of the derivative and the conditions at the minimum

\[ f(x) < f(x - \delta) \]

and

\[ f(x) < f(x + \delta) \]

for small \( \delta > 0 \).

The derivative is also zero at a local or global maximizer, so obviously this condition is not sufficient to identify a minimum. The derivative may also be zero at a point that is neither a minimum nor a maximum, for example, at \( x = 0 \) for \( f(x) = x^3 \). Such a point is called an inflection point. Any point \( x_* \) such that \( f'(x_*) = 0 \) is called a stationary point. Stationary points can be nuisances when we attempt to find maxima or minima.

If a function \( f \) of a scalar argument is twice-differentiable, information about a stationary point can be obtained from the second derivative. If \( x_* \) is a stationary point; and if

\[ f''(x_*) > 0, \]

the stationary point is a minimum; if

\[ f''(x_*) < 0, \]

the stationary point is a maximum; otherwise it is an inflection point. Again, this is easily seen by using the definition of the derivative of \( f' \) at \( x_* \) and the fact that \( f'(x_*) = 0 \). At a minimum, for example,

\[ f'(x - \delta) < 0 \]

and

\[ f'(x + \delta) > 0 \]
gradient of a function

Global Maximum

Inflection Point

Global Minimum

Local Minimum

Figure 6.4. Stationary Points of a Continuous Function

for small $\delta > 0$.

For a differentiable function of a vector argument, the derivatives also provide information about the local shape of the function. In this case, we consider the vector of derivatives, the gradient,

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

(Recall that we often write a vector in the horizontal notation as in the first equation above, but whenever we perform multiplication operations on vectors or subsetting operations on matrices, we consider a vector to be a column vector; that is, it behaves in many ways as a matrix with one column.)

If $x_*$ is a minimum point of the function $f$, and $f$ is continuously differentiable in a neighborhood of $x_*$, then, similar to the derivative of a function of a scalar, $\nabla f(x_*) = 0$. If this were not the case, we could let $p = -\nabla f(x_*)$ and write

$$p^T \nabla f(x_*) = -(\nabla f(x_*))^T \nabla f(x_*)$$

$$< 0,$$

and then, because $\nabla f(x)$ is continuous in a neighborhood of $x_*$, for some positive $t_0$ we would have

$$p^T \nabla f(x_* + tp) < 0$$

for all $t \in [0, t_0]$.

Now, if we choose $t_1 \in (0, t_0]$, from the mean-value theorem (Taylor’s formula with remainder), we have
\[ f(x_\ast + t_1p) = f(x_\ast) + t_1p^T \nabla f(x_\ast + t_2p) \quad \text{for some } t_2 \in (0, t_1). \]

This, however, would mean \( f(x_\ast + t_1p) < f(x_\ast) \) contradicting the assumption that \( x_\ast \) is a minimum.

The fact that \( \nabla f(x_\ast) = 0 \) at a minimum point is sometimes called the “first-order necessary condition” for \( x_\ast \) to be a local minimum.

If \( \nabla f(x_\ast) = 0 \), \( x_\ast \) is a stationary point; but it may or may not be a minimum. Also, of course, if it is a minimum, it may only be a local minimum. A stationary point may be a local or global minimum, a local or global maximum, an inflection point, or a saddlepoint.

A saddlepoint is a point, \( x_\ast \), in \( \mathbb{R}^m \) such that, given vectors \( v_1 \) and \( v_2 \),

\[ f(x_\ast) < f(x_\ast + \alpha v_1) \]

and

\[ f(x_\ast) > f(x_\ast + \alpha v_2), \]

for \( 0 < |\alpha| < \alpha_0 \) for a given \( \alpha_0 \).

---

**Figure 6.5. Function with a Saddlepoint** gro105

In the case of a function with a discrete domain, a saddlepoint can be defined in a similar fashion, although the usual situation is that for some points in a neighborhood of \( x_\ast \) the function values are greater than \( f(x_\ast) \) and for other points in that neighborhood the function values are less.

As in the scalar case, if a function \( f \) of a vector argument is twice-differentiable, more information about a stationary point can be obtained
from the second derivatives, which comprise a matrix, called the Hessian, which is denoted by $H_f$, and defined as

$$H_f = \nabla^2 f(x)$$

$$= \left( \frac{\partial^2 f(x)}{\partial x_i \partial x_j} \right)$$

$$= \frac{\partial^2 f(x)}{\partial x \partial x^T}.$$ 

Notice that the Hessian is a function, so we often specify the point at which it is evaluated in the ordinary function notation, $H_f(x)$. The symbol $\nabla^2 f(x)$ is also sometimes used to denote the Hessian, but because $\nabla^2 f(x)$ is often used to denote the Laplacian (which yields the diagonal of the Hessian), we will use $H_f(x)$ to denote the Hessian.

If $x_*$ is a minimum point and $H_f(x)$ is continuous in a neighborhood of $x_*$, then $H_f(x_*)$ is positive semidefinite. If $H_f(x_*)$ were not positive semidefinite, we could choose a vector $p$ such that $p^T H_f(x_*) p < 0$. In this case, because $H_f(x)$ is continuous near $x_*$, there would be an interval $[0,t_0]$ such that for any (scalar) $t \in [0,t_0]$, $p^T H_f(x_* + tp) p < 0$. Now, if we choose $t_1 \in [0,t_0]$ and express $f(x_* + t_1 p)$ in a first-order Taylor formula with remainder, we would have for some $t_2 \in (0,t_1)$,

$$f(x_* + t_1 p) = f(x_*) + t_1 p^T \nabla f(x_*) + \frac{1}{2} t_1^2 p^T H_f(x_* + t_2 p) p$$

$$= f(x_*) + \frac{1}{2} t_1^2 p^T H_f(x_* + t_2 p) p < f(x_*).$$

But this contradicts the assumption that $f(x_*)$ is a minimum; hence, we must conclude $H_f(x_*)$ is positive semidefinite.

The fact that $H_f(x_*)$ is positive semidefinite at a minimum point is sometimes called the “second-order necessary condition” for $x_*$ to be a local minimum.

A sufficient condition for $x_*$ to be a strict local minimum, given the continuity assumptions about the derivatives, is that $\nabla f(x_*) = 0$ and $H_f(x_*)$ is positive definite. Because $H_f(x)$ is continuous in a neighborhood of $x_*$, $H_f(x)$ is positive definite within a sufficiently small distance, say $r$, of $x_*$. For any vector $p$ such that $\|p\| < r$,

$$f(x_* + p) = f(x_*) + p^T \nabla f(x_*) + \frac{1}{2} p^T H_f(x_* + \alpha p) p$$

$$= f(x_*) + \frac{1}{2} p^T H_f(x_* + \alpha p) p,$$ 

where $0 < \alpha < 1$. Because $x_* + \alpha p$ is within $r$ of $x_*$, $H_f(x_* + \alpha p)$ is positive definite and so $p^T H_f(x_* + \alpha p) p > 0$. This implies $f(x_* + p) < f(x_*)$ for any vector $p$ small enough; hence $f(x_*)$ is a local minimum.
6.1 Basic Definitions and Properties

A maximum of a function, either a global or local maximum, is sometimes called its mode. This term is often used in reference to a probability density. A function with no local maxima other than a single strict maximum is called unimodal. If $x_\Lambda$ is the mode of $f$, then $f(x_\Lambda) > f(x)$ for $x \neq x_\Lambda$, $f(x)$ is an increasing function for $x \leq x_\Lambda$, and $f(x)$ is a decreasing function for $x \geq x_\Lambda$.

A multivariate function may be unimodal in some directions and not in others. When we say a function is unimodal, we mean it is unimodal in all directions in its domain. Some functions may be unimodal in some projections, or along some slices. A function of $m$ variables is said to be orthounimodal at the mode $x_\Lambda$ in $\mathbb{R}^m$, if $f(x_\Lambda) > f(x)$ for $x \neq x_\Lambda$ and for each $i$, $f(x_1, x_2, \ldots, x_m)$ is an increasing function in $x_i$ for $x_i \leq x_\Lambda i$ and a decreasing function in $x_i$ for $x_i \geq x_\Lambda i$. See Dharmadhikari and Joag-Dev (1988) for discussion of variations.

Bounds and Other Constraints in Optimization Problems

If there are no restrictions on $x$, the problem is an unconstrained optimization problem. Often, however, there are limits on $x$; for example, the elements of $x$ must be such that $l_i \leq x_i \leq u_i$. These simple limits on the decision variables are called bounds. In addition to such simple bounds, $x$ may be required to satisfy other constraints. Constraints are of the general form $g(x) \leq b$, where $g$ is a vector-valued function. Bounds are just simple constraints.

An equality constraint such as $g_1(x) = b_1$ can be formulated as two inequality constraints, $g_1(x) \leq b_1$ and $-g_1(x) \leq -b_1$.

In some cases, the constraints may just be the domain of the objective function; that is, the objective function may be undefined outside of the region specified by the constraints.

A point that satisfies all constraints is said to be feasible. In an unconstrained optimization problem, all points are feasible.

The general optimization problem with bounds and constraints is written as

\[
\min_x f(x) \\
\text{s.t. } -x \leq -l \\
\quad \quad \quad x \leq u \\
\quad \quad \quad g(x) \leq b.
\] (6.4)

This general optimization problem is called a mathematical programming problem. In this standard formulation, the notation “s.t.” means “such that”. Some constraints may be difficult to express in terms of a function, $g$. The decision variables may be required to take on values only in the set of integers, for example. To simplify the notation, sometimes the bounds or other conditions on $x$ are used to define a set $S$, so the constraints are just stated as $x \in S$, properly defined. The set $S$ is called the feasible region.
The inequalities in the constraints allow equality. For continuous variables the problem would not be well-defined otherwise. Any constraint that is satisfied by equality at a given point is called an active constraint at that point, and the point is a boundary point. A point that satisfies all constraints, but is not a boundary point is an interior point.

**General Approaches to Optimization**

Various special cases of the general optimization problem require very different approaches. Although there are general purpose algorithms that can be used on almost any optimization problem, for a given special case, these algorithms are likely to be quite inefficient relative to an algorithm designed for that case. If, for example, \( f \) and \( g \) in problem (6.4) are linear, the problem is called a linear program or a linear programming problem, and there are very efficient methods for solving such linear programming problems that take advantage of the special linear structure. (Notice that the word “program” is sometimes used more-or-less synonymously with “problem”.) If the values of the decision variables are constrained to a set of integers, the problem is called an integer program or a mixed integer program, and in this case, other specialized algorithms must be used.

An unconstrained problem is generally simpler to solve than one with constraints. One general approach to solving an optimization problem with constraints is to begin from an unconstrained optimal solution and then, through a series of steps, to move the solution into the region that satisfies the constraints. This approach may not be reliable if the objective function is not well-behaved outside of the feasible region. Another general approach is to begin inside the feasible region, and to move within the region in directions that decrease the objective function. The optimal solution is often on the boundary of the feasible region, so some common approaches, called active set methods, move along the boundary of the feasible region, seeking decreases in the objective function.

**6.2 Solution of Nonlinear Systems of Equations**

Because of the special properties of derivatives of functions at the optima of the functions, we may solve an optimization problem by solving a simpler problem, namely finding a point at which a function is zero. More generally, a common problem in scientific computing is to solve a nonlinear system

\[
g(x) = b,
\]

where \( g \) is a vector-valued function of a vector argument; that is, \( x \) is an \( m \)-vector and \( b \) is an \( n \)-vector. This is the general case of the linear system \( Ax = b \), where \( A \) is a matrix. By writing \( f(x) = g(x) - b \), we change the problem to one of solving the equation,
“Solving the equation” means finding the value of $x$, say $x_0$, that makes the equation true. The point $x_0$ is called a “root” or a “zero” of the function.

If the objective function in an optimization problem is quadratic in the decision variables, as in the least squares problem, the derivatives in terms of the decision variables are linear. The optimum, therefore, may be found by solving a linear system. When the objective function is not quadratic, but the function is differentiable, the optimum may be found by solving a system of nonlinear equations. Because of this, methods for solution of nonlinear systems of equations are the basic underlying methods of many optimization algorithms.

We describe several general methods. Each of the methods may be the best for some given problem, and it is important to understand how these methods work. There are some specialized methods, such as for finding the roots of a polynomial, but we will not discuss them. We first consider methods for a single equation in a scalar variable.

### 6.2.1 Basic Methods for a Single Equation

Let us first consider the special case of equation (6.5) in which $f$ is a scalar-valued function $f$ of a scalar variable $x$.

If there is no closed form for the inverse $f^{-1}()$, and if $f$ is continuous, then the solution is effected by an iterative process. This iterative process must have a convergence criterion or stopping criterion to decide when the solution is “close enough”. (See the discussion on page 64 in Chapter 3.) In some cases, the primary interest is in the values of the decision variables; and in other cases, the main interest is in the value of the objective function. The criterion may be based on a small positive number, $\epsilon$, to bound the distance of the computed point of the minimum from $x_0$, or to bound the value of $|f(x)|$ at the computed minimum point. The number of iterations allowed must also be bounded; in fact, if there is no stopping criterion independent of the “goodness” of the solution, the method of solution is not an “algorithm”, in a common definition of that term.

In the following discussion, we assume $f$ is a continuous function, and that a solution $x_0$ exists. We will illustrate the methods with the function

\[ f(x) = x^3 - 4x^2 + 18x - 115, \]  

which has a single root at $x = 5$. (As we mentioned earlier, there are special algorithms for polynomials, which we will not discuss. The standard algorithms are given by Jenkins and Traub, 1970a, 1970b, and 1972. A program by Jenkins, 1975, is available in the ACM CALGO and in the IMSL Libraries. Hull and Mathon (1996) describe a modification of the basic method that works better in the case of a multiple root of the polynomial.)
Fixed-Point Method

A general type of iteration for problems such as (6.5) is called a *fixed-point method*. In this problem the fixed-point method uses the fact that at the solution

\[ x_0 = f(x_0) + x_0. \]

The fixed-point iteration is then

\[ x_0^{(k+1)} = f \left( x_0^{(k)} \right) + x_0^{(k)}, \]

after starting with any value \( x_0^{(0)} \).

Bisection Method

One of the simplest iterative methods for solving \( f(x) = 0 \) is the *bisection method*. The method begins with two values that bracket the solution, and then tightens the interval by halves. We assume that there are values \( x_l \) and \( x_u \), with \( x_l < x_u \), such that

\[ f(x_l) \leq 0, \]
\[ f(x_u) \geq 0, \]

and

\[ f(x_0) = 0. \]

(If \( f(x_l) \geq 0 \) and \( f(x_u) \leq 0 \), we can relabel the points.) The method is shown in Algorithm 6.1.

**Algorithm 6.1 Bisection to Find a Root of an Equation**

0. Set \( k = 0 \), and

find an interval \([x_l, x_u]\) in which a solution lies.

1. Set \( k = k + 1 \) and set \( x_0^{(k)} = (x_u + x_l)/2 \).

2. If \( \text{sign} \left( f \left( x_0^{(k)} \right) \right) = \text{sign}(f(x_l)) \), then

2.a. set \( x_l = x_0^{(k)} \);
otherwise

2.b. set \( x_u = x_0^{(k)} \).

3. If \( x_u - x_l \leq \epsilon \)

return the solution as \( x_0^{(k)} \);
otherwise,

if \( k < k_{\text{max}} \),

set \( k = k + 1 \) and go to step 1;
otherwise,

issue message that

‘algorithm did not converge in \( k_{\text{max}} \) iterations’.
In the example in Figure 6.6, the interval is successively halved, first by moving the upper bound down, then moving the lower bound up, then moving the lower bound up again, and so on. In each step the approximation to the solution \( x_0^{(k)} \) is the midpoint of an interval, and then becomes an endpoint of the interval in the next step.

Figure 6.6. Bisection to Find \( x_0 \), so that \( f(x_0) = 0 \)

The bisection method is very easy to understand and to implement. The solution always remains within a known interval. After \( k \) steps, the length of that interval is \( 2^{-k} \) times its initial length, so the error of the approximation is of order \( 2^{-k} \). Each iteration gains one more bit of accuracy. Because the ratios of the lengths of successive intervals is constant, the bisection method converges linearly. The iterations beginning with those shown in Figure 6.6, and continuing until 11 digits of accuracy are shown in Table 6.1. The length of the interval is 7 initially. After 35 steps, it is approximately \( 7 \cdot 2^{-35} \).

The stopping rule in Algorithm 6.1 is based on the length of the interval. It is clear that the algorithm must converge using this stopping rule; in fact, beginning with \( x_l \) and \( x_u \), the algorithm terminates after exactly

\[
\lceil \log_2 (x_u - x_l) / \epsilon \rceil
\]

steps.

At the solution, \( \left| f \left( x_0^{(k)} \right) \right| \) should be close to zero. An alternative stopping rule could be based on this value; that is, for a given \( \epsilon > 0 \), stop when
Newton’s Method

Newton’s method for a differential function is based on the first-order Taylor series of the function about a point near the solution:

\[ f(x) \approx f(x_0^{(k)}) + (x - x_0^{(k)}) f'(x_0^{(k)}). \]

As before, the solution is approached through the iterates, \( x_0^{(k)}, x_0^{(k+1)}, \ldots \). The update is obtained by assuming

\[ f(x_0^{(k+1)}) = 0, \]

and solving the Taylor series approximation

\[ f(x_0^{(k+1)}) \approx f(x_0^{(k)}) + (x_0^{(k+1)} - x_0^{(k)}) f'(x_0^{(k)}). \]

If \( f'(x_0^{(k)}) \neq 0 \), this approximation yields

\[ |f(x_0^{(k)})| \leq \epsilon. \]

The bisection method requires that the function be continuous within the initial interval. The function need not be differential, however.
Newton’s method uses the slope of the function at one point to choose the next point, which is the direction of a smaller value of the function, indicated by the slope. The method is given in Algorithm 6.2.

Algorithm 6.2 Newton’s Method to Find a Root of an Equation

0. Set \( k = 0 \), and determine an approximation \( x^{(k)}_0 \).
1. Solve for \( x^{(k+1)}_0 \) in
\[
0 = x^{(k+1)}_0 - f \left( x^{(k)}_0 \right) \frac{f'(x^{(k)}_0)}{f'(x^{(k)}_0)}
\]
that is, set
\[
x^{(k+1)}_0 = x^{(k)}_0 - \frac{f(x^{(k)}_0)}{f'(x^{(k)}_0)},
\]
if \( f'(x^{(k)}_0) \) exists.
2. If \( \left| x^{(k+1)}_0 - x^{(k)}_0 \right| \leq \epsilon \)
   return the solution as \( x^{(k+1)}_0 \),
   otherwise,
   if \( k < k_{\text{max}} \),
   set \( k = k + 1 \) and go to step 1;
   otherwise,
   issue message that
   ‘algorithm did not converge in \( k_{\text{max}} \) iterations’.

The stopping rule in Algorithm 6.2 is based on the interval between two successive approximations, just as the stopping rule of the bisection method is based on the length of the interval. As we mentioned in discussing the bisection method, there may also be some interest in \( \left| f(x^{(k)}_0) \right| \). This value should be near zero, and it could also be used as a stopping criterion.

Newton’s method is easy to understand and to implement if the derivative is available. In Figure 6.7, we show Newton’s method applied to the same function we used bisection on in Figure 6.6. In the example in Figure 6.7, Newton’s method proceeds in an orderly fashion toward the zero of the function.

Notice in Figure 6.7 that the derivatives (the slopes) are decreasing, as the solution is approached from the right side. This could cause some problems with the method, because the denominator in step 1 of Algorithm 6.2 becomes small. In this example the derivative,
\[
f'(x) = 3x^2 - 8x + 18,
\]
Figure 6.7. Newton’s Method to Find $x_0$, so that $f(x_0) = 0$ gro220

is not zero at the solution. (See Exercise 6.10, page 376. The derivative of the function in Exercise 6.10b is zero at the solution.)

To investigate the convergence of Newton’s method, consider the first-order Taylor series with remainder, expanded about a point near the solution, $x_0^{(k)}$, and evaluated at the solution $x_0$:

$$f(x_0) = f\left(x_0^{(k)}\right) + \left(x_0 - x_0^{(k)}\right)f'\left(x_0^{(k)}\right) + \frac{1}{2} \left(x_0 - x_0^{(k)}\right)^2 f''(\xi) = 0.$$ 

Using equation (6.7), we have

$$\frac{x_0 - x_0^{(k+1)}}{x_0 - x_0^{(k)}}^2 = \frac{1}{2} \frac{f''(\xi)}{f'(x_0^{(k)})}.$$ 

So, if the limit, as $k \to \infty$, of the ratio on the right exists, the convergence is quadratic. It is clear that if $f'(x_0^{(k)}) = 0$ at any point, the method may fail.

Even if the derivatives are not zero, however, Newton’s method may diverge unless the starting point is sufficiently close to the solution. Two ways in which Newton’s method can go wrong are illustrated in Figures 6.8 and 6.9.

In both of these examples, the failure of Newton’s method occurs because the starting point is too far away from the zero. The possibility of this occurring makes the choice of starting value very important. In the bisection
method, we do not have to be concerned about this, so long as we can find values that bracket the solution.

Figure 6.8. Failure of Newton’s Method gro221

Figure 6.9. Failure of Newton’s Method; Another Example gro222
A modification of Newton’s method is to use an approximation to the derivative:

\[ f'(x_0^{(k)}) \approx \frac{f(x_0^{(k)} + h) - f(x_0^{(k)})}{h}. \]

This is sometimes called the “discrete Newton’s method”. It is also essentially the same as the next method we discuss.

**Secant Method**

The *secant method* is similar to Newton’s method in using the slope to determine successive points in the iteration. Newton’s method uses the derivative or the tangent at a given point, and the secant method uses the slope of the function between two given points to choose the next point. The method is given in Algorithm 6.3.

**Algorithm 6.3 Secant Method to Find a Root of an Equation**

1. **Set** \( k = 1 \), and determine approximations \( x_0^{(k-1)} \) and \( x_0^{(k)} \).
2. **Set** \( x_0^{(k+1)} = x_0^{(k)} + \frac{f(x_0^{(k)}) - f(x_0^{(k-1)})}{f(x_0^{(k)}) - f(x_0^{(k-1)})} \).
3. **If** \( |x_0^{(k+1)} - x_0^{(k)}| \leq \epsilon \), return the solution as \( x_0^{(k+1)} \);
   otherwise,
   - if \( k < k_{\text{max}} \), set \( k = k + 1 \) and go to step 1;
   - otherwise, issue message that ‘algorithm did not converge in \( k_{\text{max}} \) iterations’.

The intersection of the line between the two points on the function and the \( x \)-axis is taken as the next point at which to evaluate the function, as we see in Figure 6.10. The choice of \( x_0^{(0)} \) and \( x_0^{(1)} \) is arbitrary, although just as in Newton’s method, if they are too far away from the solution, the method may not converge.

The two points in the secant method may or may not bracket a root.

**Regula Falsi Method**

The *regula falsi* or *false position* method is similar to the secant method, except that the two starting points are chosen so as to bracket a solution, and as in the bisection method, each successive point is chosen so that it together with one of the two previous points brackets a solution. The method given in Algorithm 6.4 is a slight modification of the ordinary regula falsi method, and is sometimes called the “modified” regula falsi method. Because
the “unmodified” regula falsi method (which omits steps 2.a.ii and 2.b.ii in Algorithm 6.4) should not even be used, we just refer to the method given here as regula falsi. Algorithm 6.4) is also sometimes called the “Illinois method”.

Algorithm 6.4 Regula Falsi to Find a Root of an Equation

0. Set \( k = 0 \);
   find an interval \([x_l, x_u]\) in which a solution lies;
   set \( f_l = f(x_l) \);
   set \( f_u = f(x_u) \); and
   set \( x_0^{(k)} = x_l \).
1. Set \( x_0^{(k+1)} = \frac{x_l f_u - x_u f_l}{f_u - f_l} \).
2. If \( f_l f_0^{(k+1)} < 0 \), then
   2.a.i. set \( x_u = x_0^{(k+1)} \) and \( f_u = f(x_0^{(k+1)}) \).
   2.a.ii. if \( f\left(x_0^{(k)}\right) f\left(x_0^{(k+1)}\right) > 0 \), then set \( f_l = f_l / 2 \).
   Otherwise,
   2.b.i. set \( x_l = x_0^{(k+1)} \) and \( f_l = f(x_0^{(k+1)}) \).
   2.b.ii. if \( f\left(x_0^{(k)}\right) f\left(x_0^{(k+1)}\right) > 0 \), then set \( f_u = f_u / 2 \).
3. If \( x_u - x_l \leq \epsilon \),
   return the solution as \( x_0^{(k+1)} \);
   otherwise,
if \( k < k_{\text{max}} \),
set \( k = k + 1 \) and go to step 1;
otherwise,
issue message that
‘algorithm did not converge in \( k_{\text{max}} \) iterations’.

The regula falsi method generally converges more slowly than the secant method, but it is more reliable, because the solution remains bracketed. Figure 6.11 illustrates two iterations of the method.

**Figure 6.11.** Regula Falsi to Find \( x_0 \), so that \( f(x_0) = 0 \)

---

**Stochastic Approximation**

In practical applications we often cannot evaluate \( f(x) \) precisely. Instead, we make observations that are contaminated with random errors or noise. At \( x_0^{(k)} \), instead of \( f\left(x_0^{(k)}\right) \), we observe

\[
y_0^{(k)} = f \left(x_0^{(k)}\right) + \epsilon_k.
\]

A fixed-point iteration of the form

\[
x_0^{(k+1)} = x_0^{(k)} + \hat{f}\left(x_0^{(k)}\right)
\]  

(6.8)
Robbins-Munro procedure

Multiple roots of a function

Robbins and Monro (1951) suggested the recursion

\[ x_{0}^{(k+1)} = x_{0}^{(k)} + \alpha^{(k)}y_{k}, \tag{6.9} \]

where \( \alpha^{(k)} \) is a decreasing sequence of positive numbers similar to \( 1/f'(x_{0}^{(k)}) \) in Newton’s method (6.7), page 363, when the approach is from the left. Convergence in the Robbins-Munro procedure is not deterministic because of the random variables. Our interest must be in convergence in probability or convergence with probability 1. In order to guarantee convergence, the deterministic sequence of \( \alpha^{(k)} \) must satisfy certain norm properties (see Kushner and Yin, 1997).

Multiple Roots

It is possible that the function has more than one root, and we may want to find them all. A common way of addressing this problem is to use different starting points in the iterative solution process. Plots of the points evaluated in the iterations may also be useful. In general, if the number of different roots is unknown, there is no way of finding all of them with any assurance.

If the number of roots in a given interval is known, however, and if the function is twice continuously differentiable in the interval, a “guided” bisection algorithm of Kavvadias and Vrahatis (1996) can be used to find all of them with certainty. We refer the interested reader to their paper for the details.
6.2.2 Accuracy of the Solution

As with most problems in numerical computations, the accuracy we can expect in finding the roots of a function varies from problem to problem; some problems are better conditioned than others. A measure of the condition of the problem of finding the root \( x_0 \) can be developed by considering the error in evaluating \( f(x) \) in the vicinity of \( x_0 \). Suppose a bound on this error is \( \epsilon \), so

\[
|\hat{f}(x_0) - f(x_0)| \leq \epsilon,
\]
or

\[
|\hat{f}(x_0)| \leq \epsilon,
\]

where \( \hat{f}(x_0) \) is the computed value approximating \( f(x_0) \). Let \([x_l, x_u]\) be the largest interval about \( x_0 \) such that

\[
|f(x)| \leq \epsilon, \quad \text{if} \quad x \in [x_l, x_u]. \tag{6.10}
\]

Within this interval, the computed value \( \hat{f}(x) \) can be either positive or negative just due to error in computing the value. A stable algorithm for finding the root of the function yields a value in the interval, but no higher accuracy can be expected. If \( f(x) \) can be expanded in a Taylor series about \( x_0 \), we have

\[
f(x) \approx f(x_0) + f'(x_0)(x - x_0),
\]
or

\[
f(x) \approx f'(x_0)(x - x_0).
\]

Now applying the bound in (6.10) to the approximation, we have that the interval is approximately

\[
x_0 \pm \frac{1}{f'(x_0)} \epsilon,
\]

if the derivative exists and is nonzero. Therefore, if the derivative exists and is nonzero, a quantitative measure of the condition of the problem is

\[
\frac{1}{f'(x_0)}.
\]

This quantity is a condition number of the function \( f \) with respect to finding the root \( x_0 \). In Figure 6.12, we can see the sensitivity of a root-finding algorithm to the condition number.

Wilkinson (1959) considered the polynomial

\[
f(x) = (x - 1)(x - 2)\cdots(x - 20)
\]

for studying rounding error in determining roots. Very small perturbations in the coefficients of the polynomial lead to very large changes in the roots; hence, we referred to the problem as ill-conditioned. The derivative of that
function in the vicinity of the roots is very large, so the condition number defined above would not indicate any conditioning problem. As we pointed out, however, that problem is ill-conditioned because of the extreme variation in the magnitude of the coefficients. This kind of situation is common in numerical analysis. Condition numbers do not always tell an accurate story; they should be viewed only as indicators, not as true measures of the condition.

6.2.3 Systems of Equations

If the argument of the function is an \( m \)-vector and the function value is an \( n \)-vector, equation (6.5),

\[
f(x) = 0,
\]

represents a system of equations:

\[
\begin{align*}
 f_1(x_1, x_2, \ldots, x_m) &= 0 \\
 f_2(x_1, x_2, \ldots, x_m) &= 0 \\
 \vdots & \vdots \\
 f_n(x_1, x_2, \ldots, x_m) &= 0.
\end{align*}
\]  

(6.11)

Each of the functions \( f_i \) is a scalar-valued function of the vector \( x \). Solution of systems of nonlinear equations can be a significantly more computationally intensive problem than solution of a single equation.

Whether or not the system of equations (6.11) has a solution is not easy to determine. A system that has a solution is said to be consistent, just as
a consistent linear system. If \( n > m \), the system may be overdetermined (just as the linear system (5.26) on page 243); and it is very likely that no solution exists. In this case, a criterion, such as least squares, for a good approximate solution must be chosen. Even if \( n = m \), we do not have easy ways of determining whether a solution exists, as we have for the linear system.

**Newton’s Method**

As we have seen in the previous sections, the solution of nonlinear equations proceeds iteratively to points ever closer to zero. The derivative or an approximation to the derivative is used to decide which way to move from a given point. For a scalar-valued function of several variables, say \( f_1(x) \), we must consider the slopes in various directions, that is, the gradient \( \nabla f_1(x) \).

In a system of equations such as (6.11), we must consider all of the gradients; that is, the slopes in various directions of all of the scalar-valued functions. The matrix whose rows are the transposes of the gradients is called the Jacobian. We denote the Jacobian of the function \( f \) by \( J_f \). The transpose of the Jacobian, that is, the matrix whose columns are the gradients, is denoted by \( \nabla f \) for the vector-valued function \( f \). (Note that the symbol \( \nabla \) can denote either a vector or a matrix, depending on whether the function to which it is applied is scalar- or vector-valued.) Thus, the Jacobian for the system above is

\[
J_f = \begin{bmatrix}
\frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_m} \\
\frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_m} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_m}
\end{bmatrix}
\]

\[= (\nabla f)^T. \tag{6.12}\]

Notice that the Jacobian is a function, so we often specify the point at which it is evaluated in the ordinary function notation, \( J_f(x) \). Newton’s method described above for a single equation in one variable can be used to determine a vector \( x_0 \) that solves this system, if a solution exists, or to determine that the system does not have a solution.

For the vector-valued function in the system of equations (6.11), the first-order Taylor series about a point \( x_0^{(k)} \) is

\[f(x) \approx f(x_0^{(k)}) + J_f(x_0^{(k)})(x - x_0^{(k)}).\]

This first-order Taylor series is the basis for Newton’s method, shown in Algorithm 6.5.
Algorithm 6.5 Newton’s Method for a System of Equations

(Compare with Algorithm 6.2, page 363.)

0. Set \( k = 0 \), and determine an approximation \( x_0^{(k)} \).
1. Solve for \( x_0^{(k+1)} \) in
\[
J_f \left( x_0^{(k)} \right) \left( x_0^{(k+1)} - x_0^{(k)} \right) = f \left( x_0^{(k)} \right).
\]
2. If \( \left\| x_0^{(k+1)} - x_0^{(k)} \right\| \leq \epsilon \),
   return the solution as \( x_0^{(k+1)} \);
   otherwise,
   if \( k < k_{\text{max}} \),
   set \( k = k + 1 \) and go to step 1;
   otherwise,
   issue message that ‘algorithm did not converge in \( k_{\text{max}} \) iterations’.

Notice in general that \( m \) and \( n \) are not equal, and the system in step 1 is \( n \) equations in \( m \) unknowns. If, however, \( m = n \), and the Jacobian is nonsingular, the solution in step 1 is
\[
x_0^{(k+1)} = x_0^{(k)} - \left( J_f \left( x_0^{(k)} \right) \right)^{-1} f \left( x_0^{(k)} \right).
\] (6.13)

It is important to remember that this expression does not imply that the Jacobian matrix should be inverted. Linear systems are not solved this way. (See page ?? in Chapter 5.) Expressions involving the inverse of a matrix provide a compact representation, and so we often write equations such as (6.13).

Sometimes, the Jacobian is replaced by a finite-difference approximation,
\[
\left( \frac{\partial f_i}{\partial x_j} \right) \approx \left( \frac{f_i(x_1, x_2, \ldots, x_j + h, \ldots x_m) - f_i(x_1, x_2, \ldots, x_j, \ldots x_m)}{h} \right),
\]
for \( h > 0 \). Use of this approximation in place of the Jacobian is called the “discrete Newton’s method”. This, of course, doubles the number of function evaluations per iteration, but it does avoid the computation of the derivatives.

The number of computations in Newton’s method may be reduced by assuming that the Jacobian (or the discrete approximation) does not change much from one iteration to the next. A value of the Jacobian may be used in a few subsequent iterations.

The number of computations can also be reduced if the Jacobian has a special structure, as is often the case in important applications, such as in solving systems of differential equations. It may be sparse or banded. In these cases, use of algorithms that take advantage of the special structure will reduce the computations significantly.

Other ways of reducing the computations in Newton’s method use an estimate of the derivative that is updated within each iteration. This kind of
method is called quasi-Newton. We will discuss quasi-Newton methods for optimization problems in Section 7.1.6.

If the ranges of the variables in a nonlinear system are quite different, the solution may not be very accurate. The accuracy can often be improved considerably by scaling the variables and the function values so that they all have approximately the same range. Scaling of a variable \( x_i \) is just a multiplicative transformation: \( y_i = \sigma x_i \). Of course, the ranges of the values of the variables may not be known in advance, so it may be necessary to do some preliminary computations in order to do any kind of useful scaling.

**Stochastic Approximation**

The Robbins-Munro stochastic approximation (see page 369),

\[
x_0^{(k+1)} = x_0^{(k)} + \alpha^{(k)} y_k,
\]

extends immediately to the case in which \( x \) and \( y \) are vectors. The weights in the update, \( \alpha^{(k)} \), can either be constant for each element of \( y_k \), or the weights may be elements of a diagonal matrix that allows different weights for each element of \( y_k \).

More generally, a matrix that reflects the correlational structure of \( y_k \) may be used in the update. We have a recursion similar to the Newton update (6.13) with \( \left( J f (x_0^{(k)}) \right)^{-1} \) replaced by some \( A^{(k)} \), and \( f (x_0^{(k)}) \) replaced by \( y_k \). A further extension allows the observations on the underlying random vector \( Y \) to be correlated.

We will encounter the Robbins-Munro procedure again in Section 7.1.11, but for more details on it and related methods we refer the reader to Kushner and Yin (1997).

**Exercises**

6.1. Give an example of a set of points in \( \mathbb{R}^3 \) that constitute a simplex.

6.2. Consider the function \( f(x, y) = ax^2 + bxy + cy^2 \).
   a) What is the Hessian, \( H_f \)?
   b) Under what conditions on \( a \), \( b \), and \( c \) is \( f \) convex? *Hint: see section 5.1.1*
   c) Under what conditions on \( a \), \( b \), and \( c \) would \( f \) have a saddlepoint?

6.3. For the function in Exercise 6.2, choose values of \( a \), \( b \), and \( c \) that make \( f \) convex, and plot the function over a rectangular domain centered on \((0, 0)\).

6.4. For the function in Exercise 6.2, choose values of \( a \), \( b \), and \( c \) that cause \( f \) to have a saddlepoint, and plot the function over a rectangular domain centered on \((0, 0)\).
6.5. In what essential way does the Hessian for your function in Exercise 6.3 differ from the Hessian for your function in Exercise 6.4?

6.6. Now consider a more interesting function:

\[ g(x, y) = \sin\left(\sqrt{x^2 + y^2}\right) / \sqrt{x^2 + y^2}, \quad \text{if} \quad x^2 + y^2 \neq 0; \]
\[ = 1, \quad \text{otherwise}. \]

a) Plot \( g(x, y) \) for \(-10 \leq x \leq 10\) and \(-10 \leq y \leq 10\).
b) Plot the Hessian \( H_g(x, y) \) over the same range.
c) Determine a value of \( c \) such that \( H_g(x, y) \) is positive semidefinite for \( x^2 + y^2 = c^2 \). Identify this circle in your plot of \( g(x, y) \).
d) Determine a value of \( d \) such that \(-H_g(x, y)\) is positive semidefinite for \( x^2 + y^2 = d^2 \). Identify this circle in your plot of \( g(x, y) \).

6.7. Suppose \( f(x) \) and \( g(x) \) are convex functions over \( \mathbb{R}^d \). Show that

\[ h(x) = f(x) + g(x) \]

is a convex function over \( \mathbb{R}^d \).

6.8. If \( X \) is a random variable with probability density function \( p(x) \), the expected value of any function of \( X \), say \( g(X) \), is

\[ E(g(X)) = \int_{-\infty}^{\infty} g(x)p(x) \, dx, \]

provided the integral exists. Show that if \( g \) is a twice-differentiable convex function and \( E(X) \) is finite,

\[ g(E(X)) \leq E(g(X)). \]

This is another form of Jensen’s inequality. It also holds if \( g \) is not twice-differentiable, but the proof is more difficult.

Exercises 6.9 through 6.12 require you to write simple programs to find the zeros of functions. Use Fortran, C, Matlab, R, S-Plus, PV-Wave, or any other general-purpose language. Your program modules should be independent of the function and should allow the user to input starting values and stopping criteria.

6.9. Write a program module to implement the bisection method to find a root of a given function, which is input together with values that bracket a root, and an epsilon as the stopping criterion. Use bisection to determine the first zero of the Bessel function of the first kind, of order 0:

\[ J_0(x) = \frac{1}{\pi} \int_0^\pi \cos(x \sin \theta) \, d\theta. \]

(This function is available in Matlab, \texttt{besselj}; in PV-Wave, \texttt{besselj}; in the IMSL Library, \texttt{bsj0/dbsj0}; and in the Unix math library, \texttt{j0}.)
6.10. Write a program module similar to that of Exercise 6.9 to implement Newton’s method to find a root of a given function, which is input together with its derivative, a starting value, and two stopping criteria: an epsilon and a maximum number of iterations.

a) Observe the performance of the method on the function

\[ f(x) = x^3 - 14x^2 + 68x - 115, \]

which is the function used in the examples in Section 6.2. Start with \( x_0^{(0)} = 9 \). Print \( x_0^{(k)} \) to 10 digits, and observe the number of correct digits at each iteration until the solution is accurate to 10 digits. Produce a table similar to Table 6.1 on page 362. What is the rate of convergence?

b) Now observe the performance of the method on the function

\[ f(x) = x^3 - 15x^2 + 75x - 125, \]

whose solution is also 5. Again start with \( x_0^{(0)} = 9 \). What is the rate of convergence? What is the difference?

6.11. Write a program module similar to that of Exercise 6.9 to implement the secant method to find a root of a given function, which is input together with two starting values, and two stopping criteria: an epsilon and a maximum number of iterations. Observe the performance of the method on the function

\[ f(x) = x^3 - 14x^2 + 68x - 115. \]

Produce a table similar to Table 6.1 on page 362.

6.12. Write a program module similar to that of Exercise 6.9 to implement the regula falsi method to find a root of a given function, which is input together with two starting values, and two stopping criteria: an epsilon and a maximum number of iterations. Your program should check that the two starting values are legitimate. Observe the performance of the method on the function

\[ f(x) = x^3 - 14x^2 + 68x - 115. \]

Produce a table similar to Table 6.1 on page 362.

6.13. Compare the performance of the three methods in Exercises 6.10 through 6.12 and that of the bisection method for the given polynomial. Consider such things as rate of convergence and ease of use of the method.

6.14. Now consider the same function \( f \) as in the previous exercises, except assume that the value of \( f(x) \) can not be observed exactly. More precisely, suppose when we attempt to compute \( f(x) \), we get the value

\[ \hat{f}(x) = f(x) + \epsilon, \]

where \( \epsilon \) is a realization of a normal random variable with mean 0 and variance 0.0001. Compare the performance of the three methods in Exercises 6.9 through 6.12 for finding a zero of \( \hat{f}(x) \). Consider such things as
rate of convergence and ease of use of the method. What other issues are relevant?
Optimization and Nonlinear Systems

Optimization problems arise in many areas of science and industry. Business decisions may be made in an attempt to maximize profit or to minimize risk. Modeling and estimation are usually optimization problems. Scientific models may be developed based on maximum entropy or minimum energy principles. Methods of data analysis may be chosen to maximize likelihood or to minimize residuals. The best fit of a model may be defined in terms of a minimum of a norm, such as least squares. Estimates of parameters may be chosen to maximize a probability density at the point of the observed data.

The methods of optimization depend on the functions involved. The first consideration is the domain of the function. Many functions of interest are over continuous (dense) domains, and many of the methods of optimization are developed for such functions. Many methods for such functions also assume the functions are differentiable, so for functions with dense domains, the differentiability of the functions are of primary concern. The simplest differentiable functions to work with are polynomials, and among these, linear and quadratic functions are particularly simple. Optimization methods for differentiable functions are often developed using a quadratic function as a prototype function.

Many functions of interest have discrete domains. Points in the domain are combinations of allowable values of a set of variables, and the problem is one of combinatorial optimization.

For either discrete or dense domains, restrictions within the domain can make a relatively simple problem much more difficult.

Because of the differences in optimization problems resulting from differences in the properties of the functions and of their domains, the research literature in optimization is quite diverse. Some journals that emphasize optimization, especially the computational aspects, are *Journal of Optimization Theory and Applications*, *Journal of Global Optimization*, *Mathematical Programming*, *Optimization Methods and Software*, and *SIAM Journal on Optimization*. In addition, journals on numerical analysis, operations research or in various fields of application often contain articles on optimization.

7.1 Unconstrained Descent Methods in Dense Domains

We now return to the problem of finding $x_*$ such that

$$\min_x f(x) = f(x_*),$$

where $x$ is an $m$-vector and $f$ is a real scalar-valued function.

In this section we consider the continuous optimization problem in which $x$ is a vector in a dense subset of $\mathbb{R}^m$. In Section 7.2 we discuss discrete optimization, in which $x$ is restricted to a countable subset. In this section we generally assume the function is differentiable in all variables, and we often assume it is twice differentiable in all variables. The properties of derivatives and their characterization of stationary points discussed in Section 6.1 are the basis for most optimization methods for differentiable functions. Sometimes, rather than using the exact derivatives it is more efficient to use approximations such as finite differences. If the function is not differentiable, but is “well-behaved”, the methods based on finite differences often allow us to determine the optimum.

For the time being we will consider the problem of unconstrained optimization. The methods we describe are the basic ones whether constraints are present or not.

Solution of an optimization problem is usually an iterative process, moving from one point on the function to another. The basic things to determine are

- direction or path, $p$, in which to step and
- how far to step. (The step length is $\|\alpha p\|$, for the scalar $\alpha$.)

7.1.1 Direction of Search

For a differentiable function, from any given point, an obvious direction to move is the negative gradient, or a direction that has an acute angle with the negative gradient. We call a vector $p$ such that
a descent direction at the point $x$. For a function of a single variable, this direction of course is just the sign of the derivative of $f$ at $x$.

If $\nabla f(x) \neq 0$, we can express $p$ as

$$Rp = -\nabla f(x),$$

for some positive definite matrix $R$. A particular choice of $R$ determines the direction. A method that determines the direction in this manner is called a “gradient method”.

Numerical computations for quantities such as $p^T \nabla f(x)$ that may be close to zero must be performed with some care. We sometimes impose the requirement

$$p^T \nabla f(x) < -\epsilon,$$

for some positive number $\epsilon$, so as to avoid possible numerical problems for quantities too close to zero.

Once a direction is chosen, the best step is the longest one for which the function continues to decrease.

These heuristic principles of choosing a “good” direction and a “long” step guide our algorithms, but we must be careful in applying the principles.

### 7.1.2 Line Searches

Although the first thing we must do is to choose a descent direction, in this section we consider the problem of choosing the length of a step in a direction that has already been chosen. In subsequent sections we return to the problem of choosing the direction.

We assume the direction chosen is a descent direction. The problem of finding a minimum is similar to, but more complicated than, the problem of finding a zero of a function that we discussed in Section 6.2. In finding a root of a continuous function of a single scalar variable, two values can define an interval in which a root must lie. Three values are necessary to identify an interval containing a local minimum. Nearby points in a descent direction form a decreasing sequence, and any point with a larger value defines an interval containing a local minimum.

After a direction of movement $p^{(k)}$ from a point $x^{(k)}$ is determined, a new point, $x^{(k+1)}$, is chosen in that direction:

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)},$$

where $\alpha^{(k)}$ is a positive scalar, called the step length factor. (The step length itself is $\|\alpha^{(k)} p^{(k)}\|. $)

Obviously, in order for the recursion (7.2) to converge, $\alpha^{(k)}$ must approach 0. A sequence of $\alpha^{(k)}$ that converges to 0, even in descent directions, clearly
golden section search does not guarantee that the sequence \( x^{(k)} \) will converge to \( x_\ast \), however. This is easily seen in the case of the function of the scalar \( x \),

\[
f(x) = x^2,
\]

starting with \( x^{(0)} = 3 \) and \( \alpha^{(0)} = 1 \), proceeding in the descent direction \(-x\), and updating the step length factor as \( \alpha^{(k+1)} = \frac{1}{2} \alpha^{(k)} \). The step lengths clearly converge to 0, and while the sequence \( x^{(k)} \) goes in the correct direction, it converges to 1, not to the point of the minimum of \( f \), \( x_\ast = 0 \).

Choice of the “best” \( \alpha^{(k)} \) is an optimization problem in one variable:

\[
\min_{\alpha^{(k)}} f(x^{(k)} + \alpha^{(k)} p^{(k)}), \tag{7.3}
\]

for fixed \( x^{(k)} \) and \( p^{(k)} \). An issue in solving the original minimization problem for \( f(x) \) is how to allocate the effort between determining a good \( p^{(k)} \) and choosing a good \( \alpha^{(k)} \). Rather than solving the minimization problem to find the best value of \( \alpha^{(k)} \) for the \( k \)th direction, it may be better to get a reasonable approximation, and move on to choose another direction from the new point.

One approach to choosing a good value of \( \alpha^{(k)} \) is to use a simple approximation to the one-dimensional function we are trying to minimize:

\[
\rho(\alpha) = f(x^{(k)} + \alpha p^{(k)}).
\]

A useful approximation is a second- or third-degree polynomial that interpolates \( \rho(\alpha) \) at three or four nearby points. The minimum of the polynomial can be found easily, and the point of the minimum may be a good choice for \( \alpha^{(k)} \).

A simpler approach, assuming \( \rho(\alpha) \) is unimodal over some positive interval, say \([\alpha_l, \alpha_u]\), is just to perform a direct search along the path \( p^{(k)} \). A bisection method or some other simple method for finding a zero of a function as we discussed in Section 6.2.1 could be modified and used.

Another approach for developing a direct search method is to choose two points \( \alpha_1 \) and \( \alpha_2 \) in \([\alpha_1, \alpha_u]\), with \( \alpha_1 < \alpha_2 \), and then, based on the function values of \( \rho \), to replace the interval \( I = [\alpha_1, \alpha_u] \) with either \( I_l = [\alpha_1, \alpha_2] \) or \( I_u = [\alpha_1, \alpha_u] \). In the absence of any additional information about \( \rho \), we choose the points \( \alpha_1 \) and \( \alpha_2 \) symmetrically, in such a way that the lengths of both \( I_l \) and \( I_u \) are the same proportion, say \( \tau \), of the length of the original interval \( I \). This leads to \( \tau^2 = 1 - \tau \), the golden ratio. The search using this method of reduction is called the golden section search, and is given in Algorithm 7.1.
Algorithm 7.1 Golden Section Search

0. Set $\tau = (\sqrt{5} - 1)/2$ (the golden ratio).
   Set $\alpha_1 = \alpha_l + (1 - \tau)(\alpha_u - \alpha_l)$ and set $\alpha_2 = \alpha_l + \tau(\alpha_u - \alpha_l)$.
   Set $\rho_1 = \rho(\alpha_1)$ and $\rho_2 = \rho(\alpha_2)$.

1. If $\rho_1 > \rho_2$,
   1.a. set $\alpha_l = \alpha_1$,
       set $\alpha_1 = \alpha_2$,
       set $\alpha_2 = \alpha_l + \tau(\alpha_u - \alpha_l)$,
       set $\rho_1 = \rho_2$, and
       set $\rho_2 = \rho(\alpha_2)$;
   otherwise,
   1.b. set $\alpha_u = \alpha_2$,
       set $\alpha_2 = \alpha_1$,
       set $\alpha_1 = \alpha_l + (1 - \tau)(\alpha_u - \alpha_l)$,
       set $\rho_2 = \rho_1$, and
       set $\rho_1 = \rho(\alpha_1)$.

2. If $\alpha_u - \alpha_l > \epsilon$ (a preset tolerance)
   go to step 1;
   otherwise,
   return the solution as $\alpha_1$.

The golden section search is robust, but it is only linearly convergent, like
the bisection method of Algorithm 6.1. (This statement about convergence
applies just to this one-dimensional search, which is a subproblem in our
optimization problem of interest.)

Another criterion for a direct search is to require
\[ f(x^{(k)} + \alpha^{(k)} p^{(k)}) \leq f(x^{(k)}) + \tau \alpha^{(k)} (p^{(k)})^T \nabla f(x^{(k)}), \tag{7.4} \]
for some $\tau$ in $(0, \frac{1}{2})$. This criterion is called the sufficient decrease condition,
and the approach is called the Goldstein-Armijo method after two early inves-
tigators of the technique. After choosing $\tau$, the usual procedure is to choose
$\alpha$ as the largest value in 1, $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots$ that satisfies the inequality.

If the step length is not too long, the descent at $x^{(k)}$ in the given direction
will be greater than the descent in that direction at $x^{(k)} + \alpha^{(k)} p^{(k)}$. This leads
to the so-called curvature condition:
\[ \left| (p^{(k)})^T \nabla f(x^{(k)} + \alpha^{(k)} p^{(k)}) \right| \leq \eta \left| (p^{(k)})^T \nabla f(x^{(k)}) \right|, \tag{7.5} \]
for some $\eta$ in $(0, 1)$.

More and Thuente (1994) describe other ways of doing the line search, and
provide some empirical results on the performance of the searches.

7.1.3 Steepest Descent

We now turn to the problem of choosing a descent direction. Most methods
we will consider are gradient methods, that is, they satisfy (7.1):
From a given point \( x^{(k)} \), the function \( f \) decreases most rapidly in the direction of the negative gradient, \( -\nabla f(x^{(k)}) \). A greedy algorithm uses this steepest descent direction; that is,

\[
p^{(k)} = -\nabla f(x^{(k)}),
\]

and so the update in equation (7.2) is

\[
x^{(k+1)} = x^{(k)} - \alpha^{(k)} \nabla f(x^{(k)}).
\]

The step length factor \( \alpha^{(k)} \) is chosen by a method described in Section 7.1.2.

The steepest descent method is robust so long as the gradient is not zero. The method, however, is likely to change directions often, and the zigzag approach to the minimum may be quite slow (see Exercise 7.1a). For a function with circular contours, steepest descent proceeds quickly to the solution. For a function whose contours are ellipses, as the function in Exercise 7.1 (page 441), for example, the steepest descent steps will zigzag toward the solution. A matrix other than the identity may deform the elliptical contours so they are more circular. In Newton’s method discussed next, we choose the Hessian.

### 7.1.4 Newton’s Method

To find the minimum of the scalar-valued function \( f(x) \), under the assumptions that \( f \) is convex and twice differentiable, we can seek the zero of \( \nabla f(x) \) in the same way that we find a zero of a vector-valued function using the iteration in equation (6.13), page 373. We begin by forming a first-order Taylor series expansion of \( \nabla f(x) \), which is the second-order expansion of \( f(x) \).

In place of a vector-valued function we have the gradient of the scalar-valued function, and in place of a Jacobian, we have the Hessian \( H_f \), which is the Jacobian of the gradient.

This first-order Taylor series expansion of \( \nabla f \) is equivalent to a second-order Taylor series expansion of \( f \). Setting the gradient to zero, we obtain an iteration similar to equation (6.13):

\[
x^{(k+1)} = x^{(k)} - (H_f(x^{(k)}))^{-1} \nabla f(x^{(k)}).
\]

Use of this recursive iteration is Newton’s method. The method is also often called the Newton-Raphson method.

In one dimension, the Newton recursion is just

\[
x^{(k+1)} = x^{(k)} - \frac{\nabla f(x^{(k)})}{\nabla^2 f(x^{(k)})} = x^{(k)} - \frac{f'(x^{(k)})}{f''(x^{(k)})}.
\]
The second-order Taylor series approximation to $f$ about the point $x_*$,

$$f(x) \approx f(x_*) + (x - x_*)^T \nabla f(x_*) + \frac{1}{2}(x - x_*)^T H_f(x_*)(x - x_*), \quad (7.8)$$

is exact if $f$ is a quadratic function. In that case, $H_f$ is positive definite, and the terms in equation (7.7) exist and yield the solution immediately. When $f$ is not quadratic, but is sufficiently regular, we can build a sequence of approximations by quadratic expansions of $f$ about approximate solutions. This means, however, that the Hessian may not be positive definite and its inverse in (7.7) may not exist.

Once more, it is important to state that we do not necessarily compute each term in an expression. We choose mathematical expressions for their understandability; we choose computational method for their robustness, accuracy, and efficiency. Just as we commented on page 373 concerning inversion of the Jacobian, we comment here that we do not compute the Hessian and then compute its inverse, just because that appears in equation (7.7). We solve the linear systems

$$H_f(x^{(k)})p^{(k)} = -\nabla f(x^{(k)}) \quad (7.9)$$

by more efficient methods such as Cholesky factorizations. Once we have the solution to equation (7.9), equation (7.7) becomes

$$x^{(k+1)} = x^{(k)} + p^{(k)}. \quad (7.10)$$

Newton’s method, by scaling the path by the Hessian, is more likely to point the path in the direction of a local minimum, whereas the steepest descent method, in ignoring the second derivative, follows a path along the gradient, that does not take into account the rate of change of the gradient. This is illustrated in Figure 7.1.

For functions that are close to a quadratic within a region close to the minimum, Newton’s method can be very effective so long as the iterations begin close enough to the solution. In other cases Newton’s method may be unreliable. The problems may be similar to those illustrated in Figures 6.8 and 6.9 (page 365) for finding a root.

One way of increasing the reliability of Newton’s method is to use a damped version of the update (7.10),

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)},$$

for which a line search is used to determine an appropriate step length factor $\alpha^{(k)}$.

When the function is not quadratic, the Hessian may not be positive definite, and so a modified Cholesky factorization may be used. In this approach, positive quantities are added as necessary during the decomposition of the Hessian. This changes the linear system (7.9) to the system

$$\left( H_f(x^{(k)}) + D^{(k)} \right) p^{(k)} = -\nabla f(x^{(k)}), \quad (7.11)$$
Another method of increasing the reliability of Newton’s method is to restrict the movements to regions where the second-order Taylor expansion (7.8) is a good approximation. This region is called a “trust region”. At the $k^{\text{th}}$ iteration, the second-order Taylor series approximation provides a scaled quadratic model $q^{(k)}$:

\[ q^{(k)}(s) = f(x^{(k)}_*) + s^T \nabla f(x^{(k)}_*) + \frac{1}{2} s^T H f(x^{(k)}_*) s, \quad (7.12) \]

where $s = x - x^{(k)}_*$.

When the Hessian is indefinite, $q^{(k)}$ is unbounded below, so it is obviously not a good model of $f(x^{(k)}_* + s)$ if $s$ is large. We therefore restrict $\|s\|$, or better we restrict $\|D^{(k)}s\|$ for some scaling matrix $D^{(k)}$. For some $\tau^{(k)}$, we require

\[ \|D^{(k)}s\| < \tau^{(k)}, \quad (7.13) \]

and we choose $s^{(k)}$ as the point where the quadratic $q^{(k)}$ achieves its minimum subject to this restriction. How much we should restrict $s$ depends on how good the quadratic approximation is. If

\[ \frac{f(x^{(k)}_*) - f(x^{(k)}_* + s^{(k)})}{f(x^{(k)}_*) - q^{(k)}(s^{(k)})} \]
7.1 Unconstrained Descent Methods in Dense Domains

is close to 1, that is, if the approximation is good, we increase $\tau^{(k)}$; if it is small or negative, we decrease $\tau^{(k)}$. Implementation of these methods requires some rather arbitrary choices of algorithm parameters.

7.1.5 Accuracy of Optimization Using Gradient Methods

The problem of finding a minimum of a function is somewhat more difficult than that of finding a zero of a function discussed in Section 6.2. Our intuition should tell us this is the case. In one dimension, a zero of a function can be determined by successively bracketing a zero with two points. An interval containing a minimum of a function requires three points to determine it.

Another way of comparing the accuracy of the solution of a nonlinear equation and the determination of the minimum of such an equation is to consider the Taylor expansion:

$$f(x) = f(\tilde{x}) + (x - \tilde{x})f'(\tilde{x}) + \frac{1}{2}(x - \tilde{x})^2f''(\tilde{x}) + \cdots.$$ 

In the problem of finding a zero $x_0$, $f'(x_0)$ is nonzero, and for $\tilde{x}$ close to $x_0$, $(f(x) - f(\tilde{x}))$ is approximately proportional to $(x - \tilde{x})$, where the constant of proportionality is $f'(\tilde{x})$. A small value of the difference $(x - \tilde{x})$ results in a proportionate difference $(f(x) - f(\tilde{x}))$. On the other hand, in the problem of finding the minimum $x^*$, $f'(x^*)$ is zero, and for $\tilde{x}$ close to $x^*$, $(f(x) - f(\tilde{x}))$ is approximately proportional to $(x - \tilde{x})^2$, where the constant of proportionality is $f''(\tilde{x})$. A small value of the difference $(x - \tilde{x})$ results in a smaller difference $(f(x) - f(\tilde{x}))$. In finding roots of an equation we may set a convergence criterion proportional to the machine epsilon, $\epsilon_{\text{mach}}$. In optimization problems, we often set a convergence criterion proportional to $\sqrt{\epsilon_{\text{mach}}}$.

7.1.6 Quasi-Newton Methods

All gradient descent methods determine the path of the step by the system of equations,

$$R^{(k)}p^{(k)} = -\nabla f(x^{(k)}).$$

The steepest descent method chooses $R^{(k)}$ as the identity, $I$, in these equations. As we have seen, for functions with eccentric contours, the steepest descent method traverses a zigzag path to the minimum. Newton’s method chooses $R^{(k)}$ as the Hessian, $H_f(x^{(k)}_*)$, which results in a more direct path to the minimum. Aside from the issues of consistency of the resulting equation (7.11) and the general problems of reliability, a major disadvantage of Newton’s method is the computational burden of computing the Hessian, which is $O(m^2)$ function evaluations, and solving the system, which is $O(m^3)$ arithmetic operations, at each iteration.

Instead of using the Hessian at each iteration, we may use an approximation, $B^{(k)}$. We may choose approximations that are simpler to update and/or
quasi-Newton method
variable metric
method

that allow the equations for the step to be solved more easily. Methods using such approximations are called quasi-Newton methods or variable metric methods.

Because

\[ H_f(x^{(k+1)})(x^{(k+1)} - x^{(k)}) \approx \nabla f(x^{(k+1)}) - \nabla f(x^{(k)}) , \]

we choose \( B^{(k)} \) so that

\[ B^{(k+1)}(x^{(k+1)} - x^{(k)}) = \nabla f(x^{(k+1)}) - \nabla f(x^{(k)}) . \quad (7.14) \]

This is called the secant condition. (Note the similarity to the secant method for finding a zero discussed in Sections 6.2.1 and 6.2.3.)

We express the secant condition as

\[ B^{(k+1)}s^{(k)} = y^{(k)} , \quad (7.15) \]

where

\[ s^{(k)} = x^{(k+1)} - x^{(k)} \]

and

\[ y^{(k)} = \nabla f(x^{(k+1)}) - \nabla f(x^{(k)}) . \]

The system of equations in (7.15) does not fully determine \( B^{(k)} \) of course. Because \( B^{(k)} \) is approximating \( H_f(x^{(k)}) \), we may want to require that it be symmetric and positive definite.

The most common approach in quasi-Newton methods is first to choose a reasonable starting matrix \( B^{(0)} \) and then to choose subsequent matrices by additive updates,

\[ B^{(k+1)} = B^{(k)} + B^{(k)}_a , \]

subject to preservation of symmetry and positive definiteness.

The general steps in a quasi-Newton method are

0. Set \( k = 0 \) and choose \( x^{(k)} \) and \( B^{(k)} \).
1. Compute \( s^{(k)} \) as \( \alpha^{(k)}p^{(k)} \), where
   \[ B^{(k)}p^{(k)} = -\nabla f(x^{(k)}) . \]
2. Compute \( x^{(k+1)} \) and \( \nabla f(x^{(k+1)}) \).
3. Check for convergence and stop if converged.
4. Compute \( B^{(k+1)} \).
5. Set \( k = k + 1 \), and go to 1.

Within these general steps there are two kinds of choices to be made: the way to update the approximation \( B^{(k)} \), and, as usual, the choice of the step length factor \( \alpha^{(k)} \).

There are several choices for the update \( B^{(k)}_a \) that preserve symmetry and positive definiteness (or at least nonnegative definiteness). One simple choice is the rank-one symmetric matrix
This update results in a symmetric matrix that satisfies the secant condition no matter what the previous matrix \( B^{(k)} \) is. (You are asked to do the simple algebra to show this in Exercise 7.3.) If \( B^{(k)} \) is positive definite, this update results in a positive definite matrix \( B^{(k+1)} \) so long as \( c^{(k)} \leq 0 \), where \( c^{(k)} \) is the denominator:

\[
c^{(k)} = (y^{(k)} - B^{(k)}s^{(k)})^T s^{(k)}.\]

Even if \( c^{(k)} > 0 \), positive definiteness can be preserved by shrinking \( c^{(k)} \) to \( \tilde{c}^{(k)} \) so that

\[
\tilde{c}^{(k)} < \frac{1}{(y^{(k)} - B^{(k)}s^{(k)})^T (B^{(k)})(-1)(y^{(k)} - B^{(k)}s^{(k)})}.
\]

Although this adjustment is not as difficult as it might appear, the computations to preserve positive definiteness and, in general, good condition of the \( B^{(k)} \) account for a major part of the effort in quasi-Newton methods.

Other, more common choices for \( B_a^{(k)} \) are the rank-two Broyden updates of the form

\[
B_a^{(k)} = \frac{1}{(s^{(k)})^TB^{(k)}s^{(k)}} B^{(k)}s^{(k)}(B^{(k)}s^{(k)})^T + \frac{1}{(y^{(k)})^T s^{(k)}} y^{(k)}(y^{(k)})^T + \sigma^{(k)} ((s^{(k)})^T B^{(k)}s^{(k)}) y^{(k)}(y^{(k)})^T,
\]

where \( \sigma^{(k)} \) is a scalar in \([0, 1]\), and

\[
y^{(k)} = \frac{1}{(y^{(k)})^T s^{(k)}} y^{(k)} - \frac{1}{(s^{(k)})^T} B^{(k)}s^{(k)} B^{(k)}s^{(k)}.
\]

Letting \( \sigma^{(k)} = 0 \) in (7.17) yields the Broyden-Fletcher-Goldfarb-Shanno (BFGS) update, which is one of the most widely used methods. If \( \sigma^{(k)} = 1 \), the method is called the Davidon-Fletcher-Powell (DFP) method.

The Broyden updates will preserve the positiveness of \( B^{(k)} \) so long as

\[
(y^{(k)})^T s^{(k)} > 0.
\]

This is the curvature condition (see (eq:opt327) on page 383). If the curvature condition is not satisfied, \( s^{(k)} \) could be scaled so as to satisfy this inequality. (Scaling \( s^{(k)} \) of course changes \( y^{(k)} \) also.) Alternatively, the update of \( B^{(k)} \) can just be skipped, and the updated step is determined using the previous value, \( B^{(k)} \). This method is obviously quicker, but it is not as reliable.

Inspection of either the rank-one updates (7.16) or the rank-two updates (7.17) reveals that the number of computations is \( O(m^2) \). If the updates are done to the inverses of the \( B^{(k)} \)'s or to their Cholesky factors, the
computations required for the updated directions are just matrix-vector multiplications and hence can also be computed in $O(n^2)$ computations.

It is easily seen that the updates can be done to the inverses of the $B^{(k)}$'s using the Sherman-Morrison formula, equation (5.87) on page 303, for rank-one updates, or the Woodbury formula, equation (5.89), for more general updates. Using the Woodbury formula, the BFGS update, for example, results in the recursion,

$$
(B^{(k+1)})^{-1} = \left( I - \frac{1}{(y^{(k)})^T s^{(k)}} s^{(k)}(y^{(k)})^T \right) (B^{(k)})^{-1} \left( I - \frac{1}{(y^{(k)})^T s^{(k)}} s^{(k)}(y^{(k)})^T \right)
+ \frac{1}{(y^{(k)})^T s^{(k)}} s^{(k)}(y^{(k)})^T.
$$

The best way of doing the inverse updates is to perform them on the Cholesky factors instead of on the inverses. The expression above for updating the inverse shows that this can be done.

Another important property of the quasi-Newton methods is that they can be performed without explicitly storing the $B^{(k)}$'s, which could be quite large in large-scale optimization problems. The storage required in addition to that for $B^{(k)}$ is for the vectors $s^{(k)}$ and $y^{(k)}$. If $B^{(k)}$ is a diagonal matrix, the total storage is $O(m)$. In computing the update at the $(k+1)^{th}$ iteration, limited-memory quasi-Newton methods assume that $B^{(k-j)}$ is diagonal at some previous iteration. The update for the $(k+1)^{th}$ iteration can be computed by vector-vector operations beginning back at the $(k-j)^{th}$ iteration. In practice, diagonality is assumed at the fourth or fifth previous iteration; that is, $j$ is taken as 4 or 5.

Quasi-Newton methods are available in most of the widely-used mathematical software packages. Broyden updates are the most commonly used in these packages, and of the Broyden updates, BFGS is probably the most popular.

Nocedal (1992) discusses the various choices for updates in quasi-Newton methods and provides some comparisons. Khalfan, Byrd, and Schnabel (1993) and Byrd, Nocedal, and Schnabel (1994) also provide comparisons of update methods. Their results showed that the simple rank-one update (7.16) is often a superior method.

**Truncated Newton Methods**

Another way of reducing the computational burden in Newton-type methods is to approximate the solution of the path direction

$$
R^{(k)} p^{(k)} = -\nabla f(x^{(k)}),
$$

where $R^{(k)}$ is either the Hessian, as in Newton’s method, or an approximation, as in a quasi-Newton method. In a truncated Newton method, instead of
solving for $p^{(k)}$, we get an approximate solution using only a few steps of an iterative linear equation solver, such as the conjugate gradient method (see Algorithm 5.2 on page 297). The conjugate gradient method is particularly suitable because it uses only matrix-vector products, so the matrix $R^{(k)}$ need not be stored. This can be very important in large-scale optimization problems that involve a large number of decision variables. How far to continue the iterations in the solution of the linear system is a major issue in tuning a truncated Newton method.

### 7.1.7 Fitting Models to Data Using Least Squares; Gauss-Newton Methods

One of the most important applications that involve minimization is the fitting of a model to data. In this problem, we have a function $f$ that relates one variable, say $y$, to other variables, say the $m$-vector $t$. The function involves some unknown parameters, say the $d$-vector $\theta$:

$$y = f(t; \theta). \quad (7.18)$$

The data consists of $n$ observations on the variables $y$ and $t$.

Fitting the model is usually done by minimizing some norm of the vector of residuals

$$r_i(\theta) = y_i - f(t_i; \theta). \quad (7.19)$$

The decision variables are the parameters $\theta$. The optimal values of $\theta$, often denoted as $\hat{\theta}$, are called “estimates”.

Because the data are observed and so are constants, the residuals are functions of $\theta$ only. The vector-valued function $r(\theta)$ maps $\mathbb{R}^d$ into $\mathbb{R}^n$.

The most common norm to minimize to obtain the fit is the $L_2$ or Euclidean norm. The scalar-valued objective function then is

$$s(\theta) = \sum_{i=1}^{n} (y_i - f(t_i; \theta))^2 = \sum_{i=1}^{n} (r_i(\theta))^2 = (r(\theta))^T r(\theta). \quad (7.20)$$

This problem is called least squares regression. If the function $f$ is nonlinear in $\theta$, the functions $r_i$ are also nonlinear in $\theta$, and the problem is called nonlinear least squares regression.

**“Modified” Gauss-Newton Method**

The gradient and the Hessian for a least squares problem have special structures that involve the Jacobian of the residuals, which is a vector function of the parameters. The gradient of $s(\theta)$ is
\[ \nabla s(\theta) = (J_r(\theta))^T r(\theta). \]

The Jacobian of \( r \) is also part of the Hessian of \( s \):

\[ H_s(\theta) = (J_r(\theta))^T J_r(\theta) + \sum_{i=1}^{n} r_i(\theta)H_{r_i}(\theta). \quad (7.21) \]

In this maze of notation the reader should pause to remember the shapes of these arrays, and their meanings in the context of fitting a model to data. Notice, in particular, that the dimension of the space of the optimization problem is \( d \), instead of \( m \) as in the previous problems. We purposely chose a different letter to represent the dimension so as to emphasize that the decision variables may have a different dimension from that of the independent (observable) variables. The space of an observation has dimension \( m + 1 \) (the \( m \) elements of \( t \), plus the response \( y \)); and the space of the observations as points \( y_i \) and corresponding model values \( f(t_i, \theta) \) has dimension \( n \).

- \( t_i \) is an \( m \)-vector. In the modeling context, these are the independent variables.
- \( y \) is an \( n \)-vector, and it together with the \( n t_i \) vectors are constants in the optimization problem. In the modeling context, these are observations.
- \( \theta \) is a \( d \)-vector. This is the vector of parameters.
- \( r(\cdot) \) is an \( n \)-vector. This is the vector of residuals.
- \( J_r(\cdot) \) is an \( n \times d \) matrix.
- \( H_{r_i}(\cdot) \) is a \( d \times d \) matrix.
- \( s(\cdot) \) is a scalar. This is the data-fitting criterion.
- \( \nabla s(\cdot) \) is a \( d \)-vector.
- \( H_s(\cdot) \) is a \( d \times d \) matrix.

In the vicinity of the solution \( \theta_* \), the residuals \( r_i(\theta) \) should be small, and \( H_s(\theta) \) may be approximated by neglecting the second term in equation (7.21). Using this approximation and the gradient descent equation, we have

\[ (J_r(\theta^{(k)}))^T J_r(\theta^{(k)}) p^{(k)} = -(J_r(\theta^{(k)}))^T r(\theta^{(k)}). \quad (7.22) \]

It is clear that the solution \( p^{(k)} \) is a descent direction; that is, if \( \nabla s(\theta^{(k)}) \neq 0 \),

\[ (p^{(k)})^T \nabla s(\theta^{(k)}) = -\left( (J_r(\theta^{(k)}))^T p^{(k)} \right)^T (J_r(\theta^{(k)}))^T p^{(k)} < 0. \]

The update step is determined by a line search in the direction of the solution of equation (7.22):

\[ x^{(k+1)} - x^{(k)} = \alpha^{(k)} p^{(k)}. \]

The search is usually required to satisfy the sufficient decrease condition (7.4) and the curvature condition (7.5).
This method is called the **Gauss-Newton algorithm**. Because many years ago (prior to Hartley, 1961), the step was often taken simply as \( p^{(k)} \), a method that uses a variable step length factor \( \alpha^{(k)} \) is sometimes called a “modified Gauss-Newton algorithm”. It is the only kind to use, so we just call it the “Gauss-Newton algorithm”.

In the case of a linear model, equation (7.18) becomes

\[
y = t^T \theta.
\]

The data, consisting of \( n \) observations on \( y \) and the \( m \)-vector \( t \), results in an \( n \)-vector of residuals,

\[
r = y - T \theta,
\]

where \( T \) is the \( n \times m \) matrix whose rows are the observed \( t^T \). The Gauss-Newton algorithm for this linear least squares problem yields the solution in one step, as in equation (5.93) on page 304.

If the residuals are small and if the Jacobian is nonsingular, the Gauss-Newton method behaves much like Newton’s method near the solution. The major advantage is that second derivatives are not computed.

If the residuals are not small or if \( J_r(\theta^{(k)}) \) is poorly conditioned, the Gauss-Newton method can perform very poorly. If \( J_r(\theta^{(k)}) \) is not of full rank, just as we do in the linear case, we could choose the solution corresponding to the Moore-Penrose inverse, which has the shortest Euclidean length:

\[
p^{(k)} = \left( (J_r(\theta^{(k)}))^T \right)^+ r(\theta^{(k)}).
\]

(Compare equation (5.98) on page 306.) If the matrix is nonsingular, the Moore-Penrose inverse is the usual inverse.

**Levenberg-Marquardt Method**

Another possibility, which is similar to what is done in linear ridge regression (see Exercise 6.2 on page 179 of NLA, 1998), is to add a conditioning matrix to \( (J_r(\theta^{(k)}))^T J_r(\theta^{(k)}) \) in equation (7.22). A simple choice is \( \tau I_d \), and the equation for the update becomes

\[
\left( (J_r(\theta^{(k)}))^T J_r(\theta^{(k)}) + \tau I_d \right) p^{(k)} = - (J_r(\theta^{(k)}))^T r(\theta^{(k)}).
\]

A better choice may be a scaling matrix, \( S^{(k)} \), that takes into account the variability in the columns of \( J_r(\theta^{(k)}) \); hence we have for the update

\[
\left( (J_r(\theta^{(k)}))^T J_r(\theta^{(k)}) + \lambda(\theta^{(k)}) S^{(k)} S^{(k)} \right) p^{(k)} = - (J_r(\theta^{(k)}))^T r(\theta^{(k))}.
\]

The basic requirement for the matrix \( (S^{(k)})^T S^{(k)} \) is that it improve the condition of the coefficient matrix. There are various way of choosing this matrix.
One is to transform the matrix \((J_r(\theta^{(k)}))^TJ_r(\theta^{(k)})\) so it has 1’s along the diagonal (this is equivalent to forming a correlation matrix from a variance-covariance matrix), and to use the scaling vector to form \(S^{(k)}\). The nonnegative factor \(\lambda^{(k)}\) can be chosen to control the extent of the adjustment. The sequence \(\lambda^{(k)}\) must go to 0 for the solution to converge.

Equation (7.24) can be thought of as a Lagrangian multiplier formulation of the constrained problem (see Section 7.3):

\[
\begin{align*}
\min_x & \frac{1}{2}\|J_r(\theta^{(k)})x + r(\theta^{(k)})\| \\
\text{s.t.} & \|S^{(k)}x\| \leq \delta_k,
\end{align*}
\]

The Lagrange multiplier \(\lambda^{(k)}\) is zero if \(p^{(k)}\) from equation (7.23) satisfies \(\|p^{(k)}\| \leq \delta_k\); otherwise it is chosen so that \(\|S^{(k)}p^{(k)}\| = \delta_k\).

Use of an adjustment such as in equation (7.24) is called the Levenberg-Marquardt algorithm. This is probably the most widely used method for nonlinear least squares. The method can be thought of as a trust region method, with \(\delta_k\) being the radius of the trust region, comparable to \(\tau^{(k)}\) in (7.13).

Just as in ridge regression (see Exercise 6.2b on page 179 of NLA, 1998), the computations for equation (7.24) can be performed efficiently by recognizing that the system is the normal equations for the least squares fit of

\[
\begin{pmatrix}
\mathbb{I} & J_r(\theta^{(k)}) \\
0 & \sqrt{\lambda^{(k)}} (S^{(k)})^{-1}
\end{pmatrix} p.
\]

**Variance-Covariance of the Parameter Estimators**

In fitting models to data, we always wish to know how dependent our estimated parameters are to the particular set of data used. A careful assessment of this dependence is usually predicated on formulation of the model (7.18) as

\[y = f(t; \theta) + E,\]

where \(E\) is a random variable with an assumed probability distribution. The estimates, \(\hat{\theta}\), are therefore functions of realizations of the random variable, and their variability that results from the variability in the data can be assessed by the variance-covariance matrix of the random variable of which \(\hat{\theta}\) is a realization. (We usually use slightly less precise terminology, and refer to the variance-covariance matrix of \(\hat{\theta}\).)

In simple cases where \(f\) is linear, \(E\) has a normal distribution with mean 0 and constant variance, \(\sigma^2\), and the observations are independent, the problem is particularly simple. In the more familiar notation of linear regression, \(y = X\beta\), where \(y\) is a vector of observations, and \(X\) is a matrix of the corresponding observations, the estimates are the solution to \(X^TX\beta = X^Ty\). Furthermore, the variance-covariance matrix for \(\hat{\beta}\) is \((X^TX)^{-1}\sigma^2\). A good estimate of \(\sigma^2\),
7.1 Unconstrained Descent Methods in Dense Domains

\[ \hat{\sigma}^2, \text{ is } (y - X\hat{\beta})^T (y - X\hat{\beta}) / (n - m), \] where \( n \) is the number of observations and \( m \) is the length of \( \beta \).

In problems of fitting models to data, we often make the assumption that the residuals are independently and identically distributed as normals. Even with this assumption, however, it may not be possible to write a simple expression for the variance-covariance matrix of \( \hat{\theta} \). Using a linear approximation that follows from the linear regression model described above, we may approximate the variance-covariance matrix as

\[
\left( (J_r(\hat{\theta}))^T J_r(\hat{\theta}) \right)^{-1} \hat{\sigma}^2,
\]

from the analogue, \( X^TX\beta = X^Ty \), of equation (7.22). The estimate of \( \sigma^2 \) is taken as the sum of the squared residuals, divided by \( n - m \), where \( m \) is the number of estimated elements in \( \theta \).

From equation (7.21), if the residuals are small, the Hessian is approximately equal to the cross-product of the Jacobian, and so an alternate expression for the variance-covariance matrix is

\[
(\mathbf{H}_r(\hat{\theta}))^{-1} \hat{\sigma}^2.
\]

This later expression would be more useful if Newton’s method or a quasi-Newton method is used in the solution of the least squares problem.

### 7.1.8 Iteratively Reweighted Least Squares

Often in applications, the residuals in equation (7.19) are not given equal weight in fitting the model. This may be because the reliability or precision of the observations on \( y \) and \( t \) may be different. For weighted least squares, instead of (7.20) we have the objective function

\[
s_w(\theta) = \sum_{i=1}^{n} w_i (r_i(\theta))^2. \tag{7.25}
\]

The weights add no complexity to the problem, and the Gauss-Newton methods of the previous section apply immediately, with

\[ \hat{r}(\theta) = Wr(\theta), \]

where \( W \) is a diagonal matrix containing the weights.

The simplicity of the computations for weighted least squares suggests a more general usage of the method. Suppose for fitting the model (7.18) we choose to minimize some other \( L_p \) norm of the residuals \( r_i \) in (7.19). The objective function then is
iteratively reweighted least squares
IRLS (iteratively reweighted least squares)
conjugate gradient method

\[ s_p(\theta) = \sum_{i=1}^{n} |y_i - f(t_i; \theta)|^p \]
\[ = \sum_{i=1}^{n} \left( \frac{1}{|y_i - f(t_i; \theta)|^{2-p}} \right) |y_i - f(t_i; \theta)|^2 \]

This leads to an iteration on the least squares solutions. Beginning with \( y_i - f(t_i; \theta^{(1)}) = 1 \), we form the recursion that results from the approximation

\[ s_p(\theta^{(k+1)}) \approx \sum_{i=1}^{n} \left( \frac{1}{|y_i - f(t_i; \theta^{(k)})|^{2-p}} \right) |y_i - f(t_i; \theta^{(k+1)})|^2. \]

Hence, we solve a weighted least squares problem, and then form a new weighted least squares problem using the residuals from the previous problem. This method is called *iteratively reweighted least squares* or IRLS. The iterations over the residuals are outside the loops of iterations to solve the least squares problems, so in nonlinear least squares, IRLS results in nested iterations.

There are some problems with the use of reciprocals of powers of residuals as weights. The most obvious problem arises from very small residuals. This is usually handled by use of a fixed large number as the weight.

Iteratively reweighted least squares can also be applied to other norms,

\[ s_p(\theta) = \sum_{i=1}^{n} \rho(y_i - f(t_i; \theta)), \]

but the approximations for the updates are not as good. Green (1984) and Street, Carroll, and Ruppert (1988) discuss IRLS methods for more general norms. Heiberger and Becker (1992) address some of the software development issues in using IRLS in regression programs.

For the overdetermined linear systems, such as we discuss in Section 5.2.7, beginning on page 303, the iterations for IRLS are performed on solutions of linear least squares problems that can be written in closed form. We do not need to use nested iterations.

7.1.9 Conjugate Gradient Methods

A quadratic function is well-behaved for the problem of finding its optimum. For this reason, as we have seen, many optimization methods are developed in the context of a quadratic function,

\[ f(x) = \frac{1}{2} x^T A x + x^T b + c, \]

in which \( A \) is positive definite. The methods also often work for other types of functions for which optima exist, because the quadratic function is a good
7.1 Unconstrained Descent Methods in Dense Domains

local model of the other functions. Functions with singularities or with extreme variability in the neighborhood of the optimum generally present difficult optimization problems.

Using the quadratic function above, we can describe another method closely related to quasi-Newton methods. The updates, as usual, are

\[ x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)}, \]

hence, at the \( k \)th step we have the linear combination,

\[ x^{(k)} = x^{(0)} + \alpha^1 p^1 + \cdots + \alpha^{(k-1)} p^{(k-1)}. \]

In the conjugate gradient method, these steps are chosen so that

\[ (p^{(k)})^T A p^{(i)} = 0, \quad \text{for } i = 1, \ldots, k; \]

that is, \( p^{(k)} \) is “A conjugate” to \( p^{(1)}, p^{(2)}, \ldots p^{(k-1)} \). Thus, in the case of the quadratic objective function, the steps are orthogonal to each other with respect to the Hessian. This orthogonality of the directions makes the steps more efficient.

At each iteration, we determine the optimal step length by a line search.

On page 297, we formulated the problem of solving the linear system \( Ax = b \) as a problem of finding the minimum of the function

\[ f(x) = \frac{1}{2} x^T A x - x^T b. \]

conjugate gradient method shown in Algorithm 5.2 to solve it.

7.1.10 Derivative-Free Methods

In the previous discussions we have generally assumed that the objective function is differentiable. If the function is differentiable, and the derivatives are available, methods that use the gradient are generally the most efficient ones; although depending on the cost of evaluation of derivatives, more efficient algorithms may avoid evaluation of the derivatives at every iteration. If the function is differentiable but the derivatives are not available, numerical derivatives or other approximations to the gradient should generally be used.

For continuous functions that are not differentiable or whose derivatives are difficult to compute or to approximate, we need derivative-free methods. Also in the case of noisy functions that cannot be evaluated exactly, methods that do not directly use derivatives may be better.

Nelder-Mead Simplex Method

The Nelder-Mead simplex method (Nelder and Mead, 1965) is a derivative-free, direct search method. The steps are chosen so as to ensure a local descent,
but neither the gradient nor an approximation to it is used. In this method, to find the minimum of a function, \( f \), of \( m \) variables, a set of \( m + 1 \) extreme points (a simplex) is chosen to start with, and iterations proceed by replacing the point that has the largest value of the function with a point that has a smaller value. This yields a new simplex and the procedure continues. The method is shown in Algorithm 7.2.

**Algorithm 7.2 Nelder-Mead Simplex Method**

0. Set tuning factors: reflection coefficient, \( \alpha > 0 \); expansion factor, \( \gamma > 1 \); contraction factor, \( 0 < \beta < 1 \); and shrinkage factor, \( 0 < \delta < 1 \).

Choose an initial simplex, that is, \( m + 1 \) extreme points (points on the vertices of a convex hull).

1. Evaluate \( f \) at each point in the current simplex, obtaining the values \( f_1 \leq f_2 \leq \cdots \leq f_m \leq f_{m+1} \).

Label the points correspondingly, that is, let \( x_{m+1} \) correspond to \( f_{m+1} \), and so on.

2. Reflect the worst point: let \( x_r = (1 + \alpha)x_a - \alpha x_{m+1} \), where \( x_a = \frac{1}{m} \sum_{i=1}^{m} x_i \), and let \( f_r = f(x_r) \).

3. If \( f_1 \leq f_r \leq f_m \), accept reflection: replace \( x_{m+1} \) by \( x_r \), and go to step 6.

4. If \( f_r < f_1 \), compute expansion: \( x_e = \gamma x_r + (1 - \gamma)x_a \).

   4.a. accept expansion: replace \( x_{m+1} \) by \( x_a \); otherwise,

   4.b. replace \( x_{m+1} \) by \( x_r \).

Go to step 6.

5. If \( f_m < f_r < f_{m+1} \), let \( f_h = f_r \); otherwise, let \( f_h = f_{m+1} \). Let \( x_h \) be the corresponding point. Compute contraction: \( x_c = \beta x_h + (1 - \beta)x_a \).

   5.a. accept contraction: replace \( x_{m+1} \) by \( x_c \); otherwise,

   5.b. shrink simplex: for \( i = 2, 3, \ldots, m + 1 \), replace \( x_i \) by \( \delta x_i + (1 - \delta)x_1 \).

6. If convergence has not occurred (see below) or if a preset limit on the number of iterations has not been exceeded, go to step 1; otherwise, return the solution as \( x_1 \).

There are three common ways of assessing convergence of the Nelder-Mead algorithm. All three, or variations of them, may be used together.

- The amount of variation in the function values at the simplex points. This is measured by the sample variance,

\[
s_f^2 = \frac{1}{m + 1} \sum (f_i - \bar{f})^2,
\]

where \( \bar{f} \) is the average function value at the simplex points.
where $\bar{f}$ is the sample mean of $f_1, f_2, \ldots, f_{m+1}$. Convergence is declared if $s_f^2 < \epsilon$. This stopping criterion can lead to premature convergence, just because the simplex points happen to lie close to the same level curve of the function.

- The total of the norms of the differences in the points in the new simplex and those in the previous simplex. (In any iteration except shrinkage, there is only one point that is replaced.) This is one of several stopping criteria proposed by Parkinson and Hutchinson (1972).
- The size of the simplex. Dennis and Woods (1987) suggested measuring this by

$$\frac{\max \|x_i - x_1\|}{\max(1, \|x_1\|)}$$

and terminating when this measure is sufficiently small.

Figure 7.2 illustrates one iteration of the algorithm in a two-dimensional problem. In two dimensions, the iterations are those of a triangle tumbling downhill vertex over edge and deforming itself as it goes.

Figure 7.2. One Nelder-Mead Iteration. In this step, “x2” becomes “x3”; “x1” becomes “x2”, and “xr” becomes “x1”. gro325

Although the Nelder-Mead algorithm may be slow to converge, it is a very useful method for several reasons. The computations in any iteration of the algorithm are not extensive. No derivatives are needed; in fact, not even the function values themselves are needed, only their relative values. The method
is therefore well-suited to noisy functions; that is functions that cannot be evaluated exactly.

There have been many suggestions for improving the Nelder-Mead method. Most have concentrated on the stopping criteria or the tuning parameters. The various tuning parameters allow considerable flexibility, but there are no good general guidelines for their selection.

Barton and Ivey (1996) describe modifications of the Nelder-Mead algorithm for stochastic functions; that is, noisy functions or functions whose values have an additive random component. Their modifications include reevaluating the stochastic functions at the points in the simplex considered for replacement.

It is a simple matter to introduce randomness in the decisions made at various points in the Nelder-Mead algorithm. As we discuss in Section 7.4.1, this may be useful for finding the global optimum of a function with many local optima. If some decisions are made randomly, however, the convergence criteria must be modified to reflect the fact that the iterations may no longer be strictly descending.

**Price Controlled Random Search Method**

Price (1977) proposed a method called controlled random search, in which a simplex is chosen randomly from a fixed set of points, and a random point in the simplex is reflected to obtain a new candidate point. In this method, to find the minimum of a function, $f$, of $m$ variables, first a random set of $n$ points is chosen, where $n$ is an arbitrary number greater than $m$ (Khuri, 1993, has suggested $n = 10m$, and Krývý and Tvrdič, 1995, recommended $\max(10, m^2)$). The function is evaluated at each of the points, and the best point in the set $x^{(0)}_*$ (yielding $f^{(0)}_*$) is determined. In the $k^{th}$ iteration, from the set of $n$ points, $m$ are chosen randomly and their centroid, $x_a = \frac{\sum_{i=1}^{m} x_i}{m}$, is chosen. Another point $x_{m+1}$ is chosen randomly from the remaining set of $n - m$ points, and is reflected through the centroid, to obtain $x_r$; that is, $x_r = (1 + \alpha)x_a - \alpha x_{m+1}$ (Price chose $\alpha = 1$). If $f(x_r) < f^{(k-1)}_*$, then $f^{(k)}_*$ is updated and $x_*$ is replaced by $x_r$; otherwise, $x_r$ is discarded. The iterations are continued until a stopping criterion is satisfied.

Aside from a convergence criterion, there are only two parameters to be chosen in the controlled random search method, the number of fixed points to retain, and the reflection parameter. The number of points to chose, $n$, obviously should increase as the number of variables, $m$, increases. The optimal relationship increases faster than a polynomial, so while a linear increase may work well in low dimensions, in higher dimensions, larger sets of fixed points must be maintained. The larger the value of $n$, the less likely that the iterations will become stuck in a local minimum. One of the advantages of the controlled random search method is that it is likely to find a global minimum even if the function has multiple local minima.
The choice of the reflection parameter, $\alpha$, depends on the smoothness of the function. It can be chosen larger for smoother functions. Tvrdík and Krývý (1995) suggested that $\alpha$ be chosen randomly, uniformly over $(0, 8)$, and that this choice worked well for most problems they considered. Krývý and Tvrdík (1995) suggested various other modifications to the basic controlled random search algorithm, including one that used ideas of genetic algorithms. Ali, Törm, and Viitanen (1997) incorporated local optimizing searches within the steps of the controlled random search algorithm.

Krývý, Tvrdík, and Krpec (2000) report numerical experiments in fitting 14 different nonlinear models using two of their modified controlled random search algorithms and the standard algorithms in four different statistical packages (NCSS, Systat, R, and SPSS). Surprisingly, they found the modified controlled random search methods to work better. The software packages used Gauss-Newton, Levenberg-Marquardt, or simplex methods.

One stopping criterion for the controlled random search method is a maximum scaled range of the function values of the points in the fixed set. As with any iterative method, of course, a limit on the number of iterations is also a stopping criterion.

One possibility for speeding up the controlled random search method include selection of the point to be reflected as the point out of the $m + 1$ points defining the simplex with the smallest function value. Another possibility is to select the $m + 1$ simplex points from the $n$ possible points with different probabilities, so as to favor the points with smaller function values. While both of these modifications may speed convergence in some cases, they do so at the risk of becoming stuck in a local minimum. For finding the global optimum of a function with many local optima, we could, in fact introduce more randomness in the procedure. It is a simple matter to introduce randomness in the decision of whether or not to accept the candidate point at each iteration. As we discuss in Section 7.4.1, more randomness in the optimization method increases the chances of finding a global optimum.

**Ralston-Jennrich Dud Method for Least Squares**

A secant method that often works well for least squares was proposed by Ralston and Jennrich (1978a). The method is a modification of the Gauss-Newton algorithm (see Section 7.1.7) that uses secant hyperplanes instead of the tangent hyperplanes defined by the gradient or approximations of the gradient. This secant method is called “dud” (doesn’t use derivatives).

The least squares problem is

$$
\min_{\theta} s(\theta) = \sum_{i=1}^{n} r_i^2(\theta). \tag{7.27}
$$

This problem usually arises as a natural criterion for fitting a model to data by selecting an optimal value for the parameter vector $\theta$:
\[ r_i(\theta) = y_i - f(t_i; \theta) \]

where \( y_i \) and the \( m \)-vector \( t_i \) are known (observations) and \( \theta \) is a \( d \)-vector to be determined (see page 391).

The dud method approximates \( f(t_i; \theta) \) by a secant hyperplane defined by \( d + 1 \) points on the surface of \( f \). To prepare for the \( (k + 1) \)th iteration, given \( d + 1 \) values of \( \theta \), of which any \( p \) are linearly independent, label them in such a way that \( \theta_{p+1}^{(k)} \) yields the smallest value for \( r(\theta) \) in equation (7.27). Then, with the \( p + 1 \) given points

\[ \theta_1^{(k)}, \theta_2^{(k)}, \ldots, \theta_{p+1}^{(k)} \]

and the corresponding values of \( s(\theta) \),

\[ s_1^{(k)}, s_2^{(k)}, \ldots, s_{p+1}^{(k)} \]

we express the hyperplane as

\[
\begin{align*}
    h(\alpha) &= \sum_{j=1}^{d} \alpha_j f(t_i; \theta_j^{(k)}) + \left(1 - \sum_{j=1}^{d} \alpha_j\right)f(t_i; \theta_{d+1}^{(k)}), \\
    (7.28)
\end{align*}
\]

and likewise the variables as

\[
\begin{align*}
    \theta &= \sum_{j=1}^{d} \alpha_j \theta_j^{(k)} + \left(1 - \sum_{j=1}^{d} \alpha_j\right)\theta_{d+1}^{(k)}. \\
    (7.29)
\end{align*}
\]

Then, in the \( (k + 1) \)th iteration, the dud method

1. finds the value \( \alpha_* \) that minimizes the distance between \( h(\alpha) \) and \( y \);
2. finds a new point \( \theta_N \) from equation (7.29); and
3. replaces one of the \( \theta_j^{(k)} \) with \( \theta_N \).

There are several possible ways of proceeding, and various possibilities for certain computational details. The computations can be defined easily as a linear least squares problem by first expressing equation (7.28) as

\[
h(\alpha) = f(t_i; \theta_{d+1}^{(k)}) + F^{(k)} \alpha
\]

where \( F^{(k)} \) is the \( n \times d \) matrix with columns

\[
\begin{pmatrix}
    f(t_i; \theta_j^{(k)}) - f(t_i; \theta_{d+1}^{(k)})
\end{pmatrix},
\]

where \( f(t_i; \cdot) \) is the vector with elements \( f(t_i; \cdot) \). We likewise express equation (7.29) as

\[
\begin{align*}
    \theta &= \theta_{d+1}^{(k)} + G^{(k)} \theta, \\
    \text{where } G^{(k)} \text{ is the } d \times d \text{ matrix with columns}
\end{align*}
\]
The minimizer of the function

$$(y - h(\alpha))^T(y - h(\alpha))$$

is of the familiar form,

$$\alpha_* = \left( (F^{(k)})^TF^{(k)} \right)^{-1}(F^{(k)})^T(y - f(t, \theta^{(k)}_{d+1})),$$

and the new point is

$$\theta_N = \theta^{(k)}_{d+1} + G^{(k)}\alpha_*.$$

As usual in this kind of iteration, we may encounter a nearly singular $F^{(k)}$. In that case, $\alpha_*$ can be chosen as any least squares solution. Again we remark that the expression for $\alpha_*$ above does not imply a computational method; we certainly do not invert the matrix in equation (7.30).

The choice of the point to be replaced by $\theta_N$ requires some care so as to ensure that the points are systematically replaced. Another important consideration is how the initial starting points are to be selected. The simplest, and probably the most common, method is just to evaluate the function $r$ over a grid of values and choose $d+1$ points from the grid that have generally smaller values, but which also provide some reasonable coverage of the grid. One of the main problems of course is the existence of local minima. If often better to restart the algorithm with a different set of $d + 1$ points than to attempt to provide such a good starting set that one convergence of the algorithm would be expected to find a global minimum. Inspection of the function values within planes of the $d$-dimensional grid can be helpful in identifying likely points of local minima. Various computational considerations and suggestions are discussed by Ralston and Jennrich (1978b).

Powell (1965) described an efficient method for updating in a nonlinear least squares algorithm that used numerical approximations to derivatives. That method could also be used in dud, but it would result in only minimal improvement in most cases, especially if the number of observations $n$ is large and if it is expensive to evaluate $f$.

This algorithm lends itself to weighted and iteratively reweighted least squares, as described in Section 7.1.8. This allows its use in a data fitting problem even when the criterion of fit is the minimum of some other norm of the residual vector.

### 7.1.11 Stochastic Search Methods

The Robbins-Munro stochastic approximation (see pages 369 and 374), can also be applied to optimization problems. In the update equation (6.9),

$$x_*^{(k+1)} = x_*^{(k)} + \alpha^{(k)}y_k,$$
the random element $y_k$ is the negative gradient or an approximation to it. Kiefer and Wolfowitz (1952) used stochastic approximation with finite differences for a simple regression problem, and so methods like this are called often Kiefer-Wolfowitz procedures. Chin (1993) reviews some of the methods for general optimization problems.

Spall (1992) describes a method called simultaneous perturbation stochastic approximation (SPSA) that differs from the Kiefer-Wolfowitz procedure by using only two evaluations of the objective function to approximate the gradient. The method is given in Algorithm 7.3.

**Algorithm 7.3 Simultaneous Perturbation Stochastic Approximation (SPSA)**

0. Set $k = 0$, and choose an initial point, $x^{(k)}$.

1. Generate a random vector $p^{(k)}$, whose components are independent and from a suitable distribution with a mean of 0.

2. Compute the simultaneous perturbation approximation $\hat{g}^{(k)}$ to the gradient, $g(x^{(k)}) = \nabla f(x^{(k)})$:

$$\hat{g}^{(k)} = \left( \frac{f(x^{(k)}+c^{(k)}p^{(k)}) - f(x^{(k)}-c^{(k)}p^{(k)})}{2c^{(k)}p^{(k)}_1}, \ldots, \frac{f(x^{(k)}+c^{(k)}p^{(k)}) - f(x^{(k)}-c^{(k)}p^{(k)})}{2c^{(k)}p^{(k)}_m} \right)$$

3. Update:

$$x^{(k+1)}_{*} = x^{(k)}_{*} - \alpha^{(k)}\hat{g}^{(k)}.$$ 

4. If $|x^{(k+1)}_{*} - x^{(k)}_{*}| \leq \epsilon$, return the solution as $x^{(k)}_{*}$; otherwise, if $k < k_{\text{max}}$, set $k = k + 1$ and go to step 1; otherwise, issue message that 'algorithm did not converge in $k_{\text{max}}$ iterations'.

The efficiency in this method arises from the simplicity of the gradient approximation. The objective function is only evaluated at two points to compute the approximation.

Spall (1992) recommends that the sequence $\alpha^{(k)}$ and $c^{(k)}$ be chosen as

$$\alpha^{(k)} = \alpha_0(k + 1)^{-p_\alpha}$$

and

$$c^{(k)} = c_0(k + 1)^{-p_c}.$$
While \( \alpha_0 \) and \( c_0 \) may require some numerical experimentation for the given problem, based on general empirical results, Spall recommends \( p_\alpha = 0.602 \) and \( p_c = 0.101 \).

Spall (1992) recommends that the elements of \( p^{(k)} \) be chosen in step 1 from a symmetric Bernoulli distribution with mass points \((-1, 1)\). Other distributions also satisfy the requirements to guarantee convergence. The normal distribution and the uniform distribution, however, do not.

Spall and Cristion (1994) described some modifications to the simultaneous perturbation stochastic approximation method for use in situations in which the objective function is changing over time.

### 7.1.12 Summary of Continuous Descent Methods

Descent methods in dense domains are based on iterations consisting of two choices:

1. direction in which to step
2. how far to step

A gradient descent method chooses the direction \( p \) based on (7.1):

\[
R p = -\nabla f(x).
\]

In the major variants of gradient methods, the directions are chosen as follows.

- **steepest descent** (7.6), page 384,
  \[
p^{(k)} = -\nabla f(x^{(k)}).
\]

- **Newton’s method** (or Newton-Raphson) (7.9), page 385,
  \[
  H_f(x^{(k)}) p^{(k)} = -\nabla f(x^{(k)})
  \]

- **quasi-Newton methods** (7.1) and (7.14), page 388,
  \[
  B^{(k)} p^{(k)} = -\nabla f(x^{(k)})
  \]

- **Gauss-Newton methods** for least-squares problems (7.22), page 392,
  \[
  (J_r(\theta^{(k)}))^T J_r(\theta^{(k)}) p^{(k)} = -(J_r(\theta^{(k)}))^T r(\theta^{(k)}).
  \]

- **Levenberg-Marquardt modifications** of Gauss-Newton methods (7.24),
  \[
  \left( (J_r(\theta^{(k)}))^T J_r(\theta^{(k)}) + \lambda^{(k)} (S^{(k)})^T S^{(k)} \right) p^{(k)} = -(J_r(\theta^{(k)}))^T r(\theta^{(k)}).
  \]
The idea behind the Levenberg-Marquardt modification of the Gauss-Newton methods can also be applied in Newton’s method and in quasi-Newton methods. In applications, the adjustments are often taken as \( \lambda(k)I \); that is, the identity is used in place of the squared scaling matrix \( (S(k))^T S(k) \).

These methods may also use approximations to the gradient direction \( \nabla f(x^{(k)}) \). Other descent methods, such as SPSA (page 404) and dud (page 401), are based on explicit approximations to the gradient direction. The Nelder-Mead (page 397) and Price (page 400) methods use a simplex in a secant hyperplane to determine the direction of the step.

In each method, after a direction has been determined, the length of the step must be determined. The gradient methods usually employ some type of line search, as described in Section 7.1.2.

### 7.2 Unconstrained Combinatorial Optimization; Other Direct Search Methods


== must use stochastic methods – cannot explore the space because it’s too large.

If the objective function is differentiable and the derivatives are available, methods described in the previous section that make use of the gradient and Hessian or simple approximations to the gradient and Hessian are usually the most effective ones. Even if the derivatives are not available or do not exist everywhere for a continuous objective function, the methods that use approximations to gradients are usually best. If the objective function is not differentiable, however, or if it is very rough, some kind of direct search for the optimum may be necessary. In some cases the objective function is noisy, perhaps with an additive random error term that prevents exact evaluation. In these cases also it may not be effective to use gradient or approximate-gradient methods. The Nelder-Mead simplex method may work in these cases. Stochastic search methods, such as SPSA which uses a type of gradient approximation method, may also be effective. Other stochastic search methods such as described in this section in the context of a countable domain may be useful for rough or noisy functions.

Another important type of optimization problem are those in which the decision variables are discrete. The solution may be a configuration of a finite set of points, that is, a graph. In the traveling salesperson problem, for example, we seek a configuration of cities that provides a path with minimal total length that visits each point in a set. In the vehicle routing problem, a fleet of vehicles stationed at a depot must make deliveries to a set of cities and it is desired to route them so as to minimize the time required to make all the deliveries. In a resource scheduling problem, a set of machines or workers are to be assigned to a set of tasks, so as to minimize the time required to
complete all the tasks, of so as to minimize idle time of the resources. These kinds of problems are examples of combinatorial optimization.

Direct search methods move from point to point using only the values of the function; they do not use derivative information, or approximations to derivatives. In some methods new points are chosen randomly, and then the decision to move to a new point is based on the relative values of the function at the old and new points. A tree or other graph of points may help to organize the points to visit in the search.

Sometimes, based on points that have already been evaluated, sets of other points can be ruled out. In tree-based search methods, such fathoming or branch-and-bound techniques may greatly enhance the overall efficiency of the search. “Tabu” methods keep lists of points that are not likely to lead to an optimum.

There are several variations of direct searches. Some search methods use heuristics that mimic certain natural systems. The articles in the collection by Aarts and Lenstra (1997) describe several types of search algorithms and discuss various applications to which the methods have been applied.

In all direct search methods the new points are accepted or not based on the objective function values. Some search methods allow iterations that do not monotonically decrease the objective function values. These methods are especially useful when there are local minima. In these iterations, if the new point is better, then it is used for picking a point in the next iteration. If the new point is not better, there are three possible actions:

- discard the point and find another one to consider
- accept the new point anyway
- declare the search to have converged

### 7.2.1 Simulated Annealing

Simulated annealing is a method that simulates the thermodynamic process in which a metal is heated to its melting temperature and then is allowed to cool slowly so that its structure is frozen at the crystal configuration of lowest energy. In this process the atoms go through continuous rearrangements, moving toward a lower energy level as they gradually lose mobility due to the cooling. The rearrangements do not result in a monotonic decrease in energy, however. The density of energy levels at a given temperature ideally is exponential, the so-called Boltzmann distribution, with a mean proportional to the absolute temperature. (The constant of proportionality is called “Boltzmann’s constant”). This is analogous to a sequence of optimization iterations that occasionally go uphill. If the function has local minima, going uphill occasionally is desirable.

Metropolis et al. (1953) developed a stochastic relaxation technique that simulates the behavior of a system of particles approaching thermal equilibrium. (This is the same paper that described the Metropolis sampling algorithm.) The energy associated with a given configuration of particles is
compared to the energy of a different configuration. If the energy of the new configuration is lower than that of the previous one, the new configuration is immediately accepted. If the new configuration has a larger energy, it is accepted with a nonzero probability. This probability is larger for small increases than for large increases in the energy level. One of the main advantages of simulated annealing is that the process is allowed to move away from a local optimum.

Although the technique is heuristically related to the cooling of a metal, as in the application of Metropolis et al. (1953), it can be successfully applied to a broader range of problems. It can be used in any kind of optimization problem, but it is particularly useful in problems that involve configurations of a discrete set, such as a set of particles whose configuration can continuously change, or a set of cities in which the interest is an ordering for shortest distance of traversal. Kirkpatrick, Gelatt, and Vecchi (1983) discussed various applications, and the method became widely following the publication of that article. Collins, Eglese, and Golden (1988) provide an annotated bibliography for the development of the method as well as for a variety of problems in which it has found application.

The Basic Algorithm

In simulated annealing, a “temperature” parameter controls the probability of moving uphill; when the temperature is high, the probability of acceptance of any given point is high, and the process corresponds to a pure random walk. When the temperature is low, however, the probability of accepting any given point is low; and in fact, only downhill points are accepted. The behavior at low temperatures corresponds to a gradient search.

As the iterations proceed and the points move lower on the surface (it is hoped), the temperature is successively lowered. An “annealing schedule” determines how the temperature is adjusted.

In the description of simulated annealing in Algorithm 7.4, recognizing the common applications in combinatorial optimization, we refer to the argument of the objective function as a “state”, rather than as a “point”.

Algorithm 7.4 Simulated Annealing

0. Set \( k = 1 \) and initialize state \( s \).
1. Compute \( T(k) \).
2. Set \( i = 0 \) and \( j = 0 \).
3. Generate state \( r \) and compute \( \delta f = f(r) - f(s) \).
4. Based on \( \delta f \), decide whether to move from state \( s \) to state \( r \).
   - If \( \delta f \leq 0 \), accept;
   - otherwise, accept with a probability \( P(\delta f, T(k)) \).
   - If state \( r \) is accepted, set \( i = i + 1 \).
5. If \( i \) is equal to the limit for the number of successes at a given temperature, go to step 1.
6. Set \( j = j + 1 \). If \( j \) is less than the limit for the number of iterations at given temperature, go to step 3.
7. If \( i = 0 \),
   deliver \( s \) as the optimum; otherwise,
   if \( k < k_{\text{max}} \),
   set \( k = k + 1 \) and go to step 1;
   otherwise,
   issue message that
   ‘algorithm did not converge in \( k_{\text{max}} \) iterations’.

For optimization of a continuous function over a region, the state is a point in that region. A new state or point may be selected by choosing a radius \( r \) and point on the \( d \)-dimensional sphere of radius \( r \) centered at the previous point. For a continuous objective function, the movement in step 3 of Algorithm 7.4 may be a random direction to step in the domain of the objective function. In combinatorial optimization, the selection of a new state in step 3 may be a random rearrangement of a given configuration.

**Parameters of the Algorithm: The Probability Function**

There are a number of tuning parameters to choose in the simulated annealing algorithm. These include such relatively simple things as the number of repetitions or when to adjust the temperature. The probability of acceptance and the type of temperature adjustments present more complicated choices.

One approach is to assume that at a given temperature, \( T \), the states have a known probability density (or set of probabilities, if the set of states is countable), \( p_S(s, T) \), and then to define an acceptance probability to move from state \( s_k \) to \( s_{k+1} \) in terms of the relative change in the probability density from \( p_S(s_k, T) \) to \( p_S(s_{k+1}, T) \). In the original application of Metropolis et al., the objective function was the energy of a given configuration, and the probability of an energy change of \( \delta f \) at temperature \( T \) is proportional to \( \exp(-\delta f / T) \).

Even when there is no underlying probability model, the probability in step 4 of Algorithm 7.4 is often taken as

\[
P(\delta f, T(k)) = e^{-\delta f / T(k)},
\]

(7.31)

although a completely different form could be used. The exponential distribution models energy changes in ensembles of molecules, but otherwise it has no intrinsic relationship to a given optimization problem.

The probability can be tuned in the early stages of the computations so that some reasonable proportion of uphill steps are taken. In empirical studies of optimization of continuous functions, Bohachevsky, Johnson, and Stein (1986) found that early acceptance rates of 50% to 90% of uphill moves worked well. They suggest use of a factor that reduces the probability as
the state moves closer to the optimum. In some optimization problems, the value of the function at the optimum, \( f^* \), is known, and the problem is only to determine the location of the optimum. In such cases, they use a factor \((f - f^*)^p\) in the exponent. If the value \( f^* \) is not known but a reasonable estimate is available, they suggest use of the estimate. The estimate could be updated as the algorithm proceeds.

**Parameters of the Algorithm: The Cooling Schedule**

There are various ways the temperature can be updated in step 1.

The probability of the method converging to the global optimum depends on a slow decrease in the temperature. In practice, the temperature is generally decreased by some proportion of its current value:

\[
T(k + 1) = b(k)T(k).
\]  

(7.32)

We would like to decrease \(T\) as rapidly as possible, yet have a high probability of determining the global optimum. Geman and Geman (1984) showed that under the assumptions that the energy distribution is Gaussian and the acceptance probability is of the form (7.31), the probability of convergence goes to 1 if the temperature decreases as the inverse of the logarithm of the time, that is, if \(b(k) = (\log(k))^{-1}\) in equation (7.32). Under the assumption that the energy distribution is Cauchy, a similar argument allows \(b(k) = k^{-1}\), and a uniform distribution over bounded regions allows \(b(k) = \exp(-c_k k^{1/d})\), where \(c_k\) is some constant, and \(d\) is the number of dimensions (see Ingber, 1989).

A constant temperature is often used in simulated annealing for optimization of continuous functions. Alrefaei and Andradóttir (1999) also suggested use of a constant temperature for optimization of noisy functions. The additive and multiplicative adjustments, \(c(k)\) and \(b(k)\) are usually taken as constants, rather than varying with \(k\). Van Laarhoven and Aarts (1987), Collins, Eglese, and Golden (1988), and Hajek (1988) describe several other methods of updating the temperature.

For functions of many continuous variables, Siarry et al. (1997) suggest using the basic simulated annealing approach on a sequence of lower-dimensional spaces. This approach can reduce the total number of computations, and would be particularly useful when the cost of evaluation of the function is very high.

**Other Variations**

A method somewhat similar to simulated annealing was developed by Aluffi-Pentini, Parisi, and Zirilli (1988a, 1988b). Their method, which is designed for continuous optimization problems, searches along solution trajectories of stochastic differential equations that govern a diffusion process. The cooling is continuous. Their method also does well in moving away from local optima.
The differences in this method and the standard simulated annealing seem to depend more on values of tuning parameters than on any fundamental difference between the two methods.

In some cases it may desirable to exercise more control over the random walk that forms the basis of simulated annealing. For example, we may keep a list of “good” points, perhaps the $b$ best points found so far. After some iterations, we may return to one or more of the good states and begin the walk anew.

Gelfand and Mitter (1989) and Gutjahr and Pflug (1996) studied the performance of simulated annealing for optimization of noisy functions. They derived convergence properties that depend on the manner in which the temperature is decreased. Alrefaie and Andradóttir (1999) suggested simulated annealing algorithms for noisy functions that uses a constant temperature. One of their procedures uses the number of times a point is visited to estimate the optimal solution.

Simulated annealing is often used in conjunction with other optimization methods. Brooks and Morgan (1994) suggest using simulated annealing to determine starting points for other optimization methods, and Brooks (1995) provides a program that implements the simulated annealing selection of a number of starting points. Multiple starting points may allow the subsequent optimization method to identify several local optima.

When gradient information is available, even in a limited form, simulated annealing is generally not as efficient as other methods that use that information. The main advantages of simulated annealing include its simplicity, its ability to move away from local optima, and the wide range of problems to which it can be applied. Corana, Marchesi, Martin, and Ridella (1987) compared a version of simulated annealing with other methods, including Nelder-Mead, and found the simulated annealing method to be more robust but more expensive in terms of number of function evaluations.

Ingber (1989) suggests periodically “re-annealing”, by adjusting the temperature periodically, based on numerical derivatives computed during the previous iterations in the algorithm. When the exponential cooling schedule, $T(k + 1) = \exp(-c_k k^{1/d}) T(k)$, mentioned above is also used, he calls this “very fast re-annealing” or “adaptive simulated annealing”.

Simulated annealing proceeds as a random walk through the domain of the objective function. There are many opportunities for parallelizing such a process. The most obvious is starting multiple walks on separate processors. Aarts and Korst (1989) discuss various ways of performing simulated annealing on parallel processors.

Applications

Simulated annealing has been successfully used in a range of optimization problems, including probability density smoothing (Deutsch, 1996), classifi-
The Canonical Example: The Traveling Salesperson Problem

The traveling salesperson problem can serve as a prototype of the problems in which the simulated annealing method has had good success. In this problem, a state is an ordered list of points ("cities") and the objective function is the total distance between all the points in the order given (plus the return distance from the last point to the first point. One simple rearrangement of the list is the reversal of a sublist, that is, for example,

\[(1, 2, 3, 4, 5, 6, 7, 8, 9) \rightarrow (1, 6, 5, 4, 3, 2, 7, 8, 9).\]

Another simple rearrangement is the movement of a sublist to some other point in the list, for example,

\[(1, 2, 3, 4, 5, 6, 7, 8, \uparrow 9) \rightarrow (1, 7, 8, 2, 3, 4, 5, 6, 9)\]

(Both of these rearrangements are called “2-changes”, because in the graph defining the salesperson’s circuit, exactly two edges are replaced by two others. The circuit is a Hamilton closed path.)

7.2.2 Evolutionary Algorithms

There are many variations of methods that use evolutionary strategies. These methods are inspired by biological evolution, and often use terminology from biology. Genetic algorithms mimic the behavior of organisms in a competitive environment in which only the fittest and their offspring survive. Decision variables correspond to “genotypes” or “chromosomes”; a point or a state is represented by a string (usually bit strings); and new values of the decision variables are produced from existing points by “crossover” or “mutation”. The set of points at any stage constitute a “population”. The points that survive from one stage to another are those yielding lower values of the objective function.

The ideas of numerical optimization using processes similar to biological evolution are old (see Mühlenbein, 1997, for some prehistory), but the current algorithms derive from the work of Rechenberg (1973) and Holland (1975 and 1992). Back (1996) provides descriptions and discussions of various evolutionary algorithms.

Genetic Algorithms

In most iterations it is likely that the new population includes a higher proportion of fit organisms (points yielding better values of the objective function).
than the previous population, and that the best of the organisms is better than the best in the previous population.

The biological analogy of a single iteration of a genetic algorithm is represented in Figure 7.3.

![Figure 7.3. The Biological Analogy of One Iteration of a Genetic Algorithm](image)

Parametrizations

The first consideration in applying a genetic algorithm to an optimization problem is how to represent the problem in terms of the “chromosomes”, “populations”, and “fitness” of an evolutionary process. To use a genetic algorithm in the standard optimization problem

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

the values of the decision variables are represented in binary notation as substrings of a bit string of length $l$ that corresponds to a representation of a chromosome, and the fitness is a function $g$ that is monotonically related to $f$.

If $g$ is fitness and we are to minimize $f$, then logically $g$ would increase as $f$ decreases. We usually do not interpret the relationship so literally. For practical purposes, $g$ is often chosen so $g(x) = f(x)$, and improved fitness is interpreted as a decrease in $g$. If the decision variables are continuous, they are discretized as necessary to fit in a reasonable bit string. Discrete decision
variables may be allocated a number of bits sufficient to represent all of their possible values.

Consider the minimization of the function

$$f(x_1, x_2) = x_1 - 2x_1^2 + 3x_1x_2 - x_2^2.$$  

Obviously, for this problem we would not use a genetic algorithm or any other stochastic method, but we can use it to illustrate a parametrization of the problem. Suppose we represent values of both $x_1$ and $x_2$ in binary notation using strings of length 8 in which the fourth position from the left is the unit position (for example, $3.5 = 00111000$). An organism is represented by a pair of such bit strings. We choose the fitness, $g$, as $-f$. Some organisms and their fitness are shown in Table 7.1. We see that organisms $s_2$ and $s_3$ are most fit within this population. The corresponding chromosomes would be good candidates for propagation.

<table>
<thead>
<tr>
<th>Organism</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$g$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>00010000</td>
<td>00010000</td>
<td>1.0</td>
</tr>
<tr>
<td>$s_2$</td>
<td>00111000</td>
<td>00010000</td>
<td>11.5</td>
</tr>
<tr>
<td>$s_3$</td>
<td>00110000</td>
<td>00000000</td>
<td>15.0</td>
</tr>
<tr>
<td>$s_4$</td>
<td>00100000</td>
<td>00110000</td>
<td>-3.0</td>
</tr>
<tr>
<td>$s_5$</td>
<td>00100000</td>
<td>00100000</td>
<td>-4.0</td>
</tr>
<tr>
<td>$s_6$</td>
<td>01001000</td>
<td>01010000</td>
<td>-6.5</td>
</tr>
</tbody>
</table>

**Evolution Strategies**

Rechenberg formalized *evolution strategies*, two of which are called

- $(\mu + \lambda)$-ES
- $(\mu, \lambda)$-ES

In a $(\mu + \lambda)$-ES procedure, $\mu$ parents produce $\lambda$ offspring and the best $\mu$ of the parents and offspring survive to the next generation. In a $(\mu, \lambda)$-ES procedure, $\mu$ survivors are selected only from the offspring. The former method is the more commonly used evolution strategy, often with $\mu = \lambda = 1$. The latter method is more similar to the method of Holland, which is sometimes identified as the genetic algorithm. Terminology varies somewhat, and we will not attempt to sort it out here, but rather proceed to describe the basic genetic algorithm (also called the “canonical genetic algorithm”).
Evolution Method

Algorithm 7.5 provides an outline of a genetic algorithm. There are several decisions that must be made in order to apply the algorithm. The first, as mentioned above, is to decide how to represent the values of decision variables in terms of chromosomes, and to decide how to evaluate the objective function in terms of a chromosome. Then, an initial population must be chosen.

Algorithm 7.5 Genetic Algorithm

0. Determine a representation of the problem, and define an initial population, \( x_1, x_2, \ldots, x_n \), for \( n \) even.
1. Assign probabilities \( p_i \) to each item in the population and choose (with replacement) a probability sample of size \( n \). This is the reproducing population.
2. Randomly pair all items in the reproducing population.
   Form a new population of size \( n \) from the \( n/2 \) pairs in the reproducing population, using various mutation and recombination rules.
3. If convergence criteria are met, stop, and deliver \( s \) as the optimum. otherwise, go to step 1.

Mutation and Recombination Rules

There are several possibilities for producing a new generation of organisms from a given population. Some methods mimic sexual reproduction, that is, the combining of chromosomes from two organisms, and some methods are like asexual reproduction or mutation. A genetic algorithm may involve all of these methods, perhaps chosen randomly with fixed or varying probabilities.

Three simple methods are crossover, for combining two chromosomes, and inversion and mutation, for yielding a new chromosome from a single one. In crossover of two chromosomes each containing \( m \) bits, for a randomly selected \( j \) from 1 to \( l \), the first \( j \) bits are taken from the chromosome of the first organism and the last \( l - j \) bit are taken from the chromosome of the second organism. In inversion, for \( j \) and \( k \) randomly selected from 1 to \( l \), the bits between positions \( j \) and \( k \) are reversed, while all others remain the same. In mutation, a small number of bits are selected randomly and are changed, from 0 to 1 or from 1 to 0. The number of bits to change may be chosen randomly, perhaps from a Poisson distribution, truncated at \( l \). These operations are illustrated in Table 7.2.

In the example operations shown in Table 7.2, crossover occurs between the third and fourth bits; inversion occurs for the bits between (and including) the third and the sixth; and mutation occurs at the second and fourth bits.

As with simulated annealing, indeed, as with almost any optimization method, for a given problem, genetic algorithms may require a good deal of ad hoc tuning. Grefenstette (1986) has suggested general guidelines for selecting these control parameters. In the case of genetic algorithms, there are
7 Optimization and Nonlinear Systems

416  7 Optimization and Nonlinear Systems

Table 7.2. Reproduction Rules for a Genetic Algorithm

<table>
<thead>
<tr>
<th>Generation $k$</th>
<th>Generation $k + 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crossover</td>
<td></td>
</tr>
<tr>
<td>$s_1^{(k)}$</td>
<td>11001001</td>
</tr>
<tr>
<td></td>
<td>$s_2^{(k)}$</td>
</tr>
<tr>
<td></td>
<td>00111010</td>
</tr>
<tr>
<td></td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td></td>
<td>$s_1^{(k+1)}$</td>
</tr>
<tr>
<td></td>
<td>11011010</td>
</tr>
<tr>
<td>Inversion</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$s_1^{(k)}$</td>
</tr>
<tr>
<td></td>
<td>11101011</td>
</tr>
<tr>
<td></td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td></td>
<td>$s_2^{(k+1)}$</td>
</tr>
<tr>
<td></td>
<td>11010111</td>
</tr>
<tr>
<td>Mutation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$s_1^{(k)}$</td>
</tr>
<tr>
<td></td>
<td>11101011</td>
</tr>
<tr>
<td></td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td></td>
<td>$s_1^{(k+1)}$</td>
</tr>
<tr>
<td></td>
<td>10111011</td>
</tr>
</tbody>
</table>


Genetic algorithms can be implemented in parallel rather directly. Some of the issues in parallelizing genetic algorithms are discussed by Mühlenbein (1992).

An interesting application of genetic algorithms is in genetic programming, as developed by Koza (1992, 1994b) and Koza, Bennett, and Andre (1999). Genetic algorithms are used to develop a computer program, given only a description of the problem to be solved. Genetic programming proceeds to structure program elements consisting of simple computations and control structures such as loops and iterations into a complete program. The fitness is the proximity of the output of the program to the desired solution. See Section 3.4.4.

7.2.3 Other Combinatorial Search Methods

There are a number of other methods of combinatorial optimization. One general type of method are guided direct search methods, in which at each stage there is an attempt to use the history of the search to choose new directions to explore.

Tabu search simulates the human memory process in maintaining a list of recent steps. The list is called a tabu list. The purpose of the list is to prevent the search from backtracking. Before a potential step is selected the search procedures checks the tabu list to determine if it is in the recent path to this point. The tabu list can be implemented by penalizing the objective function.

The length of the tabu list determines how well the procedure works. A short list may result in some cycling because of the shorter memory. A long list, on the other hand, increases the computational burden. An “aspiration function” allows the tabu status of a potential step to be overridden if the
aspiration level is attained. This also could allow for cycling. The aspiration function can be implemented by rewarding the objective function. The reader is referred to Glover (1986, 1993) for extensive discussion of the tabu search method.

Artificial neural networks are another type of algorithm for decision making that is analogous to a biological process. Haykin (1994) provides a good general discussion of the methodology. Ripley (1996) discusses the application of neural nets to pattern recognition.

A number of other stochastic combinatorial search methods have been developed. Some of these methods derive from the stochastic approximations in the Robbins-Munro procedure (equation (6.9)). Kushner and Yin (1997) describe these algorithms. As we have mentioned, stochastic algorithms are particularly useful in noisy function optimization.

The book by Cook et al. (1997), covers many additional topics and methods in combinatorial optimization. Cook et al. emphasize the kernel algorithms that are utilized in optimization methods.

### 7.3 Optimization under Constraints

The general optimization problem for a scalar-valued function in $m$ variables with $r$ constraints is

$$
\min_{x} f(x) \quad \text{s.t. } g(x) \leq b,
$$

(7.33)

where $x$ is $m$-dimensional and $g(x) \leq b$ is a system of $r$ inequalities. We sometimes write separately any simple bounds as in problem (6.4) on page 357, but problem (7.33) can of course include bounds on individual variables. Problem (7.33) also can include equality constraints by expressing an equality as two inequalities.

A point satisfying the constraints is called a feasible point, and the set of all such points is called the feasible region. For a given point $x_j$, a constraint $g_i$ such that $g_i(x_j) = b_i$ is called an active constraint.

Any of the unconstrained optimization methods we have described can be modified to handle constraints by first insuring that the starting points satisfy the constraints and then explicitly incorporating checks at each iteration to insure that any new point also satisfies the contraints. If the new point does not satisfy the constraints, then some of the parameters of the algorithm may be adjusted and a new point generated (this is a possible approach in the Nelder-Mead simplex method, for example), or, in random methods such as the Price controlled random search method, the new point is simply discarded and a new point chosen. Although this is a straightforward procedure, it is unlikely to be very efficient computationally.
Unconstrained methods can be used efficiently if a sequence of unconstrained problems that converges to problem of interest can be defined. Although there may be problems with the objective function in regions that are not feasible, this method can often be very effective.

Another approach to solving constrained problems is to incorporate the constraints into the objective function. One way in which this is done is by use of supplementary variables, as discussed below. Another way is to define transformations of the variables so that the objective increases rapidly near constraint boundaries. See Box (1965) for discussion of this type of approach.

### 7.3.1 Constrained Optimization in Dense Domains

In a constrained optimization problem over a dense domain, the most important concerns are the shape of the feasible region and the smoothness of the objective function. The problem is much easier if the feasible region is convex, and fortunately most constrained real-world problems have convex feasible regions. The smoothness of the objective function is important, because if it is twice-differentiable, we may be able to use the known properties of derivatives at function optima to find those optima. Some methods of dealing with constraints incorporate the constraints into the objective function. For such a method the shape of the feasible region is important because the derivatives of the combined objective function depend on the functions defining the constraints.

#### Equality Constraints

We will first consider some simple problems. Equality constraints are generally much easier to handle than inequalities, and we generally write the constraints explicitly as equalities, rather than as a pair of inequalities in the form of problem (7.33):

\[
\begin{align*}
\min_x & \quad f(x) \\
\text{s.t.} & \quad g(x) = b.
\end{align*}
\]

An optimization problem with equality constraints can often be transformed into an equivalent unconstrained optimization problem.

For any feasible point, all equality constraints are active constraints.

An important form of equality constraints are linear constraints, \(Ax = b\), where \(A\) is an \(r \times m\) (with \(r \leq m\)) matrix of rank \(s\). With \(g(x) = Ax\), we have

\[
\begin{align*}
\min_x & \quad f(x) \\
\text{s.t.} & \quad Ax = b.
\end{align*}
\]
If the linear system is consistent (that is, rank([A|b]) = s; see page 246), the feasible set is nonnull. The rank of A must be less than m, or else the constraints completely determine the solution to the problem. If the rank of A is less than r, however, some rows of A and some elements of b could be combined into a smaller number of constraints. We will therefore assume A is of full row rank; that is, rank(A) = r.

If $x_c$ is any feasible point, that is, $Ax_c = b$, then any other feasible point can be represented as $x_c + p$, where $p$ is any vector in the null space of $A, N(A)$. The dimension of $N(A)$ is $m - r$, and its order is $m$. If $B$ is an $m \times m - r$ matrix whose columns form a basis for $N(A)$, all feasible points can be generated by $x_c + Bz$, where $z \in \mathbb{R}^{m-r}$.

Hence, we need only consider the restricted variables

$$x = x_c + Bz,$$

and the function

$$h(z) = f(x_c + Bz).$$

The argument of this function is a vector with only $m - r$ elements, instead of $m$ elements, as in the original function $f$. The unconstrained minimum of $h$, however, is the solution of the original constrained problem.

Now, if we assume differentiability, the gradient and Hessian of the reduced function can be expressed in terms of the the original function:

$$\nabla h(z) = B^T \nabla f(x_c + Bz) = B^T \nabla f(x),$$

and

$$H_h(z) = B^T H_f(x_c + Bz)B = B^T H_f(x)B.$$

The relationship of the properties of stationary points to the derivatives, as described in Section 6.1, are the conditions that determine a minimum of this reduced objective function; that is, $x_*$ is a minimum if and only if

- $B^T \nabla f(x_*) = 0$,
- $B^T H_f(x_*)B$ is positive definite, and
- $Ax_* = b$.

These relationships then provide the basis for the solution of the optimization problem. This simple constrained optimization problem could be solved using the same methods as discussed in Section 7.1.

Because the $m \times m$ matrix $[B|A^T]$ spans $\mathbb{R}^m$, we can represent the vector $\nabla f(x_*)$ as a linear combination of the columns of $B$ and $A^T$, that is,

$$\nabla f(x_*) = Bz_* + A^T \lambda_*,$$
where \( z^* \) is an \((m-r)\)-vector and \( \lambda^* \) is an \(r\)-vector. Because \( \nabla h(z^*) = 0 \), \( Bz^* \) must also vanish, and we have
\[
\nabla f(x^*) = A^T \lambda^*
\]
\[
= J_g(x^*)^T \lambda^*.
\]
(7.35)

Thus, at the optimum, the gradient of the objective function is a linear combination of the columns of the Jacobian of the constraints. The elements of the linear combination vector \( \lambda^* \) are called Lagrange multipliers.

The condition expressed in (7.35) implies that the objective function cannot be reduced any further without violating the constraints. We can also see this in another simple example with equality constraints. In this example the objective function is linear, and the single equality constraint is quadratic:

\[
\min_x f(x) = 2x_1 + x_2
\]
\[
s.t. g(x) = x_1^2 - x_2 = 1.
\]

The optimum is \( x^* = (-1, 0) \). The gradient of \( f(x) \) is \( \nabla f(x) = (2, 1) \), that of \( g(x) \) is \( \nabla g(x) = (2x_1, -1) \), and \( \nabla g(x^*) = (-2, -1) \). As we see in Figure 7.4 at the optimum,
\[
\nabla f(x^*) = -\nabla g(x^*)
\]
\[
= -J_g(x^*)^T.
\]

The Lagrangian Function

The relationship between the gradient of the objective function and the Jacobian of the constraint function, motivates the definition of the Lagrangian function:
\[
L(x, \lambda) = f(x) + \lambda^T (g(x) - b),
\]
(7.36)
where \( \lambda \) is an \(m\)-vector, the elements of which are called Lagrange multipliers.

The derivatives of the Lagrangian function can be analyzed in a manner similar to the analysis of the derivatives of the objective function in Section 6.1 to determine necessary and sufficiency conditions for a minimum subject to equality constraints.

General Constrained Optimization over Dense Domains

Inequality constraints present significant challenges in optimization problems. The extent of the difficulty depends on the type of the constraint. The simplest constraints are “box constraints”, or simple bounds on the variables. Next are linear constraints of the form \( l \leq Ax \leq u \). Finally, general nonlinear constraints are the most complicated.
As in other cases of optimization over dense domains, we will usually assume that the objective function is twice differentiable in all variables. We will only indicate some of the general approaches, and refer the interested reader to other literature such as Nash and Sofer (1996) or Nocedal and Wright (1999) for more extensive discussions.

When there are both equality and inequality constraints, it is more convenient for the discussion to write the equality constraints explicitly as equalities, rather than as a pair of inequalities in the form of problem (7.33):

\[
\begin{align*}
\min_x f(x) \\
\text{s.t. } &g_1(x) = b_1, \\
&g_2(x) \leq b_2,
\end{align*}
\]

For any feasible point all equality constraints are active, while the any of the inequality constraints \(g_2(x) \leq b_2\) may or may not be active.

The following well-known theorem is proved in Nocedal and Wright (1999).

Let \(L(x, \lambda)\) be the Lagrangian and let \(x_*\) be a solution to problem (7.37). If the gradients of the active constraints at \(x_*\), \(\nabla g_2^{(a)}(x_*)\), are linearly independent, then there exists \(\lambda_*\) such that

\[
\nabla_x L(x_*, \lambda_*) = 0,
\]

and for all active constraints, \(g_2^{(a)}\) with corresponding \(\lambda^{(a)}\),
Karush-Kuhn-Tucker conditions
sequential unconstrained minimization techniques
SUMT quadratic programming

These necessary conditions are called the Karush-Kuhn-Tucker conditions, or just Kuhn-Tucker conditions. The Karush-Kuhn-Tucker conditions allow identification of potential solutions. These conditions, together with sufficient conditions involving second derivatives of $L(x, \lambda)$, form the basis for a variety of algorithms for constrained optimization of differentiable functions. The reader is referred to Nocedal and Wright (1999) for details.

As mentioned previously, another approach to solving constrained problems is to formulate a sequence of simpler problems that converges to problem of interest. Fiacco and McCormick (1968) described a method of formulating a sequence of unconstrained problems that converges to the given constrained problem, called the sequential unconstrained minimization technique (SUMT). See Nash (1998). for further discussions of the method. A possible problem arises in this approach if the behavior of the objective function is different outside the feasible region from its behavior when the constraints are satisfied.

**Quadratic Objective Function with Linear Inequality Constraints**

A common form of the general constrained optimization problem (7.33) has a quadratic objective function and linear inequality constraints:

$$
\min_x c^T x + x^T H x \\
\text{s.t. } Ax \leq b.
$$

(7.38)

This is called a quadratic programming problem. If $H$ is positive semidefinite, the problem is particularly simple, and there are efficient methods for solving a quadratic programming problem that make use of the fact that if $x_*$ is a solution, then there exists $\lambda_*$ such that

$$
2H x_* + A^T \lambda_* = c^T.
$$

(7.39)

Goldfarb and Idnani (1983) described an algorithm that uses this approach, and some software packages that solve quadratic programming problems require the user to formulate the problem in that form (see, for example, Schrage, 1997).

Quadratic programming has been used extensively in portfolio analysis following the work of Markowitz (1952). The optimization problem is defined in terms of the recent rates of growth and the covariances of those rates of growth for a set of assets under consideration for inclusion in the portfolio. See Exercise 7.12.
A number of algorithms based on sequential quadratic programming problems are used for more general constrained optimization problems. As in the unconstrained sequences, the violations of the constraints are built into the objective functions of later stages. Schittkowski (1985) gave a program NLPQL that implements sequential quadratic programming. Fan, Sarkar, and Lasdon (1988) developed a sequential algorithm called successive quadratic programming that is somewhat more robust.

As mentioned above, a disadvantage of a formulation of a sequence of approximate problems is that the problems generally do not maintain feasibility of the solution to the original problem. In some cases the objective function may not even be defined outside of the feasible region. Panier and Tits (1993) described a sequential approach to quadratic programming problems whose solutions are feasible. The method is called feasible sequential quadratic programming. They gave a program called FSQP that implements the method.

---

**7.3.2 Constrained Combinatorial Optimization**

Constraints in combinatorial optimization problems are usually handled by restricting the mechanism that generates new points to generate only feasible points.

**The Simplex Method in Linear Programming**

The basic linear program, which is often written as

\[ \min_x z = c^T x \]

subject to

\[ x \geq 0 \]

\[ Ax \leq b, \]

is a problem over a dense domain. A solution to the problem, however, occurs at a vertex of the polytope formed by the constraints. Because this is a finite set, the solution can be determined by inspecting a finite number or possibilities. It is in this sense that the linear programming problem is similar to other combinatorial optimization problems.

The linear programming problem is generally most easily solved by a simplex method, which steps through the vertices efficiently. We will not describe the simplex method here, but rather refer the reader to texts on linear programming, for example, Dantzig (1963), Murtagh (1981), Chvátal (1983) or sequential quadratic programming NLPQL (software) successive quadratic programming SQP (software) FSQP (software) EM algorithm linear programming simplex algorithm, linear programming
Nash and Sofer (1996). The points labeled \( s^{(k)} \), \( s^{(k+1)} \), and so on in Figure 7.6 may represent the progress of a simplex algorithm along the extreme points of the feasible region toward the solution \( x^* \).

More efficient methods for very large-scale linear programs are based on interior-point methods such as developed by Karmarkar (1984) (see Gonzaga, 1992, or Nash and Sofer, 1996, for a description). An interior-point method may proceed along points such as those labeled \( i^{(k)} \), \( i^{(k+1)} \), and so on in Figure 7.6 until the algorithm appears to slow, and then move to a vertex at \( i^{(k+4)} \) and switch over to a simplex algorithm for the final iterations toward the solution \( x^* \). The interior-point method uses a barrier function to proceed through the dense interior of the feasible region. This approach treats the problem as one in combinatorial optimization only in the latter stages.

Linear programming is a good example of how a specialized algorithm can perform very differently for some variation of the underlying optimization problem.

Special formulations of the simplex method make very significant differences in the speed of the solution. The problem of fitting a linear regression under the criterion of least absolute values is a linear programming problem, but its solution is much more efficient when the simplex method is accelerated by taking into account its special structure, such as done by Barrodale and Roberts (1974). Arthanari and Dodge (1981) discuss this and other optimization problems in statistics that can be formulated in such a way that their special structure leads to more efficient mathematical programming problems.
An important variation of linear programming is integer programming, in which the decision variables are restricted to be integers. In mixed integer programming some variables are restricted to be integers and others are not.

**Network Optimization**

Variations of the basic linear programming problem also include the transportation problem, the traveling salesman problem, and other network problems. The methods for linear programming can be applied directly. Sometimes, however, it is more efficient to use one of the methods discussed for combinatorial optimization in Section 7.2.

**7.4 Multiple Extrema and Multiple Objectives**

In practical applications, an optimization problem can rarely be stated in absolute terms. A local optimum may be preferable to a global optimum, because of issues that may not even be apparent until the optima are identified. Likewise, in many applications, the constraints are not necessarily essential. After inspecting alternative near optimal solutions and solutions to an unconstrained problem that almost satisfy the constraints, the penalty for violating the constraints may not be as important as the gain in the optimal solution. Also, in most practical applications, the objective is not just some simple function; there are multiple objectives.
7.4.1 Multiple Extrema and Global Optimization

It is possible that the function has more than one extreme point, or local optimum. As in the case of solving a system of linear equations that we discussed earlier, a common way of addressing this problem is to use different starting points in the iterative solution process. Plots of the points evaluated in the iterations may also be useful. In general, there is no way of finding all of the extreme points or the global optimum with any assurance. In fact, by analyzing any given deterministic method for finding a global optimum, an objective function could be constructed with a global optimum (probably a spike) that will not be found.

The way to find a global optimum is to cause the optimization method to take different paths toward a point at which it will converge. In the absence of specific knowledge of the shape of the objective function, randomly diverting the course of the iterations is likely to be the best way of searching for the global optimum. There are three places in the algorithms at which randomization can be introduced:

- random selection of starting points
- random selection of subsequent points to be considered
- random acceptance of a point under consideration

For the deterministic descent methods discussed in Section 7.1 random selection of starting points is the obvious method to choose. The stochastic methods discussed in Section 7.1.11 and the controlled random search method of Section 7.1.10 use both random starting points and random steps. Most of the methods discussed in Section 7.2 use randomization in all three ways. It is also easy to add random acceptance to some of the other methods, for example, Nelder-Mead and controlled random search.


The difficulty of the problem obviously depends on the number and distribution of local optima. If the number of extrema in a given interval is known, and if the function is twice continuously differentiable in the interval, a “guided” bisection algorithm of Kavvadias and Vrahatis (1996) can be used to find all of them with certainty. We refer the interested reader to their paper for the details. The *Journal on Global Optimization* is devoted to research in this area.

For the case of the statistical application of maximum likelihood, Gan and Jiang (1999) describe a statistical test that a maximum of the likelihood function is the global optimum. Their test is based on the fact that, under
suitable regularity conditions, the log-likelihood $l$ with parameter $\theta$ satisfies the equation
\[ E \left( \frac{\partial^2 l}{\partial \theta^2} \right) + E \left( \frac{\partial l}{\partial \theta} \right)^2 = 0. \]

### 7.4.2 Optimization with Multiple Criteria

The objective function in a given application may actually be quite complicated. For example, in a statistical procedure based on least squares, the effect of a single outlier on the solution to the minimization problem may be unacceptable. The more appropriate objective function may be least squares for residuals that are small or moderate, and least squares of scaled residuals for the larger residuals. Many specific objective functions have been proposed to allow for differential weighting of the residuals or to use a different function of the residuals, rather than the square function. For robust statistics, formulation of an appropriate objective function is usually the primary issue.

In statistical procedures that attempt to achieve a minimum mean squared error (MSE), there are two things that are minimized: the square of the bias and the variance. The objective function is just the sum of these two quantities, so it is just a simple and natural generalization of the objective function in minimum variance unbiased estimation when the feasible space is extended beyond unbiased estimators.

A simple generalization of a single objective function is a set of objectives. Optimization procedures that explicitly recognize the existence of multiple objectives can then be developed.

Whereas a standard optimization problem usually has an objective of the form $\min \{f(x) = z\}$, the general multicriteria problem can be formulated as
\[
\begin{align*}
\min \{ f_1(x) = z_1 \} \\
\min \{ f_2(x) = z_2 \} \\
\vdots \\
\min \{ f_n(x) = z_n \}
\end{align*}
\]

s.t. $x \in S$.  

The vector of $z_i$’s in (7.41) is called the criterion vector. A criterion vector is nondominated if there does not exist another feasible criterion vector all of whose elements are less than the given vector. (The terms dominate and dominated are then defined by the common language semantics.) In most nontrivial multicriteria problems there exists a set of nondominated criterion vectors. Although no solution is “best”, for any solution that does not result in a nondominated criterion vector, there is a “better” solution. Techniques for multiple criteria optimization generally prescribe some systematic exploration of the set of nondominated criterion vectors.
Within the set of parameter vectors, the concept of dominance leads to that of efficiency. A point \( x^* \in S \) is efficient if and only if there does not exist another feasible point yielding a criterion vector that dominates the criterion vector associated with \( x^* \). Other terms synonymous with efficiency are Pareto optimality and admissibility.

The most common way of addressing the problem of optimizing with respect to more than one criterion is to form a weighted sum of the objective functions, and then to proceed as in a standard problem in mathematical programming. There are also other ways, such as the reference point method, for solving this problem. Steuer (1986) describes these methods, and also discusses the practical problems of using an approach that effectively weights the criteria a priori. The problems arise because we usually do not have an explicit utility function. Even if a reasonable a priori formulation of a single objective were possible, it is generally desirable to explore the space of tradeoffs within the feasible region that contains near-optimal points. Human intervention is almost always involved in multiple criteria optimization.

Steuer (1986) discusses interactive procedures for multiple criteria optimization. Some of the procedures only work for linear objective functions, and others make implicit assumptions about the user’s utility function. The methods generally employ iterative projections of an unbounded line segment in the criterion space onto the nondominated surface of the feasible region (see also Korhonen and Wallenius, 1986). The available computer programs implementing this general method only work for linear problems. An important aspect of the methods is a graphical display that aids the user in interacting with the computations. This strategy is also applicable to nonlinear problems by replacing the linear programming module with a nonlinear code. The underlying computations for the nonlinear problem are more extensive, of course; and there may be a need to provide more than one nonlinear programming module. Any of the methods could be improved with more integrated graphics. For a nonlinear problem, the graphics to display the tradeoffs among the various criteria may be far more complicated.

Most of the work in multicriteria optimization has involved both linear objective functions and linear constraints. There has been some work in the area of multicriteria optimization for nonlinear problems (see, for example, the survey by Weistroffer and Narula, 1991, and the book by Miettinen, 1999). Any approach to multicriteria optimization involves solution of one or more ordinary optimization problems, and a variety of algorithms is available for solving the basic nonlinear optimization problems.

7.4.3 Optimization under Soft Constraints

Because of the standard formulation of optimization problems as a single “objective” function together with a set of “constraints”, practitioners generally set up their problems in this way. In many real-world applications, however, given a choice between a solution that satisfies all of the constraints and a
point that slightly violates some constraints but yields a much better value of the objective function, the practitioner would rethink the constraints and possibly accept the “nonfeasible” point. The way to approach most applications is to place a value or cost on results or decisions. The objective function does this; it represents the value of something. Hard constraints ignore gradations of value; they are either satisfied or they are not.

Often it is better to allow the constraints to be violated, but to construct the objective function so as to attempt to satisfy them. We thus treat the constraints as “soft”. For example, we may modify the constrained problem,

$$\min_x f(x)$$
$$\text{s.t. } g(x) \leq b.$$  

to the unconstrained problem,

$$f(x) + h(g(x) - b),$$

where $h$ is a function whose minimum occurs at all points $x$ such that $g(x) < b$. If $h$ is simple function with only two values, say 0 and $M$, and $M$ is a very large number relative to the range of $f$, the unconstrained problem is very similar to the constrained problem. We may choose $h$ to be an increasing function in $g(x) - b$ for $g(x) \geq b$. This will have the effect of insuring that the solution “nearly” satisfies the constraints.

Not all constraints can or should be treated as soft constraints. Some constraints represent physical limitations and must be satisfied.

### 7.5 Software for Optimization

Most of the comprehensive scientific software packages such as the IMSL Libraries, Matlab, and R have modules for solution of systems of nonlinear equations and for optimization.

It is possible for a user to access computational servers for optimization over the internet, so that the user client does not need to run the software. Czyzyk, Mesnier, and Moré (1998) describe a system called NEOS that provides server capability for optimization problems. Problems can be submitted to the NEOS system by email, or by interfaces available over the internet.

Casanova and Dongarra (1998) describe a system called Netsolve that uses the NEOS system with remote procedure calls. Ferris, Mesnier, and Moré (2000) describe a simpler interface using Condor (see Epema et al., 1996), together with NEOS.
It is difficult to design general-purpose software for optimization problems because the problems tend to be somewhat specialized and different solution methods are necessary for different problems. There are several specialized software packages for optimization. Some address general optimization problems for continuous nonlinear functions, with or without constraints. There are several packages for linear programming. These often also handle quadratic programming problems, as well as other variations, such as mixed integer problems and network problems.

Another reason it is difficult to design general-purpose software for optimization problems is because the formulation of the problems in simple computer interfaces is difficult.

Finally, the need for an initial guess may complicate the design of optimization software, especially for the unsophisticated user. The software would be hardpressed to decide on a reasonable starting value, however. Sometimes an obvious default such as \( x^{(0)} = 0 \) will work, and there are some software packages that will choose such a starting value if the user does not supply a value. Most packages require, or at least allow, the user to input a starting value.

Because the development of a mathematical model that can be communicated easily to the computer is an important, but difficult aspect of optimization problems, there are packages that implement modeling languages, and many of the general packages accept optimization problems expressed in these languages.

Moré and Wright (1993) provide a survey of software for nonlinear systems and optimization problems, both linear and nonlinear. Hans Mittelmann and Peter Spellucci maintain a guide to non-commercial optimization software

http://plato.la.asu.edu/guide.html

The magazine ORMS Today, published by Informs, periodically surveys software for optimization. A survey for nonlinear programming, for example, is in the June, 1998, issue, pages 36–45; and a survey for linear programming is in the August, 1999, issue, pages 64–71.

### 7.5.1 Fortran and C Libraries

There are a number of reliable optimization algorithms available for use in Fortran and C. Many of these programs are available in the ACM TOMS collection at netlib:

http://www.netlib.org/liblist.html

Two of the most widely used Fortran and C libraries are the IMSL Libraries and the Nag Library. They provide a large number of routines for optimization. Both libraries are available in both Fortran and C versions, and in both single and double precision.
The Optimization Subroutine Library (OSL) is an IBM product that provides a collection of tools for solving linear programming (LP), quadratic programming (QP), and mixed integer programming (MIP) problems. The MIP solver is capable of handling either a linear or quadratic objective function. Individual OSL components implement state-of-the-art algorithms in code that takes special advantage of the characteristics of the platforms on which they run. These components can be combined into applications as simple as “input, solve, output,” or as complicated as a knowledgeable practitioner may care to create.

Both serial and parallel versions are available. OSL subroutines are written primarily in portable FORTRAN, with a few assembler language routines to enhance performance. OSL includes routines for linear programming, quadratic programming, network problems, and mixed-integer programming. There are a number of utility routines for input/output, matrix manipulation, and control querying and setting. There are also routines for performing sensitivity and parametric analyses. Data may be input in any format, or generated as needed, and passed on to OSL modules in internal arrays.

OSL is available on numerous platforms, from PC’s to mainframes. It is available either as standalone solvers or as a library for developing custom applications.

There are a large number of other optimization software packages in either Fortran or C. One of the widely-used ones is GRG2 reduced gradient methods (Lasdon, Waren, Jain, and Ratner, 1978), which is distributed by Windward Technologies Inc. (1995) and Frontline Systems. There is also a version for large sparse problems LSGRG2 (Smith and Lasdon, 1992).

Conn, Gould, and Toint (1992) developed a package called LANCELOT for solving very large-scale nonlinear optimization problems.

Some widely-used constrained optimization programs based on sequential quadratic programming are NLPQL (Schittkowski, 1985), FSQP (Panier and Tits, 1993), and NPSOL (Gill et al., 1992). NLPQL does not maintain feasibility and so may not be as reliable, especially for objective functions that may not be well-behaved when the constraints are violated. NPSOL has an option to require it to maintain feasibility. FSQP ("Feasible Sequential Quadratic Programming") always maintains feasibility. NPSOL is available in the Nag Library as E04UCF. Lukšan and Vlček (2001) describe four different optimization subroutines that do not require derivatives. The programs use variations of sequential quadratic programming and quasi-Newton methods. All of these programs were written in Fortran. The subroutines of Lukšan and Vlček are available in the ACM TOMS collection at netlib.

Another widely-available package is NL2SOL that has been successively modified over the years (Dennis, Gay, and Welsch, 1981a, 1981b; Gay, 1983; Gay and Welsch, 1988; and Bunch, Gay, and Welsch, 1993).

Many of the Fortran and C subprogram libraries also have interactive interfaces that reduce the programming burden in using such libraries.
Examples of Use of the IMSL Libraries

The IMSL Libraries have eleven routines for unconstrained optimization and twelve routines for constrained optimization. The documentation provides a decision tree to identify the appropriate routine for a given problem. The first node in the tree is for unconstrained or constrained. Under the unconstrained branch, the next node is for univariate or multivariate. Under the unconstrained multivariate branch, there are two special branches for least squares problems and for very large-scale problems. For unconstrained problems not in these categories, the next choice is between smooth and nonsmooth objective functions. For smooth functions, the next choice is whether a first derivative is available, and if it is, the next choice is whether a second derivative is available.

The branch of the decision tree corresponding to constrained problems likewise has a variety of choices depending on type of objective function, type of constraints, derivative information, and smoothness of the function. For constrained optimization, the first question is whether or not the constraints are linear. For linear constraints, a further question is whether the constraints are simple box constraints, that is, bounds on the variables. Other factors determining the routine to use are the type of derivative information available, the size of the problem, and the smoothness of the objective function.

Consider the problem of determining the unconstrained minimum of the two-dimensional Rosenbrock function,

\[ f(x) = (1 - x_1)^2 + 100(x_2 - x_1^2)^2. \]

The IMSL Fortran routine UMING/DUMING uses a quasi-Newton method to solve an unconstrained problem of this type. The single precision routine is invoked by the statement

```fortran
    call uming (fcn, grad, n, xguess, xscale, fscale, iparam, rparam, x, fvalue)
```

To use a Fortran program to solve this problem, we first write either a Fortran subroutine or a function for the mathematical function and either a Fortran subroutine or a function for its gradient. These are EXTERNAL modules. The routine UMING/DUMING requires these functions to be passes as subroutines with specific forms:

- **FCN**, a user-supplied **SUBROUTINE** to evaluate the function to be minimized. The usage is
  ```fortran
  CALL FCN (N, X, F), where
  N is the length of X (input),
  X is the vector at which the function is to be evaluated (input, and not to be changed by FCN),
  F is the function value at the point X (output).
  ```

- **GRAD**, a user-supplied **SUBROUTINE** to evaluate the gradient at the point \( x \). The usage is
CALL GRAD (N, X, G), where
N is the length of X and of G (input),
X is the vector at which the function is to be evaluated (input, and not to be
changed by GRAD),
G is the gradient vector at the point X (output).

The Fortran modules for the Rosenbrock function and its gradient are
shown in Figure 7.7.

The other arguments in UMIN/DUMING are:
N, the dimension of the problem (input),
XGUESS, vector of length N containing the initial guess of the minimum (in-
put),
XSCALE, vector of length N containing the scaling factors for the variables
(input),
FSCALE, scaling factors for the function and gradient (input),
IPARAM, parameter vector of length 7 (input/output),
RPARAM, parameter vector of length 7 (input/output),
X, the point at which the minimum occurs (output),
FVALUE, the value of the function at the minimum (output).

A program to solve the minimization problem is shown in Figure 7.7. The
scales are set to 1 for both the variable and the function; and the default
values are used for the parameter vectors IPARAM and RPARAM.

The IMSL C function to solve this problem is min_uncon_multivar, which
is available in two precisions:

```
float *imsl_f_min_uncon_multivar
double *imsl_d_min_uncon_multivar.
```

There only two required arguments for *imsl_f_min_uncon_multivar:

```
float fcn (int n, float x[])
int n, the number of variables.
```

The same arguments as in the Fortran version are also available, but they all
have default values. If the gradient is not supplied, numerical approximations
are used. A C program to solve the minimization problem using the same
settings as in the Fortran program is shown in Figure 7.8. The final 0 in the
invocation of imsl_f_min_uncon_multivar is required to indicate the end of
the argument list.

Some of the most common constrained optimization problems are quadratic
programming problems of the form (7.38) on page 422. The IMSL Libraries
provide a quadratic programming routine, qprog (Fortran) or quadratic_prog
(C), that implements the method of Goldfarb and Idnani (1983).

The IMSL Libraries also provide utility routines for finite approximations
to the gradient, Hessian, and Jacobian. These routines, cdgrd, fdgrd, fdhes,
and fdjac are useful when building an optimization program, and they are
Fortran 77 program

```fortran
parameter (n=2)
integer iparam(7)
real f, fscale, rparam(7), x(n), xguess(n), xscale(n)
external rosbrk, rosgrd, uming

data xguess/-1.0,2.0/, xscale/1.0,1.0/, fscale/1.0/

iparam(1) = 0
call uming(rosbrk, rosgrd, n, xguess, xscale, fscale, iparam, &
            rparam, x, f)
print *, ' The solution is ', x, //,
            ' The function value is ', f, //,
            & ' The number of iterations was ', iparam(3),
            & ' The number of function evaluations was ', iparam(4),
            & ' The number of gradient evaluations was ', iparam(5)
end

The two-dimensional Rosenbrock function

subroutine rosbrk (n, x, f)
integer n
real x(n), f
f = (1.0 - x(1))**2 + 100.0 * (x(2) - x(1)*x(1))**2
return
end

The two-dimensional Rosenbrock function

subroutine rosgrd (n, x, f)
integer n
real x(n), g(n)
g(1) = -2. * (1.0 - x(1)) - 400.*(x(2)-x(1)*x(1))*x(1)
g(2) = 200.*((x(2)-x(1)*x(1))
return
end
```

Figure 7.7. IMSL Fortran Program to Find an Unconstrained Minimum

used internally in some of the IMSL optimization programs. It is generally better to provide program modules that actually compute the derivatives, rather than numerical approximations to them. Packages that do symbolic computations, such as Maple and Mathematica, can be used to determine mathematical expressions for the derivatives. Functions to compute the derivatives
7.5 Software for Optimization

/* C program */
#include <imsl.h>
#include <stdio.h>
main()
{ 
    int i, n = 2;
    float *result, fx;
    static float rosbrk(int, float[]);
    static void rosgrd(int, float[], float[]);
    static float xguess[2] = {-1.0e0, 2.0e0};
    static float grad_tol = 0.0001;

    result = imsl_f_min_uncon_multivar (rosbrk, n, IMSL_XGUESS, xguess,
                            IMSL_GRAD, rosgrd,
                            IMSL_GRAD_TOL, grad_tol,
                            IMSL_FVALUE, &fx, 0)
    printf (" The solution is ");
    for (i=0; i<n; i++) printf("%8.3f", result[i]);
    printf ("n The function value is %8.3f\n", fx);
} /* end of main */

/* The two-dimensional Rosenbrock function */
static float rosbrk (int n, float x[])
{ 
    return (1.0 - x[0])*(1.0 - x[0]) + 100.0 * (x[1] - x[0]*x[0])*(x[1] - x[0]*x[0]);
} /* end of function */

/* The gradient of the two-dimensional Rosenbrock function */
static void rosgrd (int n, float x[], float g[])
{ 
    g[0] = -2. * (1.0 - x[0]) - 400.*(x[1]-x[0]*x[0])*x[0];
    g[1] = 200.*(x[1]-x[0]*x[0]);
} /* end of function */

Figure 7.8. IMSL C Program to Find an Unconstrained Minimum

can also be written using software for automatic differentiation, as we discuss on page 438.

7.5.2 Optimization in General-Purpose Interactive Systems

General-purpose interactive systems such as Matlab, R, Gauss, and PV-Wave usually provide some functions for optimization. These are generally easier to use than the Fortran or C libraries, but the types of problems they solve
are often more limited, and there are fewer available options to control the computations.

An example of the use of the Matlab function to solve the same two-dimensional unconstrained Rosenbrock problem is shown in Figure 7.9.

M-File:

```matlab
function f = rosbrk(x)
    f = (1 - x(1))^2 + 100 * (x(2) - x(1)^2)^2;
```

Statements to find and print minimum:

```matlab
[x, out] = fmins('rosbrk', [-1,2]);
x
rosbrk(x)
```

**Figure 7.9.** Matlab Statements to Find an Unconstrained Minimum

In Matlab the function is defined in a Matlab M-file. The function `fmins` in Figure 7.9 uses a Nelder-Mead simplex method.

PV-Wave is a general-purpose interactive system that provides many of the capabilities of the IMSL Libraries. The C function `min_uncon_multivar` used in Figure 7.8, for example, is available in PV-Wave with a simpler interface.

Many other routines from Fortran and C Libraries are available with interactive interfaces. A graphical user interface is available for OSL on some systems. This provides some of the OSL functionality in a point and click environment in which no programming is required.

### 7.5.3 Software for General Classes of Optimization Problems

Because of the general complexity of optimization problems, special-purpose software has been developed for different types of problem. This reduces the complexity both of the user interface and of the computational algorithms. We will mention some software packages for various types of problems, but for a more extensive survey, we refer to Moré and Wright (1993).

**General Optimization Problems**

The difficulty in defining a user interface for optimization problems of a wide variety of types means that most of the software that addresses general optimization problems are Fortran or C libraries. The user interface requires specification of the problem in one of those languages. The IMSL Libraries, the Nag Library, and the Optimization Subroutine Library (OSL) discussed earlier are the most widely-used packages that provide capabilities for solving a wide range of types of optimization problems.
Most packages provide a choice of computational methods. Many include the Nelder-Mead simplex method as one of the choices because of the simplicity of its interface (no derivatives) and because of its robustness.

**Linear Programming and Quadratic Programming**

Although Fortran and C libraries often provide routines for linear programming, the special structure of the problems makes it easy to define simpler user interfaces than Fortran or C modules.

Well-developed software for linear programming has been available for a long time. An early very large-scale package produced by the IBM Corporation was called MPS. The format for specifying the problem and providing the data that this package required is called MPS format, and most software packages for linear programming allow for this format.

Currently a very popular format for linear programming packages is a spreadsheet format, which allows specification of the problem and input of the data in a spreadsheet that is compatible with the very popular spreadsheet programs such as Excel and Lotus.

The magazine *ORMS Today*, published by Informs, periodically provides surveys of linear programming packages. The August, 1999, issue describes several systems and gives contact information for the distributors of the packages.

Some of the more commonly used linear programming packages include OSL, particularly through its “non-programming” interface; Cplex; MINOS; Lindo and Lingo; and SAS. These linear programming software packages also solve quadratic programming problems.

The modeling systems GAMS and AMPL are also widely used for linear programming problems.

**Least Squares**

There is a wide range of software for least squares problems. Most of the general-purpose software includes special routines for least squares. Packages for statistical data analysis often include functions for nonlinear least squares. For example, in the IMSL Libraries the routine `rnlin` performs least squares fits of general models and in R the function `nls` performs the computations for nonlinear least squares regression. A more general function in R, `ms`, minimizes a sum of nonlinear functions over parameters.

Many of the packages for linear programming, such as OSL, also include abilities for quadratic programming. The linear objective function is replaced by a quadratic function and otherwise the interface for linear programming and quadratic programming are the same.

Bunch, Gay, and Welsch (1993) give Fortran subroutines for nonlinear least squares in nonlinear regression models. These routines also perform maximum likelihood and robust fitting of nonlinear regression models.
Automatic Differentiation

For algorithms that require the derivative of a function, it is often convenient to use software that performs symbolic differentiation, such as Maple. Even for functions that appear relatively simple, it is a good idea to double check the derivative by performing the differentiation on the computer. The software for the symbolic differentiation will also produce the appropriate Fortran or C code for the derivative of the function.

Another approach is to use a software system that operates directly on the Fortran or C code for the function to produce corresponding code for the derivative of the function. Dobmann, Liepelt, and Schittkowski (1995) and Bischof et al. (1996) have developed systems that perform automatic differentiation on functions written in Fortran. Griewank, Juedes, and Utke (1996) have developed a package for automatic differentiation of functions written in C and C++.

The systems of Dobmann, Liepelt, and Schittkowski (1995) and Griewank, Juedes, and Utke (1996) are available in CALGO (see page 651).

The ADIFOR system of Bischof et al. (1996) accepts the user’s Fortran 77 source code for the function and a specification of dependent and independent variables. ADIFOR then generates code that computes the partial derivatives of all of the specified dependent variables with respect to all of the specified independent variables. ADIFOR is available at http://www.mcs.anl.gov/adifor/

Bischof, Roh, and Mauer (1997) also produced a version of ADIFOR for C, called ADIC.

Griewank (2000) discusses various techniques of automatic differentiation, including methods for sparse problems, higher derivatives, and nonsmooth problems. He also discusses software for automatic differentiation.

7.5.4 Modeling Languages and Data Formats

Many computational problems in science and statistics can be stated in a very straightforward manner. For software for these problems, there are simple inputs that are just scalars or dense arrays, and the output is likewise simple, a few scalars or arrays. Optimization problems by their very nature are somewhat more difficult to set up for input to computer software; the input consists of functions and gradients, relationships, and initial guesses. Often we want more than just a point solution; we want to know something about the progress toward the solution and we want to know the sensitivity of the problem to other values in the neighborhood of the solution.

In many applications in which optimization problems arise, the structure of the problem is fixed, and values required to define the problem are sparse in input that consists of potentially very large arrays or other data structures. Adoption of a standard format in which to specify the problem can greatly facilitate the input of data. Likewise, a standard format in which to describe the
solution helps the analyst to understand the results, and perhaps to try other scenarios. The IBM package for linear programming that dates to the 1960’s, called MPS, defined a standard format for specifying a linear programming problem. This MPS format is still widely used and most software packages for linear programming allow for this format. The format is very efficient for large-scale problems.

Currently a very popular format for linear programming problem is a spreadsheet format, which allows specification of the problem and input of the data in a spreadsheet. Many linear programming software packages such as OSL have input/output abilities for both MPS and spreadsheet formats.

GAMS (The Scientific Press, 1988) and AMPL (Fourer, Gay, and Kernighan, 1993) are systems built on modeling languages. The GAMS language, which is somewhat similar to Fortran in appearance, provides concise algebraic statements that are readily comprehensible to persons with a mathematics background. AMPL provides an interactive command environment for defining optimization problems. Both of these packages provide for input and output of problem specifications in standard formats such as MPS. These packages are often used as front-end interfaces for other optimization packages. AMPL provides an easy interface for several optimization programs and nonlinear solvers, including Cplex, GRG2, LANCELOT, Minos, NPSOL, and OSL. An complete list, as well as additional information about AMPL, is available at [http://www.ampl.com/](http://www.ampl.com/)

### 7.5.5 Testbeds for Optimization Software

As we discuss in Section 3.4.5, there are two different kinds of issues involved in gaining confidence in the results of numerical computations. One type of question relates to the quality of the algorithm and the software. Analysis of the algorithm, review of the software, and finally empirical testing of the software can provide general answers to the question of whether our computations are correct. For a given problem, however, the real question is how good is the value we have computed. One approach is to perturb the problem in a way that has a known effect on the exact solution. Comparing the computed solution of the original problem to the computed solution of the perturbed problem can alert us to possible inaccuracies; or, conversely, it can give us confidence in the computed solutions. The simplest perturbation that sometimes works is to change the precision. This does not perturb the mathematical problem, so the exact solution does not change. (The exact solution to the problem that the software is actually given may change. Our interest, however, is not in the question of whether the software did well with the approximations in the input data. In the real world, we want real solutions.)

It is exceptionally difficult to verify the solution to an optimization problem. One way of gaining confidence in our computed solution is to use different starting values and see if we converge to the same solution. This approach is
also used to help to get a “better” optimum in the case of multiple optima. When we try different starting values and converge to different solutions, we are left with the uncomfortable feeling that the problem has many optima, and we have just visited some local optima.

Because of the difficulties of assessing the accuracy of computed results in a given optimization problem, it is important to have a wide variety of test datasets to use in validating optimization software. The development of such testbeds began many years ago. Hoffman et al. (1953) described several test problems, and gave their solutions. More recently, Bongartz et al. (1995) describe an extensive testbed for optimization called CUTE. Many of the test sets are available from netlib (see page 651 for information on netlib).

Floudas et al. (1999) present a collection of test problems gathered from a wide range of applications. They provide input files either in the GAMS format or the MINOPT format.

A very simple problem that is often used to test optimization software is the Rosenbrock function:

\[
f(x) = \sum_{j=1,3,...,d-1} ((1 - x_j)^2 + 100(x_{j+1} - x_j^2)^2),
\]

for an even integer \(d\). This function is one generalization of the original Rosenbrock function that has two variables (see Rosenbrock, 1969). The function has a banana-shaped valley, with a minimum at \((1, 1, \ldots, 1)\). A slightly different generalization of Rosenbrock function is one of five test functions sometimes called De Jong’s tests that are widely used in testing combinatorial algorithms, especially genetic and other evolutionary algorithms. (See Kenneth A. De Jong, 1976, Analysis of the behavior of a class of genetic adaptive systems, unpublished Ph.D. dissertation, University of Michigan, Ann Arbor.)

The other four tests in De Jong’s test suite include a simple circular quadratic,

\[
f(x) = \sum_{j=1}^d x_j^2;
\]

a step function,

\[
f(x) = 6d + \sum_{j=1}^d [x_j];
\]

a quartic with noise,

\[
f(x) = \sum_{j=1}^d (jx_j^4 + e_j),
\]

where \(e_j\) is a realization of a normal \((0,1)\) random variable; and a version of the “Shekel foxholes” function,

\[
f(x) = \frac{1}{500} + \sum_{j=1}^{25} \frac{1}{j + \sum_{i=1}^2 (x_i - a_{ij})^p},
\]
where \( p \) is an even integer, usually 2, and
\[
a_{1j} = -32, -16, 0, 16, 32, -32, -16, 0, 16, 32, \ldots
\]
and
\[
a_{2j} = -32, \ldots, -32, -16, \ldots, -16, 0, \ldots, 0, 16, \ldots, 16, 32, \ldots, 32.
\]

Kennedy and Gentle (1980, pages 493–498) list seventeen other test problems from the literature; and Hock and Schittkowski (1985) and Schittkowski (1987) provide extensive collections of test problems.

Test problems are useful for developers of algorithms, but because the developers have tested their software on standard collections of test problems, these test problems can give the user of the software a false sense of security. The test problems may or may not be representative of the user’s problem.

In addition to a fixed set of test problems, it is important to be able to generate problems with specific characteristics, so as to test the ability of the optimization software to solve problems with given features. Dembo and Steihaug (1985) describe a test problem generator for unconstrained optimization problems. Facchinei, Júdice, and Soares (1997a, 1997b) describe a method of generating test datasets with box constraints, beginning with unconstrained problems with known solutions.

**Exercises**

7.1. Consider the function
\[
f(x) = x_1^2 + 5x_2^2,
\]
whose minimum obviously is at \((0, 0)\).

a) Plot contours of \( f \). (You can do this easily in R or Matlab, for example.)

b) In the steepest descent method, determine the first 10 values of \( \alpha^{(k)} \), \( f(x^{(k)}) \), \( \nabla f(x^{(k)}) \), and \( x^{(k)} \), starting with \( x^{(0)} = (5, 1) \). For the step length, use the optimal value (equation (7.3), page 382).

c) Plot contours of the scaled quadratic model (7.12) of \( f \) at the point \((5, 1)\).

d) Repeat Exercise 7.1b using Newton’s method. (How many steps does it take?)

e) Repeat Exercise 7.1b using SPSA (Algorithm 7.3).

7.2. Now consider a modification of Exercise 7.1. Suppose the function and its derivatives are measured with a random Gaussian error. The function actually observed is
\[
f(x) = x_1^2 + 5x_2^2 + \epsilon,
\]
where \( \epsilon \) has a \( N(0, 0.01) \) distribution, that is a normal distribution with a mean of 0 and a standard deviation of 0.1. The minimum of the expected
value of the function is at \((0, 0)\). Also, any measurement of the derivative has a random additive \(\epsilon\) term.

a) In the steepest descent method, determine the first 10 values of \(\alpha^{(k)}, f(x^{(k)}), \nabla f(x^{(k)}),\) and \(x^{(k)}\), starting with \(x^{(0)} = (5, 1)\). For the step length, use the optimal value.

b) Repeat Exercise 7.2a using Newton’s method.

c) Repeat Exercise 7.2a using SPSA.

7.3. Show that the rank-one update of equation (7.16), page 389, results in a matrix \(B^{(k+1)}\) that satisfies the secant condition (7.14).

7.4. Derive an expression for \((B^{(k+1)})^{-1}\) in terms of \((B^{(k)})^{-1}\) when the rank-one update of equation (7.16), page 389, is used.

7.5. Formulate a quasi-Newton method with the rank-one update (equation (7.16), page 389) to find the minimum of the function in Exercise 7.1. Start with \(B^{(0)} = I\) and \(x^{(0)} = (5, 1)\).

7.6. Many natural phenomena, such as phosphorescence or radioactive emissions, decay exponentially over time. Suppose the following measurements (with appropriate units) of the variable \(y\) were made with a crude instrument at regular 5 second time intervals beginning at \(t_0 = 5\):

\[
1.71, 1.07, 0.62, 0.65, 0.17, 0.14, 0.08, 0.09
\]

a) Use least squares to fit the model

\[
y = \theta_1 \exp(\theta_2 t).
\]

i. Formulate the Gauss-Newton method for this problem, and show the first two steps, beginning with \(\theta = (1, 1)\).

ii. Now use a program for nonlinear least squares, such as the IMSL Fortran routine \texttt{rnlin}, the C routine \texttt{nonlinear regression}, or the R function \texttt{nls}, for example. Plot the data and your fitted model.

b) Transform the model by taking logs of both sides, and again fit the model with least squares. What is the difference in this fit and the one using the raw model and data?

c) Now use the same software that you used in Exercise 7.6a to fit the model using least absolute values. Plot the data and your fitted model. What is the difference in the fit using least absolute values and that using least squares? Which observation contributes most to the difference in the fits?

7.7. According to Maxwell-Boltzmann theory, the probability density of the velocity of a gas molecule is proportional to

\[
(m/(kT))^{(3/2)}e^{-(mv^2)/(2kT)}v^2,
\]

where \(v\) is the velocity, \(T\) is the absolute temperature, \(m\) is the molecular mass, and \(k\) is Boltzman’s constant.
Determine the mode of this distribution (the point where it achieves its maximum value – the “most likely” velocity). Your solution is called the rms velocity. (Make sure you choose the correct critical point for the maximum.)

7.8. The now classic application of simulated annealing is the traveling salesperson problem. The objective is to develop an ordered list of a set of cities, so that each city occurs at least once in the list, and that if the cities are visited in the order of the list, the total distance is minimized. The method, as indicated in the text, is to begin with an initial ordered list and to compute its total distance, to make a random change in the list and compute its distance, and then to accept the new list if its distance is less than the previous distance or else to accept the new list with a probability that is decreasing in the difference of the old distance and the new distance. Design and write a simulated annealing program for the traveling salesperson problem.

a) Use your program to determine the optimal order in which to visit the cities in the mileage chart below. Assume you return to the starting city.

<table>
<thead>
<tr>
<th></th>
<th>Alexandria</th>
<th>Blacksburg</th>
<th>Charlottesville</th>
<th>Culpeper</th>
<th>Fairfax</th>
<th>Front Royal</th>
<th>Lynchburg</th>
<th>Manassas</th>
<th>Richmond</th>
<th>Roanoke</th>
<th>Williamsburg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alexandria</td>
<td>↓</td>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blacksburg</td>
<td>263</td>
<td>↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Charlottesville</td>
<td>117</td>
<td>151</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Culpeper</td>
<td>70</td>
<td>193</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fairfax</td>
<td>15</td>
<td>249</td>
<td>102</td>
<td>55</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Front Royal</td>
<td>71</td>
<td>203</td>
<td>124</td>
<td>44</td>
<td>57</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lynchburg</td>
<td>178</td>
<td>94</td>
<td>66</td>
<td>108</td>
<td>163</td>
<td>157</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manassas</td>
<td>28</td>
<td>238</td>
<td>91</td>
<td>44</td>
<td>23</td>
<td>45</td>
<td>157</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Richmond</td>
<td>104</td>
<td>220</td>
<td>71</td>
<td>89</td>
<td>106</td>
<td>133</td>
<td>110</td>
<td>96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Roanoke</td>
<td>233</td>
<td>41</td>
<td>120</td>
<td>164</td>
<td>218</td>
<td>174</td>
<td>52</td>
<td>207</td>
<td>189</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Williamsburg</td>
<td>148</td>
<td>257</td>
<td>120</td>
<td>133</td>
<td>150</td>
<td>177</td>
<td>165</td>
<td>140</td>
<td>51</td>
<td>215</td>
<td></td>
</tr>
</tbody>
</table>

b) Determine an optimal order for beginning at Alexandria and ending at Williamsburg (not returning to Alexandria).

7.9. K-means clustering is a method of clustering observations into a preset number of groups $k$ in such a way as to minimize the total of the within sums-of-squares,

$$
\sum_{g=1}^{k} \sum_{j=1}^{m} \sum_{i=1}^{n_g} \left( x_{ij(g)} - \bar{x}_{j(g)} \right)^2,
$$

where $n_g$ is the number of observations in the $g^{th}$ group, $x_{ij(g)}$ is the $i^{th}$ observation on the $j^{th}$ variable in the $g^{th}$ group, and $\bar{x}_{j(g)}$ is the mean of the $j^{th}$ variable in the $g^{th}$ group. (See Section 14.2.1, page 617.) Write a simulated annealing program to perform K-means clustering; that is to minimize the objective function above. Use your program to form four clusters of the data.
\[
\begin{bmatrix}
  x_1 \\
  x_2 \\
  x_3 \\
  x_4 \\
  x_5 \\
  x_6 \\
  x_7 \\
  x_8 \\
  x_9 \\
  x_{10} \\
  x_{11} \\
  x_{12} \\
  x_{13} \\
  x_{14} \\
  x_{15} \\
  x_{16}
\end{bmatrix} =
\begin{bmatrix}
  1 & 1 \\
  1 & 2 \\
  1 & 3 \\
  1 & 4 \\
  2 & 1 \\
  2 & 2 \\
  2 & 3 \\
  2 & 4 \\
  3 & 1 \\
  3 & 2 \\
  3 & 3 \\
  3 & 4 \\
  4 & 1 \\
  4 & 2 \\
  4 & 3 \\
  4 & 4
\end{bmatrix}.
\]

See Zeger, Vaisey, and Gersho (1992) for discussion of a simulated annealing algorithm in this context.


7.11. Consider the quadratic programming problem,

\[
\begin{align*}
\min_x & \quad 3x^2 + 2y^2 + z^2 + 2xy - xz - 0.8yz \\
\text{s.t.} & \quad x + y + z = 1 \\
& \quad 1.3x + 1.2y + 1.08z \geq 1.12 \\
& \quad x \leq 0.75 \\
& \quad y \leq 0.75 \\
& \quad z \leq 0.75
\end{align*}
\]

(This is the form of a simple portfolio optimization problem. Because \(x\), \(y\), and \(z\) are not restricted to be nonnegative, short-selling is allowed.)

Put this problem in the form of equation (7.38), and identify all of the variables in the new formulation of the problem. (This would be the first step in solving a quadratic programming problem using some software packages.)

7.12. In the allocation of financial assets it is generally desirable to maximize expected returns (growth) and to minimize risk. A commonly accepted definition of risk is the variability in the rates of return. The expected return is generally measured by the sample mean of past rates of return. The variability is generally measured by a sample variance of past rates of return.
Suppose we have \( n \) possible assets to choose from. If the individual assets have average growth rates \( g_1, g_2, \ldots, g_n \), and we put a proportion \( p_i \) into the \( i^{th} \) asset, the overall growth rate of the portfolio of these assets is

\[
d(p) = \sum_{i=1}^{n} p_i g_i.
\]

The variance of the portfolio depends on the variances of the individual assets, and on their covariances (expressing how they tend to move in the same or opposite directions). Assume the individual assets have standard deviations of \( s_1, s_2, \ldots, s_n \), and the \( n \times n \) matrix \( R \) contains the correlations of the individual assets, measured over some time in the past that is considered to be replicated in the present and near-future.

Adopt this notation: \( g \) is the vector of the \( g_i \)'s; \( p \) is the vector of the \( p_i \)'s; \( s \) is the vector of the \( s_i \)'s; and \( S = \text{diag}(s) \) is the \( n \times n \) diagonal matrix containing the \( s_i \)'s along the diagonal.

The overall variance then is

\[
v(p) = p^T SRSp.
\]

For a set of possible investments, the “optimal” allocation is usually defined as the \( p \) vector that minimizes \( v \) for a given value of \( d \). Because the \( p_i \)'s represent proportions, we assume \( \sum p_i = 1 \).

For 8 assets, a sample of rates of return yielded

\[
g = (0.39, 0.88, 0.53, 0.88, 0.79, 0.71, 0.25, 0.27)
\]

\[
s = (5.50, 7.03, 6.22, 7.04, 6.01, 4.30, 2.01, 1.56)
\]

and

\[
R = \begin{bmatrix}
1.00 & 0.41 & 0.30 & 0.25 & 0.58 & 0.71 & 0.33 \\
0.41 & 1.00 & 0.62 & 0.42 & 0.54 & 0.44 & 0.26 \\
0.30 & 0.62 & 1.00 & 0.35 & 0.48 & 0.34 & 0.28 \\
0.25 & 0.42 & 0.35 & 1.00 & 0.40 & 0.22 & 0.16 \\
0.58 & 0.54 & 0.48 & 0.40 & 1.00 & 0.25 & 0.29 \\
0.71 & 0.44 & 0.34 & 0.22 & 0.56 & 1.00 & 0.42 \\
0.33 & 0.26 & 0.28 & 0.16 & 0.29 & 0.42 & 1.00 \\
0.26 & 0.22 & 0.14 & 0.25 & 0.36 & 0.25 & 0.92
\end{bmatrix}
\]


a) Under the definition of optimality and the given data, determine the optimal portfolio that has \( d = 0.7 \). You may wish to use the IMSL routine `qprog` or `quadratic_prog`. 
b) Now consider the quadratic objective function, \( v(p) - \lambda d(p) \), where \( \lambda \) is a constant. For various choices of \( \lambda \), determine the optimal portfolio, \( p^*_\lambda \) and the corresponding values of \( v^*_\lambda \) and \( d^*_\lambda \). Plot a parametric curve of \( d^*_\lambda \) versus \( v^*_\lambda \). This curve is called the “efficient frontier”.

c) The most obvious problem in the use of this procedure to select an “optimal” portfolio is the relevance of a portfolio that was optimal over some past period of time to future performance. Given some assumption of stationarity, however, there remain several issues about the quantities used in the problem formulation. The most basic of these issues concerns measurement methods. Again, given some assumptions of the appropriate ways to measure things such as past growth rate, however, additional problems arise because of the inherent randomness. From our most basic assumptions in the formulation of the problem, we must recognize that \( g, s, \) and \( R \) are realizations of random variables. Issues relating to the use of these realizations of random variables as parameters in an optimization problem have often been discussed in the financial literature, for example, Jobson and Korkie (1981) and Chopra and Ziemba (1993). Several possible ways of using simulation to deal with this problem come to mind. Resampling of the original data to generate new realizations of \( g, s, \) and \( R \) would probably be the most effective. A simpler way would be to assume that the realized value of \( g \) is in fact the mean of a random variable with variance-covariance matrix \( SRS \), and merely to simulate realizations of \( g \) from a multivariate normal distribution with this mean and covariance (see page 209). Use this approach and generate 1,000 realizations of \( g \). For each, repeat Exercise 7.12b. This yields 1,000 efficient frontier curves. Now determine the curve formed by the means \( d^*_\lambda \) and \( v^*_\lambda \), where the means are taken over the 1,000 values on the efficient frontiers. Compare this curve with the efficient frontier curve of Exercise 7.12b. Jobson and Korkie (1981) carried out a similar Monte Carlo study.

7.13. Consider the optimization problem in Exercise 7.11. Define an optimization problem in which it is desired to minimize the same objective function, but \( x, y, \) and \( z \) are required to be nonnegative; the upper bounds on \( x, y, \) and \( z \) are not hard constraints, but are “desirable”; and it is desired to maximize the linear combination \( 1.3x + 1.2y + 1.08z \).

7.14. Consider the optimization problem in Exercise 7.11. Define an optimization problem in which it is desired to minimize the same objective function, but \( x, y, \) and \( z \) are required to be nonnegative; the upper bounds on \( x, y, \) and \( z \) are not hard constraints, but are “desirable”; and it is desired to maximize the linear combination \( 1.3x + 1.2y + 1.08z \).
Quadrature and Evaluation of Special Functions

One of the most common mathematical operations in scientific computing application is quadrature, the evaluation of a definite integral. It is used to determine volume, mass, or total charge, for example. In the evaluation of probabilities, of expectations, and of marginal or conditional densities, integration is the basic operation.

A straightforward way of developing models of relationships among variables is to represent the change in some variables as a function of some other variables. This approach leads to differential equations, and these are the basic models for studying many natural processes.

Most of the integrals and differential equations of interest in real-world applications do not have closed-form solutions; hence, their solutions must be approximated numerically.

The quadrature approximation is generally based on a function approximation such as discussed in Sections 8.2, page 458, and 8.3, page 463. The approximation of a given function is formed by some combination of other functions. With a basis set \( \{ f_i \} \), the function \( f(x) \) in the space spanned by the basis set can be represented exactly as

\[
f(x) = \sum_{i=1}^{\infty} c_i f_i(x),
\]

and so a finite series expansion such as

\[
f(x) \approx \sum_{i=1}^{k} c_i f_i(x)
\]

may be used. When the \( c_i \) in equation (8.2) are the Fourier coefficients, that is, the coefficients from equation (8.1), the approximation in equation (8.2) is the least squares approximation of the given form (see Section 8.3).

The problem of approximating a definite integral is called numerical quadrature.
8.1 Numerical Quadrature

Although some of the more interesting problems are multivariate and the region of integration is not rectangular, we begin with the simple integral,

\[ I = \int_{a}^{b} f(x) \, dx. \]  

(8.3)

The definition of the integral (8.3) involves a limit of sums of products of values of the function at given arguments and the distance between successive argument values.

8.1.1 Newton-Cotes Quadrature

One way of approaching the problem of evaluating (8.3) is to approximate it directly by a sum of areas under the curve. The Riemannian definition of the integral leads to a set of rectangles, the sum of whose areas approximates the integral. More generally an approximation of the integral results from an approximation of \( f(x) \) by piecewise simpler functions that can be integrated in closed form. If the piecewise approximants are step functions, the approximation is similar to a Riemann sum.

The Trapezoid Rule

Instead of a simple step function, the function \( f(x) \) may be approximated as shown in Figure 8.1 by a piecewise linear function \( p_1(x) \) that agrees with \( f \) at each of the points \( a = x_0 < x_1 < x_2 < \ldots < x_n = b \). In that case, the integral (8.3) can be approximated by a sum of integrals,

\[ \int_{a}^{b} f(x) \, dx \approx \sum_{i=0}^{n} \int_{x_i}^{x_{i+1}} p_1(x) \, dx, \]

each of which is particularly easy to evaluate. Because \( p_1 \) is linear over each interval, the integral in the \( i^{th} \) interval is just the area of the trapezoid, that is,

\[ h(f(x_i) + f(x_{i+1}))/2. \]

The integral (8.3) is therefore approximated by

\[ h(f(a) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(b))/2. \]  

(8.4)

The expression (8.4) is called the trapezoid rule.

Figure 8.1 shows how the areas in the trapezoids may be used to approximate areas under the curve.

A simple choice for the points is to make them equally spaced, that is,

\[ (x_{i+1} - x_i) = (b - a)/n. \]
The number of points \( n \), or the width of the interval, say \( h \),

\[
h = (b - a)/n,
\]
is a tuning parameter of the quadrature algorithm. Subject to the rounding induced by working with floating point numbers instead of real numbers, the larger is \( n \), or the smaller is \( h \), the better the approximation of the finite sum to the integral. Because of the rounding, after a certain level of refinement, no further gains can be achieved by simply making the intervals smaller.

Many other quadrature rules can be built using this same idea of using an approximating function that agrees with \( f \) at each of some set of points \( a = x_0 < x_1 < x_2 < \ldots < x_n = b \). Quadrature formulas that result from this kind of approach are called Newton-Cotes formulas.

**Figure 8.1.** The Trapezoid Rule grq210

**Simpson’s Rules**

Rather than the linear functions of the trapezoid rule, a more accurate approximation would probably result from use of quadratic functions that agree with \( f \) at each of three successive points. If \( p_2(x) \) is a quadratic that agrees with \( f(x) \) at the equally-spaced points \( x_i, x_i + h, x_i + 2h \), then the piece of the integral (8.3) from \( x_i \) to \( x_{i+2} \) can be approximated by

\[
\int_{x_i}^{x_{i+2}} p_2(x) \, dx = \frac{1}{3} h \left( f(x_i) + 4f(x_i + h) + f(x_i + 2h) \right).
\]  (8.5)
If an even number of intervals is chosen, and the integrals like (8.5) are summed, the integral (8.3) is approximated by

\[ \frac{1}{3} h \left( f(a) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots + 4f(x_{n-1}) + f(b) \right). \tag{8.6} \]

The formula (8.6) is called Simpson’s \( \frac{1}{3} \) rule.

Figure 8.2. Simpson’s 1/3 Rule grq215

Figure 8.2 shows how the areas under the quadratics may be used to approximate areas under the curve. Comparison with Figure 8.1 indicates that approximation by a higher degree polynomial (quadratic instead of linear) does not always guarantee an improvement.

A better approximation can be constructed using the same idea, except instead of using three points at a time and fitting a quadratic, we use four points at a time, and fit a cubic function. For equal-spaced points, the analog to (8.5) is

\[ \int_{x_i}^{x_{i+3}} p_3(x) \, dx = \frac{3}{8} h \left( f(x_i) + 3f(x_i + h) + 3f(x_i + 2h) + f(x_i + 3h) \right), \]

and over the entire range of integration that has been divided into a multiple of three equal-length intervals, we have, analogously to (8.6),

\[ \frac{2}{8} h \left( f(a) + 3f(x_1) + 3f(x_2) + 2f(x_3) + 3f(x_4) + \cdots + 3f(x_{n-1}) + f(b) \right). \tag{8.7} \]

The formula (8.7) is called Simpson’s \( \frac{3}{8} \) rule.
8.1 Numerical Quadrature 451

**Error in Newton-Cotes Quadrature**

Using results of linear operators and functions from Section 8.2.1, page 459, ==

- Rice, 119, 211
  - local error; total error
- trapezoid, $O(h^2)$, if equal spaced ... Hoffman (6.27), p.189, (6.36), (6.45)
- Simpson’s $\frac{1}{3}$ rule, $O(h^4)$
- Simpson’s $\frac{3}{8}$ rule, $O(h^4)$, same

**Extrapolation in Quadrature Rules**

Richardson extrapolation
Romberg

8.1.2 Adaptive Quadrature Rules+

Self-validating Corliss and Rall (1987)

8.1.3 Gaussian Quadrature+

A finite sum of a form similar to the Riemann definition of (8.3),

$$
\sum_{i=0}^{n} w_i g(x_i),
$$

(8.8)

can be used to approximate the integral

$$
I = \int_{a}^{b} f(x) \, dx
= \int_{a}^{b} g(x) w(x) \, dx.
$$

If $f$ is a polynomial, it is obiously possible to represent the integral in the form (8.8). A method based on this formula in which the $x_i$’s and $w_i$’s are chosen so that the approximation is correct when $f$ is a polynomial, is called *Gaussian quadrature*.

Let $q_0, q_1, \ldots$ be a sequence of polynomials orthogonal to the weight $w$ over $(a,b)$, as described in Section 8.3.2 (page 465), with full sets of distinct real roots in $(a,b)$. If $g(x)$ is a polynomial of degree $2n - 1$ or less, and if we choose the $x_i$ in (8.8) as the distinct roots of $q_n(x)$, then the weights are given by

$$
w_i = -\frac{1}{c_n} \frac{c_{n+1}}{q_{n+1}(x_i) q_n'(x_i)},
$$

where $c_j$ is the coefficient of the term of degree $j$ in $q_j$.

*** weights are positive
Error in Gaussian Quadrature

*** error is if \( g \) is \( 2n \) times continuously differentiable,

\[
\int_a^b f(x) \, dx - \sum_{i=0}^{n} w_i g(x_i) = \int_a^b g(x)w(x) \, dx - \sum_{i=0}^{n} w_i g(x_i) = \frac{g^{(2n)}(x^*)}{(2n)!c_n^2},
\]

for some point \( x^* \) in \((a, b)\).

*** give example ***

8.1.4 Monte Carlo Methods+

In Section 2.1, pages 7 through 17, we discussed Monte Carlo methods for evaluating a definite integral. In this section we will elaborate on that discussion.

In the Monte Carlo method of quadrature we first formulate the integral to be evaluated as an expectation of a function of a random variable, then simulate realizations of the random variable, and take the average of the function evaluated at those realizations. This is analogous to a standard method of statistical estimation, in which we use a sample mean to estimate a parameter of the distribution of a random variable. In applications, the realizations of the random variable are pseudorandom numbers; nevertheless, our analysis relies on statistical estimation theory.

The deterministic methods of quadrature, such as Newton-Cotes and Gaussian, yield approximations; Monte Carlo quadrature yields estimates. Use of Monte Carlo methods for quadrature is sometimes called stochastic integration.

An advantage of Monte Carlo quadrature is that the nature of the domain of integration is not as critical as in the other quadrature methods we have discussed above. We consider an integral similar to (8.3), except with a more general domain of integration, \( D \). To estimate the integral using Monte Carlo, we first formulate the integral as

\[
I = \int_D f(x) \, dx = \int_D h(x)p_X(x) \, dx, \tag{8.9}
\]

where \( p_X \) is the probability density function of a random variable \( X \) with support on \( D \). This step may involve some familiarity with probability density functions. For one-dimensional interval domains, appropriate probability density functions are similar to the weight functions for orthogonal polynomials shown in Table 8.1, page 468, depending on whether neither, one, or both limits of integration are infinite.
If $D$ is the interval $(a, b)$, as in (8.3), $a$ and $b$ are finite, the trivial uniform density may always be used. The uniform density over $[a, b]$ is the constant $1/(b - a)$, so one possible formulation (8.9) is

$$I = (b - a) \int_a^b f(x) \frac{1}{b - a} \, dx.$$

As we will see below, however, this may not be a good choice because the performance of the Monte Carlo method degrades for $h(x)$ with large variation, so the Monte Carlo estimates will be better (in a sense to be defined below) if $h(x)$ is nearly constant. Also, if $a$ or $b$ is infinite, the uniform density cannot be used because of the $(b - a)$ factor.

The decomposition in (8.9) results in

$$I = \int_D h(x) p_X(x) \, dx = E(h(X)),$$

where $E(h(X))$ is the expectation (or “average” over the full distribution of $X$) of the function $h(X)$.

If $x_1, x_2, \ldots, x_m$ is a random sample (or pseudorandom sample) of the random variable $X$, the sample average,

$$\bar{h}(x_i) = \frac{1}{m} \sum_{i=1}^m h(x_i),$$

is an estimate of the integral, $E(h(X))$, or $I$. We often denote an estimate of $I$ as $\hat{I}$. In this case

$$\hat{I} = \frac{1}{m} \sum_{i=1}^m h(x_i). \quad (8.10)$$

If we formulate the estimator $\hat{I}$ as a sum of functions of independent random variables, each with density $p_X$, instead of a sum of realizations of random variables, the estimator itself is a random variable. An obviously desirable property of this random variable is that its expectation be equal to the quantity being estimated. Assuming the expectations exist, this is easily seen to be the case:

$$E(\hat{I}) = E\left(\frac{1}{m} \sum_{i=1}^m h(X_i)\right)$$

$$= \frac{1}{m} \sum_{i=1}^m E(h(X_i))$$

$$= \frac{1}{m} \sum_{i=1}^m I$$

$$= I.$$

We therefore say the estimator is unbiased.
Variance of Monte Carlo Estimators

Monte Carlo methods are sampling methods; therefore the estimates that result from Monte Carlo procedures have associated sampling errors. The fact that the estimate is not equal to its expected value (assuming the estimator is unbiased) is not an “error” or a “mistake”; it is just a result of the variance of the random (or pseudorandom) data. The sampling errors mean that we get different estimates of the integral if we evaluate it on different occasions.

The variance of the estimator \( \hat{I} \) is a rather complicated function involving the original integral (assuming the integrals exist):

\[
V(\hat{I}) = \frac{1}{m} \mathbb{E}\left( (h(X) - \mathbb{E}(h(X)))^2 \right) = \frac{1}{m} \int_D \left( h(x) - \int_D h(y) p_X(y) \, dy \right)^2 p_X(x) \, dx. \tag{8.11}
\]

Loosely speaking, this variance is a measure of how variable the Monte Carlo estimates would be if we were to evaluate the integral on different occasions.

We see that the magnitude of the variance depends on the variation in \( h(x) - \int_D h(y) p_X(y) \, dy \), which depends in turn on the variation in \( h(x) \). If \( h(x) \) is constant, the variance of \( \hat{I} \) is 0. Of course, in this case, we do not need to do the Monte Carlo estimation; we have the solution \( I = h(\cdot) \).

While the variance in (8.11) is complicated, we have a very simple estimate of the variance; it is the sample variance of the elements composing the estimate of the integral, divided by the sample size \( m \):

\[
\hat{V}(\hat{I}) = \frac{1}{m} \left( \frac{1}{m-1} \sum_{i=1}^{m} (h(x_i) - \overline{h(x)})^2 \right). \tag{8.12}
\]

The second factor in this expression is the sample variance of the observations \( h(x_i) \).

An important fact to be observed in equation (8.11) is that a similar expression would hold if the integrand was multivariate. Therefore, the variance of the Monte Carlo estimate is independent of the dimensionality. This is one of the most important properties of Monte Carlo quadrature.

Reducing the Variance

As we see from equation (8.11) the variance of the Monte Carlo estimator is linear in \( m^{-1} \); hence, the variance is reduced by increasing the Monte Carlo sample size. More effective methods of variance reduction include use of antithetic variates, importance sampling, and stratified sampling, as discussed in Section 2.1.4, beginning on page 12.
Combining Monte Carlo Estimators

The Monte Carlo estimator (8.10) is linear in $h(x_i)$. This implies that the estimator can be evaluated as separate partial sums, either computed in parallel or computed at different times. Separate computations yield separate estimators, $\hat{I}_1, \hat{I}_2, \ldots, \hat{I}_k$, which can be combined to yield

$$\hat{I} = \sum_{i=1}^{k} a_i \hat{I}_i,$$

(8.13)

where the $a_i$ are constants. If each of the $\hat{I}_i$ is unbiased, this estimator is unbiased so long as $\sum_{i=1}^{k} a_i = 1$:

$$E(\hat{I}) = E\left(\sum_{i=1}^{k} a_i \hat{I}_i\right) = \sum_{i=1}^{k} a_i E(\hat{I}_i) = \sum_{i=1}^{k} a_i I = I.$$

If all of the individual estimators are uncorrelated, the variance of the combined estimator is

$$\sum_{i=1}^{k} a_i^2 \text{Var}(\hat{I}_i).$$

To minimize the variance of the linear combination (8.13), the $a_i$’s are chosen inversely proportional to the variances of the component random variables, that is,

$$a_i = \frac{c}{\text{Var}(\hat{I}_i)},$$

for some $c > 0$. If each $\hat{I}_i$ is computed as in equation (8.10) with $m = m_i$, then the optimal $a_i$’s are given by

$$a_i = \frac{m_i}{\sum_j m_j}.$$

Error in Monte Carlo Quadrature

As we have emphasized, Monte Carlo quadrature differs from quadrature methods such as Newton-Cotes methods and Gaussian quadrature in a fundamental way; Monte Carlo methods involve random (or pseudorandom) sampling. The expressions in the Monte Carlo quadrature formulas do not involve
any approximations, so questions of bounds of the error of approximation do not arise. Instead of error bounds or order of the error as some function of the integrand as we discuss for the deterministic methods on pages 452 and 451, we use the variance of the random estimator to indicate the extent of the uncertainty in the solution.

The square root of the variance, that is, the standard deviation of the estimator is a good measure of the range within which different estimators of the integral may fall. Under certain assumptions, using the standard deviation of the estimator, we can define statistical “confidence intervals” for the true value of the integral \( I \). Loosely speaking, a confidence interval is an interval about an estimator \( \hat{I} \) that in repeated sampling would include the true value \( I \) a specified portion of the time. (The specified portion is the “level” of the confidence interval, and is often chosen to be 90% or 95%. Obviously, all other things being equal, the higher the level of confidence the wider must be the interval.)

Because of the dependence of the confidence interval on the standard deviation the standard deviation is sometimes called a “probabilistic error bound”. The word “bound” is misused here, of course, but in any event, the standard deviation does provide some measure of a sampling “error”.

The important thing to note from equation (8.11) is the order of error in the Monte Carlo sample size; it is \( O(m^{-\frac{3}{2}}) \). This results in the usual diminished returns of ordinary statistical estimators; to halve the error, the sample size must be quadrupled.

We should be aware of a very important aspect of this discussion of error bounds for the Monte Carlo estimator. It applies to random numbers. The pseudorandom numbers we actually use only simulate the random numbers, so “unbiasedness” and “variance” must be interpreted carefully.

**Variations of Monte Carlo Quadrature**

The method of estimating an integral described above is sometimes called “crude Monte Carlo”. Another method, which may be more familiar, called “hit-or-miss” Monte Carlo is not to be recommended (see Gentle, 2003, Exercise 7.2, page 271).

Another Monte Carlo method can be developed as suggested in Exercise ??, page ??, To estimate the integral

\[
I = \int_{a}^{b} f(x) \, dx
\]

first generate a random sample of uniform order statistics \( x(1), x(2), \ldots, x(n) \) on the interval \( (a, b) \), and define \( x(0) = a \) and \( x(n+1) = b \). Then estimate \( I \) by \( \hat{I} \):

\[
\frac{1}{2} \left( \sum_{i=1}^{n} (x(i+1) - x(i-1))f(x(i)) + (x(2) - a)f(x(1)) + (b - x(n-1))f(x(n)) \right)
\]
8.1 Numerical Quadrature

This method is similar to approximation of the integral by Riemann sums, except in this case the intervals are random. Advantages: The order of the variance is $O(n^{-2})$. Disadvantages: The order is obviously dependent on the dimension of the integral, however, and in higher dimensions, it is $O(n^{-2/d})$.

Importance sampling in the Riemann sums [Philippe (2000)]

Siegel and O’Brien (1985) describe Monte Carlo methods (see [Thisted, page 305])

**Improper Integrals**

The use of a probability density as a weighting function allows us to apply the Monte Carlo method to improper integrals, that is, integrals with infinite ranges of integration.

**General Issues**

Monte Carlo Genz and Monahan (1998)

**Higher Dimensions**

The Monte Carlo quadrature methods extend directly to multivariate integrals, although, obviously, it takes larger samples to fill the space. It is, in fact, only for multivariate integrals that Monte Carlo quadrature should ordinarily be used. The preference for Monte Carlo in multivariate quadrature results from the independence of the pseudoprobabilistic error bounds and the dimensionality mentioned above.

An important property of the standard deviation of a Monte Carlo estimate of a definite integral is that the order in terms of the number of function evaluations is independent of the dimensionality of the integral. On the other hand, the usual error bounds for numerical quadrature are $O(m^{-d/2})$, where $d$ is the dimensionality.

- one-dimension relatively trivial
- Evans and Swartz (2000)
- compendium of articles in Flournoy and Tsutakawa (1991) most of which have appeared elsewhere.
- Cartesian products
- Order of error bounds $O(m^{-2/d})$
- curse of dimensionality
- Monte Carlo $O(m^{-1})$.

**8.1.5 Quasi Monte Carlo Methods**

Quasirandom sequences

- higher dimensions
- lack of variance estimate

improper integral
curse of
dimensionality
quasi Monte Carlo
method
Sobol’ sequence
Laplace approximation
saddlepoint approximation

8.1.6 Other Methods+

Laplace approximations

Saddlepoint Approximations

The so-called saddlepoint approximation provides useful approximations to the densities of various useful statistics such as the mean, a maximum likelihood estimator, a likelihood ratio statistic, and a score statistic. It can also be used in approximating tail probabilities for various distributions.

Although approximations similar to the saddlepoint approximation had been used in various applications previously, Daniels (1954) derived it in its currently-used form and illustrated its usefulness for the density of a sample mean. (The name “saddlepoint” is due to the method he used in the derivation.)

Jensen (1995)
Reid (1988) survey Barndorff-Nielsen and Cox (1979) really got it started
***** several papers in Sidney Interface proceedings

8.2 Function Approximation

We often need to approximate a given function $f$ by another function $g$. This may be because we know $f$ only at some specific points. We may approximate $f$ so as to approximate an integral of $f$, a derivative of $f$, or just to approximate some values of $f$ that may be unknown or may be difficult to evaluate directly.

The functions used to approximate other functions are often
8.2 Function Approximation

- polynomials or truncated power series
- piecewise polynomials
- ratios of polynomials
- trigonometric functions
- exponential functions

Generally, an approximation is formed using the values of the given function at a finite set of points.

The approximating function $g$ should be easy to evaluate, easy to differentiate, and easy to integrate. It should also of course be easy to determine, given the function to approximate $f$, and it should be “close to” the given function, that is, $\|f - g\|$ should be small.

In function approximation the norm of interest is often the Chebyshev norm, $\sup |f - g|$. This norm is determined by the value of the functions at one point. Chebyshev approximation is approximation in which this norm is minimized over a set of approximating functions $g$.

8.2.1 Linear Operators

Approximations of functions are often formed by use of a functional or an operator. A functional is a mapping of a function space into a vector space; for our purposes we will consider a functional to be a mapping of a function into the finite-dimensional vector space $\mathbb{R}^d$. An operator is a mapping of a function space into a function space. We will denote functions using an upper case letter, and operators using a caligraphic font, for example, we may write $v = L(f)$ and $g = \mathcal{L}(f)$, where $f$ is a function, $v$ is a real vector, and $g$ is a function. Depending on the emphasis or need for clarity, we also may write the functions with formal arguments, $g(t) = \mathcal{L}(f(x))$; or with actual arguments, $g(t_0) = \mathcal{L}(f(x_0))$; and we may write the operator without parentheses, $g = \mathcal{L}f$ or $g(t) = \mathcal{L}f(x)$.

For a given function $f$, the measure of the goodness of the function $g$ to approximate $f$ is the functional $L(g) = \|f - g\|$.

There are many similarities between functionals and operators. The most commonly used functionals and operators are linear. The functional $L$ is a linear functional if for any two functions $f_1$ and $f_2$ within the domain of $L$ and for any constant $c$,

$$L(cf_1 + f_2) = cL(f_1) + L(f_2).$$

Likewise, the operator $\mathcal{L}$ is a linear operator if for any two functions $f_1$ and $f_2$ within the domain of $\mathcal{L}$ and for any constant $c$,

$$\mathcal{L}(cf_1 + f_2) = c\mathcal{L}(f_1) + \mathcal{L}(f_2).$$

An example of a linear operator on functions with domain $[a, b]$ is the one that results in a straight line through the function values at $a$ and $b$. A similar, more general linear operator is
The norm of a functional

\[ \mathcal{L}(f) = \sum_{i=1}^{n} c_i f(x_i). \]

The Lagrange interpolating polynomial (equation (9.24)) is an example of this linear operator.

Another example is a finite Taylor series approximation of a differentiable function:

\[ \mathcal{L}(f) = f(x_0) + (t - x_0)f'(x_0). \]

Convolutions and covariances are linear operators, for example, the convolution of \( f \) and \( g(x) = (t - x)^2 \) over the interval \([0, 1]\):

\[ \mathcal{L}(f) = \int_{0}^{1} (t - x)^2 f(x) \, dx. \]

We define norms of functionals and operators. Because of the way we use functionals and operators in approximation, the most useful norms would be those that capture maximal deviations. Hence, we define the norms as Chebyshev norms. The norm of functional, \( \| L \| \), in terms of a normalized function, is the vector norm:

\[ \| L \| = \max_{\| f \| = 1} |L(f)|. \]

The norm of an operator, \( \| \mathcal{L} \| \), in terms of a normalized function, is the function norm:

\[ \| \mathcal{L} \| = \sup_{\| f \| = 1} \| \mathcal{L}(f) \|. \]

It is clear that these norms satisfy the properties that define a norm (see page 257).

For example, the norm of the linear interpolant operator mentioned above that is the straight line between the points \((a, f(a))\) and \((b, f(b))\) is easily seen to be \( \max(|f(a)|, |f(b)|) \).

The norm of the Lagrange interpolating polynomial (9.24) is the sum of the norms of the individual Lagrange polynomials; that is, if

\[ \mathcal{L}(f)(x) = \sum_{i=1}^{n} f(x_i) l_i(x), \]

then

\[ \| \mathcal{L} \| = \sum_{i=1}^{n} \| l_i(x) \|. \]

This is easily seen by observing
8.2 Function Approximation

\[ \|L\| = \left\| \sum_{i=1}^{n} f(x_i) l_i(x) \right\| \]
\[ \leq \sum_{i=1}^{n} |f(x_i)| \|l_i(x)\| \]
\[ \leq \max |f(x_i)| \sum_{i=1}^{n} \|l_i(x)\| \]
\[ \leq \|f\| \sum_{i=1}^{n} \|l_i(x)\|. \]

Take \( f \) as a function such that \( \|f\| = 1 \), and the result follows.

We will consider some other important linear operators in Section 12.2.

8.2.2 Polynomial Approximation

The most commonly-used approximating functions are polynomials. The Weierstrass approximation theorem insures that a continuous function on \([a, b]\) can be uniformly approximated by polynomials.

**Weierstrass Approximation Theorem.** Let \( f \) be a continuous real function defined on \([a, b]\) and let \( \epsilon > 0 \) be given. Then there exists a polynomial \( q \) with real coefficients such that \( |f(x) - q(x)| < \epsilon \) for all \( x \) in \([a, b]\).

(See any standard text on real analysis, for example, Hewitt and Stromberg, 1969, for a proof.) If the set is not compact, the approximation cannot be guaranteed to be uniform. In practice, however, polynomials can be used effectively in approximating functions that are continuous over open intervals.

Polynomial approximations are usually built from a sum of orthogonal polynomials. We address the methods in Section 8.3.2.

\[ f(x) \approx \frac{p(x)}{q(x)}. \]

This type of approximation, which is an instance of a rational approximation, is called a **Padé approximation.** A Padé approximation is particularly useful when the function to be approximated contains poles.

and Remez algorithms – see Ralston and Wilf (1960)
8.2.3 Multivariate Approximation

The methods discussed above were generally based on an implicit assumption that the domain of the function is an interval in the reals. The approximation method uses the function values at a finite set of points within the interval.

Conceptually, most of the discussion applies immediately to multivariate functions defined over subspaces of $\mathbb{R}^d$. Practical difficulties, however, prevent direct methods of multivariate approximation from being very useful.

In multivariate function approximation, the function values are usually known at points on a grid in $\mathbb{R}^d$. The most common way of approximating a multivariate function is by successive univariate approximations. If the function to be approximated is, say, the bivariate function $f(x, y)$, and the values of the function are known at a grid of points $(x_1, \ldots, x_n) \times (y_1, \ldots, y_m)$, suppose we wish to approximate $f(x, y)$ at the point $(x^*, y^*)$. In successive univariate approximation, we first approximate $f(x_i, y)$ at each given point $x_i$ by a function $g_{x_i}(y)$. (This is univariate approximation in the variable $y$.) Now, with the values $g_{x_i}(y^*)$ for each $i$, we approximate $f(x^*, y^*)$ using univariate approximation in the variable $x$.

The method of successive univariate approximation extends to higher dimensions, but the number of computations obviously grows exponentially.

8.2.4 Edgeworth Expansions

Two reasons that the normal distribution prevades statistical theory and methods for two reasons. The first is that this distribution serves so well to model natural phenomena. Even if finite samples follow some other distribution, it is likely that following a suitable transformation, the normal distribution is a good asymptotic approximation. The second reason follows from the first. A wealth of methods have been developed that are directly applicable to the normal distribution. If another distribution can be related to the normal, the vast array of statistical methods for the normal distribution become available.

Edgeworth expansions

$$X_n \xrightarrow{d} N(0, \sigma^2) \text{ with CDF } P_{X_n}$$

$$P_{X_n}(x) = \Phi(x) + \left( \frac{1}{n \pi} f_1(x) + \cdots + \frac{1}{n \pi} f_k(x) \right) \phi(x) + o\left( \frac{1}{n \pi} \right)$$


8.2.5 Recursive Schemes for Accumulating Series

Tietjen (1998) recursive schemes for calculating common cumulative distributions,
8.3 Orthogonal Systems

In Section 5.1.1 on page 233 we discuss systems of orthogonal vectors. Orthogonality is defined in terms of the dot product, and two vectors are said to be orthogonal if $\langle u, v \rangle = 0$. Orthogonality of other systems can be defined similarly.

Two functions $f$ and $g$ in $L^2(a, b)$ are said to be orthogonal if

$$\langle f, g \rangle = 0.$$ 

Two normal functions that are orthogonal are said to be orthonormal.

In Section 5.1.1 we mention the use of a weight matrix for defining orthogonality of vectors (in which case we use the phrase “conjugate vectors”). The use of a weight is more common in working with orthogonal functions. We say that $f$ and $g$ orthogonal with respect to the weight function $w$ if

$$\int_a^b f(x)g(x)w(x)\,dx = 0.$$ 

(In this case, we also require that the integral of the weighted squared functions exist.)

8.3.1 Basis Functions

If each function in a linear space can be expressed as a linear combination of the functions in a set $G$, then $G$ is said to be a generating set, a spanning set, or a basis set for the linear space. (These three terms are synonymous.) The basis sets for finite-dimensional vector spaces are finite; for most function spaces of interest, the basis sets are infinite.

Basis sets consisting of orthonormal functions are generally easier to work with, and can be formed from any basis set. Given two nonnull, linearly independent functions, $f_1$ and $f_2$, two orthonormal vectors, $\tilde{f}_1$ and $\tilde{f}_2$, that span the same space can be formed as

$$\tilde{f}_1(\cdot) = \frac{1}{\|f_1\|} f_1(\cdot)$$

$$\tilde{f}_2(\cdot) = \frac{1}{\|f_2 - \langle f_2, f_1 \rangle f_1\|} (f_2(\cdot) - \langle f_2, f_1 \rangle f_1(\cdot)).$$

(8.14)

These are called Gram-Schmidt transformations. They can easily be extended to more than two functions to form a set of orthonormal functions from any set of linearly independent functions.
An arbitrary function \( f \) in \( L^2(a, b) \) often can be expressed as a linear combination of functions in a set of orthonormal functions \( \{f_i\} \), just as an arbitrary vector is expressed as a linear combination of a basis set:

\[
f(x) = \sum_{i=1}^{\infty} c_i f_i(x).
\]

(8.15)

This representation may not be possible for an arbitrary function \( f \). When this expansion is possible, however, it is clear that \( c_i \) in (8.15) must satisfy

\[
c_i = \langle f, f_i \rangle.
\]

(8.16)

(Compare this with the similar result for vectors in Exercise 5.2 in Chapter 5, page 342.) The coefficients \( \{c_i\} \) are called the Fourier coefficients of \( f \) with respect to the orthonormal functions \( \{f_i\} \).

If the function \( f \) has the expansion (8.15), the square of the norm of the function is the sum of squares of the Fourier coefficients:

\[
\langle f, f \rangle = \left( \sum_{i=1}^{\infty} c_i f_i(x) \right) \cdot \left( \sum_{i=1}^{\infty} c_i f_i(x) \right)
= \int \left( \sum_{i=1}^{\infty} c_i f_i(x) \right)^2 w(x) \, dx
= \sum_{i=1}^{\infty} |c_i|^2.
\]

(8.17)

In practice, of course, for representing an arbitrary function in terms of the orthogonal basis, we use a finite truncation of just the first \( k \) terms of the convergent series (8.15). The error due to finite truncation at \( k \) terms of the infinite series is the residual function \( f - \sum_{i=1}^{k} c_i f_i \). The mean squared error over the domain \( (D) \), is the scaled, squared \( L^2 \) norm of the residual,

\[
\frac{1}{d} \left\| f - \sum_{i=1}^{k} c_i f_i \right\|^2,
\]

(8.18)

where \( d \) is some measure of the domain \( D \). (If the domain is the interval \([a, b]\), for example, one choice is \( d = b - a \).)

A very important property of Fourier coefficients is that they yield the minimum mean squared error for a given set of basis functions \( \{f_i\} \); that is, for any other constants, \( \{a_i\} \), and any \( k \),

\[
\left\| f - \sum_{i=1}^{k} c_i f_i \right\|^2 \leq \left\| f - \sum_{i=1}^{k} a_i f_i \right\|^2
\]

(8.19)

(see Exercise 12.3).
Another important property of the residuals, analogous to that of the linear least squares estimator (see equation (5.94), page 304), is that the residual or error, \( f - \sum_{i=1}^{k} c_i f_i \), is orthogonal to the approximation, that is,

\[
\left\langle \sum_{i=1}^{k} c_i f_i, f - \sum_{i=1}^{k} c_i f_i \right\rangle = 0.
\]  
(8.20)

Partial sums of squares of Fourier coefficients, \( \sum_{i=1}^{k} c_i^2 \), for any \( k \) are bounded by \( \| f \|^2 \), that is,

\[
\sum_{i=1}^{k} |c_i|^2 \leq \| f \|^2.
\]  
(8.21)

This is called Bessel’s inequality, and, it follows from

\[
0 \leq \left\| f - \sum_{i=1}^{k} c_i f_i \right\|^2 = \| f \|^2 - \sum_{i=1}^{k} |c_i|^2.
\]

To insure that the linear combination of orthonormal functions converges to the given function, the system \( \{ f_i \} \) must be complete: an orthonormal system \( \{ f_i \} \) is complete in \( L^2(D) \) if \( g \in L^2(a,b) \) and \( \langle g, f_i \rangle = 0 \) for all \( i \) imply that \( g = 0 \) almost everywhere.

For any function \( f \in L^2(D) \), a linear combination formed by Fourier coefficients with functions from a complete orthonormal system converges to \( f \); that is, if \( \{ f_i \} \) is a complete orthonormal system in \( L^2(D) \), and \( c_i = \langle f, f_i \rangle \), then

\[
\left\| f - \sum_{i=1}^{k} c_i f_i \right\| \to 0 \quad \text{as} \quad k \to \infty.
\]  
(8.22)

If \( D \) is a linear function space with basis set \( G_1 \) and \( E \) is a linear function space with basis set \( G_2 \), the set of all tensor products of functions in \( G_1 \) with those in \( G_2 \) is a basis set for the the tensor product of \( D \) and \( E \).

8.3.2 Orthogonal Polynomials+

Very useful systems of orthonormal functions can be constructed as polynomials. The Weierstrass approximation theorem insures that a continuous function on \( [a, b] \) can be uniformly approximated by polynomials.

We use the notation \( q_i(x) \) to denote a polynomial of degree \( i \); hence, in this case, the first item in the sequence has an index of 0. (As I have indicated before, this kind of indexing can be confusing, unless it is a natural indexing, as I believe it to be in this case.)
Gram-Schmidt transformation
recursion formula for orthogonal polynomials

The simplest set of polynomials, that is, the monomials,

\[ 1, \ x, \ x^2, \ldots \]

can be orthogonalized and normalized by Gram-Schmidt transformations. The first three transformations, analogous to those of equation (5.56) are:

\[ q_0(x) = 1 \]
\[ q_1(x) = \frac{x - \langle 1, x \rangle}{\|x - \langle 1, x \rangle\|} \]  
\[ (8.23) \]
\[ q_2(x) = \frac{x^2 - \langle 1, x^2 \rangle - \langle q_1(x), x^2 \rangle q_1(x)}{\|x^2 - \langle 1, x^2 \rangle - \langle q_1(x), x^2 \rangle q_1(x)\|} \]

The specific form of these polynomials depends on the domain and the weight function, which determine the inner products in these transformations. Often the polynomials are not normalized.

It is clear that for the \( k \)th polynomial in the orthogonal sequence, we can choose an \( a_k \) that does not involve \( x \), such that

\[ q_k(x) - a_k x q_{k-1}(x) \]

is a polynomial of degree \( k - 1 \).

Because any polynomial of degree \( k - 1 \) can be represented by a linear combination of the first \( k \) members of any sequence of orthogonal polynomials, we can write

\[ q_k(x) - a_k x q_{k-1}(x) = \sum_{i=0}^{k-1} c_i q_i(x). \]

Because of orthogonality, all \( c_i \) for \( i < k - 2 \) must be 0. Therefore, collecting terms, we have, for some constants \( a_k, b_k, \) and \( c_k \), we have the three-term recursion that applies to any sequence of orthogonal polynomials:

\[ q_k(x) = (a_k x + b_k) q_{k-1}(x) - c_k q_{k-2}(x) = 0, \quad \text{for } k = 2, 3, \ldots \]  
\[ (8.24) \]

This recursion formula is often used in computing orthogonal polynomials. The coefficients in this recursion formula depend on the specific sequence of orthogonal polynomials, of course.

This three-term recursion formula can also be used to develop a formula for the sum of products of orthogonal polynomials \( q_i(x) \) and \( q_i(y) \):

\[ \sum_{i=0}^{k} q_i(x) q_i(y) = \frac{1}{a_{k+1}} \frac{q_{k+1}(x)q_k(y) - q_k(x)q_{k+1}(y)}{x - y}. \]  
\[ (8.25) \]
This expression, which is called the Christoffel-Darboux formula, is useful in evaluating the product of arbitrary functions that have been approximated by finite series of orthogonal polynomials.

In some applications it is important that the orthogonal polynomials have full sets of distinct real roots. Also, applications are often simpler if the coefficient of the term of largest degree is 1.

A system of orthogonal polynomials is defined by the weight function and the domain. The main thing that determines which system to use is the domain, although the shape of the weight function may be important in achieving better finite series approximations.

Systems of orthogonal polynomials can also be developed from series solutions to differential equations (see Chapter 9, page 482).

Orthogonal vectors can be formed by evaluating orthogonal polynomials over a grid. These orthogonal vectors are discrete versions of the corresponding polynomials.

The standard treatment of orthogonal polynomials is Szegő (1958), in which several other systems are described and more properties of orthogonal polynomials are discussed. A general reference on multivariate orthogonal polynomials is Dunkl and Yu (2001).

**Standard Series of Univariate Orthogonal Polynomials**

There are several widely-used complete systems of univariate orthogonal polynomials. The different systems are characterized by the one-dimensional intervals over which they are defined and by their weight functions. The Legendre, Chebyshev, and Jacobi polynomials are defined over $[-1, 1]$, and hence can be scaled into any finite interval. The weight function of the Jacobi polynomials is more general, so a finite sequence of them may fit a given function better, but the Legendre and Chebyshev polynomials are simpler and so are often used. The Laguerre polynomials are defined over the half line $[0, \infty)$ and the Hermite polynomials are defined over the reals, $(-\infty, \infty)$. Table 8.1 summarizes the ranges and weight functions.

Most of these systems have particularly simple expressions for the coefficients in the recurrence relation (8.24), so they are relatively simple to compute. The $k^{th}$-degree polynomial in each system has $k$ distinct real roots. The usefulness of the standard orthogonal polynomials derives from their use in approximations and also from their use as solutions to standard classes of differential equations, as we discuss in Chapter ??.

These systems of orthogonal polynomials are described below. For some systems, different forms of the weight functions are used by other authors. The properties of the orthogonal polynomials are essentially the same for the differing forms of the weight functions, but the coefficients of the polynomials are different. In some cases, also, the polynomials may be normalized.
Legendre polynomial
Legendre's differential equation
discrete Legendre polynomials

Table 8.1. Orthogonal Polynomials

<table>
<thead>
<tr>
<th>Polynomial Series</th>
<th>Range</th>
<th>Weight Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Legendre</td>
<td>$[-1, 1]$</td>
<td>1 (uniform)</td>
</tr>
<tr>
<td>Chebyshev</td>
<td>$[-1, 1]$</td>
<td>$(1 - x^2)^{1/2}$</td>
</tr>
<tr>
<td>Jacobi</td>
<td>$[-1, 1]$</td>
<td>$(1 - x)^\alpha (1 + x)^\beta$ (beta)</td>
</tr>
<tr>
<td>Laguerre</td>
<td>$[0, \infty)$</td>
<td>$x^{\alpha-1} e^{-x}$ (gamma)</td>
</tr>
<tr>
<td>Hermite</td>
<td>$(-\infty, \infty)$</td>
<td>$e^{-x^2}$ (normal)</td>
</tr>
</tbody>
</table>

Legendre Polynomials

The *Legendre polynomials* have a constant weight function and are defined over the interval $[-1, 1]$. Building the Legendre polynomials from the monomials, it is easy to see that the first few Legendre polynomials are

\[
\begin{align*}
P_0(x) &= 1 \\
P_1(x) &= x \\
P_2(x) &= (3x^2 - 1)/2 \\
P_3(x) &= (5x^3 - 3x)/2 \\
P_4(x) &= (35x^4 - 30x^2 + 3)/8 \\
P_5(x) &= (63x^5 - 70x^3 + 15x)/8
\end{align*}
\]

(8.26)

Graphs of these polynomials are shown in Figure 8.3.

The recurrence formula (8.24) for the Legendre polynomials is

\[
P_k(x) - \frac{2k - 1}{k} x P_{k-1}(x) + \frac{k - 1}{k} P_{k-2}(x) = 0.
\]

(8.27)

As mentioned above, orthogonal polynomials can also be developed as solutions to differential equations. The standard series of orthogonal polynomials arise from differential equations that are important in applied mathematics. The Legendre polynomials are the solutions to *Legendre’s equation*,

\[
(1 - x^2)u'' - 2xu' + k(k + 1)u = 0.
\]

This equation describes an inverse $r^2$ potential.

Also as mentioned above, vectors whose elements are orthogonal polynomials evaluated over a grid are discrete versions of the orthogonal polynomials. The discrete Legendre polynomials can be formed easily by setting $x_i$ to the values of a grid over $[-1, 1]$, forming a Vandermonde matrix evaluated at those grid points, and then forming the $QR$ decomposition (see Section 5.2.2, page 287) of the matrix.
8.3 Orthogonal Systems

Chebyshev polynomial
Jacobi polynomial
beta weight function

Chebyshev Polynomials

The *Chebyshev polynomials* have a weight function proportional to the Chebyshev density, \( w(x) = (1 - x^2)^{-1/2} \). They are defined over the interval \([-1, 1]\). The first few Chebyshev polynomials are

\[
T_0(x) = 1 \quad T_1(x) = x \\
T_2(x) = 2x^2 - 1 \quad T_3(x) = 4x^3 - 3x \\
T_4(x) = 8x^4 - 8x^2 + 1 \quad T_5(x) = 16x^5 - 20x^3 + 5x.
\]

(8.28)

The recurrence formula (8.24) for the Chebyshev polynomials is

\[
T_k(x) - 2xT_{k-1}(x) + T_{k-2}(x) = 0.
\]

(The sequence of Chebyshev polynomials is the only sequence for which the coefficients in formula (8.24) are the same for all \( k \).)

These polynomials are sometimes called Chebyshev polynomials of the first kind; similar polynomials with weight function \( w(x) = (1 - x^2)^{1/2} \) are called Chebyshev polynomials of the second kind.

Jacobi Polynomials

The *Jacobi polynomials* are defined over the interval \([-1, 1]\), with a beta weight function, \( w(x) = (1 - x)^\alpha (1 + x)^\beta \), for \( \alpha, \beta > -1 \). The Legendre polynomials are Jacobi polynomials with \( \alpha = \beta = 0 \), and the Chebyshev polynomials are Jacobi with \( \alpha = \beta = -1/2 \).
Laguerre Polynomials

The Laguerre polynomials are defined over \([0, \infty)\), with a gamma weight function, \(w(x) = x^{\alpha - 1}e^{-x}\). The \(j^{th}\) Laguerre polynomial is often denoted by \(L_j^{(\alpha)}(x)\). The most commonly used series of Laguerre polynomials have \(\alpha = 1\), and in this case, the notation \(L_j(x)\) is used. The first few Laguerre polynomials are

\[
\begin{align*}
L_0(x) &= 1 \\
L_1(x) &= -x + 1 \\
L_2(x) &= (x^2 - 4x + 2)/2 \\
L_3(x) &= (-x^3 + 9x^2 - 18x + 6)/6 \\
L_4(x) &= (x^4 - 16x^3 + 72x^2 - 96x + 24)/24 \\
L_5(x) &= (-x^5 + 25x^4 - 200x^3 + 600x^2 - 600x + 120)/120
\end{align*}
\]  

The recurrence formula (8.24) for these Laguerre polynomials is

\[
L_k(x) - \left(\frac{2k - 1}{k} + \frac{x}{k}\right)L_{k-1}(x) + \frac{k - 1}{k}L_{k-2}(x) = 0.
\]

Hermite Polynomials

The Hermite polynomials are defined over \((-\infty, \infty)\) with a weight function proportional to the error function density, \(w(x) = e^{-x^2}\). Building the Hermite polynomials from the monomials, it is easy to see that the first few Hermite polynomials are

\[
\begin{align*}
H_0(x) &= 1 \\
H_1(x) &= 2x \\
H_2(x) &= 4x^2 - 2 \\
H_3(x) &= 8x^3 - 12x \\
H_4(x) &= 16x^4 - 48x^2 + 12 \quad H_5(x) = 32x^5 - 160x^3 + 120x
\end{align*}
\]  

Graphs of these polynomials are shown in Figure 8.4.

In an alternative definition of Hermite polynomials, the normal weight function, \(w(x) = e^{-x^2/2}\), is used. Hermite polynomials are sometimes denoted in the same way as the ones defined above, \(H_0, H_1, \ldots\), although they are often denoted by \(He_0, He_1, \ldots\). We will use the notation \(He_0, He_1, \ldots\). The polynomials are related by

\[
He_k(x) = 2^{-k/2}H_k(x/\sqrt{2}).
\]

This alternate definition of the Hermite polynomials are often used by statisticians, because the weight function is proportional to the normal density (see, for example, Kendall and Stuart, 1969). Also, because the coefficient of the term of largest degree is 1, some applications involve simpler expressions. The first few alternate Hermite polynomials are

\[
\begin{align*}
He_0(x) &= 1 \\
He_1(x) &= 2x \\
He_2(x) &= 4x^2 - 2 \\
He_3(x) &= 8x^3 - 12x \\
He_4(x) &= 16x^4 - 48x^2 + 12 \quad He_5(x) = 32x^5 - 160x^3 + 120x
\end{align*}
\]
8.3 Orthogonal Systems

The recurrence formula (8.24) for these Hermite polynomials is particularly simple:

\[ H^e_k(x) - xH^e_{k-1}(x) + (k - 1)H^e_{k-2}(x) = 0. \]  

Orthogonal Functions Related to the Orthogonal Polynomials

Sometimes applications are simpler when the weight function is incorporated into the orthogonal functions. The resulting functions are orthogonal with respect to a constant weight. For example, if

\[ h_k(x) = H_k e^{-x^2/2}, \]  

the set \( h_0, h_1, h_2, \ldots \) is an orthogonal set with respect to a constant weight:

\[ \int_{-\infty}^{\infty} h_i(x)h_j(x)dx = 0 \quad \text{for} \ i \neq j. \]

These orthogonal functions are called Hermite functions.
Even and Odd functions

An even function is a function $f$ such that $f(-x) = f(x)$, and an odd function is one such that $f(-x) = -f(x)$. The Legendre, Chebyshev, and Hermite polynomials are either even or odd functions, depending on their degree. In each case, a polynomial of even degree is an even function and one of odd degree is an odd function; for example, the Chebyshev polynomials satisfy the relation

$$T_i(-x) = (-1)^i T_i(x).$$

Expansion of Functions in Orthogonal Polynomials

Complicated functions or functions that are intractable for certain operations can often be approximated with a finite sum of orthogonal polynomials. An important application of this type of approximation is in evaluation of integrals by expansion of the integrand using orthogonal polynomials. This method of numerical integration is called Gaussian quadrature, and is discussed in Section 8.1.3, page 451.

To represent a given function in a series of orthogonal polynomials, the first consideration is the domain of the function. The three types of domain and the possible polynomials to use are

- a finite interval, $[a, b]$ — Jacobi polynomials (or the special cases, Legendre and Chebyshev);
- a half-infinite interval, $[a, \infty]$ or $[-\infty, b]$ — Laguerre polynomials;
- an infinite interval, $[-\infty, \infty]$ — Hermite polynomials.

**** give some examples here ****

expansion of a probability density function in terms of a normal Gram-Charlier series

$$p(x) = \phi(x) + \frac{1}{6} \gamma_1 \Phi'''(x) + \frac{1}{24} \gamma_2 \Phi''''(x) + \cdots$$

Edgeworth series

Cornish-Fisher expansion

$$y \approx \mu + \sigma w$$

where

$$w = x + \gamma_1 h_1(x) + \gamma_2 h_2(x) + \gamma_3^2 h_{11}(x) + \gamma_3 h_3(x) + \gamma_1 \gamma_2 h_{12}(x) + \gamma_3^3 h_{111}(x) + \gamma_4 h_4(x) + \gamma_2^3 h_{22}(x) + \gamma_1 \gamma_3 h_{13}(x) + \gamma_2^2 \gamma_2 h_{112}(x) + \gamma_4^2 h_{1111}(x) + \cdots$$

where
8.3 Orthogonal Systems

Chebyshev approximation
minimax approximation

\[ h_1(x) = \frac{1}{6} H_2^s(x) \]
\[ h_2(x) = \frac{1}{24} H_3^s(x) \]
\[ h_{11}(x) = -\frac{1}{36} (2H_3^s(x) + H_1^s(x)) \]
\[ h_3(x) = \frac{1}{120} H_4(x) \]
\[ h_{12}(x) = -\frac{1}{24} (H_4^s(x) + H_2^s(x)) \]
\[ h_{111}(x) = -\frac{1}{324} (12H_4^s(x) + 19H_2^s(x)) \]
\[ h_4(x) = \frac{1}{720} H_5^s(x) \]

Smoothing Data with Orthogonal Polynomials

In the previous section we considered the approximation of a function with known form by a series of orthogonal polynomials. In many applications, we do not have a function with a closed form; we may have a discrete function composed of observations with two components corresponding to an argument, \(x_i\), and a function value, \(y_i\). If we assume the data represent exact values, we may interpolate the data to form a continuous function. If, however, the data are assumed to arise from a process with noise, we may build a smooth approximation of the function as a finite series of orthogonal polynomials,

\[ f(x) = \sum_{j=1}^{k} c_j f_j(x). \]

Because we do not know the form of \(f(x)\), we choose the \(c_j\) so as to minimize the differences

\[ y_i - \sum_{j=1}^{k} c_j f_j(x_i). \]

Instead of a function norm, as in equation (8.18), we consider a norm of the vector:

\[ \left\| y_i - \sum_{j=1}^{k} c_j f_j(x_i) \right\|. \]

The norm is most often chosen as the \(L_2\) norm, and the resulting approximation is the least squares fit. Other appropriate norms include the \(L_1\) norm, resulting in an approximation with the least absolute deviations, and the \(L_\infty\) norm, resulting in an approximation with the minimum maximum deviation. The latter type of fit is called a minimax or Chebyshev approximation.

******* example **** also see exercises
8.4 Evaluation of Special Functions

Some of the most important and most common numerical computations in scientific applications are the evaluation of the so-called “special functions”. These functions, such as the exponential and logarithmic functions, the trigonometric functions, and functions that arise in the solutions of differential equations, are rarely evaluated for their own sake; rather, their evaluations are generally performed as part of some larger computational problem. The functions are often evaluated repeatedly in the solution of the larger problem. It is therefore important not only that they be evaluated accurately, but that the computations be very efficient.

Abramowitz and Stegun (1964) describe computational methods for evaluating many special functions. The basic algorithms they discussed are in many cases still the best ways of evaluating the special functions. Abramowitz and Stegun classified special functions into various groups that remain the general organization of numerical libraries for the functions. The Guide to Available Mathematical Software (GAMS) (see the bibliography), follows the general structure of Abramowitz and Stegun (1964). The major groupings of the special functions are

- elementary transcendental functions
  - trigonometric, inverse trigonometric
  - exponential, logarithmic
  - hyperbolic, inverse hyperbolic
- exponential and logarithmic integrals
- gamma
  - gamma, log gamma, reciprocal gamma
  - beta, log beta
  - psi (logarithmic derivative of gamma)
  - incomplete gamma
  - incomplete beta
- error function
  - error functions, inverses, integrals
  - Fresnel integrals
  - Dawson’s integral
- Bessel functions
  - J, Y, H^{(1)}, H^{(2)}
  - I, K
  - Kelvin functions
  - Airy functions
- elliptic integrals
- Weierstrass elliptic integrals
- Mathieu functions

There is also another grouping of special functions in GAMS for probability distributions.
It is not our intent in this section to describe algorithms for each of these special functions. Rather, we mention some general techniques, and refer to other sources. GAMS is a good source of information about software for evaluating the special functions. Programs for evaluation of special functions are available in the IMSL Libraries (a function is available for each entry in the list above), in the Maple and Mathematica packages, and in CALGO (see page 651), as well as in more specialized collections, such as Cody (1993) or Cody and Coonen (1993).

Spanier and Oldham (1987) and Thompson (1997) provide general descriptions of many special functions. Both books discuss relationships among the special function and describe methods for evaluating the special functions. They also contain many graphs of the functions. Abramowitz and Stegun (1964) provide tables of the values of special functions for many arguments.

8.4.1 General Methods

Expansions

Use of moments and systems of frequency curves Bowman and Shenton (1979) give percentage points of Pearson distributions.


Comparisons – Pearson, Johnson, Burr (1979)

Use of recursion relations Gautschi (1999) recursive calculation of incomplete gamma functions

Table-Driven Evaluation

table-driven evaluation of special functions **** see Tang ACM TOMS 1989, page 144 ****

8.4.2 Evaluation of Distribution Functions and Their Inverses

Application of most methods of statistical inference involve computation of probabilities or of rejection regions corresponding to an estimator or a test statistic. Two simple examples are the computation of the probability that a standard normal random variable $Z$ is larger in absolute value than some given value $z_0$, and the computation of the value $t_0$ such that a Student’s $t$ random variable with given degrees of freedom would have a given probability of being greater than $t_0$. For the more common distributions, tables with three or four decimals of precision for these probabilities and critical values have been available for many years. For setting confidence intervals and doing significance tests, this level of precision for the distribution function of the random variable being used is quite adequate.
It is not always the case, however, that three or four decimals of precision is adequate for a distribution function. Perhaps the simplest example of the need for high precision is in the evaluation of the distribution function of an order statistic. The distribution function of the $k^{th}$ order statistic in a sample of size $n$ from a population with distribution function $P(\cdot)$ is

$$\Pr(X_{(k)} \leq x_0) = \sum_{j=k}^{n} \binom{n}{j} (P(x_0))^j (1 - P(x_0))^{n-j}.$$ 

Obviously, if this relationship is used, even if only three or four decimals of precision is required for the distribution function of the order statistic, it is necessary to have much greater precision in the evaluation of $P(\cdot)$.

In addition to the computational problems resulting from the need for higher precision in computing a standard, relatively simple distribution function, distribution functions such as for the doubly noncentral $F$ random variable are exceedingly complicated and require different algorithms for evaluation at different points in the argument/parameter space.

**Saddlepoint Approximations**

Jensen (1995)
Reid (1988) survey Barndorff-Nielsen and Cox (1979) really got it started

**Normalization of the Approximation**

**Approximation of Tail Probabilities**

Lugannani and Rice (1980) for tail probabilities
McCullagh (1987)
Davison and Hinkley (1988)
Daniels (1983)
Daniels (1987)
Tsuchiya and Konishi (1997), Communications is Statistics A 26 2541.
General saddlepoint approximations and normalizing transformations for multivariate statistics,

**Exercises**

8.1. Derive the Newton-Cotes 4-point formula (called the Milne rule), the 6-point formula (called the Weddle rule), and the 8-point formula (used in the Matlab function `quad8`). A symbolic package such as Maple may be useful for this. See Gander and Gruntz (1999), who write the Maple function
8.2. Use Newton-Cotes formulas to evaluate $\int_{-\infty}^{1} \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \, dx$. Because the limit of integration is infinite, you must use some symmetry property along with the Newton-Cotes formulas.
   a) Use the trapezoid rule with $h = 0.10$ and $h = 0.01$.
   b) Use Simpson’s $\frac{1}{3}$ rule with $h = 0.10$ and $h = 0.01$.
   c) Use Simpson’s $\frac{3}{8}$ rule with $h = 0.10$ and $h = 0.01$.

Compare the accuracy of the results obtained from the trapezoid rule and Simpson’s $\frac{1}{3}$ rule. Thinking of the problem as a statistical sampling problem, which rule would you think would be better, and why? (More equal weighting of the sample points?) Does this heuristic reasoning correspond with your empirical results obtained above?

8.3. Consider the definite integrals

$\begin{align*}
I_1 &= \int_0^1 \sin(\pi x) \, dx, \\
I_2 &= \int_0^1 \int_0^2 y \sin(\pi x) \, dy \, dx, \\
I_3 &= \int_0^1 \int_0^2 \int_0^2 zy \sin(\pi x) \, dz \, dy \, dx.
\end{align*}$

Determine the error in various quadrature methods as a function of the number of points at which the integrand is evaluated. Let the number of evaluations be $n = 10, 20, \ldots, 200$. Evaluate each integral using
   a) Newton-Cotes formulas
   b) crude Monte Carlo using pseudorandom numbers
   c) crude Monte Carlo using Halton sequences (from Exercise ??, page ??)
   
For each integral and each method, compute the error and plot the error as a function of $n$ (nine plots). Write a brief interpretation of your plots.

8.4. The change in enthalpy of a substance from temperature $T_1$ to $T_2$ is given by

$$\int_{T_1}^{T_2} C_p \, dT,$$

where $C_p$ is the heat capacity; and the change in entropy is given by

$$\int_{T_1}^{T_2} \frac{C_p}{T} \, dT.$$

Measurements taken on a certain alloy are shown below.
noncentral gamma distribution, evaluating probabilities “it Exercise”refex:qde200:
Hermite’s equation
Hermite polynomial

<table>
<thead>
<tr>
<th>$C_p$(J · mol$^{-1}$ · K$^{-1}$)</th>
<th>$T$(K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2271</td>
<td>10</td>
</tr>
<tr>
<td>1.563</td>
<td>20</td>
</tr>
<tr>
<td>4.655</td>
<td>30</td>
</tr>
<tr>
<td>8.221</td>
<td>40</td>
</tr>
<tr>
<td>10.98</td>
<td>50</td>
</tr>
<tr>
<td>13.78</td>
<td>60</td>
</tr>
<tr>
<td>15.05</td>
<td>70</td>
</tr>
<tr>
<td>16.92</td>
<td>80</td>
</tr>
</tbody>
</table>

a) Compute the changes in enthalpy and entropy of the alloy from 10K to 80K. State your answers with the correct number of significant digits.
b) What kinds of errors are your solutions subject to?

8.5. Use a standard computer function (as in R or the IMSL Libraries, for example) to determine the probability that a standard normal random variable is greater than 12. (Hint: The answer is not 0!)
a) Why did I give the hint?!
b) For an important random variable $X$ (for example, one widely used as a test statistic), why is it often important to provide a software routine to compute the distribution function $P(x)$ as well as the complementary distribution function $1 - P(x)$?

8.6. Consider the noncentral gamma distribution with density

$$p(x) = e^{-\lambda - \theta x} \sum_{r=0}^{\infty} \frac{\lambda^r}{r! \Gamma(r + \gamma)} x^{r + \gamma - 1}$$

(Hougaard, 1988).
a) Derive the saddlepoint approximation to the noncentral gamma.
b) For $\gamma = 2, \theta = 1, \text{and} \lambda = 4$, compute the 0.90, 0.95, and 0.99 fractiles.

(Other approximations are given in Cox and Reid, 1987.)

8.7. Use the method of undetermined coefficients, as illustrated beginning on page 482, to determine solutions to Hermite’s equation:

$$u'' - 2xu' + \lambda u = 0$$

over $\mathbb{R}$. With proper scaling, your solutions should be the Hermite orthogonal polynomials given in equation (8.30) on page 470.
Solution of Differential Equations

Differential equations are some of the most important tools in modeling physical phenomena. A differential equation expresses the rate of change of one or more variables in terms of the values of those and other variables. The rate of change is measured in terms of “independent variables”. The differential equation

\[ \frac{du}{dx} = au + bx, \]

for example, states that the rate of change of the response variable \( u \) is equal to a multiple of the current value of \( u \) and some other multiple of the value of the independent variable \( x \). Often the quantities \( a \) and \( b \) in an equation such as this may also depend on \( x \). If they do, we generally emphasize this by writing the equation as

\[ \frac{du}{dx} = a(x)u(x) + b(x)x. \]

We often use the notation \( u' \) in place of \( \frac{du}{dx} \), when it is clear what the independent variable is. We also use the notation \( u(x) \) or \( u = f(x) \) to express the dependence of \( u \) on \( x \). Without some dependence, the differential equation is null.

If a differential equation has only one independent variable, it is called an ordinary differential equation (ODE); if it has more than one independent variable, it is called a partial differential equation (PDE). In the most common applications of partial differential equations, which model changes of some quantity in time or in ordinary three-dimensional space, there are only a small number of independent variables.

The order of a differential equation is the highest order of the derivative appearing in the equation; thus, \( u' = u + x \) is a first order ordinary differential equation and \( u'' = u' + u + x \) is a second order ordinary differential equation.

The degree of a differential equation is the largest power of a derivative. A differential equation is *linear* if the dependent variables and all derivatives are
linear in the equation; otherwise, the differential equation is nonlinear. (Notice
the degree is defined in terms of the power of the derivatives only, whereas
linearity is defined in terms of both derivatives and dependent variables.)
Thus, \( u' = u + x \) and \( u'' = u + x \) are linear, and \( (u')^2 = u + x \) and \( u'' = u^2 + x \)
are nonlinear.

### More definitions
- complementary solution
- particular solution
- autonomous equation
- homogeneous equation

### 9.1 Ordinary Differential Equations

#### Systems of Differential Equations

A system of ODEs is a set of differential equations with the same independent
variable:

\[
\begin{align*}
    u'_1 &= f_1(x, u_1, u_2, \ldots, u_n) \\
    u'_2 &= f_2(x, u_1, u_2, \ldots, u_n) \\
    \cdots &= \cdots \\
    u'_n &= f_n(x, u_1, u_2, \ldots, u_n)
\end{align*}
\]

or

\[ Au' = f(x, u), \]

where \( A \) is a matrix and \( u \) and \( u' \) are vectors.

Higher-order ODEs can generally be replaced by systems of first-order
ODEs by introducing new variables for the higher-order derivatives. For ex-
ample, the ODE

\[ u''' = f(x, u, u', u'') \]  \hspace{1cm} (9.1)

is equivalent to the system

\[
\begin{align*}
    v'_1 &= v_2 \\
    v'_2 &= v_3 \\
    v'_3 &= f(x, v_1, v_2, v_3)
\end{align*}
\]  \hspace{1cm} (9.2)

which is obtained by setting
Solutions of Differential Equations

A solution of a system of differential equations is a system of equations without derivatives that expresses relationships that would lead to the differential equations. Obviously, many equations can lead to the same differential equation because the derivative of a constant is zero. Additional information such as a fixed relationship at some point must be used to arrive at a unique solution to a differential equation.

Although many differential equations must be solved numerically, a differential equation should always be inspected and simplified or solved analytically if possible. A particularly simple form of a differential equation is the linear form

\[ u' = p(x)u + r(x) \]

with

\[ u(x_0) = u_0. \]

It is easy to see that the solution of this differential equation is

\[ u = e^{P(x)}u_0 + e^{P(x)} \int_{x_0}^{x} e^{-P(v)} r(v) dv, \]

where

\[ P(x) = \int_{x_0}^{x} p(w) dw. \]

For example, a solution to the equation \( u' = u + x \) that goes through the point \((x_0, u_0)\) is

\[ u = e^{x-x_0} \left( u_0 + \frac{1}{2} e^{-u+u_0} (x^2 - x_0^2) \right). \]

The trick that leads to this solution is to multiply both sides by the “integrating factor”, \( e^{P(x)} \), where \( P'(x) = p(x) \) in equation (9.4).

Another simple form of a differential equation is one that is separable:

\[ u' = p(x)/q(u) \]

with

\[ u(x_0) = u_0. \]

The solution of this differential equation is obtained by evaluating the two integrals in the equation

\[ \int_{u_0}^{u} q(v) dv = \int_{x_0}^{x} p(w) dw. \]
9 Solution of Differential Equations

Series Solutions of Differential Equations

A general approach to solving a differential equation analytically is to express the dependent variable as a power series in the independent variable:

\[ u = \sum_{j=0}^{\infty} c_j x^j. \]

From this series we have

\[ u' = \sum_{j=0}^{\infty} j c_j x^{j-1}, \]
\[ u'' = \sum_{j=0}^{\infty} j(j-1)c_j x^{j-2}, \]

and so on. Notice that the lower limit on the sums can be increased because the elements with factors of \( j \), \( j-1 \), and so on are zero.

In many cases, it is possible to determine values for the coefficients \( c_j \) that will provide a solution to the differential equation. The use of the series in this way is also called the method of undetermined coefficients.

As an example of the use of a power series, consider the solution of Legendre’s equation for inverse-\( r^2 \)-squared potential in a region of spherical symmetry:

\[ -\left((1 - x^2)u'\right)' = \lambda u, \tag{9.6} \]

for \(-1 < x < 1\). Rewriting this equation as

\[ u'' - x^2u'' - 2xu' + \lambda u = 0, \]

and substituting the power series in the appropriate places, after adjusting for the early zero terms in series for the derivatives, we have

\[ \sum_{j=0}^{\infty} \left( (j+1)(j+2)c_{j+2} - (j(j+1) + \lambda)c_j \right) x^j = 0. \tag{9.7} \]

In order for equation (9.7) to hold for all allowable \( x \), the coefficient of each term in the series must be zero. We therefore have the relationship

\[ c_{j+2} = \frac{j(j+1) - \lambda}{(j+1)(j+2)} c_j, \quad \text{for } j = 0, 1, 2, \ldots \tag{9.8} \]

Now, if the differential equation (9.6) has a solution that is a \( k \)th degree polynomial, then \( c_{j+2} = 0 \) for \( j > k - 1 \), and so from equation (9.8) it is clear that \( \lambda \) must be \( k(k+1) \). We also see from equation (9.8) that for \( k \) even, \( c_j = 0 \) for all odd values of \( j \); and for \( k \) odd, \( c_j = 0 \) for all even values of \( j \).

For each value of \( k \) and the corresponding \( \lambda \), we have a solution \( u_k(x) \) of the
differential equation (9.6). The solution \( u_k(x) \) is a \( k \)th degree polynomial. For example, if \( k = 4 \), that is, \( \lambda = 20 \), the solution is
\[
u_4(x) = c_0 + c_2x^2 + c_4x^4.
\]
Letting \( c_0 = 1 \), and using equation (9.8), we have the solution
\[
u_4(x) = 1 - 10x^2 + \frac{35}{3}x^4.
\]
Letting \( c_0 = \frac{3}{8} \), we obtain something more recognizable; indeed, for judicious choices of \( c_0 \) and \( c_1 \), we obtain the following solutions:
\[
u_0(x) = 1 \\
u_1(x) = x \\
u_2(x) = (3x^2 - 1)/2 \\
u_3(x) = (5x^3 - 3x)/2 \\
u_4(x) = (35x^4 - 30x^2 + 3)/8 \\
u_5(x) = (63x^5 - 70x^3 + 15x)/8
\]
These are the Legendre polynomials given in equations (8.26) on page 468 in Chapter ??, where they are derived in a different way.

Another example of an important differential equation that has a system of orthogonal polynomials as solutions is the Schrödinger equation describing a simple one-dimensional harmonic oscillator:
\[
\frac{d^2\psi}{dx^2} + \frac{8\pi^2m}{h^2} \left(E - \frac{1}{2}kx^2\right)\psi = 0,
\]
where \( \pi, m, h, E, \) and \( k \) are constants. Now if we let
\[
\gamma = \frac{8\pi^2mE}{h^2}
\]
and
\[
\beta^2 = \frac{4\pi^2mk}{h^2},
\]
we have
\[
\frac{d^2\psi}{dx^2} + (\epsilon - \beta^2x^2)\psi = 0.
\]
Next, making the change of variables \( \xi = \sqrt{\beta}x \), we have
\[
\frac{d^2\psi}{d\xi^2} + \left(1 - \xi^2 + \left(\frac{\epsilon}{\beta} - 1\right)\right)\psi = 0.
\]
Now, if we let \( \psi(\xi) = e^{-\xi^2/2}u(\xi) \), and take its second derivative we have \textit{Hermite’s equation}:
\[
\frac{d^2u}{d\xi^2} - 2\xi\frac{du}{d\xi} + \left(\frac{\epsilon}{\beta} - 1\right)u = 0.
\]
or, in simpler form,
\[
u'' - 2xu' + \lambda u = 0
\]
The solutions to this equation are the Hermite orthogonal polynomials given in equation (8.30) on page 470. In Exercise 8.7 you are asked to use the method of undetermined coefficients to obtain the solutions to Hermite’s equation, that is, to derive the Hermite polynomials.
Many differential equations that are useful models of physical systems cannot be solved in closed form. The process of solving the equations numerically involves the conversion of a calculus problem to an algebra problem. First, the continuous domain is discretized into a grid, and then the derivatives in the differential equations are approximated by finite differences on the grid. The system of differential equations becomes a system of algebraic equations.

By approximating the derivative $u'$ at $x$ by a finite difference, such as the forward difference
\[
\frac{u(x + h) - u(x)}{h},
\]
the backward difference
\[
\frac{u(x) - u(x - h)}{h},
\]
or the central difference
\[
\frac{u(x + h) - u(x - h)}{2h},
\]
we can approximate a differential equation by a difference equation. A difference equation is an algebraic equation. The step $h$ together with a single fixed starting value of $x$, say $x_0$, defines a one-dimensional grid, which we may index as $x_k$, where $x_k = x_0 + kh$.

9.1.1 Initial Value Problems in ODEs

As we mentioned, a differential equation does not have a unique solution unless additional constraints are placed on the solution. In very common applications of ODEs, the independent variable is time, and the differential equation expresses the time-rate of change. If a starting point is given, the ODE may have a unique solution. A differential equation together with a starting value is called an initial value problem, or IVP. This kind of ODE often arises from some kind of propagation problem in an open domain.

A first-order ODE with initial value condition is of the form
\[
\frac{du}{dx} = f(x, u), \quad u(x_0) = u_0.
\]

More generally, the first-order system is
\[
Au' = f(x, u), \quad u(x_0) = u_0, \quad \text{(9.9)}
\]
where $A$ is a matrix and $u, u'$ and $u_0$ are vectors.

The solution of the IVP is a function $u(x)$ such that $\frac{du}{dx} = f(x, u)$, and $u(x_0) = u_0$. 
Euler’s Method

The simplest approach to solving this problem is called Euler’s method. Euler’s method is not as accurate as other methods, and it is not recommended in practice, but we will consider it here because analysis of Euler’s method provides insight into general methods of solving ODEs. Euler’s method uses the forward finite difference equation:

\[ u_{k+1} = u_k + (x_{k+1} - x_k)f(x_k, u_k). \]

This is called a marching method, because we use this expression to march forward from \((x_0, u_0)\).

How well the finite difference approximates the derivative depends on the step size, \(x_{k+1} - x_k\), and how much the function actually changes over that interval. Although it may be appropriate to use smaller step sizes in regions of rapid change, we generally use a fixed step size,

\[ h = x_{k+1} - x_k, \]

that does not depend on the difference between \(f(x_{k+1})\) and \(f(x_k)\).

Figure 9.1. Euler’s Method for \(u' = au\)
9 Solution of Differential Equations

- Local error, or error per step.
  Let \( z(x) \) be the true solution of
  \[
  z' = f(x, z), \quad z(x_k) = u_k.
  \]
  Then, by Taylor’s theorem,
  \[
  z(x_{k+1}) = z(x_k) + z'(x_k)(x_{k+1} - x_k) + z''(x_k)(x_{k+1} - x_k)^2/2 + \cdots
  = u_k + f(x_k, u_k)h + O(h^2).
  \]
- Truncation error (error from true solution).
  \[
  |u(x_{k+1}) - (u(x_k) + hf(x_k, u_k))|.\]
- Global error, or discretization error.
  \[
  |u(x) - u_k|.
  \]
  Because there are \( k \) errors of order \( O(h^2) \) in computing \( u_k \), the best we would hope for is \( O(h) \). Of course this depends very much on the function \( u(x) \).
- Round-off error.
  The round-off error, as in any numerical computation, arises from the inability to represent an arbitrary real number in the set of computer numbers, \( \mathbb{F} \). Also, as in many numerical computations, the round-off error may interact with errors from the approximations used in the solution method. The cumulative effect of round-off errors can be significant in Euler’s method.

Euler’s method is a single-point method, that is, each step requires only one known point. It is an explicit method, that is, as it marches forward to a solution at \( x_{k+1} \) it does not use a value of the function beyond that at \( x_k \).

In summary, Euler’s method
- is a single-step method
- (a single finite difference method is used to advance the solution from \( k \) to \( k+1 \)),
- uses only one derivative function evaluation per step,
- has local truncation error \( O(h^2) \), and
- has global error \( O(h) \).

**Stability of Computational Methods**

**Implicit Methods**

An implicit method is one that uses a value of \( u_{k+1} \) in the difference equation used to evaluate \( u_{k+1} \). The implicit Euler’s method is
\[
 u_{k+1} = u_k + (x_{k+1} - x_k)f(x_{k+1}, u_{k+1}).
\]
Its error is essentially the same, but it may have better stability. In general, implicit methods tend to be more stable than explicit ones.
Runge-Kutta Methods

Use Maple to develop ... see Gander and Gruntz (1999) ... make exercises

The basic Runge-Kutta method is

1. Make a step of length $\alpha h$ from $x_k$ with Euler's method.
2. Use $f(x_k + \alpha h, u_{k+1}^{(1)})$ as an approximate slope, $s^{(1)}$.
3. Repeat for $s^{(2)}, \ldots, s^{(j)}$.
4. Combine all approximate slopes to obtain $u_{k+1}$.

Example, three stages:

$$
\begin{align*}
    s_1 &= hf(x_k, u_k) \\
    s_2 &= hf(x_k + \alpha_2 h, u_k + \beta_2 s_1) \\
    s_3 &= hf(x_k + \alpha_3 h, u_k + \beta_3 s_1 + \beta_3 s_2) \\
    u_{k+1} &= u_k + a_1 s_1 + a_2 s_2 + a_3 s_3.
\end{align*}
$$

There are several options within the general Runge-Kutta method:

- How many stages to use
- How long to make the steps (they can all be different)
- How to combine the information for the update

The quantities $\alpha_i, a_i$, and $\beta_{ij}$ are determined so as to make the finite difference equations agree with a Taylor series expansion up to a chosen number of terms. The largest power of $h$, for which the terms agree is called the order of the Runge-Kutta method.

The local error of a $p^{th}$ order Runge-Kutta method is $O(h^{p+1})$.

Two common ways of combining the slopes result in the methods called Runge-Kutta-Fehlberg and Runge-Kutta-Verner.

Runge-Kutta-Fehlberg, for example, chooses the slopes so as to give local error of $O(h^{p+1})$, and truncation error of $O(h^p)$, for a given $p$.

Other modifications of Runge-Kutta include adaptive step size selection. The step $h$ is rescaled depending on information that is accumulated during the solution up to any point.

Runge-Kutta methods are probably the most widely used methods.

Bulirsch-Stoer Method

- Richardson extrapolation (see page 65)
- Rational function extrapolation
- Modified midpoint method – errors contain only even powers of $h$. 
Multistep Methods

A multistep method uses information from \( x_k, x_{k-1}, x_{k-2}, \ldots \) to compute \( u_{k+1} \). The general form of the update is

\[
u_{k+1} = \sum_{i=0}^{m} a_i u_{k-i} + h \sum_{i=-1}^{m} b_i u'_{k-i}.
\]

This has \( m + 1 \) steps.

If \( b_{-1} = 0 \), the method is an explicit method; otherwise, it is an implicit method. As mentioned previously, implicit methods tend to be more stable than explicit ones.

The Adams multistep methods compute interpolating polynomials and then integrate them from \( x_k \) to \( x_{k+1} \) to obtain \( u_{k+1} \) from \( u_k \). These methods are also called Adams-Moulton methods.

Predictor-Corrector Methods

Starting with an Euler step,

\[
u_{k+1}^{(0)} = u_k + h f(x_k, u_k),
\]
a “predictor”, we form a “corrector”,

\[
u_{k+1}^{(j)} = u_k + \frac{h}{2} \left( f(x_k, u_k) + f(x_k + 1, u_{k+1}^{(j-1)}) \right).
\]

These methods work very well for relatively smooth functions.

An important side benefit of predictor-corrector methods is that they yield useful estimates of the local error in each step.

Stiff Systems

The system \( u' = f(x, u) \) is said to be stiff if some of the eigenvalues of the Jacobian matrix \( \partial f_i / \partial x_j \) are large, with negative real parts.

This is often the case for differential equations representing the behavior of physical systems such as chemical reactions proceeding to equilibrium where subspecies effectively complete their reaction in different epochs. An alternate model concerns discharging capacitors such that different parts of the system have widely varying decay rates (or time constants).

This definition of stiffness, based on the eigenvalues of the Jacobian matrix, is not very useful. Users typically identify stiff systems by the fact that numerical differential equation solvers are inefficient, or else they fail. The most common inefficiency is that a large number of evaluations of the functions \( f_i \) are required.

Implicit methods are used to solve stiff systems. The two most common classes of methods are implicit Adams-Moulton methods or Gear’s method, which uses backward differentiation formulas (BDF).
9.1.2 Boundary Value Problems in ODEs

In the boundary value problem (BVP) for ODEs, constraints on the dependent variables are given at certain values of the independent variable. For simplicity, we will assume the constraints only apply at two values of the independent variable, the “boundary”, $x_1$ and $x_2$. The constraints are the boundary values, and the problem is sometimes called a “two-point boundary value problem”. A first-order two-point boundary value problem can be written as

$$Au' = f(x,u), \quad B\begin{pmatrix} u(x_1) \\ u(x_2) \end{pmatrix} = b,$$  \hspace{1cm} (9.10)

where $A$ is an $n \times n$ matrix as before, $B$ is a matrix with $2n$ columns and a maximum of $n$ rows, and $\begin{pmatrix} u(x_1) \\ u(x_2) \end{pmatrix}$ is a $2n$-vector consisting of the $n$-vector of values of $u$ at $x_1$ and the $n$-vector of values of $u$ at $x_2$. In a simple case, $B$ is block diagonal composed of the $n_1 \times n$ matrix $B_1$ and the $n_2 \times n$ matrix $B_2$. Using this notation, $b$ is composed of the corresponding subvectors $b_1$ and $b_2$.

This kind of ODE often arises from some kind of propagation problem in a closed domain.

There are two general types of methods for the BVP. One is similar to the marching methods of the IVP. These methods are called shooting methods, because we start at one boundary and “shoot” at the other. We use the difference between the computed solution at the boundary and the known boundary condition to shoot again.

The other method is an equilibrium method that uses a finite difference approximation of the ODE at each point in the closed solution domain, including the boundaries. The system of coupled finite difference equations is solved simultaneously.

**Shooting Methods**

Let us consider the simple case in which $B$ in equation (9.10) is block diagonal and composed of the $n_1 \times n$ matrix $B_1$ and the $n_2 \times n$ matrix $B_2$, and $n_1 + n_2 = n$. This means that there are $n_1$ conditions on $u$ at $x_1$. This is equivalent to placing boundary values on exactly $n_1$ of the $u$’s and leaving the other $n_2 = n - n_1$ $u$’s as free values. We now choose a vector $v$ with $n_2$ elements that we can vary so that the $n$-vector $\begin{pmatrix} B_1u(x_1) \\ v \end{pmatrix}$ spans the $n_2$-dimensional subspace defined by the constraint $B_1u(x_1) = b_1$.

Now, for any fixed value of $v$, say $v^{(0)}$, we can use one of the methods such as Runge-Kutta, for an initial value problem discussed in Section 9.1.1.

When the marching method gets to $x_2$, however, it is unlikely that the boundary conditions $B_2u(x_2) = b_2$ will be satisfied.

We now define the function

...
Multiple shooting method

Phase plane (for ODEs)

Nullcline

\[ f(v^{(k)}) = b_2 - B_2 u(x_2), \]

which is the discrepancy in the boundary conditions at \( x_2 \) that results from our particular choice of \( v^{(k)} \).

The problem now becomes one of finding the zero of the function \( f \). We can use Newton’s method or one of the other methods discussed in Section 6.2 to find the value of \( v \) that results in the boundary condition at \( x_2 \) being satisfied. This can require many computations, because for each value \( v^{(k)} \) we must perform the integrations equivalent to solving an initial value problem. Three such integrations are illustrated in Figure 9.2.

\[ \begin{align*}
&u^{(0)}, u^{(1)}, u^{(2)}, \ldots \text{ may yield ill-conditioned initial values} \\
&\text{problems whose solutions diverge over the interval } [x_1, x_2]. \text{ One way of addressing this problem is by use of a multiple shooting method, in which the the interval } [x_1, x_2] \text{ is divided into two or more subintervals and the problem becomes two or more two-point boundary problems, in which the constraints at the interior boundaries are determined by the requirement that the derivatives are continuous across the boundaries.} \\
\end{align*} \]

9.1.3 Phase Planes in ODEs


“Phase line analysis” Mooney and Swift (1999), p 266

Nullclines
9.2 Partial Differential Equations

Partial differential equations are generally much more difficult to solve than ordinary differential equations. Because differential equations are most often used as models changes of some quantity in time or in ordinary three-dimensional space, we often define the equations in terms of time \( t \), and \( x-y-z \) space variables (actually, often only as \( x-y \) space). The \textit{domain} of a PDE is the area over which the PDE is defined. Any known values on the boundary of the domain are \textit{boundary conditions}.

The most commonly used partial differential equations are of order 2.

9.2.1 Classification of PDEs

Classes of Physical Problems Modeled by PDEs

Physical problems for which PDEs are useful models can be classified as

- equilibrium problems, which are \textit{steady state} problems in closed spatial domains,
- eigenproblems, which are special equilibrium problems for which a solution exists only for certain values of a parameter in the problem,
- propagation problems, which are \textit{initial value} problems in open domains.

Methods of solution are often dependent on the type of the PDE. The types of PDEs arise from the nature of the physical problem, as well as from the assumptions or approximations we make in modeling the physical problems. Partial differential equations of order 2 in two variables can be used as prototypes for three important classes of PDEs.

- elliptical equations
  \[
  a \frac{\partial^2 u}{\partial x^2} + \beta \frac{\partial^2 u}{\partial y^2} = g \quad (9.11)
  \]
  If \( g = 0 \), the equation is \textit{homogeneous} and is called the \textit{Laplace equation}; for general \( g \), which is a function of \( x \) and \( y \), it is called the \textit{Poisson equation}. An equation of this form governs the diffusion of energy or mass as the domain reaches equilibrium. The Laplace equation represents a steady diffusion and the Poisson equation models an unsteady diffusion, that is, diffusion with a source or sink.

- parabolic equations
  \[
  \frac{\partial u}{\partial t} = a \frac{\partial^2 u}{\partial x^2} + \beta \frac{\partial^2 u}{\partial y^2} \quad (9.12)
  \]
  This is called the \textit{heat equation} or the \textit{diffusion equation}, because it governs the diffusion or propagation of energy (heat) or mass in time.
492 9 Solution of Differential Equations

- hyperbolic equations

\[
\frac{\partial^2 u}{\partial t^2} = \alpha \frac{\partial^2 u}{\partial x^2} + \beta \frac{\partial^2 u}{\partial y^2}
\]

(9.13)

This is called the wave equation. It governs diffusion or propagation with rates that vary in time.

This classification follows from the analogous conic sections in the variables \(x, y,\) and \(z:\) \(ax^2 + by^2 = c,\) \(z = ax^2 + by^2,\) and \(z^2 = ax^2 + by^2.\)

The coefficients \(\alpha\) and \(\beta\) in the expressions above may be functions of \(x, y,\) and \(t.\) If so, various placements of the coefficients with respect to the differential operators may result in different forms within the same general class. For example, for the diffusion equation, the heat equation follows Fickian diffusion, and the form may be

\[
\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\alpha(x, y) \frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial x}\left(\beta(x, y) \frac{\partial u}{\partial x}\right).
\]

In Fokker-Planck diffusion, the form may be

\[
\frac{\partial u}{\partial t} = \frac{\partial^2}{\partial x^2}\left(\alpha(x, y)u\right) + \frac{\partial}{\partial x}\left(\beta(x, y)u\right).
\]

Boundary Conditions

For equilibrium problems with closed domains, we can identify three types of boundary conditions:

- Dirichlet boundary condition, in which the value of the function \(u\) is specified on the boundary,
- Neumann boundary condition, also called flux boundary condition, in which the value of the derivative normal to the boundary is specified, and
- mixed boundary condition, also called Robin or radiation boundary condition, in which a linear combination of the function and its derivative normal to the boundary is specified.

Any of the types of boundary conditions listed above may be periodic.

Well-Posed Problems

A physical problem is said to be well-posed if its solution exists, is unique, and depends continuously on the boundary data or the initial data.


9.2.2 Phase Planes in PDEs

use turning point problems, \( u'' = f(u) \) to introduce phase plane techniques
orbits:
- trapped
- singular
- untrapped
- damped
- point attractor
- line attractor
- control problem

\[
\frac{dx}{dt} = f(x, t, u(t))
\]

9.2.3 Methods of Solution of PDEs

The methods of solution of PDEs, as those for solution of ODEs, basically involve the formulation of a calculus problem as an algebraic problem.

**Finite Difference Methods**

There are several possibilities for setting up finite difference methods on structured grids for solving PDEs. Figure 9.3 shows some possibilities. Representations such as those in the figure are called “stencils”.

**Figure 9.3.** Structured Grids in Two Dimensions. Solution is Known at Solid Dots

**Finite Element Methods**

Reddy (1993) and Smith and Griffiths (1998) provide good introductions for programming the finite element method. The book by Smith and Griffiths has many programs and subroutine libraries written in FORTRAN 90. The
programs and most subroutines are listed in full in the text, and are all available on the Web. They give examples from a wide variety of applications, including classical structural analysis, elasticity and plasticity, steady state and transient fluid flow, linear and non-linear solid dynamics and construction processes in geomechanics.

Unstructured Grids

There are two unstructured triangulation algorithms commonly used methods to generate unstructured grids: the advancing front method, which has a heuristic simplicity, and the Delaunay triangulation, which is more efficient and more robust. The Delaunay triangulation also has some very interesting properties. It provides a unique connectivity for a given set of grid nodes. Each node is surrounded by a convex polygon, the Voronoi region, that contains all points of the plane closer to this node than to any other. The set of edges of the polygons on which points are equidistant to two nodes is called the Voronoi diagram. A unique triangulation is obtained when joining all nodes that share an edge in the Voronoi diagram, that is, their Voronoi regions border each other. For each triangle there is an associated node of the Voronoi diagram. The edges of the Voronoi diagram joined at that node are the medians of the edges of the triangle, and so the Voronoi node is the circumcenter of the triangle. Because the three regions meeting at the node are convex, there cannot be another vertex within the circle.


***** shear layers in Navier-Stokes

Resolving shear layers in Navier-Stokes calculations may require cell aspect ratios of up to 1:100000. A problem encountered when stretching a mesh is the control of large angles in the grid in order to keep the error in the solution bounded. Rather than using a Delaunay triangulation we use a triangulation that minimizes the maximum angle. In addition, as the aspect ratios grow larger, situations may arise where the Delaunay triangulation is entirely inappropriate. One way to remedy this situation is to use a constrained Delaunay triangulation such that nodes on either side of the viscous layer cannot be connected.

Another hybrid method was proposed by Löhner (1993) Matching semi-structured and unstructured grids for Navier-Stokes calculations. AIAA-paper, (93-3348-CP), 1993. The first step is to generate structured collar meshes that can possibly overlap and overturn around all components. In a second step elements are clipped from these meshes if they overlap with another element, are overturned and exhibit a negative element surface or have an aspect ratio of less than one. The open regions of the domain are triangulated with an isotropic process in a third step.
Multigrid Methods

Multigrid methods use a nested sequence of iterations over ever-finer grids to obtain a numerical solution of a differential equation. The problem is first solved on a coarse grid, and that solution is interpolated to a finer grid. These interpolated values are then used as starting values for a numerical solution on the finer grid. The solution on the finer grid is then interpolated to an even finer grid, and the process is continued. See Golub and Ortega (1993).

Domain Decomposition

If the domain of the differential equations can be partitioned in such a way that the grids or elements of the partitions can be used separately, the solution can be computed in parallel. A general discussion of domain decomposition methods with many examples is given by Quarteroni and Valli (1999).

Meshfree Methods

Another approach is to modify the internal structure of the traditional finite element method to make it more flexible and robust. Methods based on this approach are called “meshfree” or “meshless” methods. Meshfree methods may be particularly useful in applications involving moving discontinuities such as cracks and shocks, multi-scale resolution, and large material distortions.

9.2.4 Higher Dimensions

**** discuss problems

9.3 Stochastic Differential Equations

Although a growth model such as

\[ \frac{du}{dt} = s(t)u(t) \hspace{1cm} (9.14) \]
may be a good ideal model, in applications either there is some noise in the process, possibly just due to measurement error, or, more likely, there are extraneous factors affected the growth that are not included in the model. These factors may not be included in the model because we are unaware of them, or they may be excluded from the model for simplicity.

If we allow for some randomness in the model it may be a more realistic representation of the situation. The random component can account for various extraneous factors so long as their net average effect does not change the simple model without the random component. For example, suppose the rate in (9.14) is modeled as

\[ s(t) = r(t) + Z(t), \]

where \( Z(t) \) is a random variable whose distribution may change in time. A differential equation with a random component is called a stochastic differential equation.

We can write equation (9.14) in a slightly different form to express the deterministic and the random components:

\[
\frac{du}{dt} = a(u, t) + b(u, t)W(t),
\]

(9.15)

where \( W(t) \) represents some random variable whose distribution possibly changes in time. As usual, we will use upper case letters to denote random variables.

We will usually assume that the expectation of the additive random component is 0:

\[ \mathbb{E}(W(t_1)) = 0. \]

==

Rewrite in differential form:

\[ du = a(u, t)dt + b(u, t)d\mu(t), \]

where

\[ d\mu(t) = W(t)dt. \]

The solution is

\[ u(t) = u(0) + a(u, s)ds + b(u, s)d\mu(s), \]
9.4 Software for Differential Equations

Before submitting a differential equation to a computer software package for solution, the user should inspect the equation for possible problems, such as singularities, or extreme differences in the scales of the variables. Rewriting the equations will sometimes reveal problems that are not apparent in some other formulations of the same problem. Sometimes use of a software package for symbolic manipulation such as Maple or Mathematica can help to simplify a differential equation.

9.4.1 Software for ODEs

There is a wide range of software for solving differential equations. The Guide to Available Mathematical Software (GAMS) (see the bibliography) is a good source of information about software.

Although general systems for interactive scientific computations such as Matlab and R provide some capabilities for numerical solution of ODEs, it is usually necessary to write Fortran or C programs to call library routines in one of those languages. A number of routines for solution of differential equations have been published in ACM Transactions on Mathematical Software and are available in the Collected Algorithms of the ACM, CALGO (see page 651). Most of these are in some version of Fortran. An example is the set of routines rksuite_90 that implement Runge-Kutta methods in Fortran 90 for solving initial value problems (Brankin and Gladwell, 1997). Kamel, Enright, and Ma (1993) describe an expert system to select solvers for initial value problems.

The book edited by Arge, Bruaset, and Langtangen (1997) contains a number of articles on software for differential equations. Some of the articles describe specific packages, but others discuss general principles of design of software for differential equations and other problems, with an emphasis on object-oriented design.

Most software for ODEs handle systems of first order equations. For systems of higher order, it is generally necessary first to use the technique suggested in equations (9.3) on page 481 to reduce the system to one of first order equations.

IMSL Software for ODEs

The IMSL Libraries provide a number of routines and options for solving ODEs. The four basic routines are

- ivprk IVP using a Runge-Kutta-Verner method.
- ivpag IVP using an Adams or a Gear method.
- bvprfd BVP using a finite-difference method.
- bvpmv BVP using a multiple-shooting method.
The IVP routines use marching methods. Library subprograms for such methods are often designed so that the control is returned to the user after each step. IMSL routines that are intended to be used in this way have an input argument called \texttt{ido} to indicate the stage of the computations. The usual values for \texttt{ido} and their meanings are

\begin{tabular}{|c|l|}
\hline
\texttt{ido} & state \\
1 & initial entry \\
2 & normal re-entry \\
3 & final call \\
\hline
\end{tabular}

In both \texttt{ivprk} and \texttt{ivpag}, if \texttt{ido} is 1, various initializations such as allocating workspace from a stack are performed; if \texttt{ido} is 2, the routines assume that the initializations have been performed, and so just computes one additional step in the marching method; if \texttt{ido} is 3, the routines perform some final operations such as deallocating workspace. These routines also allow for continuation even if certain error conditions (such as exceeding the maximum number of function evaluations) have occurred.

The references to the IVP routines are

\begin{verbatim}
CALL IVPRK (IDO, N, FCN, T, TEND, TOL, PARAM, Y)
CALL IVPAG (IDO, N, FCN, FCNJ, A, T, TEND, TOL, PARAM, Y)
\end{verbatim}

The meanings of most of the arguments are obvious. As in many areas of application of ODEs, “y” is used for the dependent variable, and it is considered to be a function of “t”. So in the argument list, N is the number of equations; FCN is a user-supplied subroutine to evaluate the functions; T is input as the beginning value of the independent variable and is output as its ending value; \texttt{TEND} (input) is the point of the independent value where the solution is to be obtained; \texttt{TOL} (input) is a tolerance for error control (an attempt is made to control the norm of the error in the step so that the global error is proportional to \texttt{TOL}); and \texttt{Y} (output) is the solution at \texttt{TEND}.

The user can set various parameters to control the computations in the argument \texttt{PARAM}, which is a vector of length 50. All of the settings have default values, which the user can specify by assigning the corresponding value in \texttt{PARAM} to 0. The routine \texttt{ivpag} includes additional arguments to compute the Jacobian and to specify a matrix of coefficients if the user chooses an implicit method.

The routines \texttt{bvpfd} and \texttt{bvpms} solve the BVP for systems of the form

\[ u'(x) = f(x, u), \]

subject to the conditions

\[ g_1(u_1(a), \ldots, u_n(a), u_1(b), \ldots, u_n(b)) = 0. \]

The references to the BVP routines are somewhat more complicated:
CALL BVPFD (FCNEQN, FCNJAC, FCNBC, FCNPEQ, FCNPBC, N, 
NLEFT, NCUPBC, TLEFT, TRIGHT, PISTEP, TOL, 
NINIT, TINIT, YINIT, LDYINI, LINEAR, PRINT, 
MXGRID, NFINAL, TFINAL, YFINAL, LDYFIN, ERREST)

CALL BVPMS (FCNEQN, FCNJAC, FCNBC, NEQNS, TLEFT, TRIGHT, 
DTOL, BTOL, MAXIT, NINIT, TINIT, YINIT, LDYINI, 
NMAX, NFINAL, TFINAL, YFINAL, LDYFIN)

These routines are designed to be called only once for the given problem; they do not implement marching methods. The fcn** parameters are user-supplied subroutines to compute the functions, the derivatives of the functions, or boundary conditions.

Examples of Use of the IMSL Libraries

As an example of the use of an IMSL routine we will use ivpag to solve the mildly stiff problem

\[
\begin{align*}
    u_1' &= -u_1 - u_1 u_2 + 294 u_2 \\
    u_2' &= -3 u_2 + 0.01020408 (1 - u_2) u_1
\end{align*}
\]

(9.16)

with initial values

\[
\begin{align*}
    u_1(0) &= 1 \\
    u_2(0) &= 0
\end{align*}
\]

This problem was given by Enright and Pryce (1987) in a set of test problems.

A program and a required function to solve this problem are shown in Figures 9.4 and 9.5. A function fcnj must also be declared, but it is not used because of the method selected.

Note that the parameter vector param is first set to zero with IMSL routine sset, to indicate that default values are to be used for most parameters. Then other parameters, such as the error control and the method are set by various elements of param.

The last call to ivpag with ido set to 3 deallocates IMSL workspace allocated on the first call to ivpag. Because for the IMSL differential equation solvers, the only action performed in a call with ido set to 3 is to deallocate workspace, it, in this example program that ends after solving a single problem, is not necessary to release the workspace. The call to release workspace would be needed if the program included further calls to IMSL routines.

Boundary value problems are somewhat more complicated, and consequently, the software generally has several more arguments and requires more setup by the user. In the next example we solve the third-order boundary value problem,
integer mxparm, n
parameter (mxparm=50, n=2)
integer ido, istep, nout
real a(1,1), param(mxparm), x, xend, tol, u(n)
c the array a(*) is not used.
external ivpag, sset, fcn, fcnj
c set initial conditions
x = 0.0
u(1) = 1.0
u(2) = 0.0
c set error tolerance
tol = 0.001
c set param to defaults
call sset (mxparm, 0.0, param, 1)
c select absolute error control
param(10) = 1
param(12) = 2
param(13) = 2
c print header
print *,  ' istep time u1 u2')
ido = 1
istep = 0
10 continue
istep = istep + 24
tend = istep
call ivpag (ido, n, fcn, fcnj, a, x, xend, tol, param, u)
if (istep .le. 240) then
  print *, istep/24, x, u
end if
final call to release workspace
if (istep .eq. 240) ido = 3
go to 10
end if
show number of function calls.
print *,  ' Number of fcn calls:', param(35)
end

Figure 9.4. IMSL Fortran 77 Program to Solve a Mildly Stiff IVP (9.16)

\[
\frac{d^3 u}{dx^3} - 2\frac{d^2 u}{dx^2} + u' - u = \sin x, \quad (9.17)
\]

with boundary conditions
\[
\begin{align*}
  u(0) &= u(2\pi) \\
  u'(0) &= 1 \\
  u'(2\pi) &= 1
\end{align*}
\]
9.4 Software for Differential Equations

subroutine fcn (n, x, u, uprime)
  c specifications for arguments
  integer n
  real x, u(n), uprime(n)
  c specifications for save variables
  real ak1, ak2, ak3
  save ak1, ak2, ak3
  c
data ak1, ak2, ak3/294.0e0, 3.0e0, 0.01020408e0/
c  uprime(1) = -u(1) - u(1)*u(2) + ak1*u(2)
  uprime(2) = -ak2*u(2) + ak3*(1.0e0-u(2))*u(1)
return
end

Figure 9.5. The Fortran Subprogram to Define the IVP (9.16)

(The solution of this equation is easily seen to be $u = \sin x$.) First of all, the problem must be reduced to a first-order system. This is done as indicated in Equations (9.3). We have the system

\[
\begin{align*}
    u'_{1} &= u_{2} \\
    u'_{2} &= u_{3} \\
    u'_{3} &= 2u_{3} - u_{2} + u_{1} + \sin t
\end{align*}
\]

and boundary conditions

\[
\begin{align*}
    u_{1}(0) &= u_{1}(2\pi) \\
    u_{2}(0) &= 1 \\
    u_{2}(2\pi) &= 1
\end{align*}
\]

Notice that there is one boundary condition coupling the left and right endpoints.

A program and the required functions to solve this problem are shown in Figures 9.6 and 9.7.
9 Solution of Differential Equations

```fortran
! specifications for parameters
integer ldufin, lduini, mxgrid, neqns, ninit
parameter (mxgrid=45, neqns=3, ninit=10, ldufin=neqns,
  & lduini=neqns)
! specifications for local variables
integer i, j, ncupbc, nfinal, nleft, nout
real errest(neqns), pistep, xfinal(mxgrid), xinit(ninit),
  & xleft, tol, xright, ufinal(ldufin,mxgrid),
  & uinit(lduini,ninit)
logical linear, print
! specifications for externals
external bvpfd, sset, const, fcnbc, fcneqn, fcnjac

! set parameters
nleft = 1
ncupbc = 1
tol = 0.001
xleft = 0.0
xright = 2.0*const('pi')
pistep = 0.0
print = .false.
linear = .true.

! define tinit
do 10 i=1, ninit
   xinit(i) = xleft + (i-1)*(xright-xleft)/float(ninit-1)
10 continue

! set uinit to zero
do 20 i=1, ninit
   call sset (neqns, 0.0, uinit(1,i), 1)
20 continue

! solve problem
call bvpfd (fcneqn, fcnjac, fcnbc, fcneqn, fcnbc, neqns, nleft,
  & ncupbc, xleft, xright, pistep, tol, ninit, xinit,
  & uinit, lduini, linear, print, mxgrid, nfinal,
  & xfinal, ufinal, ldufin, errest)

! print results
print *, ' i  x  u1  u2  u3'
print *, i, xfinal(i), (ufinal(j,i),j=1,neqns)
end
```

Figure 9.6. IMSL Fortran Program to Solve a BVP (9.17)
subroutine fcneqn (neqns, x, u, p, dudx)
c specifications for arguments
integer neqns
real x, p, u(neqns), dudx(neqns)
c specifications for intrinsics
intrinsic sin
real sin
c define pde
dudx(1) = u(2)
dudx(2) = u(3)
dudx(3) = 2.0*u(3) - u(2) + u(1) + sin(x)
return
der
end
subroutine fcnjac (neqns, x, u, p, dupdu)
c specifications for arguments
integer neqns
real x, p, u(neqns), dupdu(neqns,neqns)
c define d(dudx)/du
dupdu(1,1) = 0.0
dupdu(1,2) = 1.0

dupdu(1,3) = 0.0
dupdu(2,1) = 0.0
dupdu(2,2) = 0.0
dupdu(2,3) = 1.0
dupdu(3,1) = 1.0
dupdu(3,2) = -1.0
dupdu(3,3) = 2.0
return
der
end
subroutine fcnbc (neqns, uleft, uright, p, f)
c specifications for arguments
integer neqns
real p, uleft(neqns), uright(neqns), f(neqns)
c define boundary conditions
f(1) = uleft(2) - 1.0
f(2) = uleft(1) - uright(1)
f(3) = uright(2) - 1.0
return
der
end

Figure 9.7. Fortran Subprograms Used to Define the BVP (9.17)
Software for Symbolic Solution of Differential Equations

Some simple differential equations can be solved analytically. The software packages for symbolic manipulation such as Maple and Mathematica can solve simple differential equations, and as mentioned above can also be used to simplify differential equations for use in numerical solvers. Both Maple and Mathematica will also provide numerical solutions.

In Maple the main function for solving ODEs is \texttt{dsolve}. For a system of ODEs, \( u' = f(x, u) \), the solution is returned either as equations in \( u(x) \) and \( x \) or in parametric form \( x = g(t); u(x) = g(t) \). The parameter is represented in Maple as \( \_T \). Any arbitrary constants are represented as \( \_C1, \_C2 \), and so on. The differential operator for expressions is \texttt{diff}, the first argument of which is an algebraic expression, and the subsequent arguments are variables of differentiation. If more than one variable of differentiation is specified the expression is differentiated sequentially. The sequence constructor \$\$ allows specification of higher-order derivatives. For example, given the initializations

\[
x := (x1, x2);
f(x) := 2*x1 + x1*exp(2*x2);
\]

then

\[
diff(f(x), x1);
\]
yields

\[
2 + e^{2x2},
\]

\[
diff(f(x), x2\$2);
\]
yields

\[
4x1e^{2x2},
\]

and

\[
diff(f(x), x1, x2);
\]
yields

\[
2e^{2x2}.
\]

The \texttt{diff} function has a user interface that will call the user’s own differentiation functions. If the procedure ‘\texttt{diff/f}’ is defined then the function call \texttt{diff(f(x, y, z), y)} will invoke ‘\texttt{diff/f}(x, y, z, y)’ to compute the derivative.

The expression in the \texttt{dsolve} function generally involves one or more instances of \texttt{diff}. For a system of ODEs, \( u' = f(x, u) \),

\[
dsolve(diff(u(x),x) = f(x,u), u(x));
\]

the solution is returned either as equations in \( u(x) \) and \( x \) or in parametric form \( x = g(t); u(x) = g(t) \). The parameter is represented in Maple as \( \_T \).

A numerical solution for given initial or boundary conditions can be specified with the keyword \texttt{numeric} in \texttt{dsolve}. When this keyword is specified,
the user must give a name for Maple to define as a procedure to evaluate the solution; for example, to solve the system (9.16), on page 499, whose solution using the IMSL routines was shown in Figures 9.4 and 9.5, the Maple statement would be

```maple
u := dsolve({diff(u1(x),x) = -u1(x) - u1(x)*u2(x) + 249*u2(x),
              diff(u2(x),x) = -3*u2(x) + 0.01020408*(1-u2(x))*u1(x),
              u1(0) = 1, u2(0) = 0}, {u1(x), u2(x)}, numeric);
```

The solution at \(x = 1\) is specified by

```maple
u(1);
```

More generally, the solution from \(x_0\) to \(x_n\) in steps of \(h\) can be printed by

```maple
for tt from x0 by h to xn do u(tt) od;
```

The procedure, which uses a Fehlberg fourth-fifth order Runge-Kutta method, cannot evaluate the solution as far as \(x = 24\) and beyond, as was done in Figure 9.4. Because the equations are stiff, it is necessary to take shorter steps and explicitly update the initial conditions in the `dsolve` statement.

Maple has several functions for working with differential equations in the `DEtools` package. There are several convenient routines in this package for plotting solutions of differential equations. The syntax is somewhat different from the `numeric` option in `dsolve`; however. For example, to plot the solution of system (9.16) from \(x_0\) to \(x_n\), after initializing the variables \(x_0\) and \(x_n\), we would use

```maple
DEplot2([diff(u1(x),x) = -u1(x) - u1(x)*u2(x) + 249*u2(x),
          diff(u2(x),x) = -3*u2(x) + 0.01020408*(1-u2(x))*u1(x)],
         [u1(x), u2(x)], x0..xn, {[u1(0) = 1, u2(0) = 0]});
```

### 9.4.2 Software for PDEs

Software for the solution of PDEs is often designed for only a particular class of problems. One of the more difficult aspects of a PDE problem is the definition of the domain. For this reason, software in Fortran or C tends to be relatively difficult to use. Some software packages allow the user to construct the domain interactively.

One of the more widely used packages for PDEs is Nastran, developed by NASA. Nastran uses a finite element approach. There are some commercial versions of Nastran, such as NE/Nastran, that generally have simpler user interfaces than the Nastran available from NASA. ANSYS is a widely used package with a problem-oriented user interface for applications in a range of areas including statics, mode frequency, stability analysis, heat transfer, magnetostatics, coupled field analysis, and modeling (see Moaveni, 1999). ANSYS also uses finite element analysis. Other finite element packages include FEMAP, Hypermesh, and PATRAN.
Machiels and Deville (1997) describe an object-oriented programming approach in Fortran 95 for the solution of PDEs.

Blom, Trompert, and Verwer (1996) and Blom and Verwer (1996) give Fortran programs that use adaptive-grid solvers for PDEs in 2D and 3D. The programs are vectorizable, and the authors show how to tune them for vector computers.

Diffpack is a library of C++ classes for numerical solution of partial differential equations. Langtangen (1999) describes Diffpack Information on Diffpack is available at

http://www.nobjects.com/Diffpack

An important area of application of PDEs is the analysis of fluid flow, computational fluid dynamics or CFD. There is a wealth of software specifically designed for CFD problems. The URL

http://icemcfd.com/cfd/CFDcodes.html

lists approximately 75 commercial CFD codes and approximately 50 public domain or freeware codes.

9.4.3 Testbeds for Software for Differential Equations

Enright and Pryce (1987) describe a set of initial value problems, including both nonstiff and stiff ones, for assessing the performance of software for IVP ODEs.

Another set of initial value problems from several application fields has been assembled by Walter Lioen and Jacques de Swart, of the Centre for Mathematics and Computer Science. It is available at

http://www.cwi.nl/cwi/projects/IVPtestset/

The test set consists of a descriptive part and a software part. The first part describes test problems and reports on the behavior of a few state-of-the-art solvers when applied to these problems. The software serves as a platform to test the performance of a solver on a particular test problem.

9.5 Differential Equation Models

In the traditional physical sciences, differential equation models are often fit by considering only the minimal number of observations necessary to fit the mathematical model. In most cases, this means a minimal number of initial conditions or of combination of boundary values and initial values. Because these models ignore real-world friction and a host of ambient conditions, the use of only the minimal number of constraints in order to fit the model is often quite optimistic.
When additional observations are available, they can and should be used in fitting the differential equations. Often these additional observations reveal the failures of the minimal set of constraints in fitting the models. The problem of fitting differential equation models is similar to the problems we discuss in Chapter ??.

When only a small number of observations are available, a common method for fitting differential equation models is to choose a variety of starting points, simulate the process with the differential equation, then adjust the starting points and try again.

### 9.5.1 Simulations Using Differential Equation Models

Many problems in science can be studied by modeling the process with differential equations that have variable parameters. For example, we may wish to study the population dynamics of two species of animals, where one species preys on the other species. Consider a simple problem involving rabbits and foxes, in which we assume that the only food source for the predator (fox) is the prey (rabbit). We will lump together all other reasons for any decline in the rabbit population. Let \( r \) be the density of rabbits in a given geographic region and let \( f \) be the density of foxes. In the absence of any predator-prey interaction, the rabbits would increase at a rate proportional to their number, and the foxes would die of starvation at a rate proportional to their number. We have

\[
\begin{align*}
    r' &= \gamma r \\
    f' &= -\rho f,
\end{align*}
\]

for some values of \( \gamma \) and \( \rho \). As a first approximation in an analysis, assume the rate at which the rabbits are eaten by the foxes is \( \sigma rf \), and the rate at which the foxes increase, because they are eating the rabbits, is \( \tau rf \), for some values of \( \rho \) and \( \tau \). All of the parameters are nonnegative. So, the model to be solved is

\[
\begin{align*}
    r' &= \gamma r + \sigma rf \\
    f' &= -\rho f + \tau rf
\end{align*}
\]

The use of this kind of system of differential equations to model growth of interacting populations was developed separately by Alfred Lotka and Vito Volterra in the 1920’s, and so are called Lotka-Volterra differential equations. The model is called a predator-prey model, but it can also be used to study other entities whose existence are mutually dependent. Notice that the derivatives do not vary in time. Models such as this, in which the derivatives are taken with respect to time, but in which time does not appear in the equations are called autonomous.

For certain values of these parameters in equations (9.18) and for some certain initial conditions, a stable steady state will occur, in which there is a
sustainable balance of the numbers of rabbits and foxes. On the other hand, for other relationships among the parameters or for certain initial conditions one or both populations will die out. In this simple model, if the conditions are such that the foxes die, then the rabbits will increase without bound. Many solutions in which the populations remain bounded but positive are highly oscillatory. Edelstein-Keshet (1988) and Mooney and Swift (1999) point out several inadequacies of the Lotka-Volterra equations as models of populations; for example, the cross-product term, $rf$, is not at all realistic if we assume the animals possess any intelligence to aid in preying or in avoiding the predator species.

One of the most obvious corrections to the equations (9.18) is to recognize the fact that the number of rabbits each fox eats has an upper bound independent of the number of rabbits present. Let the number of rabbits each fox eats per time period be $\min(\kappa, \alpha r)$, so the rate at which the rabbits are eaten by the foxes is $\min(\kappa, \alpha r)f$. The death rate of the foxes due to not having enough rabbits to eat is also unrealistic. Ivlev (1961) suggested a more realistic term may be $\exp(-\beta r/\kappa f)$ (Ivlev, V. S., 1961, Experimental Ecology of the Feeding of Fishes, Yale University Press, New Haven). We also introduce a term for the natural growth in the population of the foxes. We now have the reformulated system

\begin{align*}
r' &= \gamma_1 r - \min(\kappa, \alpha r)f \\
f' &= \gamma_2 f - e^{-\beta r/\kappa f} f. \tag{9.19}
\end{align*}

The R code to solve the modified Lotka-Volterra equations is shown in Figure 9.8. (Unfortunately, due to design and implementation problems, the R code cannot pass all of the parameters correctly, and the restart feature does not work beyond the second invocation. The workaround is to put the parameter assignments in the function definition and to use the full argument sequence in all calls.)
9.5 Differential Equation Models

Solutions for two different sets of parameters and starting values are shown in Figure 9.9. The Lotka-Volterra equations are very sensitive to the parameters and the starting values.

Figure 9.9. Modified Lotka-Volterra Model of Two Competing Species

The fox/rabbit (or coyote/rabbit, or lynx/hare) predator-prey example is one of the most common for illustrating simple population dynamics. Edelstein-Keshet (1988) and Rice (1993) discuss this model and give several variations of it.

***** discuss other simulations (without data) *****

n-body etc.

9.5.2 Qualitative Analysis of Differential Equation Models

vector fields Edelstein-Keshet (1988), ch 5

phase plane

9.5.3 Fitting Differential Equation Models with Observed Data

Predator-Prey Models

In Section 9.5.1, we discuss the solution of a system of differential equations that model the abundance of two species, a predator and a prey. The solutions of the equation indicated what to expect about the abundance of the species,
given different initial conditions and different rates of natural growth and consumption.

In this section we consider the same problem again, assuming we have some data on the abundance of the species.

Compartmental Models


Diffusion Models

product diffusion Bass (1969)

\[
\frac{du}{dt} = \left( p + \frac{qu}{m} \right) (m - u),
\]

(9.20)

where \( u(t) \) is the number of units of a product sold at time \( t \), \( m \) is the potential market for the product, and \( p \) and \( q \) are parameters controlling the rate of diffusion.

heat equation ... Fickian diffusion:

\[
\frac{\partial u}{\partial t} - \frac{\partial}{\partial x}\left( \theta(x) \frac{\partial u}{\partial x} \right) = f(x,t).
\]

fitting diffusion coefficients O’Sullivan and Wong (1987) represented \( \theta(x) \) as \( \sum_{j=1}^{p} \theta_j B_j(x) \), where the \( B_j \) form a B-spline basis, and estimated the \( \theta_j \).

Animal movements ... Fokker-Planck diffusion:

\[
\frac{\partial u}{\partial t} - \frac{\partial^2}{\partial x^2} (\mu(x)u) = f(x,t).
\]


Exercises

9.1. a) For the initial value problem, \( y' = -y, \ y(0) = 1 \), use Euler’s method with \( h = \frac{1}{3} \) to solve the differential equation over the interval \([0, 1]\).

(Your solution should give values of \( y \) at three points.)

b) Now use a standard routine to solve this problem over the same interval, but using a much finer grid. Compare your solution to \( e^{-t} \).
9.2. Consider the Robertson problem, which is a stiff system describing a chemical reaction:

\[ \begin{align*}
    y_1' &= -c_1 y_1 + c_2 y_2 y_3 \\
    y_2' &= -y_1' - y_3' \\
    y_3' &= c_3 y_2^2
\end{align*} \]

with initial conditions

\[ \begin{align*}
    y_1(0) &= 1 \\
    y_2(0) &= 0 \\
    y_3(0) &= 0.
\end{align*} \]

As might be guessed, the stability is very dependent on the values of the constants, and whether a solution exists depends on the starting values. Use

\[ \begin{align*}
    c_1 &= 0.04 \\
    c_2 &= 10^4 \\
    c_3 &= 3 \times 10^7
\end{align*} \]

and integrate to time equal to 10.

a) Use an Euler method and describe your results.
b) Use a Runge-Kutta method and describe your results.
c) Use Gear's BDF method and describe your results.
d) What term is involved in an eigenvalue of the Jacobian matrix \( \partial y_i / \partial t_j \)?

9.3. Consider the differential equation

\[ y''(t) - y'(t) - 2y(t) = 0 \]

a) For initial conditions \( y(0) = 1 \) and \( y'(0) = -1 \), show that the solution is \( y = e^{-t} \).
b) Now, for initial conditions \( y(0) = 1 + \epsilon \) and \( y'(0) = -1 \), show that the solution is

\[ y = (1 + 2\epsilon/3)e^{-t} + \epsilon e^{2t}/3 \]

c) Notice what happens to the solution in part (b) for any nonzero \( \epsilon \), as \( t \to \infty \). Explain why the solution \( y = e^{-t} \) in part (a) is said to be unstable.
d) Formulate the problem as a system of differential equations and use a standard routine to solve it.

9.4. Consider the two-point boundary-value problem:

\[ v'' + 2xv' - x^2v = x^2, \quad v(0) = 1, \quad v(1) = 0. \]
a) Let \( h = \frac{1}{4} \) and explicitly write out the centered difference equations.

b) Now do the same thing, but use one-sided approximations for \( v' \).

c) Formulate the problem as a system of differential equations and use a standard routine to solve it.

9.5. Consider the two-point boundary problem:

\[
u'' = u^2 + 2x, \quad \text{for } a \leq x \leq b,
\]

and

\[
u(a) = u_1
\]

\[
u(b) = u_2.
\]

a) Write the differential equation as a system of first-order ODEs.

b) To use a shooting method you need one more initial value. Use \( u'(a) = (u_2 - u_1)/2 \). Why is this a reasonable value?

c) Using the initial values above and a step size of \((b - a)/2\), go through one iteration of Euler’s method (that is, write the solution at \( x = b \) for the first shot).

d) What would you do next to solve the given BVP?

9.6. A logistic model has often been used to study the growth of organisms when the food supply is fixed. A discrete version of this model is

\[
x_i = x_{i-1}(c - x_{i-1}).
\]

This model, however, has rather peculiar behavior that would indicate that population sizes vary with the food supply \( c \) in unusual ways. To examine this counter-intuitive behavior, approximate the time average of the population size

\[
\bar{x} = \lim_{n \to \infty} \frac{1}{n} \sum_{i=0}^{n} x_i
\]

for various values of \( c \) in the range of 3 to 4. Write a program (Fortran, C, R, Matlab, or whatever) to simulate this model, and compute and plot your approximate \( \bar{x} \) for various values of \( c \).

(Use \( x_0 = c/2 \). One aspect of the problem is to choose a reasonable \( n \), so indicate what value of \( n \) you use, and how you decided on that value. Also, you must choose the type of plots to present. The behavior of this model is well-known. For values of \( c \) below \( 1 + \sqrt{6} \), the model behaves quite predictably; above that value, however, it displays unusual behavior.)

In the real world, it is unlikely that the food supply would be constant at \( c \); rather, it is more likely to be \( c + d_i \), where \( d_i \) is a realization of some random variable. (This would be a stationary case, time-trends in \( c \) might be even more realistic, but let’s assume stationarity.) Choose two (or more) “realistic” distributions for the random variable, and repeat the computations and plots above for this case.

Summarize and compare these two models in the light of the computations and plots you have made.
9.7. Consider the following data on monthly sales of PCs in the United Kingdom, given in Moshe Givon, Vijay Mahajan, and Eitan Muller, *Software piracy: Estimation of lost sales and the impact of software diffusion*, *Journal of Marketing* 59, 29–37.

<table>
<thead>
<tr>
<th>Yr/Mo</th>
<th>Units Sold</th>
</tr>
</thead>
<tbody>
<tr>
<td>87/01</td>
<td>40,164</td>
</tr>
<tr>
<td>87/02</td>
<td>41,303</td>
</tr>
<tr>
<td>87/03</td>
<td>42,466</td>
</tr>
<tr>
<td>87/04</td>
<td>35,525</td>
</tr>
<tr>
<td>87/05</td>
<td>36,513</td>
</tr>
<tr>
<td>87/06</td>
<td>37,521</td>
</tr>
<tr>
<td>87/07</td>
<td>34,961</td>
</tr>
<tr>
<td>87/08</td>
<td>35,913</td>
</tr>
<tr>
<td>87/09</td>
<td>36,884</td>
</tr>
<tr>
<td>87/10</td>
<td>45,856</td>
</tr>
<tr>
<td>87/11</td>
<td>47,076</td>
</tr>
<tr>
<td>87/12</td>
<td>48,319</td>
</tr>
<tr>
<td>88/01</td>
<td>52,560</td>
</tr>
<tr>
<td>88/02</td>
<td>52,560</td>
</tr>
<tr>
<td>88/03</td>
<td>53,924</td>
</tr>
<tr>
<td>88/04</td>
<td>55,310</td>
</tr>
<tr>
<td>88/05</td>
<td>46,498</td>
</tr>
<tr>
<td>88/06</td>
<td>47,670</td>
</tr>
<tr>
<td>88/07</td>
<td>48,860</td>
</tr>
<tr>
<td>88/08</td>
<td>45,769</td>
</tr>
<tr>
<td>88/09</td>
<td>46,887</td>
</tr>
<tr>
<td>88/10</td>
<td>48,020</td>
</tr>
<tr>
<td>88/11</td>
<td>60,047</td>
</tr>
<tr>
<td>88/12</td>
<td>61,462</td>
</tr>
<tr>
<td>89/01</td>
<td>76,022</td>
</tr>
<tr>
<td>89/02</td>
<td>77,745</td>
</tr>
<tr>
<td>89/03</td>
<td>79,483</td>
</tr>
<tr>
<td>89/04</td>
<td>67,270</td>
</tr>
<tr>
<td>89/05</td>
<td>68,730</td>
</tr>
<tr>
<td>89/06</td>
<td>70,198</td>
</tr>
<tr>
<td>89/07</td>
<td>66,232</td>
</tr>
<tr>
<td>89/08</td>
<td>67,601</td>
</tr>
<tr>
<td>89/09</td>
<td>66,974</td>
</tr>
<tr>
<td>89/10</td>
<td>86,915</td>
</tr>
<tr>
<td>89/11</td>
<td>88,615</td>
</tr>
<tr>
<td>89/12</td>
<td>90,315</td>
</tr>
<tr>
<td>90/01</td>
<td>96,098</td>
</tr>
<tr>
<td>90/02</td>
<td>97,866</td>
</tr>
<tr>
<td>90/03</td>
<td>96,098</td>
</tr>
<tr>
<td>90/04</td>
<td>90,315</td>
</tr>
<tr>
<td>90/05</td>
<td>96,098</td>
</tr>
<tr>
<td>90/06</td>
<td>97,866</td>
</tr>
<tr>
<td>90/07</td>
<td>90,315</td>
</tr>
<tr>
<td>90/08</td>
<td>97,866</td>
</tr>
<tr>
<td>90/09</td>
<td>90,315</td>
</tr>
<tr>
<td>90/10</td>
<td>103,640</td>
</tr>
<tr>
<td>90/11</td>
<td>105,142</td>
</tr>
<tr>
<td>90/12</td>
<td>106,618</td>
</tr>
<tr>
<td>91/01</td>
<td>100,641</td>
</tr>
<tr>
<td>91/02</td>
<td>101,958</td>
</tr>
<tr>
<td>91/03</td>
<td>103,241</td>
</tr>
<tr>
<td>91/04</td>
<td>92,712</td>
</tr>
<tr>
<td>91/05</td>
<td>93,787</td>
</tr>
<tr>
<td>91/06</td>
<td>94,827</td>
</tr>
<tr>
<td>91/07</td>
<td>95,080</td>
</tr>
<tr>
<td>91/08</td>
<td>96,037</td>
</tr>
<tr>
<td>91/09</td>
<td>96,953</td>
</tr>
<tr>
<td>91/10</td>
<td>113,410</td>
</tr>
<tr>
<td>91/11</td>
<td>114,371</td>
</tr>
<tr>
<td>91/12</td>
<td>115,279</td>
</tr>
<tr>
<td>92/01</td>
<td>123,043</td>
</tr>
<tr>
<td>92/02</td>
<td>123,886</td>
</tr>
<tr>
<td>92/03</td>
<td>124,667</td>
</tr>
<tr>
<td>92/04</td>
<td>92,329</td>
</tr>
<tr>
<td>92/05</td>
<td>92,808</td>
</tr>
<tr>
<td>92/06</td>
<td>93,238</td>
</tr>
<tr>
<td>92/07</td>
<td>107,933</td>
</tr>
<tr>
<td>92/08</td>
<td>108,312</td>
</tr>
<tr>
<td>92/09</td>
<td>78,969</td>
</tr>
<tr>
<td>92/10</td>
<td>103,640</td>
</tr>
<tr>
<td>92/11</td>
<td>105,142</td>
</tr>
<tr>
<td>92/12</td>
<td>106,618</td>
</tr>
</tbody>
</table>

a) Using the given data, determine the least squares estimates for $p$, $q$, and $m$ in the Bass diffusion model (9.20):

\[
\frac{du}{dt} = \left( p + \frac{q u}{m} \right) (m - u).
\]

b) Now, use separately the periods 87/01–89/12, 88/07–90/06, and 90/01–92/08 determine the least squares estimates for $p$, $q$, and $m$. Do these parameters appear to be stationary?

c) How might you assess seasonality in the model?

9.8. The table below shows the use of natural and man-made fibers in clothing produced in the United States for the years 1950 through 1978.
A model for this substitution of a new technology for an old one are the differential equations,

\[
\frac{dy_1}{dt} = \alpha_1 y_1 (C_1 - y_1 - \beta y_2) - \gamma_1 y_1,
\]

\[
\frac{dy_2}{dt} = \alpha_2 y_2 (C_2 - y_2 - \beta y_1) - \gamma_2 y_2,
\]

where

\(\alpha_1\) and \(\alpha_2\) represent the entry rates of the old and the new technologies,

\(\gamma_1\) and \(\gamma_2\) represent the exit rates of the old and the new technologies,

\(C_1\) and \(C_2\) represent the capacities of the economic for the old and the new technologies, and

\(\beta\) (\(0 \leq \beta \leq 1\)) indicates the extent to which the technologies employ the same resources.

Any of these parameters may be time dependent, but they are taken as constants in most studies.

Use the data given to determine least squares estimates of \(\alpha_1\), \(\alpha_2\), \(\gamma_1\), \(\gamma_2\), \(C_1\), \(C_2\), and \(\beta\).

9.9. The population of the United States has shown steady growth from the first decennial census in 1790 to the one in 1990. The table below gives the population in millions.

<table>
<thead>
<tr>
<th>Year</th>
<th>Natural</th>
<th>Synthetic</th>
<th>Year</th>
<th>Natural</th>
<th>Synthetic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1950</td>
<td>5340</td>
<td>1491</td>
<td>1965</td>
<td>4923</td>
<td>3512</td>
</tr>
<tr>
<td>1951</td>
<td>5371</td>
<td>1471</td>
<td>1966</td>
<td>5016</td>
<td>3989</td>
</tr>
<tr>
<td>1952</td>
<td>4957</td>
<td>1464</td>
<td>1967</td>
<td>4747</td>
<td>4244</td>
</tr>
<tr>
<td>1953</td>
<td>4966</td>
<td>1502</td>
<td>1968</td>
<td>4489</td>
<td>5306</td>
</tr>
<tr>
<td>1954</td>
<td>4526</td>
<td>1484</td>
<td>1969</td>
<td>4256</td>
<td>5550</td>
</tr>
<tr>
<td>1955</td>
<td>4815</td>
<td>1851</td>
<td>1970</td>
<td>4102</td>
<td>5500</td>
</tr>
<tr>
<td>1956</td>
<td>4825</td>
<td>1685</td>
<td>1971</td>
<td>4185</td>
<td>6345</td>
</tr>
<tr>
<td>1957</td>
<td>4444</td>
<td>1745</td>
<td>1972</td>
<td>4091</td>
<td>7566</td>
</tr>
<tr>
<td>1958</td>
<td>4207</td>
<td>1702</td>
<td>1973</td>
<td>3819</td>
<td>8665</td>
</tr>
<tr>
<td>1959</td>
<td>4781</td>
<td>2065</td>
<td>1974</td>
<td>3412</td>
<td>7700</td>
</tr>
<tr>
<td>1960</td>
<td>4657</td>
<td>1817</td>
<td>1975</td>
<td>3141</td>
<td>7416</td>
</tr>
<tr>
<td>1961</td>
<td>4506</td>
<td>2064</td>
<td>1976</td>
<td>3543</td>
<td>8053</td>
</tr>
<tr>
<td>1962</td>
<td>4629</td>
<td>2418</td>
<td>1977</td>
<td>3296</td>
<td>8889</td>
</tr>
<tr>
<td>1963</td>
<td>4465</td>
<td>2788</td>
<td>1978</td>
<td>3195</td>
<td>9236</td>
</tr>
<tr>
<td>1964</td>
<td>4615</td>
<td>3174</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
a) The relationship of a population size to time can often be modeled locally, that is, in relatively short time intervals, by the equation

$$\frac{dP}{dt} = rP.$$ 

Make a plot of the population data and discuss whether or not this differential equation is a good model for this time period.

b) Consider the model

$$\frac{dP}{dt} = (\theta_1 t + \theta_0)P.$$ 

Do you think this may be a better model? Why?

c) What is the solution to this differential equation? What are the parameters in your model? (There are three.)

d) The next step is to fit the parameters in your model. How could you proceed to estimate these? (There are more than one way of doing this — mention the general principles, and then some specific ways.)

e) Compute estimates of the parameters using the logs of the population size. (Tell how you get your estimates.)

f) Compute estimates of the parameters directly using the population size, that is, using the solution to the differential equation directly. What is the fundamental difference in the assumptions about the model in this approach to fitting the parameters and in the approach using logs?

g) What is your estimate of the population in 1985?

h) What is your estimate of the population in 2005? What is the fundamental difference in this question and the previous one?

i) Use a spline smoother on the data to get an estimate of the population in 1985?

j) Models need data. Most demographers and historians believe the data given above is not accurate to the precision shown. It is likely that bounds for the absolute error have widened as the populations has grown, but on the other hand, the relative error has probably decreased. Suppose the relative error has a normal distribution with
standard deviation \( r(t) = 0.5475 - 0.00025t \). What was the relative error in 1790? in 1990? Under this assumption about the random inaccuracy in the population counts, how would you change your approach to fitting the model? What is your new model?

k) What else may you consider in developing a model of the population growth of the United States over the given period? Write some alternative models.

Fit the model \( P(t) = \beta_0 + \beta_1 t + \beta_2 t^2 \). How good is the fit? What else (besides just how good is the fit) should we ask?

l) Something not included in the models above, but which is obviously relevant, is immigration and emigration. For the United States, emigration has been negligible over the period under consideration.

While the population has an instantaneous value, immigration is measured over intervals of time. Here are some data on immigration (from TIME Almanac, 1999):

<table>
<thead>
<tr>
<th>Period</th>
<th>Immigration (millions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1820–1940</td>
<td>38.3</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1941–1950</td>
<td>1.04</td>
</tr>
<tr>
<td>1951–1960</td>
<td>2.52</td>
</tr>
<tr>
<td>1961–1970</td>
<td>3.32</td>
</tr>
<tr>
<td>1971–1980</td>
<td>4.49</td>
</tr>
<tr>
<td>1981–1990</td>
<td>7.34</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1820–1996</td>
<td>61.2</td>
</tr>
</tbody>
</table>

How would you incorporate this information in your model? You may wish to mention some alternatives.

m) Write a new model that includes the immigration data. Fit the model.

Using your new model, estimate the net birth/death rate in 1920.
Part III

Data Science
Developing mathematical models is one of the most important activities in science. The objective of a model often is to describe the behavior of one variable in terms of other variables. We may use a model such as

$$y \approx f(x; \theta),$$  \hspace{1cm} (9.21)

in which $x$ is an $m$-vector of variables and $\theta$ is a vector of parameters used in the specification of the function $f$, to express the relationship of the variables $y$ and $x$. (Notice that we do not distinguish vectors and scalars by their notation.)

The variables $y$ and $x$ in (9.21) are observable, and in some applications, $x$ can be adjusted or controlled. The parameter $\theta$ is generally unknown. The model (9.21) simulates the behavior of $y$ as a function of $x$.

The need for an approximation such as the model (9.21) may arise in two different ways. We may know a functional relationship between $y$ and $x$, say $y = h(x)$, and we want to approximate the function $h$ with a simpler function $f$. Alternatively, we may only have observations on some values of $y$ that correspond to specific values of $x$, and we want to fit a model to those observations. In this chapter we will discuss the problem of building the model (9.21) from both of these perspectives.

Although the model (9.21) treats $x$ and $y$ differently ($x$ is “independent” and $y$ is a “response”), sometimes we do not distinguish the variables in this way, but rather consider a relationship of the form $g(x, y; \theta) \approx 0$.

There are two general aspects to the problem of building models. One involves selection and analysis of the form of the function $f$, and the other is the use of available information to fit $f$, that is, to specify a value of $\theta$. Determining a function from data is called an “inverse problem”.

There are many reasons to build models. When the relationships among the variables are modeled by a function, the variables can be studied further by analysis of the model. The model can be used for computer simulations of the reality it represents, and the model can be transformed so as to study different aspects of the data. Aris (1978, 1994) contains interesting essays on the general problems of building mathematical models.

**Observational Data**

We may have observations of $y$ and corresponding observations of $x$ and want to determine a model that fits the observed values well. The observational data are used to determine a functional form, that is, what $f$ to use, and then to determine $\theta$ for a given functional form $f$. Data used in this way are called “training data”, and a given observation in the dataset is called a “training example”.

When we have observational data, it is often convenient to write (9.21) as

$$y \approx f(X; \theta),$$  \hspace{1cm} (9.22)

where $y$ is an $n$-vector of observations on the variable $y$ from (9.21) and $X$ is an $n \times m$ matrix of $n$ observations on the $m$ variables of the vector $x$. 

---

**Inverse Problem**

---
The objective is to use the given $X$ and $y$ to fit the model (9.21), that is, to determine $f$ and $\theta$.

We often use the notation $(y_i, x_i)$ to represent an observation. In this notation $x_i$ is the transpose of the $i^{th}$ row of $X$. (Notice in the notation of equation (9.21), a subscript on $x$ would represent a particular element of the vector $x$. When we use $X$ to represent a matrix of data, we may use a double subscript to represent a particular element of a particular observation. For example $x_{ij}$ would be the $j^{th}$ element of the $i^{th}$ observed value of the vector $x$. The notation may seem somewhat difficult at first, but the context usually makes it clear.)

Observed data constitute a discrete function. This discrete function is an approximation of a subset of the continuous function $f$. There are two aspects of this function composed of the data that warrant comment. First, although a function is a set of ordered pairs no two of which have the same first value, the discrete function defined by the data may have multiple elements with the same first value. There are various practical ways of reconciling this with our definition of $f$ as a function. The addition of a random additive error to $f$ is one way. We consider this more carefully in Section 11.1.1.

A second aspect of the observational data that needs mention is the possible multiplicity of observations. Although it may be the case that $(y_i, x_i) = (y_j, x_j)$ while $i \neq j$, we do not consider these two observations to be the same, and reduce the cardinality of the set by 1, as we would in ordinary set theory. Each member of the set of observations contains an additional component, its index or some other identifying quantity, and so the elements are not the same.

**Models for Interpolation**

The model (9.21) can be viewed as expressing an exact relationship for a fixed set of values. The discrete function made up of the observations can be considered to be a subset of the function $f$. That exact fit would provide an approximation at other values that $x$ may assume. This approach of fitting a function to the given $y$ and $X$ is called interpolation.

There are several possibilities for choosing a continuous function to interpolate the data. An interpolant is likely not to be very smooth or else it may exhibit wide variation. The requirement to fit all data values exactly may also mean that the relationship is not a (single-valued) function. It is unlikely that a single easily-defined function, other than a polynomial, could interpolate the data. A polynomial of degree $n - 1$ can, of course, interpolate $n$ data points (assuming no two points have the same ordinate values), but such a polynomial would likely have wild swings.

Once the functional form $f(x; \theta)$ is chosen, it would be difficult or impossible to determine $\theta$ that would interpolate a given dataset, that is, a value of $\theta$ that would yield an equality at each data point. For interpolation, there must be considerable freedom to choose the function $f$. Rather than a single
function, however, we may choose a piecewise set of functions, each of which interpolates a subsequence of the given points that have adjacent abscissas. Two piecewise interpolating polynomials are shown in Figure 9.10.

For two-dimensional data, that is, data with one independent variable, the simplest interpolating function perhaps is the piecewise linear function shown on the left in Figure 9.10. A more general form of an interpolating function for two-dimensional data can be built from Lagrange polynomials. For fitting a function at the given points $x_1, x_2, \ldots, x_n$, we use $n - 1$ Lagrange polynomials of degree $n - 1$, the $j^{th}$ of which is

$$l_j(x) = \frac{p_n(x)}{(x - x_j)p'_n(x_j)}, \quad (9.23)$$

where $p_n = \prod_{i=1}^n (x - x_i)$ and $p'_n$ is the derivative of $p_n$. It is clear that

$$l_j(x_i) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases};$$

therefore, if the function values at the given points are $y_1, y_2, \ldots, y_n$, the function

$$h(x) = \sum_{i=1}^n l_i(x)y_i \quad (9.24)$$

is an interpolating polynomial of degree $n - 1$. 

Figure 9.10. Interpolation grf010a
A single Lagrange polynomial could be used over the full range, or a set of piecewise Lagrange polynomials of degree \( k - 1 \) could be used over sets of \( k \) adjacent points. The piecewise linear function in Figure 9.10 is a piecewise Lagrange polynomial of degree 1 each piece of which is defined on two points.

Another approach is to define breakpoints or “knots”, which may or may not correspond to observation points, and then to fit polynomials to points between the knots with the additional restriction that the polynomials join smoothly at the knots. This restriction is that the derivatives up to a chosen degree are equal at the knots. This is called spline interpolation. The piecewise linear function in Figure 9.10 has knots at each data point, and the zero-order derivatives agree at the knots. The cubic spline shown in Figure 9.10 has knots at the data points. The sharp increase at the first two points on the left causes large values for the derivatives.

Models for Smoothing Data

The process of selecting a relatively simple model that provides a good approximation to the data is called “smoothing”. The smoothed values of \( y \) simulate the phenomenon being modeled. For a given functional form \( f \), the parameter \( \theta \) is chosen so that the observed values \( y_i \) are close to the smoothed values \( f(x_i; \hat{\theta}) \), for some \( \hat{\theta} \). In Figure 9.11, we see two different smoothing models for the same data. In the plot on the left, we have a simple straight line that approximates the data. The straight line does not fit the observations very well. A different functional form is used in the plot on the right in Figure 9.11. This approximation seems to fit the data better, and it captures an important apparent structure in the data.

The criterion used for determining \( \hat{\theta} \) is usually to minimize some norm of the residual vector,

\[
\|y - f(X; \theta)\|.
\]

The vector norm is often chosen as the L_2 norm (see page 258), and in this case the criterion for smoothing the data is called least squares.

For approximation, just as for interpolation, we could also use different functional forms over different regions of the data. So-called “smoothing splines”, for example, are similar to interpolating splines without the requirement of passing through each data point.

Iterative Model Selection

A common approach to the general modeling problem is to assume a form for \( f(x; \theta) \), based on previous knowledge, based on “common sense”, or based on the principle of Occam’s razor (“simpler is better”); next to determine a “good” \( \theta \) for that functional form; then to inspect closely the “goodness” of the fit of the model to the data; and to iterate over this three-step process, as depicted in Figure 9.12.
The closeness of the fit of the model to the data is determined by comparing the simulated data generated by the model to the observed data. The model may involve a random component, as we discuss in Section 11.1.4, in which case multiple samples of simulated data are compared with the observed data.

In many situations in the physical sciences, we can develop models from first principles, based on accepted physical laws.

**Forms of Models**

The form of the model may be linear in both $x$ and $\theta$, for example,

$$f(x; \theta) = \theta_0 + \theta_1 x_1 + \theta_2 x_2;$$

it may be linear only in $\theta$, for example,

$$f(x; \theta) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 e^{x_2};$$

or it may be nonlinear in both $x$ and $\theta$, for example,

$$f(x; \theta) = \theta_0 e^{\theta_1 x_1 + \theta_2 x}. $$

Although the last model above can be “linearized” by taking logs of both sides (assuming $\theta_0 > 0$), the transformation changes the correspondence of the model to observed data. The residuals of the observed data from the linearized model are not additive in the original untransformed model.
The variables of interest in a model may all occur explicitly in the dataset, or they may be implicit, such as a variable indicating the order in which the observations are made, or, possibly, the spatial location of the observation.

**The Nature of the Approximation**

In equations (9.21) and (9.22), we have written the models as approximations. In alternative expressions we may represent the difference as a residual, \( r \):

\[
y = f(x; \theta) + r,
\]

or

\[
y = f(X; \theta) + r,
\]

where \( r \) is a scalar or a vector depending on the form of the model.

In statistical inference about the model, we may assume \( r \) is a random variable, and the specific assumptions we make about its probability distribution determine how we proceed in fitting the model and in making inferences about the process being modeled. We address some of these issues further in Section 11.1.

**Evolutionary Models**

In evolving processes, a single model like (9.21) or (9.22) may not be appropriate over different epochs. Distributional properties of variables or relationships among variables may change over time.

We may consider a model that is stationary (that is, constant) over a fixed period of time, but later may change to another form that may be constant
for some period of time. Even if the model changes continuously in time, a fixed model for some brief period of time may serve as a good approximation for that period. In some cases, the basic functional form may continue to serve as a good model, but the parameters may evolve over time. We may assume a model that in the $k^{th}$ epoch has the form

$$y^{(k)} \approx f(X^{(k)}; \theta^{(k)}). \quad (9.25)$$

We may assume that there is a systematic evolution of the model over time, which may be represented by

$$\theta^{(k)} \approx g(\theta^{(k-1)}, \theta^{(k-2)}, \ldots). \quad (9.26)$$

### Nonparametric Smoothing

Rather than using a model like (9.21) in which we first identify a functional form for $f$ that is dependent only on the parameter $\theta$, we may use the data to determine values of $y$ that correspond to $x$, and never explicitly develop a smoothing function corresponding to $f(x; \theta)$. This alternate approach is called *nonparametric smoothing*. In nonparametric smoothing the function may be expressed only as a set of ordered pairs $S = \{(\hat{y}_i, x_i)\}$ corresponding to the original dataset $\{(y_i, x_i)\}$.

In addition to representing the function as a set of ordered pairs, a rule for determining $\hat{y}_g$ corresponding to a given value $x_g$ not in the original dataset may be specified. This rule may be a function $\hat{f}(\cdot)$ that interpolates the function $S$.

### Criteria for Approximation and Fitting

The basic idea in model fitting is that the discrete function consisting of the observed data approximates another function, usually with a continuous domain. The process of estimating functions with continuous domains and smoothing data often involves expansions of functions in terms of other, simpler functions or it involves convolving two functions.

An approximation or a fitted model may be deemed adequate if the residual vector is “small”. Smallness may be defined in terms of a norm of this vector. The most common norm used for this purpose is the $L_2$ norm, in which case the criterion for fitting is called “least squares”. Another useful norm is the $L_1$ norm, and the criterion is called “least absolute values”. Other norms may be useful, or some other completely different criterion may be appropriate. Under certain assumptions about random distributions of residuals, some statistical property, such as maximum likelihood, may be relevant. We discuss some of these considerations in Section 11.1. Most criteria for goodness of a fit involve optimization of some function, and the methods of Chapter 7 are used extensively in the general problem of model building.
We begin this chapter with discussions of operators on functions that parallels our discussion of inner products, orthogonality, and norms of vectors in Chapter 5.
10

Regression Models

10.1 Linear Models
10.2 Least Squares Fitting
10.3 Assessing the Fit
10.4 Variable Selection

Exercises
11

Smoothing Data

11.1 General Considerations in Smoothing Data with Models

There are many variations on the theme expressed by the model (9.21) and what it means to fit the model. These variations relate to the nature of the problem being studied or to the purpose in using the model.

A fundamental property of data is whether or not there is an ordering of the data. For many kinds of statistical analysis, a random sample is assumed to be identically and independently distributed (i.i.d.). This assumption is realistic for many situations of interest. In other situations, however, there is likely to be some lack of independence in the data either because of the chronological order in which it was measured, or because of spatial relationships among the observed units.

In sequential data, such as a time series or a signal received over time, whether or not the sequential nature of the data is accompanied by a dependence is an important question to address. For example, in order for data generated by a pseudorandom number generator to be useful, the successive values should be “independent”. One way of assessing the independence is to compute the autocorrelations (although, of course, 0 correlations do not imply independence). Another way is to make some kind of domain transform, such as a finite Fourier transform.

11.1.1 Origins and Nature of the Model+

As we have indicated, the form of the model is selected based on any available knowledge about the process. Given that the model is to be used for approximation, a simple model is desirable because it is easier to use and understand.

The approximation of the model (9.21) to observed data can be thought of as arising from some kind of additive “error” or random fluctuation, so that the model is really of the form
where \( r \) accounts for the residual deviations. Another way of thinking about the model is that the observed values of \( y \) have been filtered by a convolution, so the model is really of the form

\[
y = h(x) \ast f(x; \theta),
\]

for some unknown function \( h \). \textit{************** see Kay, p. 365}

In some cases, the model (9.21) is not written as an expression involving mathematical operations; rather it is only given as a mathematical \textit{function}. (A function is a set of tuples, in our case \((m + 1)\)-tuples, such that any two tuples in the set differ by some value other than the first one. Sometimes the single-value part of the definition is relaxed.)

In many processes we can model \textit{change} more directly than we can the values of interest. Models of change may be differential equations, and so some of the models we consider arise initially as either ordinary or partial differential equations.

\subsection*{11.1.2 Fitting a Model to Observed Data}

A basic issue in fitting a model to observed data is how to measure the fit (of lack of fit) of data to the model. The fitting process is then chosen so as to optimize the fit.

\textbf{Minimizing Additive Residuals}

One familiar measure of the lack of fit of a datum is the square of the distance of the observation to the model equation. The common least squares fitting method, for a given functional form \( f \), is to determine a value of \( \theta \) that minimizes the sum of the squared distances. As we have seen in Chapters 5 and 7, this is a relatively simple optimization problem in some cases.

While different functions may be used over different ranges of the data, a fundamental issue is how to perform the fit within a range in which the same functional form and same values of the parameter \( \theta \) are to be used. This involves defining some measure of fit and then choosing \( f \) and \( \theta \) optimally with respect to that measure. In the familiar regression model, there is a single model over the full range of interest, the fit is measured in terms of squared distances of the observations from the function, and the model is fit so as to minimize the sum of those squares.

\[
y_i = f(x_i; \theta) + r_i
\]

The \( r_i \) are vertical distances as shown in Figure 11.1 for a linear model in two dimensions.
A common way of fitting the model is to minimize the residuals in some way. Usually this means to minimize $\sum g(r_i)$, for some nondecreasing function $g(\cdot)$. Often, of course, $g(r) = r^2$, and so the criterion for the fit is least squares. Fitting a model in this way is called least squares regression.

**Nonconstant Measures**

In addition to using different approximating functions over different ranges of the data, we may also measure the fit differently over different ranges of the data. This may be done so as to reduce the influence of outlying observations on the overall fit. It may also be done to reflect the uncertainty in the observations, either because of possible measurement errors or because the observations are assumed to be realizations of random variables with difference variances.

Each observation is assigned a weight, $w_i$, and the model is fit so as to minimize $\sum w_i g(r_i)$, where the $r_i$ are vertical distances as shown in Figure 11.1. Often, of course, $g(r) = r^2$. Fitting a model in this way is called weighted least squares regression.

**Orthogonal Distances**

The $d_i$ are orthogonal distances as shown in Figure 11.2. The model is fit so as to minimize $\sum g(d_i)$. Often, of course, $g(d) = d^2$, and so the fit is “orthogonal least squares”.

Fitting a model by minimizing the sum of the squared distances is called orthogonal distance regression and the criterion is sometimes called total least squares.
Linear Models

An important and common form of the model (9.21) is a linear model. For observational data, we often represent the \( i \)th observation in a linear model as

\[
y_i \approx \beta_0 + \beta_1 x_{1i} + \cdots + \beta_m x_{mi}. \tag{11.2}
\]

The least squares fit of the linear model are the \( \hat{\beta}_j \) such that

\[
\sum_{i=1}^{n} (y_i - \hat{\beta}_0 - \hat{\beta}_1 x_{1i} - \cdots - \hat{\beta}_m x_{mi})^2 \tag{11.3}
\]

is a minimum. The least squares fit is easy to determine by considering the \( \hat{\beta}_j \) in (11.3) to be variables and taking the derivative with respect to each variable and setting it equal to zero. Doing this, we find that

\[
\bar{y} = \hat{\beta}_0 - \hat{\beta}_1 \bar{x}_1 - \cdots - \hat{\beta}_m \bar{x}_m, \tag{11.4}
\]

where \( \bar{x}_j \) is the mean, \( \frac{1}{n} \sum x_{ji} \), of the \( j \)th independent variable.

Notation for Linear Models

The notation for the linear model is somewhat complicated. Analogously to equation (9.22), we could write

\[
y \approx [1|X] [\beta_0, \beta_1, \ldots, \beta_m]. \tag{11.5}
\]
11.1 General Considerations in Smoothing Data with Models

where \([1|X]\) is the matrix consisting of a column of 1’s and columns of the observations. The expression (11.5) is also rather complicated. We can simplify it by using homogeneous coordinates (see Section 5.1.1, page 273), or by fitting the model in such a way that the intercept \(\beta_0\) has some simple, known relationship to the data.

As we have noted, the least squares fit of the model is such that the equation fits the means of the dependent variable \(y\) and all the independent variables. This allows for simpler notation for centered data; that is data in which the original mean of each variable has been subtracted from the value of that variable in each observation. If we use the same \(y\) and \(X\) notation to represent centered data, instead of equation (11.5), we have the simpler expression,

\[
y \approx X\beta.
\]

One additional comment about the notation: the scalars \(x_{ij}\) in equations (11.2) and (11.3) do not correspond to the elements of the matrix \(X\) in equations (11.5) and (11.6); the subscripts are reversed. I will generally use the notation of equation (11.6). I point out these complexities because all of the notation above is standard in the literature, and the reader must be aware of the meaning.

**Least Squares Computations for Linear Models**

The least squares criterion for fitting the model (11.6) leads to minimizing

\[
(y - X\beta)^T (y - X\beta),
\]

which occurs for \(\hat{\beta}\) that satisfies

\[
X^T X \hat{\beta} = X^T y,
\]

that is, \(\hat{\beta} = (X^T X)^{-1} X^T y\), if \(X\) is of full column rank. Forming \(X^T X\) and then taking its inverse is not the way to solve for \(\hat{\beta}\), of course. If the QR decomposition of \(X\) is

\[
X = QR
= Q \begin{bmatrix} R_1 \\ 0 \end{bmatrix},
\]

then

\[
\hat{\beta} = R_1^{-1} c_1,
\]

where \(c_1\) is the \(m\)-vector in

\[
Q^T y = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}.
\]

(See page 305, and equation 5.97.)
11 Smoothing Data

**Updating Linear Least Squares Computations**

Often in applications we receive data sequentially, or in stages. At stage \( k \) we may have the \( n_k \) observations in the vector \( y^{(k)} \) and the matrix \( X^{(k)} \). In this case, we must center the data in each stage, based on the mean up to that stage. If

\[
T^{(1)} = (X^{(1)})^T X^{(1)}
\]

and

\[
\hat{\beta}^{(1)} = (X^T X)^{-1} X^T y,
\]

compare the algorithm for computing the sample mean and variance on page 59). We (See Gentle, 2007, equation (9.28) on page 340.)

**Orthogonal Distances in Linear Models**

Orthogonal distances have been studied most extensively in the case of the linear model. Use of orthogonal distances is sometimes suggested for the errors-in-variables model. This model has the form

\[
y = (X + \Delta)\beta + E,
\]

where both \( \Delta \) and \( E \) are random variables. See Fuller (1987) for a discussion of the errors-in-variables model.


Ammann and Van Ness (1988, 1989) describe an iterative method that is applicable to any norm, so long as a method is available to compute a value of \( \beta \) that minimizes the norm of the vertical distances in the model. The method is

1. determine \( b^{(k)} \) that minimizes the norm of \( (y^{(k-1)} - X^{(k-1)}\beta) \),
2. transform the matrix \( [y^{(k-1)}|X^{(k-1)}] \) to \( [y^{(k)}|X^{(k)}] \) by a rotation matrix that makes the \( k^{th} \) fit horizontal
3. set \( k = k + 1 \) and go to 1.

This is repeated until there is only a small change. An appropriate rotation matrix is \( Q \) in the \( QR \) decomposition of

\[
\begin{bmatrix}
I_m & 0 \\
(b^{(k)})^T & X \\
\end{bmatrix}.
\]
11.1 General Considerations in Smoothing Data with Models

11.1.3 Building Models

Reducing the Dimensionality

Locality

A basic issue in fitting the model is the locality of application of a single, simple form of the model. Is there a single functional form $f$ over the full range of possible values of $X$ that may be of interest? If so, is there a single value of $\theta$ over the full range of possible values of $X$ that may be of interest? If more than one model (either form or parameter value) is to be used, how is the range of $X$ divided up?

Another important consideration in fitting the model is the extent to which a given observation affects the fit. What is the sphere of influence of a given observation?

11.1.4 Stochastics

In the model (9.21) the deviations in the observed values of $y$ and the corresponding model values $f(x; \theta)$ may be thought of as an additive random variable, $E$. This of course means that the observed data $y$ are realizations of a random variable $Y$. The model is

$$Y = f(x; \theta) + E,$$

In this model the observed elements of $Y$ are stochastically related to the values of those $m$ “independent” variables in the $m$-vector $x$ (“predictors”, “regressors”, and “carriers” are terms that we use more or less synonymously). In this case the model is more than just the equation; it is the equation together with any model for the random variable $E$, that is, together with any specification of the probability distribution of the random component.

Given an assumed relationship such as (11.7), the problem is to estimate $f$ and its associated $\theta$, that is, to fit the model to the data. The model for the distribution of $E$ makes the problem one of statistical estimation and allows us to make statements about the distribution of various estimates. Without a model on the distribution of $E$, the problem has no aspects of inferential statistics; it is merely in the domain of descriptive statistics or data reduction. Our interest in this chapter is more on the mechanical aspects of getting a “good” fit.

The nature of the model of the distribution of $E$ may determine whether our approach is “distribution-free” or “nonparametric”. The fitting process may be called nonparametric, even though we estimate the parameter $\theta$. In a more extreme instance of a nonparametric approach, the fitted model is only a function in the sense of being a set of $(m+1)$-tuples, as we discuss above.
11.2 Smoothing Functions

One of the most obvious kinds of continuous function to use in fitting discrete data is a piecewise polynomial.

11.2.1 General Properties of Splines

\[ y_i \approx \mu(x_i), \]

\( \mu \) is a smooth function: for some fixed \( m \), has \( m - 1 \) continuous derivatives and square-integrable \( m \)th derivative over \( [a, b] \). \( a \leq x_1 < x_2 < \cdots < x_n \leq b \).

11.2.2 Smoothing Splines

The smoothing spline estimator of \( \mu \) satisfies a penalized least squares criterion; i.e., it is \( g \), satisfying the derivative conditions, that minimizes

\[
\frac{1}{n} \sum_{i=1}^{n} (y_i - g(x_i))^2 + \lambda \int_a^b \left( g^{(m)}(x) \right)^2 \, dx, \quad \lambda > 0.
\]


Eubank (1999)

Wahba (1990)

Splines are piecewise polynomials that join smoothly, that is, all of their derivatives are equal at the points where they join. Much of the development of splines has been motivated by their use as interpolants.

\( B \)-splines


Irvine, Marin, and Smith (1986)

Choice of the smoothing parameter, \( \lambda \), using cross-validation. Thomas (1991)

**** also in chapter 9 Kohn, Ansley, and Tharm (1991)

Pseudosplines Hastie (1996)

11.2.3 Adaptive Splines

multivariate adaptive regression splines (MARS) Friedman (1991) Zhang (1997) Good intro ... correlation structure

hybrid adaptive splines Luo and Wahba (1997) hybrid splines \( H \)-splines

Dias (1999)
11.2.4 Bézier Curves

Bézier developed a method of forming a smooth curve in two dimensions to connect two given points with a shape that depends on points in between. Bézier curves are used extensively in graphics applications and in geometric modeling because they are quick to compute. (See Bézier, 1986, and Mortenson, 1997.) For a given set of points in two dimensions, \( p_0, p_1, \ldots, p_n \), called control points, Bézier curves are required to satisfy two conditions:

1. The two endpoints \( p_0 \) and \( p_n \) must be interpolated.
2. The \( r \)th derivatives at \( p_0 \) and \( p_n \) are determined by \( r \) adjacent points to produce a smooth curve. The first derivative at \( p_0 \), for example, is the line determined by \( p_0 \) and \( p_1 \).

These conditions obviously do not uniquely determine the curves.

The Bézier curve is defined as the set of points \( p \) (that is, in two dimensions, \( p = (x, y) \)), defined by:

\[
p(u) = \sum_{i=0}^{n} p_i B_{i,n}(u),
\]

where, for integers \( i \) and \( n \) with \( 0 \leq i \leq n \), \( p_i \) is one of the points to be interpolated, and \( B_{i,n}(u) \) is the Bernstein polynomial,

\[
B_{i,n}(u) = \frac{n!}{i!(n-i)!} u^i (1-u)^{n-i} \quad \text{for} \quad u \in [0,1].
\] (11.8)

For example,

\[
\begin{align*}
B_{0,3}(u) &= (1-u)^3 \\
B_{1,3}(u) &= 3u(1-u)^2 \\
B_{2,3}(u) &= 3u^2(1-u) \\
B_{3,3}(u) &= u^3
\end{align*}
\]

The Bernstein polynomials are essentially the same as the beta weight function used in defining the Jacobi polynomials.

For \( n + 1 \) data points, \((x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\), we use the series of Bernstein polynomials \( B_{i,n}(u) \) for \( i = 0 \ldots n \).

Note that because of the form of the Bernstein polynomials, the sequence of points could be reversed without changing the curve.

Two quadratic Bézier curves each determined by three control points are shown in Figure 11.3.

11.2.5 Multivariate Smoothing+

Some univariate interpolation and approximation methods can be extended to address multivariate smoothing problems. The simplest extensions are achieved by tensor products.
quadratic Shepard method
Shepard method for interpolation
multivariate adaptive regression splines
MARS (multivariate adaptive regression splines)
smoothing
nonparametric regression

Figure 11.3. Quadratic Bézier Curves and Control Points


Interpolation

The tensor product extension Interpolating splines can interpolation: Renka (1988) quadratic Shepard method for 2-d and 3-d Berry and Minser (1999) extended to 5-d

Smoothing Splines


11.3 Smoothing and Nonparametric Regression


Usually the function \( f \) is relatively smooth (in the sense of having small derivatives), so fitting the model provides a “smoothing” of the data. An estimate of function \( f \) is called the “smoother” (although the term “smoother” is sometimes defined as a process or functional that yields an estimate of \( f \)),
and the set of values of the estimate of $f$ corresponding to the given values of the predictor is called the “smooth”. The smooth thus gives approximations of the values of the response variable $y$ for given values of the predictors.

In many cases of interest the predictor variables designate a category or group membership. If the properties of such categorical variables cannot be ordered in a meaningful way, the model of the relationship addresses group means (or other statistical measures within groups) and differences between pairs of groups or subsets of groups. For categorical variables such as sex (male or female) that cannot be ordered, the functional relationships such as (11.7) are relatively simple. We cannot investigate trends, shapes, or structures in the relationship. If the predictors are real-valued (or take on values that are relatively “dense”, but not necessarily in the mathematical sense of the word), the form of the function $f$ in equation (11.7) is of primary interest. Some categorical variables can be ordered, of course, and with such variables we can also speak of the trend and shape of the functional relationship in (11.7).

Instead of the linear model that relates $y_i$ to $x_i^T \beta$, we may assume only some vaguer relationship $y_i \approx f(x_i)$. This less specific relationship is nonparametric.

As in the case of the ordinary linear regression model with the error term, $\epsilon$, we seek a smoothing of the data that only approximates the $y$ values.

For the time being, let us consider the case of just one predictor variable. We might write the relationship expressed in (11.7) as $y_i \approx f(x_i)$, where the subscript indexes a set of observed data. The fitting problem is sometimes called “scatterplot smoothing” in this case.

If the predictor variable is a categorical variable, the smooth may just be taken as the average of the response values in each category, resulting in a bar chart. This type of smoothing is the conceptual basis for smoothing in other situations. If the predictor variable is continuous, the bar chart analog results from a local averaging of the response values, that is the average of response values corresponding to predictor values close to the point at which the fit is performed. The obvious questions are “how close?” and “how to average the response values?” (are all values weighted equally?, e.g.). Under any kind of averaging, the size of the neighborhood or the “bin width”, or “window width”, has a major effect on how smooth the fit is. A measure of the bin or window width is usually called a smoothing parameter. It is not necessary that all bins be the same width, but because it is simpler, that is often the case. When the bins are of equal width, the set of bins is sometimes called a mesh.

Instead of defining the neighborhoods of the predictor variable in terms of bins or windows, we could also define neighborhoods in terms of neighboring points; that is, a neighborhood may be the convex hull of the $m$ closest points to the predictor point of interest. The number of nearest neighbors is a smoothing parameter.

If the predictor is a scalar, an obvious modification is the symmetric nearest neighborhood, which is the minimal-length interval covering the $m/2$ points
540  11 Smoothing Data

smaller than the point of interest and the $m/2$ points larger than the point of interest. If the point of interest lies near the limits of the data, the number of nearest neighbors used is reduced on one side or the other as necessary.

The smoothing parameter is closely related to statistical properties of the estimate. Briefly, however, we note the intuitive fact that use of a wide neighborhood for the averaging would tend to impose a bias on the estimate, because it is data that may be less related to the values at the given point. On the other hand, use of a narrow neighborhood would not smooth out the variation in the original data, meaning that the estimate would have larger variance. The appropriate statistical criterion may depend on the mean squared error, that is, the sum of the bias squared and the variance. The smoothing parameter, therefore, provides the data analyst a mechanism to address the tradeoff between bias and variance.

11.3.1 Bin Smoothers

A very simple smoother, called a bin smoother, is constructed by first forming bins of values of the predictor, and then defining the smoothed value within a bin as an average of the response values corresponding to the predictors within the bin. The bins are defined by cutpoints, $c_0 < c_1 < \ldots < c_k$. The bins are the intervals

$$B_j = [c_{j-1} \leq x < c_j), \quad j = 1, 2, \ldots, k.$$ 

Let us denote an averaging function over some set $S$ by “Ave$(S)$”. This averaging function may be the mean of all the values in the set or it may be some function of the ordered values in the set, for example.

Whatever the actual form of the averaging function, for a given set of bins, the bin smoother at the point $x_0$ is just

$$\hat{f}_{bs}(x_0) = \text{Ave}(\{y_i, \text{ s.t. } x_i \in B_j \text{ and } x_0 \in B_j\}).$$

The number (or width) of the bins is the smoothing parameter.

Instead of defining the bins in terms of fixed cutpoints, an alternate approach is to define the bins as being intervals, perhaps of fixed width, centered about the point of interest. The bin or window moves over the data. Again whatever the actual form of the averaging function, the moving bin smoother at the point $x_0$ is just

$$\hat{f}_{mb}(x_0) = \text{Ave}(\{y_i, \text{ s.t. } |x_0 - x_i| \leq h\}),$$

where $h$ is the width of the window (usually constant), and is the smoothing parameter.

The bin smoother is a step function. An obvious problem with the bin smoother is the discontinuities that occur at the cutpoints (unless, of course, the cutpoints correspond to points at which the functional relationship between $x$ and $y$ does change discontinuously).
11.3.2 Nearest Neighbor and Moving Average Smoothers

Instead of defining the neighborhoods in terms of bins, we could define them in terms of nearest neighbors. A smoother based on averages over nearest neighbors would alleviate somewhat the jumps of the bin smoother, which is a step function. As we indicated earlier the meaning of “nearest neighbors” may be constrained for symmetry, for example. In the case of a time series, it may also be constrained so that only neighbors with smaller time values are considered (in fact, in this case, $x$ is often just a measure of time). If $N(x_0)$ is the set of nearest neighbors (according to any appropriate criteria) of the point $x_0$, the moving average smoother at the point $x_0$ is just

$$\hat{f}_{ma}(x_0) = \text{Ave}\{y_i, \text{ s.t. } x_i \in N(x_0)\}.$$ 

Moving average smoothers are also step functions and tend to jump around even more than the bin smoothers, although the jumps are smaller. The moving average smoother is likely to ignore trends near the limits of the data.

As we mentioned earlier, the averaging function $\text{Ave}(S)$ can be defined in various ways. One obvious way is to give lesser weight to observations farther away from the point of interest. There are of course many ways to do this downweighting.

A method of weighting that is especially popular in time series applications uses all past data (i.e., the number of nearest neighbors grows) is called “exponential smoothing”. In exponential smoothing, the estimate at $x_{i+1}$ is

$$\hat{f}_{es}(x_{i+1}) = \alpha x_i + (1 - \alpha)\hat{f}_{es}(x_{i-1})$$

Whenever the $x$’s are equally spaced, as is often the case in time series, when $x$ may even be a measure of time, there are several other Ave functions that have been defined and studied. Many of these methods and enhancements to existing methods are due to John Tukey (see Tukey, 1977, for further descriptions of some of them). A simple Ave function is just the median. This helps to negate the effect of outliers in the data. A running median smoother takes the smooth to be the median of a fixed number, often 3 or 5, of consecutive points.

- threeR (medians of threes, and splitting variations)
- Hanning
- regressogram
- nonparametric kernel regression.

11.3.3 Smoothing by Averaging of Functions
Running Line Smoothers

The step functions arising from simple averages over the points in neighborhoods can be modified to incorporate trends detectable in the individual neighborhoods. The simplest way of doing this is to use simple linear regression lines within each neighborhood. The regression lines can be fitted in various ways; the simplest is just ordinary least squares. This approach is obviously more appropriate for use when the neighborhoods are defined in terms of nearest neighbors. If \( N(x_0) \) is the set of nearest neighbors (according to any appropriate criteria) of the point \( x_0 \), and \( \hat{\alpha}(x_0) \) and \( \hat{\beta}(x_0) \) are the ordinary least squares estimates of \( \alpha \) and \( \beta \) using the data such that \( x \in N(x_0) \), the ordinary least squares running line smoother at the point \( x_0 \) is just

\[
\hat{f}_{lsrl}(x_0) = \hat{\alpha}(x_0) + \hat{\beta}(x_0)(x_0).
\]

Just as with flat-line averages, the data used in constructing running line smoothers can be weighted to reflect the distances from \( x_0 \) of the points used in determining \( \hat{\alpha}(x_0) \) and \( \hat{\beta}(x_0) \). One way of doing this was proposed by Cleveland (1979) in a method called lowess. (The Fortran program lowess, Cleveland, 1981, evolved into the S-Plus and R function \texttt{scatter.smooth}.)

Another way popular in time series applications is a generalization of the exponential smoothing we described earlier, in which a trend (or slope) component is maintained along with the moving mean. When both the mean and the slope are included, the process is sometimes called “double exponential smoothing” (which, in turn, can be generalized to “multiple exponential smoothing” by using slopes computed from previous adjacent points).

Elemental Subsets

least squares estimates are simple means of elemental fits Hoerl and Kennard (1980)

Hawkins (1993c)

11.3.4 Local Weighting

fitting by minimizing some function of the residuals

local weighting

locally weighted least squares ** optimal properties Stone (1977) Fan(1992)

lowess

Cleveland (1979)

Cleveland and Devlin (1988)

“Lowess”, which was a method for a single independent variable, evolved into “loess” (from wind-deposited soil) for more than one independent variable. Cleveland, Grosse, Shyu (1992)
**Kernel Smoothers**

recent realization that local polynomial regression is preferable to kernel smoothing and competitive with spline smoothing *** demonstrated by Fan (1992)***


Weights for smoothing can be defined generally by means of a *kernel function*, or just "kernel". The kernel is defined in such a way that the bin width is incorporated directly in the function itself, so its obvious use. For example, a kernel function for the moving bin smoother is just

\[ K(t) = \begin{cases} 1 & \text{for } |t| \leq h \\ 0 & \text{otherwise.} \end{cases} \]


*********** ??? data example??

**fit the short curve principle Kozek and Shuster (1991)**

***********

***More on properties of smoothers ... see Hastie and Tibshirani (1990), chapter 5, pp 105 and 118

Fan et al. (1996)

norms


Hastie and Tibshirani (1990)

Hastie (1996)

******

Seifert et al. (1994) Fast algorithms for nonparametric curve estimation,

---

**11.3.5 Spline Regression**

**11.3.6 Smoothing over Two Dimensions**

**** reorganize this section

*twicing*

Median polish

Median polish in 3 and 4 dimensions Cook (1985)

Kafadar (1994)

2-D

Median polish is dependent of the directions **** Carr??

For smoothing in two dimensions, Tukey and Tukey (1981c) suggest a method that is less dependent on directions of smoothing lines than median polish is. The method is similar to a running median using three points. Within
544 11 Smoothing Data

headbanging
kriging

a neighborhood of the point \((x_0, y_0)\) to be smoothed, the method uses pairs of points that are roughly collinear with it in such a way that \((x_0, y_0)\) is in the middle. Within a given number of nearest neighbors, there may be several pairs that roughly collinear with the given point, and so the pairs may be restricted so that the angle made by the points (with \((x_0, y_0)\) at the vertex) is between \(\pi - \theta\) and \(\pi + \theta\). Thus, there are two smoothing parameters, \(m\) the number of nearest neighbors, and \(\theta\), the angle defining collinearity. For a given pairs of points to be used according to the smoothing parameters, \((x_l, y_l)\) and \((x_h, y_h)\), suppose \(z_l \leq z_h\). The smooth at the given point is the median of the median of all the lower points (the \(x_l\)’s), the higher points (the \(x_l\)’s), and the given point.

*** endpoint modifications

*** what to do if no points The smoothing is iterated over all points until some convergence criterion is satisfied. Tukey and Tukey called this method headbanging. The headbanging adjustment at a single point is shown in Algorithm 11.1.

Algorithm 11.1 Headbanging Adjustment

1. Among the \(m\) nearest neighbors of \((x_0, y_0)\), determine all \(p\) pairs \((x_l, y_l)\) and \((x_h, y_h)\), such that the angle between the line defined by \((x_0, y_0)\) and \((x_l, y_l)\) and the line defined by \((x_0, y_0)\) and \((x_h, y_h)\) is between \(\pi - \theta\) and \(\pi + \theta\).

2. Let \(a\) be the median of all of the ***lower??? values in the \(p\) pairs and let \(b\) be the median of all of the ***lower??? values in the \(p\) pairs

3. Take the smoothed value of \(z\) as the median of \(\{a, z, b\}\).

locally weighted least squares modification Müller, Stadtmüller, and Tabnak (1997)

11.3.7 Kriging

*** spatial statistics

Cressie (1991)

see Ripley (1981) pp 44–54

see Hastie and Tibshirani (1990), p 71

see Laslett (1994)

Disjunctive kriging, indicator kriging, co-kriging ... Rivoirard (1994) see JASA (1996), p 435 (book review)

Høst (1999), Kriging by local polynomials,

Exercises
12

Density and Function Estimation

12.1 Basic Properties and Operations on Vectors and Functions

An $n$-vector can be thought of as a function whose argument takes on the values $1, 2, \ldots, n$, and whose values are the values of the elements of the vector. Thus, the domain of a vector is a set of integers, and two vectors of the same order have the same domain. The concepts and properties of vectors and vector spaces discussed in Chapter 5 can be extended in a straightforward way to vectors whose orders are countably infinite, or to functions whose domain is the set of integers, $\mathbb{Z}$. The vector concepts also carry over to functions whose domains are dense subsets of the reals, $\mathbb{R}$, but some of the properties are different for these functions.

Linear Combinations: Vectors or Functions

The addition operator for both vectors with real elements and real-valued functions is defined in terms of ordinary addition of reals. In both cases, the domains of the addends must be the same for the addition to be defined. In the case of vectors, the addition is defined by element-wise ordinary addition; in the case of functions, the addition is defined as the ordinary addition of the function values at each point in the common domain.

Multiplication of a real-valued function by a real-valued scalar is defined simply as the usual product of the function value and the scalar. A simple linear combination of functions, analogous to the axpy of vectors, is $a f(\cdot) + g(\cdot)$, “afpg”.

Linear combinations can be expressed as the multiplication of a matrix whose columns are the vectors being combined times a vector whose elements are the coefficients of the linear combination. Many other arithmetic operations on vectors can be expressed compactly as multiplication by a matrix or by another vector; for example, the mean of the elements of the $n$-vector $x$
can be expressed as $1'x/n$, where $1'$ is a row vector all of whose elements are 1’s.

There is also a variety of operations on functions. Many of these operations involve differentiation or integration.

Just as a set of vectors closed under linear combinations of vectors is called a vector space (see Section 5.1.1, page 229), a set of functions closed under linear combinations of functions is called a function space.

**Linear Independence: Vectors or Functions**

We define linear independence similarly for vectors and functions. If a given function can be formed by a linear combination of one or more functions, the set of functions (including the given one) is said to be linearly dependent; conversely, if in a set of functions no one function can be represented as a linear combination of any of the others, the set of functions is said to be linearly independent.

**Eigenanalysis: Eigenvalues and Eigenvectors or Eigenfunctions**

Many of the operations on vectors are performed as multiplication by a matrix. The matrix defines an operator that transforms one vector into another. If the matrix is square, the vector that results from the operation is in the same vector space as the vector operated on. Many important properties of this operator are determined by the eigenvalues of a matrix. Whether or not a matrix is nonsingular, whether it is positive definite, and whether some iterative methods can be shown to converge can be determined by the eigenvalues. If $A$ is an $n \times n$ (square) matrix, $v$ is a vector not equal to 0, and $\lambda$ is a scalar, such that

$$Av = \lambda v,$$

then $v$ is called an eigenvector of the matrix $A$ and $\lambda$ is called an eigenvalue of the matrix $A$.

The concepts of eigenvalues of matrices can be extended to other operators. If $\mathcal{F}$ is an operator and $g$ is a function such that

$$\mathcal{F}g = \gamma g$$

for some scalar $\gamma$, then $g$ is an eigenfunction and $\gamma$ is the corresponding eigenvalue of the operator $\mathcal{F}$. If, for example, $\mathcal{F}$ is the differential operator $d/dx$, then $g(x) = e^{\gamma x}$ is an eigenfunction with corresponding eigenvalue $\gamma$, because

$$\frac{d}{dx} e^{\gamma x} = \gamma e^{\gamma x}.$$
Inner Product Spaces: Vectors or Functions

For a vector space, after defining an inner product (or dot product) for vectors, we define an inner product vector space (page 231). The inner product can be used to define a norm (page 257). Likewise, we can define an inner product or dot product for functions, and the concepts of an inner product space and norms of functions follow naturally.

Because vectors are special cases of functions (with domains in $\mathbb{Z}$) properties of function spaces carry over immediately to vector spaces. We will refer to the general object as an inner product space. Because of the linearity of an inner product (properties 3 and 4 in the definition an inner product on page 231), an inner product space is a linear space.

12.1.1 Inner Products and Norms

The dot product of vectors is defined in terms of a sum of ordinary products, that is, as products that result from ordinary multiplication of real or complex numbers (see Section 5.1.1, page 230). The dot product of functions is naturally defined in terms of integrals of the products of the functions. Also, just as the dot product of vectors is limited to vectors with the same length, the dot product of functions is defined over some fixed range of integration (possibly infinite).

The inner product or dot product of the real functions $f$ and $g$ over the domain $D$, denoted by $\langle f, g \rangle_D$, or usually just by $\langle f, g \rangle$, is defined as

$$\langle f, g \rangle_D = \int_D f(x)g(x) \, dx,$$

if the (Lebesque) integral exists.

Dot products of functions (as well as of vectors and matrices) over the complex number field are defined in terms of integrals (or sums) of complex conjugates:

$$\langle f, g \rangle_D = \int_D f(x)\overline{g(x)} \, dx,$$

if the integral exists. The notation $\overline{g(\cdot)}$ denotes the complex conjugate of the function $g(\cdot)$. Often, even if the vectors and matrices in data analysis have real elements, many functions of interest are complex.

To avoid questions about integrability, we generally restrict attention to functions whose dot products with themselves exist; that is, to functions that are square Lebesque integrable over the region of interest. The set of such square integrable functions is denoted $L^2(D)$. In many cases, the range of integration is the real line, and we may use the notation $L^2(\mathbb{R})$, or often just $L^2$, to denote that set of functions and the associated inner product.

The Cauchy-Schwarz inequality (5.4), page 231, holds for the inner products of functions, just for as vectors, that is,
The inner product of two functions $f$ and $g$ is defined as

$$\langle f, g \rangle = \int_D f(x)g(x)w(x) \, dx,$$

if the integral exists. Often both the weight and the range are assumed to be understood, and the simpler notation $\langle f, g \rangle$ is used.

Scalar multiplication and function addition distribute over an inner product; if $a$ is a scalar and $f$, $g$, and $h$ are functions,

$$\langle af + g, h \rangle = a\langle f, h \rangle + \langle g, h \rangle.$$

(12.2)

The norm of a function $f$, denoted generically as $\|f\|$, is a mapping into the nonnegative reals that satisfy the properties that define a norm (see page 257). Because of the linearity of a norm, a space together with a norm is called a normed linear space.

The norm of a function is often defined in terms of an integral of some transformation of the function. The most common type of norm for a real-valued function is the $L_p$ norm, denoted as $\|f\|_p$, which is defined similarly to the $L_p$ vector norm as:

$$\|f\|_p = \left( \int_D |f(x)|^p w(x) \, dx \right)^{1/p},$$

(12.3)

if the integral exists. The set of functions for which these integrals exist is often denoted by $L^p(\mu; D)$. It is clear that these function norms satisfy the properties that define a norm.

A common $L_p$ function norm is the $L_2$ norm, which is often denoted simply by $\|f\|$. As with the $L_2$ vector norm, this norm is related to the inner product:

$$\|f\|_2 = (f, f)^{1/2}.$$

(12.4)

The space consisting of the set of functions whose $L_2$ norms over $\mathbb{R}$ exist together with this norm is denoted $L^2$.

Another common $L_p$ function norm is the $L_\infty$ norm, especially as a measure of the difference between two functions. This norm, which is called the Chebyshev norm or the uniform norm, is the limit of equation (12.3) as $p \to \infty$. This norm has the simpler relationship

$$\|f\|_\infty = \sup |f(x)w(x)|.$$
If \( g \) is to approximate \( f \), \( \|g - f\|_\infty \), is likely to be the norm of interest; however, if \( g \) is to be very different from \( f \), as in projection pursuit for example (see Section 13.4.5), \( \|g - f\|_2 \) may be the more appropriate norm.

To emphasize the measure of the weighting function, the notation \( \|f\|_\mu \) is sometimes used. (The ambiguity of the possible subscripts on \( \|\cdot\| \) is usually resolved by the context.) For functions over finite domains, the weighting function is most often the identity.

A normal function is one whose norm is 1. Although this term can be used with respect to any norm, it is generally reserved for the \( L_2 \) norm, that is, the norm arising from the inner product. A function whose integral (over a relevant range, usually \( \mathbb{R} \)) is 1 is also called a normal function. (Although this latter meaning is similar to the standard one, the latter meaning is broader because it may include functions that are not square integrable.) Density and weight functions are often normalized; that is, scaled so as to be normal.

### 12.1.2 Complete Spaces

For approximation methods it may be important to know that a sequence of functions (or vectors) within a given space converges to a function (or vector) in that space.

A sequence \( \{f^{(i)}\} \) in an inner product space is said to converge to \( f^* \) if given \( \epsilon > 0 \), there exists an integer \( M \), such that \( \|f^{(i)} - f^*\| \leq \epsilon \) for all \( i \geq M \). (This convergence of the norm is uniform convergence. There is also a condition of pointwise convergence of a sequence of functions, that depends on the argument of each function in the sequence.)

A sequence is said to be a Cauchy sequence if given \( \epsilon > 0 \), there exists an integer \( M \), such that \( \|f^{(i)} - f^{(j)}\| \leq \epsilon \) for all \( i, j \geq M \).

A space in which every Cauchy sequence converges to a member of the space is said to be complete.

A complete space together with a norm defined on the space is called a **Banach space**. A closed Banach space in which the norm arises from an inner product is called a **Hilbert space**.

The finite-dimensional vector space \( \mathbb{R}^d \) and the space of square-integrable functions \( L^2 \) are both Hilbert spaces. They are, by far, the two most important Hilbert spaces for our purposes. The convergence properties of the iterative methods we often employ in smoothing and in optimization methods generally derive from the fact that we are working in Hilbert spaces.

### 12.1.3 Moments

If \( p(x) \) is some measure of a density or weight (a probability density, a mass density, or a charge density, for example), the \( r \)th moment of the function \( f(x) \) with respect to \( p(x) \) is

\[
\langle f^r, p \rangle = \int_D f(x)^r p(x) \, dx.
\]
The \( r \)th moment may also be denoted as \( \langle f^r \rangle_p \).

The first moment is a measure of the average of \( f(x) \) with respect to the density and the second moment is a measure of how much \( f(x) \) varies. Moments are more useful measures if \( p(x) \) is normalized, and often it is assumed in the definition of moments that \( p(x) \) is normalized.

The first moment of the identity, \( f(x) = x \), that is, \( \langle x \rangle_p \), is called the centroid or the mean of the density \( p \). The second central moment of \( x \),

\[
\langle (x - \langle x \rangle_p)^2 \rangle_p = \int_D \left( x - \int_D x p(x) \, dx \right)^2 p(x) \, dx,
\]
is called the variance of the density.

### 12.1.4 Convolutions and Covariances and Correlations

The convolution of the functions \( f \) and \( g \) is

\[
(f \ast g)(t) = \int_D f(x)g(t-x) \, dx. 
\] (12.5)

The range of integration is usually either \([0, t]\) or \((-\infty, \infty)\). The convolution is a function, but often we write the convolution without the dummy argument: \( f \ast g \).

The convolution is a measure of the amount of overlap of one function as it is shifted over another function. The convolution can be thought of as a blending of one function with another.

Several properties follow immediately from the definition:

- commutativity:
  \[ f \ast g = g \ast f \]

- associativity:
  \[ f \ast (g \ast h) = (f \ast g) \ast h \]

- distribution over addition:
  \[ f \ast (g + h) = (f \ast g) + (f \ast h) \]

- distribution of scalar multiplication over convolution:
  \[ a(f \ast g) = (af) \ast g. \]

Although because the convolution is commutative, the two functions are essentially the same in a convolution, the second function (\( g \) in the definition above) is sometimes called the kernel. In communications engineering, the second function is sometimes a signal, and the first function is a response.

The convolution of the \( n \)-vectors \( u \) and \( v \) is
The indices of vectors in applications involving convolutions are often defined to begin at 0 instead of 1, and in that case, the lower limit above would be 0. The limits for the sum are simpler for infinite-dimensional vectors.

For functions $f$ and $g$ that integrate to zero, that is, if

$$\int_D f(x) \, dx = \int_D g(x) \, dx = 0,$$

the covariance of $f$ and $g$ at lag $t$ is

$$\text{Cov}(f, g)(t) = \int_D f(x)g(t+x) \, dx.$$  \hfill (12.7)

The argument of the covariance, $t$, is called the lag. The covariance of a function with itself is called its autocovariance. The autocovariance of a function at zero lag, $\text{Cov}(f, f)(0)$, is called its variance.

For functions $f$ and $g$ that integrate to zero, the correlation of $f$ and $g$ at lag $t$ is

$$\text{Corr}(f, g)(t) = \frac{\int_D f(x)g(t+x) \, dx}{\sqrt{\text{Cov}(f, f)(0) \text{Cov}(g, g)(0)}}.$$  \hfill (12.8)

In similar terminology to that of covariances, the argument of the correlation, $t$, is called the lag, and the correlation of a function with itself is called its autocorrelation.

The correlation between two functions is a measure of their similarity. If $f$ near the point $x$ has similar values to those of $g$ near the point $x + t$, then $\text{Corr}(f, g)(t)$ will be relatively large (close to 1). In this case, if $t$ is positive, then $f$ leads $g$; if $t$ is negative, then $f$ lags $g$. These terms are symmetric, because

$$\text{Corr}(f, g)(-t) = \text{Corr}(g, f)(t)$$

For functions that are used as models of physical processes, we generally can only observe or "sample" the function at discrete points. Convolutions and correlations are useful whenever point data are to be used in estimating functions. Applications include signal processing and probability density estimation. If the $n$-vectors $u$ and $v$ are centered at zero (that is, the sum of the elements of each vector is zero), the discrete correlation of $u$ and $v$ is

$$\text{Corr}(u, v)(t) = \frac{\sum_{1 \leq i, t-i \leq n} u_i v_{t-i}}{\sqrt{\sum_{1 \leq i \leq n} u_i^2 \sum_{1 \leq i, t-i \leq n} v_i^2}}.$$  \hfill (12.9)

The $n$-vectors $u$ and $v$ may contain sampled values of functions, and thus, the discrete correlation of $u$ and $v$ is an estimate of the correlation of the functions sampled.
12.1.5 Tensor Products

The tensor product of two real-valued functions is the ordinary product of the functions at each point in the cartesian product of the domains of the functions, that is, for the function \( f \), defined on domain \( D \), and the function \( g \), defined on domain \( E \), the tensor product of \( f \) and \( g \), denoted \( f \otimes g \), is defined as

\[
(f \otimes g)(w) = f(x)g(y) \quad \text{for} \quad x \in D, \quad y \in E, \quad \text{and} \quad w = (x, y). \tag{12.10}
\]

We also define the tensor product of the linear function spaces \( D \) and \( E \) to be the function space consisting of all tensor products of functions in \( D \) and \( E \), and denote it as \( D \otimes E \).

Several properties of the tensor products of functions follow immediately from the definition:

- associativity (with a loose definition of the cartesian products of the domains):
  \[
  f \otimes (g \otimes h) = (f \otimes g) \otimes h
  \]

- distribution over addition:
  \[
  f \otimes (g + h) = (f \otimes g) + (f \otimes h)
  \]

- distribution of scalar multiplication over convolution:
  \[
  a(f \otimes g) = (af) \otimes g.
  \]

The tensor product of the \( n \)-vector \( u \) and the \( m \)-vector \( v \) can be thought of as the outer product, \( uv^T \). (Of course, with appropriate definitions of the vectors and the matrix, the tensor product is the outer product.) Likewise, the Kronecker product of two matrices is a tensor product of the matrices.

12.2 Linear Transforms

For convolving functions or working with other integrals, it is often useful first to form an inner product with the given functions and another function that has an additional argument. The inner product with the function having an additional argument, being itself a function, is a transform of a given function. Because of the linearity of inner products, equation (12.2), these are linear transforms. Linear transforms arising from inner products include the familiar Laplace, Fourier, and wavelet transforms.

A linear transform of the function \( f \) is a operator \( T \) of the general form

\[
Tf(s) = \int_D \psi(s, x)f(x) \, dx, \tag{12.11}
\]
where the integral exists. We will denote a transform of the function \( f \) by the operator \( T \) as \( f^T \), that is,

\[
f^T = T f.
\]

The dummy arguments of the pair of functions \( f \) and \( f^T \) may range over different domains, which may correspond to different physical entities, such as time and frequency, for example.

The notation for functions and their transforms requires a word of clarification. All three of the symbols \( f \), \( T f \), and \( f^T \) represent functions. The corresponding notation in which the dummy arguments appear are the symbols \( f(x) \), \( T f(s) \), and \( f^T(s) \). We may also write both dummy arguments, as in \( T(f(x))(s) \), in which \( x \) is the argument of the function \( f \) to which the transform is being applied, and \( s \) is the argument of the transform, the function \( T f \).

From the definition (12.11) it is clear that a linear transform satisfies the relation

\[
T(af + g) = aT f + T g,
\]

where \( a \) is a constant, \( f \) and \( g \) are functions, and the transform is defined over an appropriate domain. This relation is why it is a linear transform, and of course is a property of any inner product, as in equation (12.2). The relation is called the linear property of the transform.

There are several useful transforms that correspond to specific functions \( \psi(s, x) \) and domains \( D \) in equation (12.11). In the Fourier transform, for example, \( \psi(s, x) = \exp(2\pi i sx) \), and the range of integration is the real line:

\[
\mathcal{F} f(s) = \int_{-\infty}^{\infty} \exp(2\pi i sx) f(x) \, dx.
\]

In this expression, \( i \) is the imaginary unit, \( \sqrt{-1} \). We also write the Fourier transform of the function \( f \) as \( f^F(s) \).

In the Laplace transform, \( \psi(s, x) = \exp(-sx) \), and the range of integration is the nonnegative real line:

\[
\mathcal{L} f(s) = \int_{0}^{\infty} \exp(-sx) f(x) \, dx.
\]

We also write the Fourier transform of the function \( f \) as \( f^L(s) \). The Laplace transforms of a few standard functions are shown in Table 12.1.

For a probability distribution with cumulative distribution function \( F \), the transform with \( \psi(s, x) = \exp(isx) \), and range of integration being the real line,

\[
\int_{-\infty}^{\infty} \exp(isx) \, dF(x),
\]

is called the characteristic function of the distribution, and is often written as \( \phi(s) \).
For transforms such as these based on improper integrals, the transform exists only if the integral converges. Because a distribution function is bounded between 0 and 1, the characteristic function always exists.

A linear transform with \( \psi(s, x) \propto (e^{sx})^c \) for some \( c \), such as the Fourier transform, the Laplace transform, and the characteristic function, satisfies the “change of scale property”:

\[
T(f(ax))(s) = \frac{1}{|a|} T(f(x)) \left( \frac{s}{a} \right),
\]

(12.13)

where \( a \) is a constant. This is easily shown by making a change of variables in the definition (12.11). This change of variables is sometimes referred to as “time scaling”, because the argument of \( f \) often corresponds to a measure of time. A similar scaling applies to the argument of the transform \( f^T \), which is sometimes called “frequency scaling”.

Transforms in which \( \psi(s, x) \propto (e^{sx})^c \) also have two useful translation properties:

- for a shift in the argument of \( f \),

\[
T(f(x - x_0))(s) = \psi(s, x_0) T(f(x))(s),
\]

(12.14)

- for a shift in the argument of the transform \( Tf \),

\[
T(f(x))(s - s_0) = T(\psi(-s_0, x)f(x))(s).
\]

(12.15)

These scaling and translation properties are major reasons for the usefulness of the Fourier and Laplace transforms and of the characteristic function in probability theory.

A periodic function is a function \( f \) for which \( f(t + T) = f(t) \), for some given constant \( T \), called the period of the function. The scaling properties
allow simplified representations of transforms such as the Laplace and the Fourier, and these transforms are particularly useful for periodic functions.

Another important transform is the wavelet transform, in which $\psi(s, x) = g((x - s)/t)/t$ and the range of integration is the real line:

$$\mathcal{W}^{g,t} f(s) = \int_{-\infty}^{\infty} \frac{1}{t} g \left( \frac{x - s}{t} \right) f(x) \, dx = f^{\mathcal{W}^{g,t}}(s).$$

The wavelet transformation has two adjustable parameters, $g$, the “detail function”, and $t$, the scaling parameter.

Linear transforms apply to multivariate functions as well as to univariate functions. In the definition of linear transforms (12.11), both $s$ and $x$ may be vectors. In most cases $s$ and $x$ are vectors of the same order, and specific transforms have simple extensions. In the characteristic function of multivariate random variable, for example,

$$\psi(s, x) = e^{i\langle s, x \rangle}.$$

### 12.2.1 Fourier Transforms

The Fourier transform of a function $f(x)$ is the function

$$\mathcal{F} f(s) = \int_{-\infty}^{\infty} e^{2\pi i sx} f(x) \, dx,$$

if the integral exists.

The inverse Fourier transform is

$$f(x) = \int_{-\infty}^{\infty} e^{-2\pi is} \mathcal{F} f(s) \, ds.$$ (12.17)

Instead of $e^{2\pi isx}$ as in equation (12.16), the Fourier transform is often defined with the function $e^{i \omega x}$, in which $\omega$ is called the “angular frequency”.

Fourier transforms are linear transforms, and thus enjoy the linearity property (12.12). Fourier transforms are inner products with a function of the form $(e^{sx})^c$, and thus enjoy the change of scale property (12.13), and the translation properties (12.14) and (12.15). Fourier transforms have additional useful properties that derive from the identity

$$\exp(i \omega s) = \cos(\omega s) + i \sin(\omega s),$$

in which the real component is an even function and the imaginary component is an odd function. Because of this, we immediately have the following:

- if $f(x)$ is even, then the Fourier transform is even
  $$\mathcal{F} f(-s) = \mathcal{F} f(s)$$
• if $f(x)$ is odd, then the Fourier transform is even
  \[ \mathcal{F}f(-s) = -\mathcal{F}f(s) \]

• if $f(x)$ is real, then
  \[ \mathcal{F}f(-s) = \overline{\mathcal{F}f(s)}, \]
  where the overbar represents the complex conjugate.

Fourier transforms are useful in working with convolutions and correlations because of the following relationships, which follow immediately from the definition of convolutions (12.5) and of correlations (12.7):

\[ \mathcal{F}(f * g)(s) = \mathcal{F}f(s)\mathcal{F}g(s). \quad (12.18) \]
\[ \mathcal{F}({\text{Corr}}(f, g))(s) = \mathcal{F}f(s)\overline{\mathcal{F}g(s)}. \quad (12.19) \]
\[ \mathcal{F}({\text{Corr}}(f, f))(s) = |\mathcal{F}f(s)|^2. \quad (12.20) \]

Equation (12.18) is sometimes called the “convolution theorem”. Some authors take this as the definition of the convolution of two functions. Equation (12.19) is sometimes called the “correlation theorem”, and equation (12.20), for the autocorrelation is sometimes called the “Wiener-Khinchin theorem”.

These relationships are among the reasons that Fourier transforms are so useful in communications engineering. For a signal with amplitude $h(t)$, the total power is the integral

\[ \int_{-\infty}^{\infty} |h(t)|^2 dt. \]

From the relations above, we have Parseval’s theorem, for the total power:

\[ \int_{-\infty}^{\infty} |h(t)|^2 dt = \int_{-\infty}^{\infty} |\mathcal{F}h(s)|^2 ds. \quad (12.21) \]

### 12.2.2 Wavelets

The wavelet transform of a function $f(x)$ is the function

\[ \mathcal{W}_f^g(s, t) = \int_{-\infty}^{\infty} \frac{1}{t} g \left( \frac{x-s}{t} \right) f(x) dx, \quad t > 0. \]

The analyzing wavelet, $g(x)$, is bounded and $\int g = 0$. The wavelet transform $\mathcal{W}_f^g$ maps the function $f(x)$ on $\mathbb{R}$ into the function $\mathcal{W}_f^g(s, t)$ on the upper halfplane.

**** describe methods ***

Meyer (1993)

Meyer, 1985, discovered the first orthogonal wavelet basis. Mallat introduced multi-resolution analysis which explained some of the mysteries of Meyer’s wavelet. Daubechies found a wavelet with compact support. Next came biorthogonal wavelets which were related to spline functions. Deslauriers and Dubuc, 1985, used dyadic interpolation to explain multi-resolution
12.2 Linear Transforms 557

Basic Properties of Continuous Wavelet Transforms*

**** describe ***

Exercises

12.1. The Student’s $t$ distribution is often used in statistical inference, and tables of its quantiles for various degrees of freedom are widely available. A partial table of values of $t(\nu, P)$ is shown below:

<table>
<thead>
<tr>
<th>$\nu$</th>
<th>0.95</th>
<th>0.99</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2.015</td>
<td>3.365</td>
</tr>
<tr>
<td>10</td>
<td>1.812</td>
<td>2.764</td>
</tr>
<tr>
<td>15</td>
<td>1.753</td>
<td>2.602</td>
</tr>
<tr>
<td>20</td>
<td>1.725</td>
<td>2.528</td>
</tr>
</tbody>
</table>

Use successive univariate approximation as described in Section 8.2.3 to approximate other values of $t(\nu, P)$.

a) Use a single interpolating polynomial (cubic) in $\nu$ at the two given values of $P$ and then use linear interpolation in $P$ to approximate the values $t(6, 0.975)$ and $t(19, 0.975)$.

b) Now first use four interpolating polynomials (linear) in $P$ and then use polynomial interpolation in $\nu$ to approximate the values $t(6, 0.975)$ and $t(19, 0.975)$.

c) How do your approximations compare with each other?

How do your approximations compare with the true values (to 4 significant figures), 2.447 and 2.093?

12.2. Prove equation (8.16) on page 464. *Hint:* Just expand the expression

$$\langle \sum_{i=1}^{\infty} c_i f_i, f_i \rangle.$$

12.3. Prove that the Fourier coefficients form the finite expansion in basis functions with the minimum mean squared error (that is, prove inequality (8.19) on page 464.) *Hint:* Write $\|f - a_1 f_1\|^2$ as a function of $a_1$, $\langle f, f \rangle - 2a_1 \langle f, f_1 \rangle + a_1^2 \langle f_1, f_1 \rangle$, differentiate, set to zero for the minimum, and determine $a_1 = c_1$ (equation (8.16)). This same approach can be done in multidimensions for $a_1, a_2, \ldots, a_k$, or else induction can be used from $a_2$ on.

12.4. Prove equation (8.20) on page 465. *Hint:* Write $\sum_{i=1}^{k} c_i f_i$ as $\sum_{i=1}^{k} c_i f_i + \sum_{i=k+1}^{\infty} 0 \cdot f_i$.
12.5. Prove the convergence statement in (8.22) on page 465. 

*Hint:* Let $g$ be the function to which \( \sum_{i=1}^{k} c_i f_i \) converges (by Bessel’s inequality); show that the Fourier coefficients of $g$ are the same as those of $f$, and hence that $f - g = 0$ almost everywhere. (The convergence has a necessary and sufficient relationship to completeness: given the conditions leading to the convergence statement in (8.21), except for completeness, if convergence is assured for any function in $L^2(a, b)$, then the orthonormal system is complete.)

12.6. Show that \( q_1(x) \) and \( q_2(x) \) in equations (8.23), page 466, are orthogonal.

12.7. a) Use the Legendre polynomials (equation (8.26), page 468) and a linear transformation of the interval $[-1, 1]$ onto the interval $[0, a]$ to determine the first 6 orthogonal polynomials over the interval $[0, a]$, using a constant weight function.

b) Use Gram-Schmidt transformations to derive the same set of orthogonal polynomials as in Exercise 12.7a.

12.8. Determine the first 5 discrete Legendre polynomials corresponding to the points

\[-1.0, -0.9, -0.8, \ldots, 0.9, 1.0.\]

12.9. a) Determine the first three Fourier coefficients for the function

\[ f(x) = e^{-x} \]

over the interval $[0, a]$ for the first three orthogonal polynomials from Exercise 12.7.

b) Now, for $a = 4$, determine the mean squared error of the approximation of Exercise 12.9a.

12.10. a) Use the three-term recurrence relation (8.24) to determine the Chebyshev polynomial $T_6$.

b) Use the Gram-Schmidt orthogonalization method to determine the Chebyshev polynomial $T_6$.

12.11. Let \( p_0 = (0, 1), \ p_1 = (1, 3), \ p_2 = (1.5, 3) \) and \( p_3 = (2, e^{-2}) \). Compute and plot the Bézier curve determined by these four points. Now, experiment with different positions for \( p_1 \) and \( p_2 \) to approximate the curve defined by \( f(x) = e^{-x} \) over $[0, 2]$. Plot the exponential curve and your Bézier curve together with its control points on the same set of axes.

12.12. Multiplication using FFT. *******************

Recognize that multiplication is a convolution. See Numerical Recipes, page 909.

12.13. Consider the problem of fitting the model, 

\[ y = e^{\beta_1 x + \beta_0}. \]

Suppose we have observations on $x$ and $y$:

\[(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\]
a) Write the objective function for the optimization problem of determining the least squares fit for $\beta_1$ and $\beta_0$.

b) Write the objective function for the optimization problem of determining the least absolute deviations fit for $\beta_1$ and $\beta_0$.

c) Why might one fit be preferable to another? ("Because it is a better fit" is not an acceptable answer!)

12.14. By taking derivatives to determine the minimum of expression (11.3), establish that the least-squares fit satisfies equation (11.4):

$$\bar{y} = \hat{\beta}_0 - \hat{\beta}_1 \bar{x}_1 - \cdots - \hat{\beta}_m \bar{x}_m.$$

12.15. Consider the small dataset:

<table>
<thead>
<tr>
<th>$y$</th>
<th>$x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

Use the method of Ammann and Van Ness to fit the model

$$y = \beta_0 + \beta_1 x$$

using two different criteria:

a) Minimize the sum of the squares of the distances from the points $(x_i, y_i)$ to the line $y = \beta_0 + \beta_1 x$ is a minimum. Compare this with the line for which the squares of the vertical distances are minimized. Plot both lines.

b) Minimize the sum of the distances from the points $(x_i, y_i)$ to the line $y = \beta_0 + \beta_1 x$ is a minimum. Compare this with the line for which the vertical distances are minimized. Plot both lines.
Advances in science depend on data in two critical ways: first, observational data suggest scientific principles or theories; and second, data are used to confirm scientific theory. A significant amount of the total scientific effort is devoted to collection of data, and currently data are being generated at very high rates.

Much of the data presently collected are rather complex, consisting of multiple components in a single observation. An important challenge in dealing with such data is to discover and analyze the relationships existing among the components of the data.

In this chapter we briefly discuss some of the important tools and methods of multivariate data analysis. Many methods are motivated by probability models built around the multivariate normal distribution. In this chapter, however, we will keep the underlying distributional theory to a minimum. This approach occasionally requires us to gloss over some important details. The reader with knowledge of multivariate statistical theory and methods can draw on this knowledge for deeper understanding of the issues we discuss. Other readers are referred to mathematically-oriented texts on multivariate analysis if they are interested in the background details.

13.1 Data Structures for Multivariate Data

We will speak of

structure in the data

and of

data structures.

Structure in the data, which may be clusters or other patterns, is of primary interest to the scientist. We discuss methods for discovery of structure in the data in Section 14 and subsequent sections.
Data structures are devices to help organize the data to provide better access to the individual data elements.

### 13.1.1 Variables and Observations in a Multivariate Dataset

We use the term “observation” to refer to data that relates to a single item, and the term “variable” to refer to individual elements that are measured or observed. For example, an observation in a set of medical records may relate to an individual person, and the observation may contain values of variables such as name, height, weight.

We will often denote observations by $x_1, x_2, \ldots, x_n$, each of which is a list or a vector of, say, $m$ elements. The elements of the list may be objects of various types, such as character strings, logic variables, or arbitrary scores, but in this chapter we will generally assume that the elements are real numbers, that is, $x_i \in \mathbb{R}^m$.

A multivariate dataset consists of $n$ observations, each of which consists of $m$ elements (or observations on “variables”). In the notation of the previous paragraph $x_i$ is an $m$-vector. Although we think of the observations as column vectors, we may often write them in a horizontal notation, as

$$x_i = (x_{i1}, x_{i2}, \ldots, x_{im}),$$

for example. When it is important for conformability of operands, we will use the vector transpose notation $x_i^T$ as necessary. We will represent the dataset as an $n \times m$ matrix $X$, that is, $X \in \mathbb{R}^{n \times m}$. This organization is an array data structure. Equation (13.1) is the picture to keep in mind:

$$X = \begin{bmatrix}
    x_1^T \\
    x_2^T \\
    \vdots \\
    x_n^T
\end{bmatrix} = \begin{bmatrix}
    x_{11} & x_{12} & \cdots & x_{1m} \\
    x_{21} & x_{22} & \cdots & x_{2m} \\
    \vdots & \vdots & \ddots & \vdots \\
    x_{n1} & x_{n2} & \cdots & x_{nm}
\end{bmatrix}. \quad (13.1)$$

An array data structure, which corresponds to a “flat file”, shows the relationships among the variables and the observations in a very simple, intuitive manner.

We will sometimes refer to a generic “data vector” $x$. Each of the observations $x_i$ is an instance of $x$. Often it is useful to think of $x$ as a vector-valued random variable.

The kinds of operations we may perform on a dataset depend on the nature of the elements it contains. If the data are numeric, appropriate numerical operations, such as summing, may be performed. As we have stated, in most of the following discussion we will assume that the data have the properties of the reals.
13.1 Data Structures for Multivariate Data

Statistical summary operations, such as finding the mean and variance, are generally applied to the variables, that is, to the columns of $X$; and we use notation that corresponds to that structure. For example, $\bar{x}_1$ denotes the mean of the first variable, or the first column of $X$. Notice the slight inconsistency in notation here: a single subscript on the raw data, such as $x_i$, refers to an observation (a row of $X$), but a single subscript on a summary statistic, such as $\bar{x}_j$, refers to a variable (a column of $X$). The meaning of the notation is usually clear from the context. We use the notation $\bar{X}$ to refer to the $n \times m$ matrix whose $j$th column is the constant $\bar{x}_j$. (Obviously, we would rarely store this matrix in the computer!)

13.1.2 Grouped Data

If the data have some structure, we may wish to rearrange the dataset so as to show the structure better. If there are groups or clusters in the data, we may put observations in the same group together. This may require a change in the indices in our basic paradigm of equation (13.1). We may also introduce an additional index to represent the group. In the expanded notation, we denote the $i^{th}$ observation in the $g^{th}$ group as

$$x_{i(g)} = (x_{i1(g)}, x_{i2(g)}, \ldots, x_{im(g)}).$$

As mentioned before, this is a column vector although we often represent it in a horizontal notation as above.

For grouped data, the data layout of equation (13.1) becomes

$$X_{(g)} = \begin{bmatrix}
  x_{11(1)} & x_{12(1)} & \cdots & x_{1m(1)} \\
  x_{21(1)} & x_{22(1)} & \cdots & x_{2m(1)} \\
  \vdots & \vdots & \ddots & \vdots \\
  x_{n1(1)} & x_{n2(1)} & \cdots & x_{nm(1)} \\

  x_{11(2)} & x_{12(2)} & \cdots & x_{1m(2)} \\
  x_{21(2)} & x_{22(2)} & \cdots & x_{2m(2)} \\
  \vdots & \vdots & \ddots & \vdots \\
  x_{n1(2)} & x_{n2(2)} & \cdots & x_{nm(2)} \\

  \vdots & \vdots & \ddots & \vdots \\

  x_{11(k)} & x_{12(k)} & \cdots & x_{1m(k)} \\
  x_{21(k)} & x_{22(k)} & \cdots & x_{2m(k)} \\
  \vdots & \vdots & \ddots & \vdots \\
  x_{nk1(k)} & x_{nk2(k)} & \cdots & x_{nk m(k)}
\end{bmatrix} \quad (13.2)$$
We use notation within a group similar to the notation for the full dataset, for example \( \bar{x}_{j(g)} \) refers to the mean, within the \( g \)th group, of the \( j \)th variable.

In some cases, we know what the groups are. Often, however, our objective is to find the groups from an unorganized dataset in \( X \), as in equation (13.1).

We could also of course group the variables. The layout similar to equation (13.2) would have vertical partitions. The question of relationships among the variables is important, but rather than merely forming groups of variables, we generally seek linear combinations of the variables, as we do in Section 13.4.

### 13.1.3 Graphs and Trees

Organization of a dataset as a graph may facilitate searches within the dataset to find observations that satisfy specified criteria of various types. We begin with some definitions, and then briefly discuss some special types of graphs. For general information on graphs, see Diestel (1997); for applications and algorithms, see Horowitz, Sahni, and Rajasekaran (1998); and for some applications in modeling, see Edwards (1995). For applications in nonparametric multivariate analysis, see Hartigan (1975) and Friedman and Rafsky (1979a, 1981).

A graph consists of two sets, \( V \) and \( E \), called vertices and edges. Vertices are the fundamental objects, and edges are pairs of vertices. Vertices are also called nodes. In applications for representing data, the vertices may be associated with observations, and the edges may be associated with relationships, such as distances or similarities, between the observations. Alternatively, the vertices may be associated with variables in the dataset, and the edges may represent relationships, such as correlations, between the variables.

The edges may be ordered or unordered. If a pair constituting an edge is ordered, the edge is said to be directed. If all the edges are directed, the graph is said to be directed. An edge is said to connect the two vertices in the pair. If every pair of vertices is included in the set of edges, the graph is called a complete graph. A path in a graph is a sequence of connected edges, that is, a sequence of the form

\[(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k).\]

In this representation, if \( v_1 = v_k \), the path is said to be closed. A graph in which every pair of vertices is connected by some path is called a connected graph. A simple path is a path in which no edge appears more than once, and an elementary path is a path in which no vertex appears in more than two edges. An Euler path is a simple path that contains all edges of the graph, and a Hamilton path is an elementary path that contains all vertices of the graph. Euler closed paths and Hamilton closed paths are defined in the obvious way. A cycle is a path such that all edges and all vertices are distinct, except the first and the last vertices and these are the same; for example,

\[(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k), (v_k, v_1).\]
If a graph has no cycles, it is called an *acyclic* graph.

A *tree* is an acyclic connected graph. (As we see below, “tree” is also used to refer to a special type of tree. Although the terminology is not standard, the type of structure is usually clear from the context.)

A subgraph of a given graph is a subset of the set of vertices and a subset of the set of edges in the given graph that contain only vertices in the subset of vertices. Subgraphs that have no nodes in common are said to be *disjoint*, and subgraphs that have no edges in common are said to be *orthogonal*.

An important application of graphs is to represent relationships among data. If the data are in a usual matrix structure $X$ with rows representing observations and columns representing variables, a complete graph is formed with either the observations or the variables as nodes. A real number representing “distance” or “similarity” is associated with each edge. For example, if the nodes are variables, the sample covariance between two variables may be assigned to the edge between the two nodes. If the nodes are observations, each edge may be assigned the Euclidean distance between the corresponding two observations.

### Rooted Trees

A *rooted tree* is a directed acyclic graph that has exactly one vertex or “node” (called the *root*) that no edges or “branches” enter, whose other nodes have exactly one entering branch, and which has a path from the root to each other node. A rooted tree is a tree. A minimal spanning tree with an ordering is a rooted tree if a node with only one edge is designated as the root. Because a rooted tree is such a commonly used structure, we often refer to it simply as a “tree”. A terminal node, that is, one from which no branches lead, is called a *leaf* of the tree. A tree with exactly two branches emanating from each non-terminal node, as shown in Figure 13.1, is called a *binary tree*.

A simple way to find items or positions within a sorted array of length $n$ is to inspect the middle element (or the $\lceil n/2 \rceil$th element), determine its relationship to the required condition, and then proceed to the subarray in which the required condition is satisfied, and repeat. This is a *binary search*. We can think of the search as a traversal of a special binary tree.

The root node of the binary tree used in the search corresponds to the median. The two branches from each node lead to nodes (called “children”, “sons”, or “daughters”) that correspond to the medians of two subarrays. If a leaf contains more than one element, it is often called a “bucket”. Each non-terminal node forms partitions of the array. The tree formed in this way is balanced, because the number of edges in a path from the root to any leaf is the same.

In more complicated situations, the array may be a set of keys or indices that are not uniformly spaced, so more efficient search schemes based on other trees may be desirable. These trees may not be balanced (that is, may not...
split at medians), and they may have more than two branches emanating from non-terminal nodes.

Another type of binary tree is a heap. A maximum heap is a complete binary tree such that the value associated with any node is at least as great as the value of any children of the node. A minimum heap is defined analogously.

An important consideration in constructing and working with trees is how a tree is updated when new data become available. The simple binary tree would require considerable rearrangement when new data arrive. A B-tree has the advantage of being easy to update. (See Knuth, 1973, for a discussion of B-trees and updating.)

### 13.1.4 Other Structures and Transformations of the Data

Other data structures are sometimes useful. If the dataset contains medical records of families, for example, a hierarchical structure in which a single observation may contain data on several generations of persons. The observations generally have unequal numbers of elements, and relations among variables are not as obvious as in an array structure. There are several important issues in choosing data structures for multivariate data. We refer the reader to Samet (1990) for a more extensive discussion.

Often the raw data have an array structure such as $X$, but we form other data structures in an attempt to extract information from the data. Many approaches to extract information from multivariate data are based on some method of dimension reduction. Statistics of lower dimensionality, computed from the raw data, are used to summarize salient aspects of the data. Dimen-
13.2 Measures of Similarity and Dissimilarity

A major objective in data analysis is to identify interesting features or structure in the data. These features may be relationships among variables or they may be groups or clusters of data based on similarity. When there are separate groups in the data, but the observations do not contain an element or an index representing group membership, identifying nearby elements or clusters in the data requires some measure of similarity (or, equivalently, of dissimilarity).

Although we often assume the data space is a subspace of \( \mathbb{R}^m \); a data space may be more general. Data, for example, may be character strings such as standardization, univariate standardization, multivariate (sphering) standardizing data sphered data centered data.
as names. These more general types of data may be mapped from the original data space to a “feature space”, which is a subspace of $\mathbb{R}^m$.

We may be interested in finding the nearest neighbors of a given observation based on their similarity; or, alternatively, we may be interested in identifying all observations within a given degree of closeness to a given observation. This problem is called a “proximity search”.

We begin by discussing various measures of similarity (or, equivalently, of dissimilarity). In later sections we discuss how these measures are used in finding clusters or other structures in data.

There are many ways of measuring the similarity or dissimilarity between two observations or between two variables. The most familiar measures of similarity are covariances and correlations. Other measures of similarity generally share the properties of covariances.

Dissimilarities are generally distances of some type. The dissimilarity or distance function is often a metric, which is a function $\Delta$ from $\mathbb{R}^m \times \mathbb{R}^m$ into $\mathbb{R}$ satisfying the properties

- $\Delta(x_1, x_2) \geq 0$ for all $x_1, x_2 \in \mathbb{R}^m$.
- $\Delta(x_1, x_2) = 0$ if and only if $x_1 = x_2$.
- $\Delta(x_1, x_2) = \Delta(x_2, x_1)$ for all $x_1, x_2 \in \mathbb{R}^m$.
- $\Delta(x_1, x_3) \leq \Delta(x_1, x_2) + \Delta(x_2, x_3)$ for all $x_1, x_2, x_3 \in \mathbb{R}^m$.

Nonmetric functions, such as ones allowing ties and which do not obey the triangle inequality, can also be used for defining dissimilarity, especially in applications in which there is some noise or in which there is some subjectivity in the data.

### 13.2.1 Similarities: Covariances and Correlations

Measures of similarity include covariances, correlations, rank correlations, and cosines of the angles between two vectors. Any measure of dissimilarity, such as the distances discussed in the next section, can be transformed into a measure of similarity by use of a decreasing function, such as the reciprocal. While the cosine of the angle formed by two vectors can be considered a measure of similarity, the sine can be considered a measure of dissimilarity.

Although we can consider similarities/dissimilarities between either columns (variables) or rows (observations), in our common data structures, we often evaluate covariances and correlations between columns and distances among rows. We speak of the covariance or the correlation between columns or between variables. The covariance between a column (variable) and itself is its variance.

For an $n \times m$ data matrix $X$ we have the $m \times m$ variance-covariance matrix (or just the covariance matrix):
13.2 Measures of Similarity and Dissimilarity

\[ S = \begin{bmatrix}
    s_{11} & s_{12} & \cdots & s_{1m} \\
    s_{21} & s_{22} & \cdots & s_{2m} \\
    \vdots & \vdots & \ddots & \vdots \\
    s_{m1} & s_{m2} & \cdots & s_{mm}
\end{bmatrix}, \quad (13.3) \]

where

\[ s_{jk} = s_{kj} = \sum_{i=1}^{n} (x_{ij} - \bar{x}_j)(x_{ik} - \bar{x}_k)/n - 1. \quad (13.4) \]

If \( \bar{X} \) is the matrix in which each column consists of the mean of the corresponding column of \( X \), we see that

\[ S = \frac{1}{n-1}(X - \bar{X})^T(X - \bar{X}). \]

The matrix \( S \) is therefore nonnegative definite (see page 256). The matrix \( X - \bar{X} \) is called the “centered data matrix”; each column sums to 0.

Assuming none of the variables is constant, the correlation is often a more useful measure, because it is scaled by the variances. For an \( n \times m \) data matrix the \( m \times m \) correlation matrix is

\[ R = \begin{bmatrix}
    1 & r_{12} & \cdots & r_{1m} \\
    r_{12} & 1 & \cdots & r_{2m} \\
    \vdots & \vdots & \ddots & \vdots \\
    r_{1m} & r_{2m} & \cdots & 1
\end{bmatrix}, \quad (13.5) \]

where

\[ r_{jk} = r_{kj} = s_{jk}/\sqrt{s_{jj}s_{kk}}, \]

that is,

\[ R = (\text{diag}(\sqrt{s_{11}}, \sqrt{s_{22}}, \ldots, \sqrt{s_{mm}}))^{-1} S (\text{diag}(\sqrt{s_{11}}, \sqrt{s_{22}}, \ldots, \sqrt{s_{mm}}))^{-1}. \]

The data matrix \( X \) together with either \( S \) or \( R \) is a complete graph. In this graph the columns of \( X \) constitute the vertices.

Notice that covariances and correlations are based on the L_2 norm. They are sometimes called “product-moment” covariances and correlations.

There are variations of these such as rank correlations and robust covariances. Rank correlations are computed by first replacing the elements of each column of \( X \) by the ranks of the elements within the column, and then computing the correlation as above. Robust covariances and correlations are computed either by using a different measure than the L_2 norm or by scaling of the covariance matrix based on an expectation taken with respect to a normal (or Gaussian) distribution. (“Robustness” usually assumes a normal or Gaussian distribution as the reference standard.) See Section 13.2.4, page 575, for a specific robust alternative to \( S \).
Similarities between Groups of Variables

We may want to combine variables that have similar values across all observations into a single variable, perhaps a linear combination of some of the original variables. This is an objective of the methods discussed in Section 13.4.

The general problem of studying linear relationships between two sets of variables is addressed by the method of canonical correlations. We will not pursue that topic here. The interested reader is referred to Kennedy and Gentle (1980) for the relevant computations.

13.2.2 Dissimilarities: Distances

There are several ways of measuring dissimilarity. One measure of dissimilarity is distance; and there are several ways of measuring distance. Some measures of distance between two points are based only on the elements of the vectors defining those two points. These distances, which are usually defined by a commutative function, are useful in a homogeneous space. Other measures of distance may be based on a structure imposed by a set of observations.

In a homogeneous space there are several commonly-used measures of distance between two observations. These include, for observations represented in the vectors \( x_i \) and \( x_k \)

- Euclidean distance: the root sum-of-squares of differences,
  \[
  \|x_i - x_k\|_2 = \sqrt{\sum_{j=1}^{m} (x_{ij} - x_{kj})^2};
  \]

- maximum absolute difference:
  \[
  \|x_i - x_k\|_\infty = \max_{j} |x_{ij} - x_{kj}|;
  \]

- Manhattan distance: the sum of absolute differences,
  \[
  \|x_i - x_k\|_1 = \sum_{j=1}^{m} |x_{ij} - x_{kj}|;
  \]
13.2 Measures of Similarity and Dissimilarity

- Minkowski or $L_p$ distance:
  \[ \| x_i - x_k \|_p \] (13.9)
  or
  \[ \left( \sum_{j=1}^m |x_{ij} - x_{kj}|^p \right)^{1/p} \]
  (Euclidean distance, maximum difference, and Mahattan distance are special cases, with $p = 2$, $p \to \infty$, and $p = 1$, respectively);
- Canberra distance (from Lance and Williams, 1966):
  \[ \sum_{j=1}^m \frac{|x_{ij} - x_{kj}|}{|x_{ij}| + |x_{kj}|} \] (13.10)
  so long as $|x_{ij}| + |x_{kj}| \neq 0$, otherwise, 0 (sometimes normalized by $m$ to be between 0 and 1);
- angular separation:
  \[ \frac{x_i^T x_k}{\|x_i\|_2 \|x_k\|_2} \] (13.11)
  or
  \[ \frac{\sum_{j=1}^m x_{ij} x_{kj}}{\sqrt{\sum_{j=1}^m x_{ij}^2 \sum_{j=1}^m x_{kj}^2}} \]
  For centered data, the angular separation is the correlation as in equation (13.5). Notice that, unlike the other measures of dissimilarity, the angular separation, being the cosine of the angle, is a decreasing function in what is intuitively the dissimilarity.
- For vectors composed of zeros and ones we define two useful distance measures:
  - Hamming distance: the number of bits that are different in the two vectors;
  - binary difference: the proportion of non-zeros that two vectors do not have in common (the number of occurrences of a zero and a one, or a one and a zero divided by the number of times at least one vector has a one);

Lance and Williams (1967a, 1967b, and 1968) provide a general framework for definitions of distances, and discuss the differences in the measures in cluster analysis.

The distances are elements of the $n \times n$ dissimilarity matrix,

\[
D = \begin{bmatrix}
0 & d_{12} & d_{13} & \cdots & d_{1n} \\
d_{21} & 0 & d_{23} & \cdots & d_{2n} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
d_{n1} & d_{n2} & d_{n3} & \cdots & 0
\end{bmatrix}
\] (13.12)
Notice that generally the distances are between the observations, while the covariances discussed above are between the variables.

The data matrix $X$ together with $D$ is a complete graph. In this graph the rows of $X$ constitute the vertices.

The measures of distance listed above are appropriate in a homogeneous space in which lengths have the same meaning in all directions. A scaling of the units in any of the cardinal directions, that is, a change of scale in the measurement of a single variable, may change the distances.

We see from equation (5.53) on page 260 that the $L_2$ norm of a vector is unchanged by an orthogonal transformation. Likewise, from equation (5.55) we see that the angle between two vectors is unchanged following an orthogonal transformation. Notice that the angular separation, as we have defined it, is based on the $L_2$ norm. A transformation that preserves $L_2$ distances and angles is called an “isometric transformation”. See page 270 and following for more discussion on invariances of transformations. Hence, if $Q$ is an orthogonal matrix, the Euclidean distance between $Qx_i$ and $Qx_k$ and the angular separation between those two vectors are the same as the distance and angle between $x_i$ and $x_k$. An orthogonal matrix is called an isometric matrix, because it preserves Euclidean distances and angles.

### Dissimilarities in Anisometric Coordinate Systems

If there are relationships among the variables whose observations comprise the columns of $X$, it may be appropriate to perform some scaling in the computations of distances. This is what is done in computing correlations. The correlation matrix in equation (13.5) can be computed as $\tilde{X}^T \tilde{X}$, where $\tilde{X}$ is the centered and sphered matrix,

\[ \tilde{X} = (X - \bar{X})H, \tag{13.13} \]

in which $H$ is the Cholesky factor of $S^{-1}$ (equation (13.3)), that is,

\[ H^T H = (n - 1)((X - \bar{X})^T(X - \bar{X}))^{-1} = S^{-1}. \]

(If the matrix is not of full rank, the generalized inverse is used in place of the inverse. In any case, the matrix is nonnegative definite, so the decomposition exists.)

In general, a structure may be imposed on the space by $(X - \bar{X})^T(X - \bar{X})$ or $S$. A very useful measure of the distance between two vectors is the Mahalanobis distance. The Mahalanobis distance between the $i^{th}$ and $k^{th}$ observations, $x_i$ and $x_k$ (the $i^{th}$ and $k^{th}$ rows of $X$) is

\[ (x_i - x_k)^T S^{-1}(x_i - x_k). \tag{13.14} \]
13.2 Measures of Similarity and Dissimilarity

Notice that the Mahalanobis distance is the squared Euclidean distance after using $S$ to scale the data. It is the squared Euclidean distance between rows in the $\tilde{X}$ matrix above.

There are other types of distance. Certain paths from one point to another can be specified. The distance can be thought of as the cost of getting from one node on a graph to another node. Although distances are usually considered to be symmetric (that is, the distance from point $x_i$ to point $x_k$ is the same as the distance from point $x_k$ to point $x_i$), a more general measure may take into account fluid flow or elevation gradients, and so the dissimilarity matrix would not be symmetric.

Another type of data that presents interesting variations for measuring dissimilarities or similarities is directional data or circular data; that is, data that contains a directional component. The directional component is often of the form of the measure of an angle. As the size of the angle increases, ultimately it comes close to a measure of 0. The angular separation (13.11) measures this of course, but often in directional data, one of the data elements is a plane angle (wind direction, for example). In this case the component of the overall distance between two observations $i$ and $j$ attributable to their angles, $\theta_i$ and $\theta_j$, could be taken as $d_{ij} = 1 - \cos(\theta_i - \theta_j)$. See Lund (1999) for further discussion of measures of similarity and dissimilarity and their use in clustering of directional data.

Properties of Dissimilarities

A dissimilarity measure based on a metric conforms generally to our intuitive ideas of distance. The norm of the difference between two vectors is a metric, that is, if

$$\Delta(x_1, x_2) = \|x_1 - x_2\|,$$

$\Delta(x_1, x_2)$ is a metric. Distance measures such as the $L_p$ distance, and the special cases of Euclidean distance, maximum difference, and Manhattan distance, which are based on norms of the difference between two vectors, have useful properties, such as satisfying the triangle inequality:

$$d_{ik} \leq d_{ij} + d_{jk}.$$

Gower and Legendre (1986) discuss several metric distances and other measures of dissimilarity and the properties of the measures.

Dissimilarities between Groups of Observations

In clustering applications, we need to measure distances between groups of observations. We are faced with two decisions: first, is the distance metric to use; and second, the points in the two groups between which we measure the distance. Any of the distances measures discussed above could be used.

Once a distance measure is chosen, the distance between two groups can be defined in several ways, such as the following.
• The distance between a central point, such as the mean or median, in one
cluster, and the corresponding central point in the other cluster.
• The minimum distance between a point in one cluster and a point in the
other cluster.
• The distance between clusters is the average of the distances between the
points in one cluster and the points in the other cluster.
• The largest distance between a point in one cluster and a point in the
other cluster.

13.2.3 Effects of Transformations of the Data

In the course of an analysis of data it is very common to apply various trans-
formations to the data. These transformations may involve various operations
on real numbers, such as scaling a variable (multiplication), summing all val-
ues of a variable (addition), and so on. Do these kinds of operations have an
effect on the results of the data analysis? Do they change the relative values
of such things a measures of similarity and dissimilarity?

Consider a very simple case in which a variable represents length, for
example. The actual data are measurements such as 0.11 meters, 0.093 meters,
etc. These values are recorded simply as the real number 0.11, 0.093, and so
on. In analyzing the data, we may perform certain operations (summing the
data, squaring the data, and so on) in which we merely assume that the data
behave as real numbers. (0.11 is a real number; but 0.11 meters is not a real
number. 0.11 meters is a more complicated object.) After noting the range of
values of the observations, we may decide that millimeters would be better
units of measurement than meters. The values of the variable are then scaled
by 1,000. Does this affect any data analysis we may do?

Although as a result of scaling, the mean goes from approximately \( \mu \) (for
some value \( \mu \)) to 1000\( \mu \), and the variance goes from \( \sigma^2 \) (for some value \( \sigma \)) to
1000000\( \sigma^2 \), the scaling certainly should not affect any analysis that involves
that variable alone.

Suppose, however, that another variable in the dataset is also length, and
that typical values of that variable are 1,100 meters, 930 meters, and so on.
For this variable, a more appropriate unit of measure may be kilometers. To
change the unit of measurement results in dividing the data values by 1,000.
The differential effects on the mean and variance are similar to the previous
effects when the units were changed from meters to millimeters; the effects on
the means and on the variances differ by a factor of 1,000. Again, the scaling
certainly should not affect any analysis that involves that variable alone.

This scaling, however, does affect the relative values of measures of similar-
ity and dissimilarity. Consider, for example, the Euclidean distance between
two observations, \( x_1 = (x_{11}, x_{12}) \) and \( x_2 = (x_{21}, x_{22}) \). The squared distance
prior to the scaling is

\[
(x_{11} - x_{21})^2 + (x_{12} - x_{22})^2.
\]
13.2 Measures of Similarity and Dissimilarity

Following the scaling it is

$$10^6(x_{11} - x_{21})^2 + 10^{-6}(x_{12} - x_{22})^2.$$ 

The net effect depends on the relative distances between $x_1$ and $x_2$ as measured by their separate components.

As we mention above, an orthogonal transformation preserves Euclidean distances and angular separations; that is, it is an isometric transformation. An orthogonal transformation also preserves measures of similarity based on the $L_2$ norm. An orthogonal transformation, however, does not preserve other measures of similarity or distance.

13.2.4 Outlying Observations and Collinear Variables

Many methods of data analysis may be overly affected by observations that lie at some distance from the other observations. Using a least-squares criterion for locating the center of a set of observations, for example, can result in a “central point” that is outside of the convex hull of all the data except for just one observation. As an extreme case, consider the mean of 100 univariate observations, all between 0 and 1, except for one outlying observation at 15. The mean of this set of data is larger than 99% of the data.

An outlier may result in one row and column in the dissimilarity matrix $D$ having very large values compared to the other values in the dissimilarity matrix. This is especially true of dissimilarities based on the $L_2$ norm. Dissimilarities based on other norms, such as the $L_1$ norm, may not be so greatly affected by an outlier.

Methods of data analysis that are not so strongly affected by outlying observations are said to be “robust”. (There are various technical definitions of robustness, which we will not consider here.) The variance-covariance matrix $S$ in equation (13.3), because it is based on squares of distances from unweighted means, may be strongly affected by outliers. A robust alternative is

$$S_R = (s_{Rjk}),$$

where the $s_{Rjk}$ are robust alternatives to the $s_{jk}$ in equation (13.4):

$$s_{Rjk} = \frac{\sum_{i=1}^{n} w_i^2(x_{ij} - \bar{x}_{Rj})(x_{ik} - \bar{x}_{Rk})}{\sum_{i=1}^{n} w_i^2 - 1},$$

where

$$\bar{x}_{Rj} = \frac{\sum_{i=1}^{n} w_i x_{ij}}{\sum_{i=1}^{n} w_i},$$

for a given function $\omega$,

$$w_i = \omega(d_i)/d_i,$$

and
\[ d_i = (x_i - \bar{x}_R)^T S_R^{-1} (x_i - \bar{x}_R). \]

(In this last expression, \( x_i \) represents the \( m \)-vector of the \( i \)th observation and \( \bar{x}_R \) represents the \( m \)-vector of the weighted means. These expressions are circular, and require iterations to evaluate them.)

The function \( \omega \) is designed to downweight outlying observations. One possibility, for given constants \( b_1 \) and \( b_2 \), is

\[
\omega(d) = \begin{cases} 
  d & \text{if } d \leq d_0 \\
  d_0 e^{-\frac{1}{2}(d-d_0)^2/b_2^2} & \text{if } d > d_0,
\end{cases}
\]

where \( d_0 = \sqrt{m} + b_1/\sqrt{2} \). See Campbell (1980) and Marazzi (1993) for further discussion on robust covariances and correlations. Ammann (1989, 1993) discusses the effects of outlying observations on multivariate analysis procedures such as principal components and projection pursuit. He recommends use of robust measures of association and then modifying the procedures by adaptive weighting so as to make them resistant to outliers.

A problem of a different type arises when the variables are highly correlated. In this case, the covariance matrix \( S \) and the correlation matrix \( R \), which are based on the \( L_2 \) norm, are both ill-conditioned. The ranking transformation mentioned on page 569 results in a correlation matrix that is better conditioned.

13.3 Ordering and Ranking Multivariate Data

The concept of order or rank within a multivariate dataset can be quite complicated. The simple approach of defining a “sort key” that consists of a priority ordering of the variables to use in ranking the data is not very useful except in simple, well-structured datasets.

13.3.1 Minimal Spanning Trees

A spanning tree for a graph is a tree subgraph that contains all nodes of the given graph. A spanning tree is not necessarily rooted. A useful graph of observations is the spanning tree whose edges have the least total distance. This is called a minimal spanning tree, or MST. It is obvious that the number of edges in a minimal spanning tree would be one less than the number of nodes. A minimal spanning tree may not be unique.

Kruskal (1956) gave the method shown in Algorithm 13.1 for forming a minimal spanning tree. The set of all edge distances should first be put into a minimum heap, so that the updating can proceed more rapidly.

**Algorithm 13.1 Formation of a Minimal Spanning Tree \( T \) from a Connected Graph with \( n \) Nodes and with Edge Distances in \( H \)**

0. Set \( T = \emptyset \) (the empty set), and set \( k = 0 \).
Dijkstra’s algorithm

1. Choose the edge $e$ from $H$ with shortest distance.
2. Remove $e$ from $H$.
3. If $e$ does not create a cycle in $T$, then add $e$ to $H$ and set $k = k + 1$.
4. If $k < n - 1$, go to step 1.

In the following few pages we will use a simple bivariate dataset for illustrations. The dataset is shown in Table 13.1. A plot of the data and a minimal spanning tree for this simple bivariate dataset is shown in Figure 13.2. In most cases when we display this dataset we will suppress the scales on the axes.

A related problem is to determine the shortest path between two given nodes. An algorithm to determine the shortest path, called Dijkstra’s algorithm, is described in Horowitz, Sahni, and Rajasekaran (1998), for example.

A tree that connects observations with nearby ones helps us to understand the distribution of the observations; to identify clusters of observations and outlying observations. The number of edges in the longest path starting at any node is called the *eccentricity* of that node or observation. The node most distant from a given node is called an *antipode* of the node; and the path between a node with greatest eccentricity and its antipode is called a *diameter* of the tree. The length of such a path is also called the diameter. (This word also carries both meanings in the familiar context of a circle.) A node with minimum eccentricity is called a *center* node, or a *median*. The median of the tree shown in the right plot in Figure 13.2 is observation number 6 (as labeled in the left plot in Figure 13.2).

<table>
<thead>
<tr>
<th>Obs.</th>
<th>Number $x_1$ $x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10 66</td>
</tr>
<tr>
<td>2</td>
<td>19 52</td>
</tr>
<tr>
<td>3</td>
<td>8 88</td>
</tr>
<tr>
<td>4</td>
<td>37 25</td>
</tr>
<tr>
<td>5</td>
<td>66 75</td>
</tr>
<tr>
<td>6</td>
<td>53 55</td>
</tr>
<tr>
<td>7</td>
<td>89 76</td>
</tr>
<tr>
<td>8</td>
<td>73 91</td>
</tr>
<tr>
<td>9</td>
<td>21 32</td>
</tr>
<tr>
<td>10</td>
<td>12 23</td>
</tr>
<tr>
<td>11</td>
<td>29 41</td>
</tr>
<tr>
<td>12</td>
<td>86 65</td>
</tr>
<tr>
<td>13</td>
<td>91 81</td>
</tr>
<tr>
<td>14</td>
<td>42 23</td>
</tr>
<tr>
<td>15</td>
<td>36 38</td>
</tr>
<tr>
<td>16</td>
<td>90 85</td>
</tr>
</tbody>
</table>

Table 13.1. Dataset for Illustrations
Friedman and Rafsky (1979a) defined a method for sorting multivariate observations based on a minimal spanning tree. The procedure is to define a starting node at an endpoint of a tree diameter, and then to proceed through the tree in such a way as to visit any shallow subtrees at a given node before proceeding to the deeper subtrees. For the tree shown in the right plot in Figure 13.2, for example, if we choose to begin on the right of the tree, the next seven nodes are in the single path from the first one. Finally, at the eighth node, we choose the shallow subtree for the ninth and tenth nodes. The eleventh node is then the other node connected to the eighth one. This ordering is shown in the left plot in Figure 13.3. Friedman and Rafsky (1979b) described an algorithm for this ranking, which they called a “linear” ranking. They also described another one-dimensional ranking that begins near the center of the minimal spanning tree, and moves outward. This ranking, which they called a “radial” ranking, is shown in the right plot in Figure 13.3. They called these one-dimensional rankings “linings”.

Friedman and Rafsky (1979b) also defined a mapping of the multivariate data to a plane in which the interpoint distances in the plane maintain the ordering of the interpoint distances of the multivariate data. They called this mapping a “planing”. They give an algorithm for planing (Friedman and Rafsky, 1979b) and describe applications of planing for producing multivariate p-p plots (Friedman and Rafsky, 1981). After planing of multivariate data, the data can be ranked using a two-dimensional scheme.
13.3 Ordering and Ranking Multivariate Data

The R function \texttt{mstree} computes a minimal spanning tree, the $x$-$y$ coordinates of a planing of multivariate data, and both the linear and radial ordering of the data.

Minimal spanning trees have a variety of uses in multivariate analysis. Zahn (1971) and Bhavasar and Ling (1988) give other examples of their use for discovering multivariate structure. Using orderings by minimal spanning trees, Friedman and Rafsky (1979) developed tests for equality of distributions of two samples, and Friedman and Rafsky (1981) used them to define multivariate p-p plots for comparing two multivariate samples. Banks and Lavine (1992) defined nonparametric regression methods based on minimal spanning trees. Kwon and Cook (1998) used minimal spanning trees in grand tour visualization.

There are of course several other ways of using a graph to define a sequence or ranking of multivariate data. (See Barnett, 1976, and Eddy, 1985, for general discussions of the problem.) A connected acyclic graph that contains only one path would be a reasonable possibility. (A cyclic path that includes every node exactly once is called a “Hamiltonian circuit”.) Such a graph with minimal diameter is the solution to the traveling salesperson problem. The traveling salesperson problem is computationally more complex than determining the minimal spanning tree.

\textbf{Figure 13.3.} Two Orderings of the Dataset Using the MST grmu531

\begin{itemize}
  \item p-p plot
  \item grand tour (in graphics)
  \item Hamiltonian circuit
\end{itemize}
Convex Container Hulls

Another way to order data is by convex hull peeling (see Eddy, 1981, 1982). The idea behind convex hull peeling, which is due to John Tukey, is that the convex hull of a dataset identifies the extreme points, and the most extreme of these is the one whose removal from the dataset would yield a much “smaller” convex hull of the remaining data. In two dimensions convex hull peeling takes as the most extreme observation the one on the convex hull with the smallest angle. Next, the convex hull of all remaining points is determined, and the second most extreme observation is the one on this convex hull with the smallest angle. This process continues until a total ordering of all observations is achieved. The first few steps are shown in Figure 13.4. The ordering by convex hull peeling tends to move around the edges of the set of points is often similar to the radial ordering in a minimal spanning tree.

Figure 13.4. Ordering of the Dataset (Table 13.1) by Peeling the Convex Hull

An algorithm for computing the convex hull of a multivariate dataset is given by Barber, Dobkin, and Huhdanpaa (1996). Various programs for computing convex hulls and other problems in computational geometry are available at the site,

www.geom.umn.edu/software/download/

The convex hull of a two-dimensional dataset is particularly easy to compute (see Eddy, 1977, for example).
Instead of a convex hull, formed by planes, we may consider a smoothed figure such as an ellipsoid with minimum volume. Hawkins (1993a); Cook, Hawkins, and Weisberg (1993); and Woodruff and Rocke (1993) give algorithms for computing the minimum volume ellipsoid. The algorithms in the last reference include various heuristic combinatorial algorithms such as simulated annealing, genetic algorithms, and tabu searches. The minimum volume ellipsoid that contains a given percentage of the data provides another way of ordering the points in a multivariate dataset.

Depth

A peeled convex hull or an ellipsoid containing a given percentage of data provides an ordering of the data from the outside in. Another approach to ordering data is to begin in the inside, that is at the densest part of the data and proceed outward. For bivariate data, John Tukey introduced the concept of halfspace location depth for a given point \( x_c \) relative to the dataset \( X \), whose rows are in \( \mathbb{R}^2 \). The halfspace location depth, \( d_{hsl}(x_c, X) \), is the smallest number of \( x_i \) contained in any closed halfplane whose boundary passes through \( x_c \). The halfspace location depth is defined for datasets \( X \) whose rows are in \( \mathbb{R}^m \) by immediate extension of the definition for the bivariate case. Figure 13.5 shows some halfspaces defined by lines, together with the counts of points on either of the lines.
There are 15 possible pairs of counts for each point in Figure 13.5 (no three points in this dataset are collinear). Some halfplanes for which point $A$ lies on the boundary contain as few as six points (one such halfplane is shown in the figure), but all contain at least six points. Thus,

$$d_{\text{hsl}}(a, X) = 6.$$  

Of all the points in this dataset, $a$ has the greatest halfspace location depth. The halfspace location depth of point $b$, $d_{\text{hsl}}(b, X)$, for example, is 4. The halfplane shown in Figure 13.5 with point $b$ on the boundary contains 5 points. A clockwise rotation of that boundary line yields a halfplane containing 4 points.

The halfspace location depth provides another way of ordering the data. There are generally many ties in this ordering.

Rousseeuw and Ruts (1996) provide an algorithm for computing the halfspace location depth for bivariate data. (See also Ruts and Rousseeuw, 1996, who discuss contours of regions with equal location depth.)

Ordering by location depth emphasizes the interior points, whereas convex hull peeling emphasizes the outer points. The outer points have a halfspace location depth of 0, and generally the first few points removed in convex hull peeling, such as points 15 and 16 in Figure 13.4, have a location depth of 0. The point that is removed next, however, has a location depth of 1; whereas there are other points in the dataset with location depths of 0.

If a single point has a greater location depth than any other point in the dataset, the point is called the depth median. If multiple points have the largest location depth of any in the dataset, the depth median is the centroid of all such points.

The depth median of the dataset shown in Figure 13.5 is the point labeled “a”. That is the same as observation number 6 as labeled in the left plot in Figure 13.2 (page 578), which was the median of the minimal spanning tree shown in the right plot in Figure 13.2.

Determination of the depth median is very computationally intensive. Rousseeuw and Ruts (1998) give an algorithm for computing the depth median in a bivariate dataset, and Struyf and Rousseeuw (2000) give an approximate algorithm for higher-dimensional data.

There are other ways of defining data depth. Liu (1990) defined the simplicial location depth as the proportion of data simplices (triangles formed by three observations in the bivariate case) that contain the given point. Rousseeuw and Hubert (1999) define and study a depth measure based on regression fits.

**Ordering by Clustering**

Clustering also provides a way of ordering data. The ordering that arises from clustering whether divisive or agglomerative, however, depends on local properties, and a global ordering is difficult to identify. A hierarchical clustering
of the data in Table 13.1 is shown in Figure 13.6. Comparison of the cluster
tree with the scatterplot in Figure 13.2 (page 578) shows how nearby points
are grouped first. In this dataset the points closest together are on the pe-
riphery of the data cloud. For a cloud of points that is concentrated around
the median, as is perhaps more common, the central points would be grouped
first.

13.3.2 Nearest Neighbors and $k$-$d$-Trees

For a given observation, $x_i$, we may want to find its “nearest neighbor”, $x_k$,
where we define a nearest neighbor as one for which some function, $f(x_i, x_k)$,
is minimized. For example, $f$ may be the Euclidean distance,

$$
\sqrt{\sum_{j=1}^{m} (x_{ij} - x_{kj})^2}.
$$

To search bivariate data for a point that is close in Euclidean distance to
a given point, a quad tree is useful. (See Knuth, 1973.)

For the more general problem of finding nearest neighbors in multivariate
data, a $k$-$d$-tree developed by Friedman, Bentley, and Finkel (1977) may be
more useful. A $k$-$d$-tree is a multivariate form of a B-tree; see Bentley and
Friedman (1979).
Consider an $n \times m$ data matrix $X$, in which columns represent variables and rows represent observations. A $k$-$d$-tree for $X$ is defined by two arrays, $v$, which contains indicators of the variables to be used as discriminators, and $p$, which contains values of the corresponding variables to be used in forming partitions. Let $b$ be the maximum bucket size, that is, the most number of elements to be left at a terminal node.

**Algorithm 13.2 Formation of a $k$-$d$-Tree**

0. Set $l = 1$ and $h = n$.
1. Let $k = \lfloor (l + h)/2 \rfloor$.
2. Let $v_k$ be the column number with maximum spread.
3. Let $p_k$ be the median in the range $[l, h]$ of the $v_k^{th}$ column.
4. Interchange the rows of $X$ so that all rows of $X$ with values in the $v_k^{th}$ column less than or equal to $p_k$ occur before (or at) the $k^{th}$ element.
5. If $k - l > b$, then form a submatrix with $h = k$.
   If $h - k - l > b$, then form a submatrix with $l = k + 1$ (with $h$ as in step 4).
   Process steps 1 through 4 for each submatrix formed and then return to step 5.

   While trees are used often at a lower level, there is not much software available at the user-level to form trees. The IMSL Fortran routine QUADT builds a $k$-$d$-tree; and the routine NGHBR uses a $k$-$d$-tree to find nearest neighbors.


### 13.3.3 Space-Filling Curves

Another approach to ordering a multivariate dataset is by means of a space-filling curve, which is a continuous map from a closed interval in $\mathbb{R}$ onto a compact set in $\mathbb{R}^d$. Because the first such curve was constructed by Peano, space-filling curves are also called Peano curves. Such curves were of great interest in late nineteenth-century and early twentieth-century mathematics, and their study led to many advances in analysis. (See Sagan, 1994, for a comprehensive treatment.) Space-filling curves have a number of applications in numerical and statistical analysis. For example, the ordering a space-filling curve induces can be used to traverse a grid in solving a partial differential equation, and the ordering can be used to determine a rotation of a coordinated system for graphical display of multivariate data. In such applications, of course, we use an approximation to a space-filling curve.

Space-filling curves are usually defined by piecewise curves that pass through all regions of each tessellation of a space in a sequence of tessellations in which the sizes of the regions go to zero. The simplest tessellation, and one commonly used in the construction of space-filling curves, is a rectangular tessellation.

One of the most widely-used space-filling curves is the *Hilbert curve*. A sequence of approximations to this curve is constructed by edges of hypercubes.
that are successively halved. Figure 13.7 shows the first three polygonal curves in the sequence whose limit is a Hilbert curve. The arrows along the edges of the squares in the left and middle diagrams indicate how the polygonal curves are constructed.

Figure 13.7. The First Three Approximating Polygons for a Hilbert Curve

The Hilbert curve is built from a sequence of the basic unit shapes shown in Figure 13.8. These shapes can be seen in the thicker lines in Figure 13.7. Their orientation is determined by the sequence of directions shown by the arrows. Each time the squares are halved, the beginning arrow in the lower left-hand corner changes direction; for example, the first arrow in the diagram on the right would point upward. The unit shapes are connected at their nearest terminal points by straight line segments, as shown in the middle and right diagram in Figure 13.7.

Figure 13.8. Four Orientations of the Unit Shape

A Hilbert curve maps the interval \([0, 1]\) onto the unit square (or any other two-dimensional rectangle). This can be seen by realizing that any point in the square (rectangle) lies in a sequence of nested closed squares, the diagonals of which shrink into the point. Corresponding to this sequence, there is a sequence of nested closed subintervals of \([0, 1]\), the lengths of which shrink to zero. The Hilbert curve, by construction, maps that limit point in \([0, 1]\) to the chosen point in the square. (The mapping is not one-to-one, however.) By a
similar argument utilizing nested sequences in \([0, 1]\) and in the square, we see that a Hilbert curve is continuous. (It is nowhere differentiable, however; see Sagan, 1994.)

A Hilbert curve can be constructed by recursive application of the steps indicated in Figure 13.7, using the orientations of the basic shape in Figure 13.8. Breinholt and Schierz (1998) give a recursive program to construct Hilbert curves.

There are a number of other space-filling curves that have been studied. The Lebesgue space-filling curve, which has a zigzag pattern over successive pairs of rectangles, is often used to traverse a rectangular grid in the solution of a partial differential equation. The Sierpiński space-filling curve is often used in fractal illustrations.

Space-filling curves can be constructed in higher dimensions, of course. A three-dimensional Hilbert curve, for example can be constructed in a similar manner as the two-dimensional curve by extending the squares into cubes.

Point sets built recursively, such as a Hilbert curve, are fractals, which are self-similar on all scales. (Because the fractal dimension of a Hilbert curve is 2, some authors would not call it a fractal, because they reserve this term for objects with nonintegral fractal dimension.)

Ordering and Ranking of Transformed Data

Minimal spanning trees depend on relative distances between points, so as we would expect, the minimal spanning tree and any ordering based on it may be different if the data are transformed. Likewise, of course, orderings based on clustering may be changed by transformations of the data.

An important property of convex hulls and the depth of data is that these are not affected by affine transformations.

13.4 Linear Structures in Data

In addition to clustering and ordering data, a major objective is often to reduce the dimension of the data, so as to understand it better. We begin with observations consisting of \(m\) components and we seek a smaller set of \(k\) components that carry most of the information in the original set. The problem is to transform the \(m\)-vector \(x\) into a \(k\)-vector \(\tilde{x}\). Dimension reduction may be useful in identifying structure in the data or in discovering properties that the data do not measure directly. We may wish to extract features in the data that been obscured by measurement error or other sources of noise.

13.4.1 Flats

The set of points \(x\) whose components satisfy a linear equation
is called a flat. Such linear structures often occur (approximately) in observational data, leading to study of the linear regression model,

\[ x_d = \beta_0 + \beta_1 x_1 + \cdots + \beta_m x_m + \epsilon. \]

A flat through the origin, that is, a set of points whose components satisfy

\[ b_1 x_1 + \cdots + b_d x_d = 0, \]

is a vector space. Such equations allow simpler transformations, and so we often transform regression models into the form

\[ x_d - \bar{x}_d = \beta_1 (x_1 - \bar{x}_1) + \cdots + \beta_m (x_m - \bar{x}_m) + \epsilon. \]

The data are centered to correspond to this model.

### 13.4.2 Rotations and Projections

Two major tools in seeking linear structure are rotations and projections of the data matrix \( X \). Rotations and projections of the observations are performed by postmultiplication of \( X \) by special matrices. We discuss linear rotation operators in general on page 271, and projections on page 250. In this section we briefly review these types of matrices for use in multivariate data analysis.

The \( m \times m \) orthogonal matrix

\[
Q_{pq}(\theta) = \begin{bmatrix}
1 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\
0 & 1 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & \cdots & 0 & \cos \theta & 0 & \cdots & \sin \theta & 0 & \cdots & 0 & \cdots \\
0 & 0 & \cdots & 0 & 0 & 1 & \cdots & 0 & 0 & \cdots & 0 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & 0 & 0 & \cdots & 1 & 0 & 0 & \cdots & 0 & \cdots \\
0 & 0 & \cdots & 0 & -\sin \theta & 0 & \cdots & \cos \theta & 0 & \cdots & 0 & \cdots \\
0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 1 & \cdots & 0 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \cdots & 0 & 0 & \cdots & 0 & 0 & \cdots & 1 & \cdots & \cdots
\end{bmatrix},
\]

in which \( p \) and \( q \) denote the rows and columns that differ from the identity, rotates the data vector \( x_i \) through an angle of \( \theta \) in the plane formed by the \( p^{th} \) and \( q^{th} \) principal axes of the \( m \)-dimensional cartesian coordinate system. This rotation can be viewed equivalently as a rotation of the coordinate system in the opposite direction. The coordinate system remains orthogonal after such
a rotation. In the matrix $XQ$ all of the observations (rows) of $X$ have been rotated through the angle $\theta$.

How a rotation can reveal structure can be seen in Figures 13.9 and 13.10. In the original data there does not appear to be any linear relationships among the variables. After applying a rotation about the third axis, however, as we see in the scatter plot in Figure 13.10 a strong linear relationship between the first and third variables of the rotated data.

**Figure 13.9.** Scatter Plot Matrix of Original Data grmu605

Rotations of the data matrix provide alternative views of the data. There is usually nothing obvious in the data to suggest a particular rotation. Dynamic rotations, however, are very useful in revealing structure.

A rotation of any plane can be formed by successive rotations of planes formed by two principal axes. See page 294 for some comments on computations involving multiple rotations.

Another way of getting useful alternative views of the data is to project the data onto subspaces. A symmetric idempotent matrix $P$ “projects” vectors onto the subspace spanned by the rows (or columns) of $P$. Except for the identity matrix, a projection matrix is of less than full rank; hence, it projects a full-rank matrix into a space of lower dimension. While we may only know that the rows of the data matrix $X$ are in $\mathbb{R}^m$, the rows of $XP$ are in the subspace spanned by the rows of $P$. It may be possible to identify relationships and structure in this space of lower dimension that are obscured in the higher dimensional space.
13.4.3 Linear Principal Components

For the $n \times d$ matrix of data $X$, we wish to determine a transformation of the row vectors $x_i^T$ so as to capture the maximum amount of variation in the row vectors. We can represent the variation of the data in terms of an $d$-dimensional ellipsoid formed by the quadratic form

$$(x - \bar{x})^T S^{-1} (x - \bar{x}),$$

where $\bar{x}$ is the vector of means of the columns of $X$, that is, $X^T 1_n / n$, and $S^{-1}$ is the inverse of the sample variance-covariance matrix,

$$S = \frac{1}{n-1} (X - 1_d \otimes \bar{x}^T)(X - 1_d \otimes \bar{x}^T)^T.$$

Here $1_d$ denotes a vector of length $d$ containing all 1’s and $\otimes$ denotes Kronecker multiplication. The major axis of an ellipsoid defined by the quadratic form represents the direction in which the variation of the data is the greatest. The next largest axis, which is perpendicular to the major axis, represents the direction in which the variation of the data is second greatest. Figure 13.13 is an illustration for an ellipse, that is, for $d = 2$. The original coordinates $x_1$ and $x_2$ are translated by shifting the origin to $(\bar{x}_1, \bar{x}_1)$, and then rotated about this origin to the principal axes of the ellipse. These axes are the principal components of the dataset $X$.

The problem of finding the direction of maximum variation is that of finding the normalized vector $a$, such that the sample variance of $a^T x$ is maximized. The sample variance of $a^T x$ is $a^T S a$, so the problem reduces to solving
From equation (5.41), page 255, we know this maximum is the maximum eigenvalue of $S$, and occurs when $a$ is the corresponding eigenvector. The first principal component, therefore, is the eigenvector of $S$, call it $z_1$, corresponding to the maximum eigenvalue of $S$. If the largest eigenvalue has multiplicity $m$, we choose the corresponding eigenvectors to be orthogonal and they together constitute the first $m$ principal components. After the first principal component or components have been determined, the second principal component is the direction orthogonal to the first principal components along which the greatest variation in the data occurs. If $c_1, c_2, \ldots, c_m$ are the $m$ largest eigenvalues with corresponding orthogonal eigenvectors $z_1, z_2, \ldots, z_m$, the next largest principal component corresponds to the largest eigenvalue of the matrix

$$S - \sum_{i=1}^{m} c_i z_i z_i^T.$$  

(See page 313.) This is also the $(m + 1)^{st}$ eigenvalue of $S$, so the problem of determining all the principal components is merely the problem of finding the eigenvectors of $S$.

Figure 13.14 shows the marginal distributions of the data in Figure 13.13 in the original coordinates $x_1$ and $x_2$ and in the coordinates of the principal components $z_1$ and $z_2$. 

**Figure 13.11.** Principal Components of Some Bivariate Data grme505
The dimension of the original data set can be reduced by using only a few of the principal components corresponding to the largest eigenvalues of $S$. A practical issue, of course, is how many principal components to retain. The decision is often based to some extent on gaps in the magnitudes of the descending sequence of eigenvalues. Axes corresponding to the smallest eigenvalues may be considered negligible. The data do not vary much along those directions.

Another major issue in the use of principal components arises from the fact that they are not invariant under linear transformations of the original data. Even separate scaling of the variables results in different principal components; thus, if the correlation matrix is used instead of the variance-covariance matrix, the principal components will be different. Because a variable with a large variance compared to that of the other variables will tend to determine the first principal component, it is often a good idea to scale the data by the standard deviations prior to extracting the principal components. This lack of invariance of principal components makes interpretation and statistical inference difficult.

**Principal Components Analysis of Data**

In the basic multivariate data structure of $X$ in equation (13.1), we often consider the rows to be realizations of some multivariate random variable, such as $Y$ in the discussion above. The variance-covariance of the underlying
random variable may be estimated by $S$ from equation (13.3), page 569. If the underlying random variable has a multivariate normal distribution, the estimate $S$ has some optimal properties among all estimators of $\Sigma$; in other cases, it is usually a reasonable estimator.

Using $S$ as an estimate of $\Sigma$, we can perform a principal components analysis of the data that follows the same techniques as above for a random variable. We determine the spectral decomposition of $S$, just as we did for $\Sigma$ in equation (13.3):

$$S = \sum_j \hat{\lambda}_j \hat{w}_j \hat{w}_j^T.$$ (13.16)

The principal components of the vector of observed variables $x$ are

$$\tilde{x}_{(j)} = \tilde{w}_j^T x.$$ (13.17)

Corresponding to the generic data vector $x$ is the generic vector of principal components,

$$\tilde{x} = (\tilde{x}(1), \tilde{x}(2), \ldots, \tilde{x}(m)).$$

For each observation $x_i$, we can compute a value of the principal components vector, $\tilde{x}_i$. From the spectral decomposition that yields the principal components, it is easy to see that the sample variance-covariance matrix of the principal components is diagonal.

Figure 13.14 shows the marginal distributions of the elliptical data cloud shown in Figure 13.13 in the original coordinates $x_1$ and $x_2$ and in the coordinates of the principal components $z_1$ and $z_2$.

The first principal component is the hyperplane that minimizes the orthogonal distances to the hyperplane as discussed on page 531, and illustrated in Figure 11.2.

**Dimension Reduction by Principal Components Analysis**

We can reduce the dimension of the data by considering the transformed variables $\tilde{x}_{(j)}$, each of which is a vector formed using the eigenvectors corresponding only to the $p$ largest eigenvalues. As before, we form the $p \times m$ transformation matrix $\tilde{W}$,

$$\tilde{W} = \begin{bmatrix}
\tilde{w}_1^T \\
\tilde{w}_2^T \\
\vdots \\
\tilde{w}_p^T
\end{bmatrix}.$$ 

For the $i^{th}$ observation $x_i$, this produces the $p$-vector $\tilde{x}_i = (\tilde{x}_1(i), \tilde{x}_2(i), \ldots, \tilde{x}_p(i))$. 
The application of this transformation matrix is sometimes called the (discrete) Karhunen-Loève transform (usually in signal analysis) or the Hotelling transform (usually in statistics).

The question arises, naturally, of how to choose $p$. This is the question of how much can we reduce the dimensionality of the original dataset. A simple approach that is often employed is to choose $p$ as the number of the ranked eigenvalues just prior to a large gap in the list. For example if $m = 6$ and the eigenvalues are $10.0, 9.0, 3.0, 2.5, 2.1, 2.0$, a logical choice of $p$ may be 2, because of the large decrease after the second eigenvalue. A plot of these ordered values or of the values scaled by their total may be useful in identifying at what point there is a large dropoff in effect. Such a plot, called a scree plot, is shown as the left plot in Figure 13.15. The scree plot can be either a line plot as in the figure or a bar chart in which the heights of the bars represent the relative values of the eigenvalues. The key feature in a scree plot is an “elbow”, if one exists. A plot of the accumulated “total variation” accounted for by the principal components, as shown in the right plot in Figure 13.15, may also be useful.

The effect of each of the original variables (the elements of $x$) on each principal component is measured by the correlation between the variable and the principal component. This is called the “component loading” of the variable on the principal component. The component loading of the $j^{th}$ variable on the $k^{th}$ principal component is the correlation.
Figure 13.14. Univariate Histograms of the Original Coordinates and the Principal Components

\[
\frac{w_{kj} \sqrt{\lambda_k}}{\sqrt{s_{jj}}}
\]

(Note that \(w_{kj}\) is the \(j^{th}\) element of the \(k^{th}\) eigenvector.)

Principal Components and Transformations of the Data

As we mentioned at the outset, variation provides information. Variables with large sample variances will tend to predominate in the first principal component. Consider the extreme case in which the variables are uncorrelated, that is, in which \(S\) is diagonal. The principal components are determined exactly by the variances, from largest to smallest. This is a natural and desirable consequence of the fact that variation provides information. In principal components analysis, the relative variation of one variable to another is the important factor in determining the rankings of the components. It is difficult, however, to measure the relative variation of one variable to another. The variance of a variable depends on the units of measurement. Suppose that one variable represents linear measurements in meters. If for some reason, the unit of measurement is changed to centimeters, the effect of that variable in determining the principal components will increase one hundredfold.

The component loadings can help in understanding the effects of data reduction through principal components analysis. Notice that the component loadings are scaled by the square root of the variance. Another approach
to scaling problems resulting from choice of unit of measurement is to use the correlation matrix, $R$ (see equation (13.5)), rather than the variance-covariance matrix. The correlations result from scaling the covariances by the square roots of the variances. The obvious should be noted, however, the principal components resulting from use of $R$ are not the same as those resulting from use of $S$.

Change of units of measurement is just one kind of simple scaling transformation. Transformations of any kind are likely to change the results of a multivariate analysis, as we see, for example, in the case of clustering in Section 14.2.8, on page 624.

**Principal Components of Observations**

Just as in Section 14, on page ??, we observed the basic symmetry between the “variables” and the “observations” of the dataset $X$, we can likewise reverse their roles in principal components analysis. Suppose, for example, that the observational units are individual persons and the variables are responses to psychological tests. Principal components analysis as we have described it would identify linear combinations of the scores on the tests. These principal components determine relationships among the test scores. If we replace the data matrix $X$ by its transpose and proceed with a principal components analysis as described above, we indentify important linear combinations of the observations, that, in turn, identify relationships among the observational
units. In the social sciences, a principal components analysis of variables is called an “R-Type” analysis and the analysis identifying relationships among the observational units is called “Q-Type”.

In the usual situation, as we have described, the number of observations, \( n \), is greater than the number of variables, \( m \). If \( X \) has rank \( m \), then the variance-covariance matrix and the correlation matrix are of full rank. In a reversal of the roles of observations and variables, the corresponding matrix would not be of full rank. Of course, the analysis could proceed mechanically as we have described, but the available information for identifying meaningful linear combinations of the observations would be rather limited. This problem could perhaps be remedied by collecting more data on each observational unit; that is by defining more variables.

**Principal Components Directly from the Data Matrix**

Formation of the \( S \) or \( R \) matrix emphasizes the role that the sample covariances or correlations play in principal component analysis. However, as we discussed in the context of overdetermined systems of equations (page 303 and following), there is no reason to form a matrix like \( (X - \bar{X})^T(X - \bar{X}) \), and indeed, we may introduce significant rounding errors by doing so.

The singular value decomposition (SVD) of the \( n \times m \) matrix \( X - \bar{X} \) yields the square roots of the eigenvalues of \( (X - \bar{X})^T(X - \bar{X}) \) and the same eigenvectors. (The eigenvalues of \( (X - \bar{X})^T(X - \bar{X}) \) are \( (n - 1) \) times the eigenvalues of \( S \).) We will assume there are more observations than variables; that is, that \( n > m \). In the SVD of the centered data matrix (see page 255), we write

\[
(X - \bar{X})^T(X - \bar{X}) = U A V^T,
\]

where \( U \) is an \( n \times m \) matrix with orthogonal columns, \( V \) is an \( m \times m \) orthogonal matrix, and \( A \) is an \( m \times m \) diagonal matrix with nonnegative entries, called the singular values of \( X - \bar{X} \).

The spectral decomposition in terms of the singular values and outer products of the columns of the factor matrices is

\[
X - \bar{X} = \sum_{i=1}^{m} \sigma_i u_i v_i^T.
\]

(See equation (5.43), page 255.) The vectors \( u_i \), called the “left eigenvectors” or “left singular vectors” of \( X - \bar{X} \), are the same as the eigenvectors of \( S \) in equation (13.16). The vectors \( v_i \), the “right eigenvectors”, are the eigenvectors that would be used in a Q-type principal components analysis. The reduced-rank matrix that approximates \( X - \bar{X} \) is

\[
\tilde{X}_p = \sum_{i=1}^{p} \sigma_i u_i v_i^T,
\]

for some \( p < \min(n, m) \).
Computational Issues

For the eigenalysis computations in PCA, if the sample variance-covariance matrix $S$ is available, it is probably best to proceed with the decomposition of it as in equations (13.16). Any of the methods for computing eigenvalues and eigenvectors discussed in Sections 5.3.1 through 5.3.3, beginning on page 311 may be appropriate. Because the interest is generally only in the largest eigenvalues, the power method may be the best method to use. If $S$ is not available, there is generally no reason to compute it just to perform PCA. The computations to form $S$ are $O(m^3)$. Not only do these computations add significantly to the overall computational burden, but, as we discussed in Section 5.2.7, page 303, $S$ is more poorly conditioned than $X$ (or $X - \bar{X}$). The SVD decomposition (13.18) is, therefore, the better procedure.

Artificial neural nets have been proposed as a method for computing the singular values in equation (13.18). A study by Nicole (2000), however, indicates that neural nets may not perform very well for identifying any but the first principal component.

PCA for Clustering

An objective of principal components analysis is to identify linear combinations of the original variables that are useful in accounting for the variation in those original variables. This is effectively a clustering of the variables. For many purposes, these derived features carry a large amount of the information that is available in the original larger set of variables. For other purposes, however, the principal component may completely lose the relevant information. For example, the information carried by the smaller set of features identified in PCA may be useless in clustering the observations. Consider the bivariate dataset in Figure 13.16. There are two clusters in this dataset, each of which appears to be a sample from an elliptical bivariate normal distribution with a small positive correlation. The two clusters are separated by a displacement of the mean of the second component. The two principal components are shown in Figure 13.16. As would be expected, the first principal component is in the direction of greater spread of the data, and the second principal component is orthogonal to the first.

The first principal component contains no information about the clusters in the data. Figure 13.17 shows histograms of the data projected onto the two principal components. The second principal component carries information about the two clusters, but the first principal component appears to be a single normal sample.

Principal components analysis emphasizes the direction of maximum variation. This is not always the most informative direction. Other techniques, such as projection pursuit (see Section 13.4.5), seek projections of the data in directions that exhibit other interesting structure, such as the bimodality in the direction of the second principal component in this example.
outlier

Figure 13.16. Principal Components of a Bivariate Dataset with Two Clusters

gmu640

Robustness of Principal Components

As we have mentioned above, outlying observations or (nearly) collinear variables can present problems in data analysis. Principal components is one way of dealing with collinear variables. These variables have large correlations among themselves. The dimension of the space of principal components will likely be reduced so that all variables that are collinear with each other are replaced by a single variable.

Outlying observations, however, may have a major affect on the principal components analysis. The first few principal components are very sensitive to these outlying observations. If the outliers were not present, or if they were perturbed slightly, a different set of the first few principal components would likely result.

There are generally two ways of dealing with outlying observations. One way is to identify the outliers and remove them temporarily. Another way is to use methods that are not much affected by outliers. Campbell (1980) suggests using the robust sample variance-covariance, $S_R$, in equation 13.15, page 575. The principal components resulting from $S_R$ are less affected by outliers than those resulting from the usual sample variance-covariance, $S$.

If outliers can be identified and removed temporarily, a standard analysis can be performed. This identification, removal, and analysis procedure can be applied in stages. The major problem, of course, is that as extreme observations are removed, the variability in the dataset is reduced, so other, valid
observations are more likely to appear as outliers. In general, a data analyst must assume that every observation carries useful information, and no observation must be discarded until its information is incorporated into the analysis. (See Barnett and Lewis, 1994, for methods of dealing with outliers.)

For purposes of PCA, outliers can be identified in a preliminary step using a clustering procedure or even by using Q-type principal components analysis. Caroni (2000) describes a way of using the robust PCA based on $S_R$ to identify outliers.

**Latent Semantic Indexing**

An interesting application of the methods of principal components, called *latent semantic indexing*, is used in matching keyword searches with documents. The method begins with the construction of a term-document matrix, $X$, whose rows correspond to keywords, whose columns correspond to documents (webpages, for example), and whose entries are the frequencies of occurrences of the keywords in the documents. A singular value decomposition is performed on $X$ (or on $X - \bar{X}$) as in equation (13.18), and then a reduced-rank matrix $\tilde{X}_p$ is defined, as in equation (13.19). A list of keywords is matched to documents by representing the keyword list as a vector, $q$, of 0’s and 1’s corresponding to the rows of $X$. The vector $\tilde{X}_p^T q$ is list of scores for the documents. Documents with larger scores are those deemed relevant for the search.
A semantic structure for the set of documents can also be identified by \( \tilde{X}_p \). Semantically nearby documents are mapped onto the same singular vectors. See Berry and Browne (1999) for further discussion of latent semantic indexing and its use in document retrieval.

A variation of latent semantic indexing is called probabilistic latent semantic indexing, or nonnegative-part factorization. This approach assumes a set of hidden variables, whose values in the matrix \( H \) correspond to the columns of \( X \) by a nonnegative matrix factorization,

\[
X = WH,
\]

where \( W \) is a matrix with nonnegative elements. See Lee and Seung (1999) for a description of this factorization and its use. Hofmann (1999) describes a similar approach from a different perspective.

### 13.4.4 Factor Analysis

Factor analysis is mechanically similar to principal components analysis. The main differences involve the probability model.

#### The Probability Model Underlying Factor Analysis

In factor analysis, we begin with a model that relates a centered \( m \)-vector random variable \( Y \) (observable) to an underlying, unobservable \( k \)-vector random variable, whose elements are called “factors”. The factors have a mean of 0. In this model, the observable vector \( Y \) consists of linear combinations of the factors plus an independent random vector of “unique errors”, which is modeled by a random variable with a mean of 0. The unique errors are independent of the factors. Now, letting \( F \) represent the vector of factors and \( E \) represent the errors, we have

\[
Y - \mu = \Gamma F + E,
\]

where \( \mu \) is the mean of \( Y \) and \( \Gamma \) is an \( m \times k \) fixed (but unknown) matrix, called the “factor loadings” matrix. Generally, the number of factors is less than the number of the observable variables. In some applications, in psychology, for example, the factors are related to some innate characteristics that are manifested in observable behavior.

We denote the variance-covariance matrix of \( Y \) by \( \Sigma \), that of \( F \) by \( \Sigma_F \), and that of \( E \) by \( \Psi \), which is diagonal by the assumptions in the model. We therefore have the relationship

\[
\Sigma = \Gamma \Sigma_F \Gamma^T + \Psi.
\]

Now, if we let \( \tilde{\Gamma} = \Gamma \Sigma_F^{-\frac{1}{2}} \) and \( \tilde{F} = (\Sigma_F^{-\frac{1}{2}})^{-1}F \), we have
\[ \Sigma = \Gamma \Gamma^T + \Psi. \]

Hence, an equivalent model to equation (13.20) is one in which we assume the underlying factors have the identity as their variance-covariance matrix, and so we have

\[ \Sigma = \Gamma \Gamma^T + \Psi. \tag{13.21} \]

The diagonal elements of \( \Psi \) are called the specific variances of the factors and the diagonal elements of \( \Gamma \Gamma^T \) are called the communalities of the factors.

The transformations above that indicate that \( \Gamma \Sigma \Gamma^T \) can be used instead of \( \Gamma \Sigma \Gamma^T \) raise the issue of more general transformations of the factors, leading to an indeterminacy in the analysis.

If we decompose \( \Sigma - \Psi \) as we did in PCA in equation (??), page ??, (with \( \Delta \) replacing \( \Lambda \)) we have

\[ \Sigma - \Psi = W^T \Lambda W. \tag{13.22} \]

The factor loading matrix therefore is

\[ \Gamma = W^T \Lambda^{1/2}. \tag{13.23} \]

**Factor Analysis of Data**

In practical applications of factor analysis, we must begin with a chosen value of \( k \), the number of factors. This is similar to choosing the number of principal components in PCA, and there are some ways of adaptively choosing \( k \), but the computational approaches we discuss below assume a fixed value for \( k \). As usual, we consider the rows of the data matrix \( X \) to be realizations of a multivariate random variable. In factor analysis, the random variable has the relationships to other random variables as \( Y \) above; hence, the observation \( x \) (a row of \( X \)) is related to the realization of two other random variables, \( f \) and \( e \), by

\[ x - \bar{x} = \Gamma f + e. \]

The objective in factor analysis is to estimate the parameters in the model (13.20), that is, the factor loadings, \( \Gamma \), and the variances, \( \Sigma \) and \( \Psi \), in equation (13.21). There are several methods for estimating these parameters. In one method the estimation criterion is least squares of the sum of the differences in the diagonal elements of \( \Sigma \) and \( S \), that is, minimize the function \( g \):

\[ g(\Gamma, \Psi) = \text{trace}((S - \Sigma)^2). \tag{13.24} \]

This criterion leads to the principal factors method. The minimization proceeds by first choosing a value \( \tilde{\Psi}^{(0)} \), and then performing a decomposition similar to that in principal components, except instead of decomposing the sample variance-covariance matrix \( S \), an eigenanalysis of \( S - \tilde{\Psi}^{(0)} \) as suggested by equation (13.22) is performed:
\[ S - \hat{\Psi}^{(0)} = \left( \hat{W}^{(0)} \right)^T \hat{\Lambda}^{(0)} \hat{W}^{(0)}. \] (13.25)

This yields the value for \( \Gamma \), analogous to equation (13.23):

\[ \hat{\Gamma}^{(0)} = \left( \hat{W}^{(0)} \right)^T \left( \hat{\Lambda}^{(0)} \right)^{\frac{1}{2}}. \]

Next the minimization problem (13.24) is solved for \( \Psi \) with the fixed value of \( \hat{\Gamma}^{(0)} \), that is,

\[ \min g(\hat{\Gamma}^{(0)}, \Psi). \] (13.26)

Newton’s method is usually used to solve this problem, leading to \( \hat{\Psi}^{(1)} \). The steps are then repeated; that is, \( S - \hat{\Psi}^{(1)} \) is decomposed, leading to

\[ \hat{\Gamma}^{(1)} = \left( \hat{W}^{(1)} \right)^T \left( \hat{\Lambda}^{(1)} \right)^{\frac{1}{2}}, \]

which is used in the next application of Newton’s method to solve \( \min g(\hat{\Gamma}^{(1)}, \Psi) \).

Convergence criteria are usually chosen based on norms of the change in the estimates from one iteration to the next.

A simple method that is often used to get started is to take \( \hat{\psi}_{jj}^{(0)} \) as

\[ \left( 1 - \frac{k}{2m} \right) (S^{-1})_{jj}^{-1}, \]

where \( (S^{-1})_{jj}^{-1} \) is the \( j^{th} \) diagonal element of \( S^{-1} \) if \( S \) is full rank; otherwise, take \( \hat{\psi}_{jj}^{(0)} = s_{jj}/2 \).

The factors derived using the principal factors method, that is, the linear combinations of the original variables, are the same as would be obtained in ordinary PCA if the variance of the noise (the unique errors) were removed from the variance-covariance of the observations prior to performing the PCA.

Another common method for estimating \( \Gamma, \Sigma \) and \( \Psi \) uses the likelihood criterion that results from the asymptotic distributions. Using the negative of the log of the likelihood, we have the minimization problem,

\[ \min l(\Gamma, \Psi) = \min \left( \log |\Sigma^{-1}S| - \text{trace}(\Sigma^{-1}S) \right). \] (13.27)

This criterion results in the method of maximum likelihood. In this method we require that \( S \) is positive definite.

Solution of the minimization problem (13.27) is also done in iterations over two stages, as we did in the least squares method above. First we choose a starting value \( \hat{\Psi}^{(0)} \). Its Cholesky factor, \( \left( \hat{\Psi}^{(0)} \right)^{\frac{1}{2}} \), is symmetric. We then decompose \( \left( \hat{\Psi}^{(0)} \right)^{\frac{1}{2}} S^{-1} \left( \hat{\Psi}^{(0)} \right)^{\frac{1}{2}} \) as in equation (13.25):

\[ \left( \hat{\Psi}^{(0)} \right)^{\frac{1}{2}} S^{-1} \left( \hat{\Psi}^{(0)} \right)^{\frac{1}{2}} = \left( \hat{W}^{(0)} \right)^T \hat{\Lambda}^{(0)} \hat{W}^{(0)}. \] (13.28)
Using the relationship (13.21) and equation (13.23) we get a value for $\hat{\Gamma}$:

$$\hat{\Gamma}^{(0)} = \left( \hat{\Psi}^{(0)} \hat{W}^{(0)} \right)^T \left( \hat{A}^{(0)} - I \right)^{\frac{1}{2}}.$$

Next the minimization problem (13.27) is solved for $\hat{\Psi}$ with the fixed value of $\hat{\Gamma}^{(0)}$. This problem may be rather ill-conditioned, and the convergence can be rather poor. The transformations

$$\theta_j = \psi_{jj}$$

can help. (The $\psi_{jj}$ are the only variable elements of the diagonal matrix $\Psi$.) Hence, the minimization problem is

$$\min l\left( \hat{\Gamma}^{(0)}, \theta \right). \quad (13.29)$$

An advantage of the maximum likelihood method is that it is independent of the scales of measurement. This results from the decomposition of $\hat{\Psi}^\frac{1}{2} S^{-1} \hat{\Psi}^\frac{1}{2}$ in equation (13.28). Suppose we make a scale transformation on the random variable, $Y$, in equation (13.20), that is, we form $T = YD$, where $D$ is a fixed diagonal matrix with positive entries. The resulting variance-covariance matrix for the unique errors, $\Psi_T$, is $D \hat{\Psi} D^T$. Likewise, the corresponding sample variance-covariance matrix, $S_T$, is $D S D^T$. The matrix to be decomposed as in equation (13.28) is

$$\left( \hat{\Psi}_T^{(0)} \right)^\frac{1}{2} S_T^{-1} \left( \hat{\Psi}_T^{(0)} \right)^\frac{1}{2} = \left( D \hat{\Psi}^{(0)} D^T \right)^\frac{1}{2} \left( D S D^T \right)^{-1} \left( D \hat{\Psi}^{(0)} D^T \right)^\frac{1}{2}$$

$$= \left( \hat{\Psi}^{(0)} \right)^\frac{1}{2} D^T D^{-1} S^{-1} D \left( \hat{\Psi}^{(0)} \right)^\frac{1}{2}$$

$$= \left( \hat{\Psi}^{(0)} \right)^\frac{1}{2} S^{-1} \left( \hat{\Psi}^{(0)} \right)^\frac{1}{2},$$

which is the same as the one for the untransformed data.

Other common methods for factor analysis include generalized least squares, image analysis (of two different types), and alpha factor analysis.

The methods for factor analysis begin with the computation of the sample variance-covariance matrix $S$ or the sample correlation matrix $R$. As we noted in the case of PCA, the results are different, just as the results are generally different following any transformation of the data.

Note that the model (13.20) does not define the factors uniquely; any rotation of the factors would yield the same model. In principal components analysis, a similar indeterminacy could also occur if we allow an arbitrary basis for the PCA subspace defined by the chosen $k$ principal components.

The factors are often rotated so as to get a basis with some interesting properties. A common criterion is parsimony of representation, which roughly means that the matrix has few significantly non-zero entries. This principle has
given rise to various rotations, such as the varimax, quartimax, and oblimin rotations.

Factor analysis is often applied to grouped data under a model with the same factors in each group, called a common factor model.

In general, because of the stronger model, factor analysis should be used with more caution than principal components analysis. For more discussion on factor analysis, the reader is referred to Reyment and Jöreskog (1996). An appendix in that book contains Matlab scripts to perform many of the computations for factor analysis.

13.4.5 Projection Pursuit

The objective in projection pursuit is to find “interesting” projections of multivariate data. Interesting structure in multivariate data may be identified by analyzing projections of the data onto lower-dimensional subspaces. The projections can be used for optimal visualization of the clustering structure of the data, or for density estimation or even regression analysis. The approach is related to the visual approach of the grand tour (page ??). Reduction of dimension is also an important objective, especially if the use of the projections is in visualization of the data. Projection pursuit requires a measure of the “interestingness” of a projection.

Diaconis and Freedman (1984) showed that a randomly selected low-dimensional projection of any high-dimensional data set will appear similar to a sample from a multivariate normal distribution. This result, which may be thought of as a central limit theorem for projections, implies that a multivariate normal dataset is the least “interesting”. In projection pursuit, therefore, the objective is to find departures from normality in linear projections of the data.

Departures from normality may include such things as skewness and “holes” in the data, or multimodality. The projection whose histogram is shown in right side of Figure 13.17 exhibits a departure from normality, while the histogram one the left appears to be of normal univariate data.

The Probability Model Underlying Projection Pursuit

Consider an $m$-vector random variable $Y$. In general, we are interested in a $k$-dimensional projection of $Y$, say $A^T Y$, such that the random variable $A^T Y$ is very different from a $k$-variate normal distribution.

Because all one-dimensional marginals of a multivariate normal are normal, and cross-products of normals are multivariate normal, we will concentrate on one-dimensional projections of $Z$. Our problem is to find $Z = a^T Y$, such that the random variable $Y$ is “most different” from a normal random variable. Two-dimensional projections are also of particular interest, especially in graphics. We will discuss just the one-dimensional projections, and refer the
interested reader to Cook, Buja, and Cabrera (1993) for more discussion of two-dimensional projections and their applications.

The structure of interest, that is, a departure from normality, can be considered separately from the location, variances, and covariances of the vector $Y$, therefore we will assume that $E(Y) = 0$ and $V(Y) = I$. Prior to applying projection pursuit to data, we center and sphere the data, so that the sample characteristics are consistent with these assumptions.

In order to quantify the objectives in projection pursuit, we need a measure, or index, of the departure from normality.

**Projection Indexes for the Probability Model**

One way to quantify departure from normality is to consider the probability density function of the projected variable and compare it to the probability density function $\phi$ of a standard normal random variable. If $p$ is the density of $Z$, we want it to be very different from $\phi$. This is an opposite problem from the function approximation problem we considered in Section 8.2, but the approaches are related. While in function approximation, the Chebyshev norm is generally of most interest, in seeking a function that is “different”, an $L_2$ norm,

$$ \int_{-\infty}^{\infty} (p(z) - \phi(z))^2 \, dz, $$

makes more sense as a measure of the difference.

The objective in projection pursuit is to find an $a$ that maximizes this norm.

It has become common practice in the literature on projection pursuit to name indexes of departure from normality by the type of orthogonal polynomials used in approximating the index. The index in (13.30) is called the **Hermite index** because Hermite polynomials are appropriate for its approximation (see Section 8.3.2, page 465). It is also called **Hall’s index** because it was studied by Hall (1989).

For a given $a$, Friedman (1987) proposed first mapping $Z = a^TY$ into $[-1, 1]$ by the transformation

$$ R = 2\Phi(Z) - 1, $$

where $\Phi$ is the CDF of a standard normal distribution. If $p_Z$ be the probability density of $Z$, then the probability density of $R$ is

$$ p_R(r) = \frac{1}{2} p_Z \left( \Phi^{-1} \left( \frac{r+1}{2} \right) \right). $$

If $Z$ has a normal distribution with a mean of 0 and variance of 1, $R$ has a uniform distribution over $(-1, 1)$, and so has a constant density of $\frac{1}{2}$. (This is the idea behind the inverse CDF method of random number generation.)
In Exercise 4.15, page 224, you were asked to prove a similar relationship.) Hence, the problem is to find $a$ such that the density, $p_R$, of $R$ is very different from $\frac{1}{2}$. The relevant $L_2$ norm is

$$L(a) = \int_{-1}^{1} \left( p_R(r) - \frac{1}{2} \right)^2 dr,$$

which simplifies to

$$L(a) = \int_{-1}^{1} p_R^2(r) dr - \frac{1}{2}. \quad (13.31)$$

This norm, which is a scalar function of $a$ and a functional of $p_R$, is sometimes called the Legendre index because Legendre polynomials are natural approximating series of orthogonal polynomials (see Table 8.1, on page 468). It is also called Friedman’s Legendre index.

Cook, Buja, and Cabrera (1993) suggested another index based on the $L_2$ norm in equation (13.30) being weighted with the normal density:

$$H(a) = \int_{-\infty}^{\infty} (p(z) - \phi(z))^2 \phi(z) dz. \quad (13.32)$$

Cook, Buja, and Cabrera call this the natural Hermite index. The index is evaluated by expanding both $p(z)$ and $\phi(z)$ in the Hermite polynomials that are orthogonal with respect to $e^{-x^2/2}$ over $(-\infty, \infty)$. The Hermite polynomials are the $H_k$ in equations (8.31), page 471. The index is called “natural” because the difference in $p$ and $\phi$ is weighted by the normal density. The natural Hermite index has some desirable invariance properties for two-dimensional projections. See their article for discussion of these properties.

Various other measures of departure from normality are possible; Jones (see Jones and Sibson, 1987) suggested an index based on ratios of moments of a standard normal distribution, and Huber (1985) suggested indexes based on entropy (called Shannon entropy or differential entropy):

$$-\int_{\mathbb{R}^m} p(z) \log p(z) dz.$$

The entropy is maximized among the class of all random variables when the density $p$ is the standard multivariate normal density (mean of zero and variance-covariance matrix equal to the identity). For any other distribution, the entropy is strictly smaller. The problem with most measures of departure from normality is the difficulty of estimation of the terms.

Morton (1992) suggested an “interpretability index” that gives preference to simple projections, that is, to linear combinations in which $a$ has more zeroes, and when comparing two combinations $a_1$ and $a_2$, the vectors are (nearly) orthogonal.

Sun (1992) reports comparisons of the use of Friedman’s Legendre index, $L(a)$, and Hall’s Hermite index (13.30). Cook, Buja, and Cabrera (1993)
give comparisons of these two indexes and the natural Hermite index, \( H(a) \). The results of these comparisons were inconclusive; which index is better in identifying departures from normality seems to depend on the nature of the nonnormality.

**Projection Pursuit in Data**

We now consider one-dimensional projection pursuit in a given set of data \( X \) (the familiar \( n \times m \) matrix in our data analysis paradigm). For each projection \( a \), we *estimate* the projection index associated with \( a \), under the assumption that the rows in \( X \) are independent realizations of a random variable \( Y \). The vector \( Xa \) contains independent realizations of the scalar random variable \( Z = a^T Y = Y^T a \).

In order to estimate the projection index, we must *approximate* an integral. As we suggested above, the indexes lend themselves to approximation by standard series of orthogonal polynomials.

For \( L(a) \), expanding one factor of \( p^2_R \) in equation (13.31) in Legendre polynomials and leaving the other unexpanded, we have

\[
L(a) = \int_{-1}^{1} \left( \sum_{k=0}^{\infty} c_k P_k(r) \right) p_R(r) \, dr - \frac{1}{2},
\]

where \( P_k \) is the \( k \)th Legendre polynomial (see page 468).

From equation (8.16), page 464, we have the Legendre coefficients for the expansion

\[
c_k = \frac{2k + 1}{2} \int_{-1}^{1} P_k(r) p_R(r) \, dr.
\]

Substituting this into the expression above, because of the orthogonality of the \( P_k \), we have

\[
L(a) = \sum_{k=0}^{\infty} \frac{2k + 1}{2} \left( \mathbb{E}(P_k(R)) \right)^2 - \frac{1}{2},
\]

where the expectation \( \mathbb{E} \) is taken with respect to the distribution of the random variable \( R \). Each term in equation (13.33) is an expectation, and therefore can be estimated easily from a random sample. The sample mean is generally a good estimate of an expectation; hence for the \( k \)th term, from the original observations \( x_i \), the projection \( a \), and the normal CDF transformation, we have

\[
\hat{\mathbb{E}} (P_k(R)) = \frac{1}{n} \sum_{i=1}^{n} P_k(r_i)
\]

\[
= \frac{1}{n} \sum_{i=1}^{n} P_k(2\Phi(a^T x_i) - 1).
\]
A simple estimate of the squared expectation is just the square of this quantity. Obviously, in practice, we must use a finite approximation to the infinite expansion of \( p_R \). After terminating the expansion at \( j \), we have the truncated Legendre projection index, \( L_j(a) \),

\[
L_j(a) = \sum_{k=0}^{j} \frac{2k+1}{2} (E(P_k(R)))^2 - \frac{1}{2}.
\] (13.34)

The approximation in equation (13.34) can be estimated easily from the sample:

\[
\hat{L}_j(a) = \sum_{k=0}^{j} (2k+1) \left( \frac{1}{n} \sum_{i=1}^{n} P_k(2\Phi(a^T x_i) - 1) \right)^2 - \frac{1}{2}.
\] (13.35)

This expression is easily evaluated. The first two Legendre polynomials are \( P_0(t) = 1 \), \( P_1(t) = t \). (The first six are shown in equation (8.26).) The recurrence formula (see equation (8.24), page 466) is

\[
P_k(t) = \frac{2k-1}{k} t P_{k-1}(t) - \frac{k-1}{k} P_{k-2}(t),
\]

and we use this relationship in computing the truncated Legendre index, especially if we are using more than 3 or 4 terms.

The problem now is to determine \( \max_a \hat{L}_j(a) \).

Scaling of \( a \) is not relevant, so we may restrict \( a \) so that the sum of its elements is some given value, 1, say. In general, this is not an easy optimization problem. There are local minima. Use of an optimization method such as Newton’s method may require multiple starting points. An optimization method such as simulated annealing may work better.

After both \( p(z) \) and \( \phi(z) \) are expanded in the Hermite polynomials, the natural Hermite index of Cook, Buja, and Cabrera, equation (13.32), reduces to

\[
\sum_{k=0}^{\infty} (d_k - b_k)^2,
\]

where the \( d_k \) are the coefficients of the expansion of \( p(z) \) and the \( b_k \) are the coefficients of the expansion of \( \phi(z) \). The \( b_k \) can be evaluated analytically. From Cook, Buja, and Cabrera, they are, for \( k = 0, 1, \ldots \)

\[
b_{2k} = \frac{(-1)^k((2k)!)^{1/2}}{2^{2k+1}k!(\sqrt{\pi})^{1/2}}
\]

\[
b_{2k+1} = 0.
\]
The $d_k$ can be represented as expected values, and are estimated from the data in a similar manner as done for the Legendre index above. The estimates are given by

$$\hat{d}_k = \sum_{i=1}^{n} H^e_k(x_i)\phi(x_i).$$  \hfill (13.36)

The index is the truncated series

$$\sum_{k=0}^{j} (\hat{d}_k - b_k)^2.$$  

The first two Hermite polynomials are $H^e_0(t) = 1$, $H^e_1(t) = t$. (The first six are shown in equation (8.31).) The recurrence formula (see equation (8.32), page 471) is

$$H^e_k(t) = tH^e_{k-1}(t) - (k - 1)H^e_{k-2}(t),$$

and we use this relationship in computing the truncated Hermite index, especially if we are using more than 3 or 4 terms.

While the more terms retained in the orthogonal series expansions, the better is the approximation, it is not necessarily the case that the better-approximated index is more useful. Friedman (1987), Hall (1989), Sun (1992), and Cook, Buja, and Cabrera (1993) have addressed this anomaly. Hall develops an asymptotic theory that suggests that optimal choice of $j$ depends on sample size and type of index. (He considers both the norm in (13.30) and in (13.31).) Sun suggests the choice of $j$ between 3 and 6 inclusive, with smaller values being chosen for smaller numbers of observations, and smaller values being chosen for larger values of the dimension $m$. Cook, Buja, and Cabrera (1993) found that the discrimination ability of the index for different values of $j$ depends on the nature of the nonnormality in the data.

Friedman (1987) addresses the statistical significance of $L(a)$; that is, the question of whether the projection using random data is significantly different from a projection of normal data. He gives a method for computing a p-value for the projection. Sun (1991) discusses the evaluation of p-values further.

### Exploratory Projection Pursuit

The most important use of projection pursuit is for initial exploratory analysis of multivariate datasets.

Cook, Buja, and Cabrera (1995) describe the use of projection pursuit in a grand tour, Cabrera and Cook (1992) discuss the relationship of projection pursuit to the fractal dimension of a dataset.

Different indexes may be useful in identifying different kinds of structure. The Legendre index is very sensitive to outliers in the data. If identification of the outliers is of specific interest, this may make the index useful. On the other hand, if the analysis should be robust to outliers, the Legendre index...
Laguerre-Fourier index, projection pursuit, sphered data

would not be a good one. The Laguerre-Fourier index, which is based on an expansion in Laguerre polynomials, is particularly useful in identifying clusters in the data.

**Example**

As an example of projection pursuit, consider the simple dataset shown in Figure 13.16. The obvious nonnormal structure in that dataset is the existence of two groups. Performing principal components analysis on the data resulted in a first principal component that completely obscured this structure (see Figure 13.17.) As it turned out, the second principal component identified the structure very well.

In projection pursuit the first step is generally to sphere the data. The result of the sphering is to make the variances in all directions almost equal. A “principal” component is no longer very relevant. As we have emphasized, sphering or any other transformation of the data is likely to have an effect on the results. Whether or not to sphere the data in projection pursuit is subject to some question; see the discussion following Jones and Sibson (1987).

The sphered data corresponding to those in Figure 13.16 are shown in Figure 13.18. (See Exercise 13.9 for the data.)

![Figure 13.18. The Sphered Bivariate Dataset with Two Clusters grmu660](image)

After sphering the data, the plot does not show the correlations nor the elliptical scatter of Figure 13.16. The principal components would be along

---

*Figure 13.18. The Sphered Bivariate Dataset with Two Clusters grmu660*
the axes. The sphered data do, however, continue to exhibit a bimodality, which is a departure from normality.

Now, consider the Legendre index for the principal components, that is, for the projections $a_1 = (1, 0)$ and $a_2 = (0, 1)$. Using equation (13.35) with $j = 5$, we obtain $\hat{L}_5(a_1) = 1.03$ and $\hat{L}_5(a_2) = 1.66$. Clearly the projection onto $z_2$ exhibits the greater nonnormality.

In most applications of projection pursuit, of course, we have datasets of higher dimension.

**Computational Issues**

Posse (1990) and Sun (1992) discuss some of the computational issues in evaluating the indexes, especially Friedman’s Legendre index. Projection pursuit involves not only the computation of an index, but the optimization of the index as a function of the linear combination vector. The optimization problem is characterized by many local maxima.

Rather than being interested in a global maximum, in data analysis with projection pursuit, we are generally interested in inspecting several projections, each of which exhibits an interesting structure, that is, some locally maximal departure from normality as measured by a projection index.

**13.4.6 Linear Discriminant Analysis**

**13.5 More General Structure**

Structure in data is defined in terms of transformations of the data. In PCA, for example, the linear structure that is identified consists of a set of one-dimensional linear projections, that is ordered by the norm of the projection of the centered dataset. In projection pursuit, the linear structure is also a set of projections, but they are order by their deviation from normality.

Nonlinear structure is generally much more difficult to detect. One approach is to generalize the methods of PCA. Identification of nonlinear structure in data is often preceded by a nonlinear mapping from the $m$-dimensional data space to a $p$-dimensional “feature space”. This mapping carries the basic data $m$-vector $x$ to a $p$-vector $f(x)$. The feature space is a subspace of $\mathbb{R}^p$, and thus is an inner product space. (The original data space may be more general.) This is essentially the approach of support vector machines. (See Cristianini and Shawe-Taylor, 2000, and Vapnik, 1999b.)

A simple type of mapping is one defined by an inner product, called a kernel function. A *kernel function* is an inner product of the feature mappings of two data vectors; that is for data vectors $x$ and $y$ and a mapping to the feature space $f$, a kernel $K$ is the function

$$K(x, y) = \langle f(x), f(y) \rangle.$$
See Cristianini and Shawe-Taylor (2000), and Vapnik (1999a) for more discussion of kernel methods in this context. Schölkopf, Smola, and Müller (1999) describe a nonlinear PCA based on kernel methods.

**Exercises**

13.1. Show that the Mahalanobis distance (13.14), page 572, between any two observations is nonnegative.

13.2. Show that the norm of the difference between two vectors is a metric, that is, if

\[ \Delta(x_1, x_2) = \|x_1 - x_2\|, \]

\( \Delta(x_1, x_2) \) is a metric.

13.3. Consider a two-dimensional surface with an orthogonal coordinate system over which there is a fluid flow with constant velocity \( f = (f_1, f_2) \). Suppose an object can move through the fluid with constant velocity \( v \), with respect to the fluid, and measured in the same units as \( f \). (The magnitude of the velocity is \( \|v\| = \sqrt{v_1^2 + v_2^2} \).) Assume \( \|v\| > \|f\| \).

a) Define a distance measure, \( d \), over the surface such that for two points \( x_i \) and \( x_j \), the distance from \( x_i \) to \( x_j \) is proportional to the time required for the object to move from \( x_i \) to \( x_j \).

b) Compare your distance measure with those listed on page 570.

c) What properties of a norm (page 257) does your distance measure possess?

13.4. Consider the clusterings of the toy dataset depicted in Figure 14.3, page 621.

a) How many clusters seem to be suggested by each?

b) Compute Rand’s statistic to compare the clustering on the left with that in the middle. Assume 4 clusters in each.

c) Compute Rand’s statistic to compare the clustering on the middle with that on the right. Assume 4 clusters in the middle, and 3 clusters on the right.

d) Compute Rand’s statistic to compare the clustering on the middle with that on the right. Assume 2 clusters in the middle, and 3 clusters on the right.

13.5. Develop a combinatorial optimization algorithm, perhaps using simulated annealing or a genetic algorithm, to perform K-means clustering in such a way that less than \( k \) groups may be formed. The objective function would need to be penalized for the number of groups. Try a modification of expression (14.2):

\[ \sum_{g=1}^{k} \sum_{j=1}^{m} \sum_{i=1}^{n_g} \left( x_{ij}(g) - \bar{x}_j(g) \right)^2 + \alpha k, \]
where \( \alpha \) is a tuning parameter. Its magnitude depends on the sizes of the sums of squares, which of course is unknown a priori. Write a program to implement your algorithm. In the program, \( \alpha \) is an input parameter. Use your program to form five or fewer clusters of the data from Exercise 7.9, page 443:

\[
\begin{bmatrix}
  x_1 \\
  x_2 \\
  x_3 \\
  x_4 \\
  x_5 \\
  x_6 \\
  x_7 \\
  x_8 \\
  x_9 \\
  x_{10} \\
  x_{11} \\
  x_{12} \\
  x_{13} \\
  x_{14} \\
  x_{15} \\
  x_{16}
\end{bmatrix} =
\begin{bmatrix}
  1 & 1 \\
  1 & 2 \\
  1 & 3 \\
  1 & 4 \\
  2 & 1 \\
  2 & 2 \\
  2 & 3 \\
  2 & 4 \\
  3 & 1 \\
  3 & 2 \\
  3 & 3 \\
  3 & 4 \\
  4 & 1 \\
  4 & 2 \\
  4 & 3 \\
  4 & 4
\end{bmatrix}.
\]

How many clusters do you get?

13.6. What happens in cluster analysis if the data are sphered prior to the analysis? Sphere the data used in Figure 14.4 (page 625), and then do the hierarchical clustering. To do this, replace the statement

\[
y \leftarrow \text{apply}(x, 2, \text{standard})
\]

with

\[
y \leftarrow x \times \text{solve} (\text{chol(var(x)))}
\]

The strange result in this case of clustering sphered data does not always occur; but the point is, sphering can have unexpected effects.

13.7. Given two \( n \)-vectors, \( x_1 \) and \( x_2 \), form a third vector \( x_3 \) as \( x_3 = a_1 x_1 + a_2 x_2 + \epsilon \), where \( \epsilon \) is a vector of independent \( \text{N}(0, 1) \) realizations. Although the matrix \( X = [x_1 \ x_2 \ x_3] \) is in \( \mathbb{R}^{n \times 3} \), the linear structure, even obscured by the noise, implies a two-dimensional space for the data matrix; that is, the space \( \mathbb{R}^{n \times 2} \).

a) Determine a rotation matrix that reveals the linear structure. That is, determine matrices \( Q \) and \( P \) such that the rotation \( XQ \) followed by the projection \( (XQ)P \) is a noisy line in two dimensions.

b) Generate \( x_1 \) and \( x_2 \) as realizations of a \( \text{U}(0, 1) \) process, and \( x_3 \) as \( 5x_1 + x_2 + \epsilon \), where \( \epsilon \) is a realization of a \( \text{N}(0, 1) \) process. What are \( Q \) and \( P \) from the previous question?
c) Do a principal components analysis (perhaps using `prcomp` or `princomp` in R) of the data in the previous part. Make a scree plot of the eigenvalues (perhaps using `screeplot` in R, which produces a barplot, rather than a line plot as shown in Figure 13.15). How many principal components would you choose?

13.8. a) Write out the gradient and the Hessian for the minimization problem (13.26) on page 602. Remember $\Psi$ is a diagonal matrix.

b) Write out the gradient and the Hessian for the minimization problem (13.29) on page 603.

13.9. The data shown in Figure 13.16 and used in the PCA and the projection pursuit examples was generated by the R commands:

```r
n <- 200
x <- rnorm(n)
y <- rnorm(n)
xx <- 10*x + y
yy <- 2*y + x
n2 <- n/2
```

a) Sphere this dataset and plot it. Your plot should look like that in Figure 13.18.

b) Determine the optimal projection $a$ using the estimated truncated Legendre index with $j = 4$.

13.10. Other indexes.

a) Derive equation (13.36), page 609, for use in the natural Hermite index, equation (13.32). (Compare equation (13.35).) See page 471 for the Hermite polynomials.

b) Determine the optimal projection of the data in Exercise 13.9 using the estimated truncated natural Hermite index with $j = 4$. 
An objective in this chapter is the identification of interesting structure in the data. Interesting structure may involve clusters of data or it may be the result of the data lying on or near a space of reduced dimension. Interesting structure may also be defined generically as properties of the data that different from expected properties if the data were a random sample from a multivariate normal distribution, or from some other standard distribution. The normal (or Gaussian) distribution lies at the heart of many methods of data analysis. The heuristic definition of structure as departure from normality can be motivated by the fact that a randomly selected low-dimensional projection of any high-dimensional dataset will appear similar to a random sample from a multivariate normal distribution (see Diaconis and Freedman, 1984).

14.1 Clustering and Classification

The objective in cluster analysis is to divide the observations into groups that are close to each other or are more homogeneous than the full set of observations. We will discuss briefly some methods for forming groups in data.

Identifying groups of similar observations in a dataset is an important step in making sense of the data and in understanding the phenomena represented by the data. Clustering, classification, and discrimination are terms that describe this activity. A good general reference on clustering and classification is Gordon (1999).

The first step in forming groups is to develop a definition of the groups. This may be based on similarities of the observations or on closeness of the observations to one another. Whenever the data are well-understood, it is possible to define the groups in a manner that is appropriate for the phenomena from which the data are taken. This is “supervised” classification.

Absent an understanding of the characteristics that define groups, we may define groups using properties of the data themselves. As more data are collected, the properties of the groups may become known from the past “train-
Mahalanobis distance is widely used in clustering and classification. It is a measure of the distance of a point from a mean, adjusted for the covariance structure of the data.

Formally, the Mahalanobis distance of a point \( x \) from the mean \( \bar{x}_i \) of the \( i \)th group is given by:

\[
(x - \bar{x}_i)^T S^{-1} (x - \bar{x}_i),
\]

where \( S \) is the sample variance-covariance matrix.

For classification, an observation can be classified by computing its Mahalanobis distance from the group means and assigning it to the closest group. This is the basis for a linear discriminant function for classification.

In clustering, the process is usually exploratory. It seeks to determine what groups are present in the data. If the groups are known from some training set, "discriminant analysis" seeks to understand what makes the groups different, and then to provide a method of classifying observations into the appropriate groups. When discriminant analysis is used to "train" a clustering method, we refer to the procedure as "supervised" classification. Discriminant analysis is mechanically simpler than cluster analysis. Clustering is "unsupervised" classification.

Because of the large number of possibilities for grouping a set of data into clusters, we generally must make some decisions to simplify the problem. One way is to decide a priori on the number of clusters; this is done in K-means clustering discussed below. Another way is to do recursive clustering; that is, once trial clusters are formed, observations are not exchanged from one cluster to another. Two pairs of observations that are in different clusters at one stage of the clustering process would never be split so that at a later stage one member of each pair is in one cluster and the other member of each pair is in a different cluster.

There are two fundamentally different approaches to recursive clustering. One way is to start with the full dataset as a single group, and based on some reasonable criterion, partition the dataset into two groups. This is called divisive clustering. The criterion may be the value of some single variable; for example, any observation with a value of the third variable larger than 5
may be placed into one group and the other observations placed in the other group. Each group is then partitioned based on some other criterion, and the partitioning is continued recursively. This type of divisive clustering or partitioning results in a classification tree, which is a decision tree each node of which represents a partition of the dataset.

Another way of doing recursive clustering is to begin with a complete clustering of the observations into singletons. Initially each cluster is a single observation, and the first multiple-unit cluster is formed from the two closest observations. This agglomerative, bottom-up approach is continued so that at each stage the two nearest clusters are combined to form one bigger cluster.

### 14.2.1 K-Means Clustering

The objective in K-means clustering is to find a partition of the observations into a preset number of groups $k$ that minimizes the variation within each group. Each variable may have a different variation, of course. The variation of the $j^{th}$ variable in the $g^{th}$ group is measured by the within sum-of-squares:

$$s_{j(g)}^2 = \frac{\sum_{i=1}^{n_g} (x_{ij(g)} - \bar{x}_{j(g)})^2}{n_g - 1},$$

(14.1)

where $n_g$ is the number of observations in the $g^{th}$ group, and $\bar{x}_{j(g)}$ is the mean of the $j^{th}$ variable in the $g^{th}$ group. There are $m$ such quantities. The variation of the observations within the $g^{th}$ group is chosen as a linear combination of the sums-of-squares for all the $m$ variables. The coefficients in the linear combination determine the relative effects the individual variables have on the clustering. The coefficients are usually chosen to be equal. The relative effects the individual variables have on the clustering also depend on their scales.

Now, to state more precisely the objective in K-means clustering, it is to find a partition of the observations into a preset number of groups $k$ that minimizes, over all groups, the total of the linear combinations of the within sum-of-squares for all variables. For linear combinations with unit coefficients, this quantity is

$$\sum_{g=1}^{k} \sum_{j=1}^{m} \sum_{i=1}^{n_g} (x_{ij(g)} - \bar{x}_{j(g)})^2.$$

(14.2)

Determining the partitioning to minimize this quantity is a computationally intensive task.

In practice, we seek a local minimum, that is, a solution such that there is no single switch of an observation from one group to another group that will decrease the objective. Even the procedure used to achieve the local minimum is rather complicated. Hartigan and Wong (1979) give an algorithm (and Fortran code) for performing the clustering. Their algorithm forms a set of initial trial clusters and then transfers observations from one cluster to another while seeking to decrease the quantity in (14.2). Simulated annealing can also
be used to do K-means clustering. (See Zeger, Vaisey, and Gersho, 1992, and see Exercise 7.9, page 443.)

The clustering depends on the variability of the variables. It may be necessary to scale the variables in order for the clustering to be sensible, because the larger a variable’s variance, the more impact it will have on the clustering. See Section 14.2.8, page 624, for further discussion of the issue of scaling variables prior to clustering.

A major issue is how many clusters should we form. Hartigan (1975) discusses some ways of deciding this and some heuristic underlying theory and some supporting empirical evidence. The question of the number of groups must generally be addressed in an ad hoc manner, however. The algorithms in current use form nonempty clusters. The objective function leads to exactly \( k \) clusters except in an extreme case of multiple observations with the same values which yields multiple solutions. A modification to the objective function that penalizes the objective function for the number of groups, and a modification to the algorithms that allows empty clusters would put less importance on the specification of \( k \).

### 14.2.2 Hierarchical Clustering

It is useful to consider a hierarchy of clusterings, form a single large cluster to a large number of very small clusters. Hierarchical clustering yields these alternative clusterings.

The results of a hierarchical clustering can be depicted as a tree, as shown in Figure 14.1. Each point along the bottom of the tree may correspond to a single observation. Nodes higher up in the diagram represent successively larger groups. The number of clusters depends on the level in the tree, as indicated in the plot.

### 14.2.3 Agglomerative Hierarchical Clustering

In agglomerative hierarchical clustering we first begin with a large number of clusters, generally as many as the number of observations so that each cluster consists of a single observation, and then we combine clusters that are nearest to each other.

In order to define distances between groups we must first consider from what points within the groups to measure distance. As we mentioned on page 573, there are several ways of doing this. One way is to measure the distance between a central point in one group, such as the mean or median of the group, and the corresponding central point in the other group. These methods often do not work very well in hierarchical clustering. In agglomerative hierarchical clustering, the distance between two clusters is usually chosen in one of three ways.
Figure 14.1. A Cluster Tree. Each Leaf Represents an Observation or a Group of Observations

- The minimum distance between a point in the first cluster and a point in the second cluster. Using this criterion results in what is sometimes called “single linkage” clustering.
- The distance between clusters is the average of the distances between the points in one cluster and the points in the other cluster.
- The largest distance between a point in one cluster and a point in the other cluster. Using this criterion results in what is sometimes called “complete linkage” clustering.

In addition to the choice of the two points to define the distance, different distance metrics can be chosen. Most of the clustering methods use an L_2 metric. Coleman et al. (1999) considered an exploratory clustering problem in which a metric, such as one based on the linear discriminant, is not chosen in advance.

By changing the distance metric and the clustering method, several different cluster trees can be created from a single dataset. No one method seems to be useful in all situations.

We can see the differences in hierarchical clustering with different distance measures between clusters using a simple example that consists of 5 observations with the distance matrix,
\[
D = \begin{bmatrix}
2 & 3 & 4 & 5 \\
1 & 4.34 & 4.58 & 7.68 & 4.47 \\
2 & 1.41 & 4.00 & 4.36 \\
3 & 5.10 & 5.00 \\
4 & 6.56 \\
\end{bmatrix}
\]

Using either type of distance measure, the first cluster is formed from observations 2 and 3, because 1.41 is the minimum in any case. The subsequent clusters are different in the three methods, as shown in Figure 14.2 by the matrices that contain distances between clusters.

Figure 14.2. Hierarchical Clustering Using Three Different Methods

In this example, we have carried the clustering to a single final cluster. The clusters at any intermediate stage except the first are different. Thus, in complete linkage, for example, after the cluster with observations 2 and 3 is formed, a separate cluster with observations 1 and 5 is formed; then these two clusters are grouped into a cluster with four observations, and finally observation 4 is added to form a single cluster.

Figure 14.3 shows the cluster trees that result from each method of clustering. The lengths of the vertical lines indicate the closeness of the clusters that are combined. In each tree, for example, the first level of combination (between observations 2 and 3) occurred at a measure of 1.41, as shown on the vertical scale. In the connected linkage, as shown in the tree on the left, the
second step was to add observation 4 to the cluster containing observations 2 and 3. This combining occurred at a measure of 4.00. On the other hand, in the compact linkage, as shown in the tree on the right, the cluster containing observations 2 and 3 was unchanged and a second cluster was formed between observations 1 and 5 at a measure of 4.47.

Figure 14.3. Cluster Trees Corresponding to the Methods in Figure 14.2

The cluster trees in Figure 14.3 differ in appearance from the one shown in Figure 14.1, in which the terminal nodes or leaves are all shown at the same level. This appearance is controlled by the `hang` keyword in the R plotting function `plclust`. The cluster trees in Figure 14.3 were produced with the following R commands:

```r
plclust(hclust(D,method="connected"))
plclust(hclust(D,method="average"))
plclust(hclust(D,method="compact"))
```

The height $h_{ik}$ at which the two observations $i$ and $k$ enter the same cluster is a measure of the closeness of the two observations. For any reasonable dissimilarity matrix and any reasonable method of linkage, the heights will satisfy the “ultrametric” inequality:

$$h_{ik} \leq \max_j (h_{ij}, h_{kj}).$$

This property is trivially satisfied by the linkages illustrated in Figures 14.2 and 14.3, as may be easily checked.
Consideration of the heights in a hierarchical clustering may suggest the appropriate number of clusters. For example, the relative heights separating clusters in the first tree (on the left) in Figure 14.3 indicates that there may be four clusters: (5), (1), (4), and (2,3), three of which are singletons. (Remember this is a toy dataset!) The tree on the right indicates that there may be three clusters: (4), (2,3), and (1,5).

The R function 
\texttt{agnes} also does agglomerative hierarchical clustering, and provides more information about the levels at which clusters are combined.

\section*{14.2.4 Divisive Hierarchical Clustering}

Most hierarchical clustering schemes are agglomerative, that is, they begin with no clusters and proceed by forming ever larger clusters. In divisive hierarchical clustering we begin with a single large cluster, and successively divide the clusters into smaller ones. Kaufman and Rousseeuw (1990) have described a divisive hierarchy in which clusters are divided until each cluster contains only a single observation. At each stage, the cluster with the largest dissimilarity between any two of its observations is selected to be divided. To divide the selected cluster, the observation with the largest average dissimilarity to the other observations of the selected cluster is used to define a “splinter group”. Next, observations that are closer to the splinter group than to their previous groups are assigned to the splinter group. This is continued until all observations have been assigned to a single cluster. The result is a hierarchical clustering. The R function \texttt{diana} determines clusters by this method.

\section*{14.2.5 Self-Organizing Maps+}

Kohonen real-time algorithm Kohonen batch algorithm

\section*{Classification Trees}

A rooted tree that is used to define a process based on a sequence of choices is called a \textit{decision tree}. This kind of tree is probably the best known type. Decision trees that are used for classifying observations are called \textit{classification trees}.

A procedure for building classification trees begins with a \textit{training set}, that is, a set of observations that have been grouped into previously identified classes. The objective is to determine a sequence of simple decisions that would ultimately divide the set of observations into the given groups. Each decision is generally in the form of a test based on the values of the variables that constitute an observation. Often the test involves only a single variable. If the variable takes on only a countable number of values, the test based on the variable may be chosen to have as many possible outcomes as the values
associated with the variable. If the variable has a continuous range of values, the test is generally chosen to have a binary outcome, corresponding to observations above or below some cutpoint. The terminal nodes of a classification tree represent groups or classes that are not to be subdivided further.

How well a test divides the observations is determined by the “impurity” of the resulting groups. There are several ways of measuring the impurity or how well a test divides the observations. Breiman et al. (1984) describe some methods, including a “twoing rule” for a binary test. For classifying the observations into \( k \) groups, at a node with \( n \) observations to classify, this rule assigns a value to a test based on the formula

\[
 t_L t_R \left( \sum_{i=1}^{k} |L_i t_L - R_i t_R| / n \right)^2,
\]

where \( t_L \) and \( t_R \) are the total numbers of observations the test assigns to the left and right child nodes respectively, and \( L_i \) and \( R_i \) are the numbers of group \( i \) assigned to the left and right respectively. Murthy, Kasif, and Salzberg (1994) discuss this and several other measures.

The classification tree can be built by a greedy divide-and-conquer recursive partitioning scheme as shown in Algorithm 14.1. See Quinlan (1986) for further discussion of such schemes, and their implementation on parallel processors.

**Algorithm 14.1 Recursive Partitioning for Classification Using a Training Set**

1. Evaluate all tests that divide the given set into mutually exclusive sets.
2. Choose the test that scores highest, and divide the set based on this test.
3. For any subset that contains observations from more than one group, repeat beginning at step 1.

The R function `tree` uses binary recursive partitioning to build a classification tree.

A similar method can also be used to build a regression tree, in which each separate group of observations is assumed to have a common mean value for some additional variable related to the ones that define the groups.

### 14.2.6 Other Divisive Clustering Schemes

Because of the computational time required in agglomerative clustering or global partitioning such as by K-means, for large datasets simpler methods are sometimes more useful. A recursive partitioning scheme can be efficient. One simple recursive method groups the observations into hyperrectangular regions based on the medians of the individual variables. In the first step of the median-split divisive scheme, the \( n \) observations are divided into two sets of \( n/2 \) based on the median of the variable with the largest range. The subsequent
steps iterate that procedure. At any stage, the numbers of observations in all clusters are nearly equal. This procedure and the motivation for it are closely related to the \( k-d \)-tree (see below). A related scheme uses the mean, rather than the median. This scheme is less intensive computationally. It does not have the property of almost equal-size clusters, however.

Wan, Wong, and Prusinkiewicz (1988) suggest an alternative divisive scheme that attempts to minimize the sum-of-squares, similar to K-means clustering. They provide empirical evidence that their method, while much faster than the K-means procedure, yields sums-of-squares very comparable to the local minima obtained by K-means. We refer the reader to the paper for details of their method.

### 14.2.7 Fuzzy Clustering

Fuzzy set theory has been applied to clustering, as it has to most problems that have a component of uncertainty. Instead of observations being grouped into definite or “crisp” clusters, they are given membership probabilities. The membership of the \( i \)th observation in the \( g \)th group is \( u_{ig} \). The memberships satisfy

\[
0 \leq u_{ig} \leq 1
\]

and

\[
\sum_{g=1}^{k} u_{ig} = 1 \quad \text{for all } i = 1, \ldots, n.
\]

The quantity analogous to (14.2) that is to be minimized is

\[
\sum_{g=1}^{k} \sum_{j=1}^{m} \sum_{i=1}^{n} u_{ig}^2 \left( x_{ij} - \bar{x}_{j(g)} \right)^2,
\]

where, as before, \( \bar{x}_{j(g)} \) is the mean of the \( j \)th element of the vectors \( x_i \) that are in the \( g \)th group. Since group membership is a weight, however,

\[
\bar{x}_{j(g)} = \frac{\sum_{i=1}^{n} u_{ig}^2 x_{ij}}{\sum_{i=1}^{n} u_{ig}^2}.
\]


### 14.2.8 Clustering and Transformations of the Data

As we discuss in Section 13.2.3, page 574, transformations on the data may change the relative values of measures of similarity. This, of course, affects any
method of analysis that depends on measures of similarity. A severe limitation of clustering results from the dependence of the clusters on the scaling of the data. In many data analytic procedures we perform various linear transformations on the data with predictable results on the analysis. For example we often perform a simple univariate standardization of the data by subtracting the sample mean and dividing by the sample standard deviation. For the typical data matrix $X$ whose columns represent variables and whose rows represent multivariate observations, we may standardize each variable by subtracting the column mean from each value in the column and dividing by the standardization of the column. Doing this, however, affects the clustering, as seen in Figure 14.4.

![Cluster Trees; Raw Data and Standardized Data](grmu427)

The cluster trees in Figure 14.4 were produced with the following R commands that first create a matrix of 7 observations with 5 variables and perform hierarchical clustering (using largest distances), and then standardize the data univariately and perform the same hierarchical clustering.

```r
set.seed(2)
x <- matrix(rnorm(35), ncol=5)
plclust(hclust(dist(x)))
standard <- function(vec) (vec-mean(vec))/sqrt(var(vec))
y <- apply(x, 2, standard)
plclust(hclust(dist(y)))
```
The dependence of the clustering on transformations of the data results from the effect on the distance measures discussed on page 567 and following. Whether one variable is measured in grams or kilograms affects the relative distances of any one observation to the other observations. If all variables in the dataset are of the same type, mass, say, it is easy to measure them all in the same units; if some are of one type and some are of another type, decisions on units are not so easy. These decisions, however, affect the results of clustering. We also observe effects of transformations of the data on other structures in the data, such as those we discuss in later sections.

### 14.2.9 Comparing Clusterings

As we have seen, various methods of clustering yield different results, and, furthermore, the same method yields different results if the data have been transformed. Which clustering is best cannot in general be determined by analysis of data with no context. The purpose of the clustering, after all, is to develop a better understanding of a phenomenon of which the data measure various aspects. Nevertheless, it is instructive to develop numerical measures of the agreement (or, equivalently, disagreement) of different clusterings of the same dataset.

A two-way contingency table can be used to represent agreement of clusterings. If the classes of one clustering are denoted as $C_{11}, C_{12}, \ldots, C_{1k_1}$ and those of a second clustering, as $C_{21}, C_{22}, \ldots, C_{2k_2}$, a two-way table of the numbers of units falling in the cells is constructed, as shown.

$$
\begin{array}{cccc}
C_{11} & C_{12} & \cdots & C_{1k_1} \\
C_{21} & n_{11} & n_{12} & \cdots & n_{1k_1} & n_{1*} \\
C_{22} & n_{21} & n_{22} & \cdots & n_{2k_1} & n_{2*} \\
\vdots & \vdots & \ddots & \ddots & \vdots \\
C_{2k_2} & n_{k_21} & n_{k_22} & \cdots & n_{k_2k_1} & n_{k_2*} \\
\hline
n_{*1} & n_{*2} & \cdots & n_{*k_2} & n
\end{array}
$$

From the cluster trees shown in Figure 14.4, there appear to be two obvious clusters in the first clustering, and three clusters in the second clustering. If we identify the clusters from left to right in each tree (so that, for example the first cluster in the first tree contains the points 2, 5, and 7, and the first cluster in the second tree contains the single point 3), we would have the table below.

$$
\begin{array}{ccc}
C_{11} & C_{12} \\
\hline
C_{21} & 0 & 1 \\
C_{22} & 3 & 0 & 3 \\
C_{23} & 0 & 3 & 3 \\
\hline
n_{*1} & n_{*2} & n
\end{array}
$$
The marginal totals are the counts for the corresponding clusters. The numbers in the cells indicate the extent of agreement of the two clusterings. Perfect agreement would yield, first of all, \( k_1 = k_2 \), and, secondly, a table in which each column and each row contains only one nonzero value.

Rand (1971) suggested a measure of the agreement of two clusterings by considering the numbers of pairs of points that are in common clusters. Of the total of \( \binom{n}{2} \) pairs of points, each pair may be

1. in the same cluster in both clusterings
2. in different clusters in both clusterings.
3. in the same cluster in one clustering but in different clusters in the other clustering.

Both the first and second events indicate agreement of the clusterings, and the third indicates disagreement. Rand’s statistic is a count of the numbers of pairs of the first and second types, divided by the total number of pairs. This statistic is obviously in the interval \([0, 1]\), and a value of 0 indicates total disagreement, and a value of 1 indicates complete agreement. Rand gave a method of computing the total numbers of pairs of the first and second types by subtracting the count of the number of the third type from the total \( \binom{n}{2} - \frac{1}{2} \left( \sum_i n_i^2 - 2 \sum_i \sum_j n_{ij}^2 + \sum_j n_j^2 \right) \).

This can be seen by expanding \( \left( \sum_i \sum_j x_{ij} \right)^2 \). For the two clusterings shown in Figure 14.4, with two clusters in the first clustering, and three clusters in the second, we see that this value is 18; hence, Rand’s statistic is 6/7.

Statistical significance may be determined in terms of the distribution of such a statistic given random clusterings. Hubert and Arabie (1985) studied and modified Rand’s measure to account for the expected values of random clusterings. Their statistic is

\[
\frac{\sum_i \sum_j \binom{n_{ij}}{2} - \sum_i \binom{n_i^*}{2} \sum_j \binom{n_j^*}{2}}{\left( \sum_i \binom{n_i^*}{2} + \sum_j \binom{n_j^*}{2} \right) / 2 - \sum_i \binom{n_i^*}{2} \sum_j \binom{n_j^*}{2}}.
\]

where the sums over \( i \) go to \( k_1 \) and the sums over \( j \) go to \( k_2 \). In practice, the statistical significance of such a comparison of two observed clusterings would be approximated by use of a few randomly-formed clusterings.

### 14.2.10 Computational Complexity of Clustering

The task of identifying an unknown number of clusters that are distinguished by unknown features is an exceedingly complex problem. In practical clustering methods there are generally tradeoffs between how clusters are defined.
and the algorithm used to find the clusters. In the hierarchical clustering algorithms we considered in Section 14.2.2, the algorithm dominates the approach to the problem. In those hierarchical clustering methods, the definition of clusters at any level is merely what results from a specified algorithm. After the $O(mn^2)$ computations to compute the distance matrix, the algorithm requires only $O(n)$ computations. Even with an algorithm-based definition, the method is computationally intensive.

K-means clustering (Section 14.2.1, page 617) begins with a reasonable definition of clusters, assuming a known number of clusters. Even with the simplifying assumption that the number of clusters is known, the definition of clusters requires a very computationally intensive algorithm. Even to compute the objective function (14.2) on page 617 for a given trial clustering requires $kmn$ computations. A trial clustering is defined by a permutation of the $n$ data elements together with a choice of $k$ nonnegative integers $n_g$, such that $\sum n_g = n$. Clearly the number of computations required to satisfy the definition of clusters, even under the assumption of a known number of clusters, is not acceptable.

Development of clustering algorithms that are feasible for large dataset is an important current activity.

14.3 Classification by Space Tessellations

Groups in data are often formed by the values of a subset of the variables. A simple example is data in which one or more variables represent location. Clusters may be defined based on location, either by methods such as we have discussed above, or by regions that tessellate the space. The tessellations may be pre-assigned regions, perhaps corresponding to administrative or geographical boundaries.

More interesting tessellations can be constructed from data. Let $S$ be a set of points, possibly all of the observations in a dataset, a random sample of the data, or some statistics formed from subsets of the data. The set $S$ may be used to form a tessellation that defines groups, and the tessellation may be used to classify additional data. The use of a subset of the data is related to the multipolar mapping defined by Kao, Bergeron, and Sparr (1998) for use in proximity searches. Formation of a subset of observations for classification is a form of dimension reduction, which, as we have noted with caveats, is one of the general approaches for understanding multivariate data.

A simple and useful tessellation constructed from a dataset is the Dirichlet tessellation or the Voronoi tessellation. (The names are synonymous.) This tiling forms regions containing single points of $S$ in such a way that all points within a region are closer to the given point than they are to any other point in $S$. The points in $S$ that form the tessellation are called generators. The points on a boundary in a Dirichlet tessellation are equidistant to two points in $S$. This type of tessellation generalizes to higher dimensions.
The set of edges of the polygons (or faces of polyhedra or hyperpolyhedra) on which points are equidistant to two points in \( S \) is called the Voronoi diagram. The Dirichlet tessellation determined by a set of six points is shown in Figure 14.5.

![Voronoi diagram](image)

**Figure 14.5.** A Dirichlet Tessellation in a Plane Formed by Six Generator Points

The other points shown in Figure 14.5 are clustered with respect to the tessellation formed by the given six points.

A unique set of simplices is obtained by joining all nodes that share an edge in the Voronoi diagram. The set of triangles (which are simplices in two dimensions) formed by the Dirichlet tessellation in Figure 14.5 is shown in Figure 14.6. This construction is called a Delaunay triangulation. The triangulation is also a tessellation, and is sometimes called a Delaunay tessellation. The Voronoi diagram and the Delaunay triangulation are duals of each other; one determines the other.

The Dirichlet tessellation or Voronoi diagram and the Delaunay triangulation have a number of interesting properties that hold in \( d \) dimensions. Ash et al. (1988), Aurenhammer (1991) and Okabe et al. (2000) discuss many of these properties. One important property of the Delaunay triangulation is that it is the unique triangulation that maximizes the minimum angle in a grid formed from a fixed set of vertices. This property is easy to see in two dimensions. (See Figure 14.7 for an example of another triangulation that obviously lacks this property, when compared to Figure 14.6.) This property makes the Delaunay triangulation very useful in various fields of scientific computations. For example, it is a good way to form a set of solution points for the numerical solution of partial differential equations. (This is an “unstructured grid”; see Section 9.2, page 491.)
Another property of the Voronoi diagram and the associated Delaunay triangulation in two dimensions is that a circle centered at a point where three Voronoi tiles meet and which passes through the vertices of the Delaunay triangle enclosing that point will not contain a vertex of any other triangle. (There are possible degeneracies when multiple points are collinear, but the property still holds when more than three tiles meet.) This property also holds in higher dimensions for spheres and hyperspheres.

Starting with \(d + 1\) points and a simplex, the algorithm proceeds by recursive insertion of nodes. For each new node,

- Find any simplices whose circumscribed hyperspheres include the new node.
- Create a cavity by eliminating these simplices (if there are any).
- Create the new set of simplices by connecting the new point to the nodes that define this cavity.

This triangulation is relatively simple to implement in two dimensions, as in the simple application of unstructured grids for the numerical solution of partial differential equations. O’Rourke (1998) and Lee (1999a, 1999b) provide general descriptions of computational methods for Delaunay triangulations as well as other problems in computational geometry. Various programs for performing the tessellations and other computations in \(d\) dimensions are available at

www.geom.umn.edu/software/download/


A special type of Voronoi tessellation is one in which the generators are the centroids of the regions of the tessellation. This is called a centroidal Voronoi tessellation. A centroidal Voronoi tessellation with \(k\) regions can be formed by an iterative routine in which a random set of \(k\) generators is chosen, the Voronoi tessellation is determined, and the centroids of the regions are taken as generators for a new tessellation. The generators, which were the centroids of the previous tessellation will not in general be the centroids of the new tessellation, so the iterations are continued until a stopping criterion is achieved. See Lloyd 1982 for descriptions and properties of the process. Kieffer (1983) proved convergence of the process for a fairly restricted class of problems. General convergence properties are open questions.

A tessellation of a finite point set, \(T\), can be defined in terms of a tiling over a continuous region. The points within a given tile form the finite set of points within a given tessellation of the set \(T\). A K-means clustering is a centroidal Voronoi tessellation of a finite point set in which the means of the clusters are the generators. See Du, Faber, and Gunzburger (1999) for further discussion and applications.

14.4 Heuristic Approaches to Clustering and Classification

The various clustering methods ==
VERI clustering
conceptual clustering

“Visual-Empirical Region-of-Influence” Clustering

“Visual-Empirical Region-of-Influence” (VERI) Clustering
Osborn et al. (1998)
Osborn and Martinez (1995)

Conceptual Clustering

Gordon, 128.
Michalski (1980)

Exercises
A

Notation and Definitions

All notation used in this work is “standard”, and in most cases it conforms to the ISO conventions. (The notable exception is the notation for vectors.) I have opted for simple notation, which, of course, results in a one-to-many map of notation to object classes. Within a given context, however, the overloaded notation is generally unambiguous. I have endeavored to use notation consistently.

This appendix is not intended to be a comprehensive listing of definitions. The Subject Index, beginning on page ??, is a more reliable set of pointers to definitions, except for symbols that are not words.

General Notation

Uppercase italic Latin and Greek letters, $A$, $B$, $E$, $\Lambda$, and so on are generally used to represent either matrices or random variables. Random variables are usually denoted by letters nearer the end of the Latin alphabet, $X$, $Y$, $Z$, and by the Greek letter $\mathcal{E}$. Parameters in models (that is, unobservables in the models), whether or not they are considered to be random variables, are generally represented by lowercase Greek letters. Uppercase Latin and Greek letters, especially $P$, in general, and $\Phi$, for the normal distribution, are also used to represent cumulative distribution functions. Also, uppercase Latin letters are used to denote sets.

Lowercase Latin and Greek letters are used to represent ordinary scalar or vector variables and functions. **No distinction in the notation is made between scalars and vectors**; thus, $\beta$ may represent a vector and $\beta_i$ may represent the $i$th element of the vector $\beta$. In another context, however, $\beta$ may represent a scalar. All vectors are considered to be column vectors, although we may write a vector as $x = (x_1, x_2, \ldots, x_n)$. Transposition of a vector or a matrix is denoted by a superscript $^T$.

Uppercase calligraphic Latin letters, $\mathcal{F}$, $\mathcal{V}$, $\mathcal{W}$, and so on, are generally used to represent either vector spaces or transforms.
Subscripts generally represent indexes to a larger structure, for example, $x_{ij}$ may represent the $(i,j)^{th}$ element of a matrix, $X$. A subscript in parentheses represents an order statistic. A superscript in parentheses represents an iteration, for example, $x_i^{(k)}$ may represent the value of $x_i$ at the $k^{th}$ step of an iterative process.

- $x_i$ : The $i^{th}$ element of a structure (including a sample, which is a multiset).
- $x_{(i)}$ : The $i^{th}$ order statistic.
- $x^{(i)}$ : The value of $x$ at the $i^{th}$ iteration.

Realizations of random variables and placeholders in functions associated with random variables are usually represented by lowercase letters corresponding to the uppercase letters; thus, $\epsilon$ may represent a realization of the random variable $E$.

A single symbol in an italic font is used to represent a single variable. A Roman font or a special font is often used to represent a standard operator or a standard mathematical structure. Sometimes, a string of symbols in a Roman font is used to represent an operator (or a standard function); for example, $exp$ represents the exponential function, but a string of symbols in an italic font on the same baseline should be interpreted as representing a composition (probably by multiplication) of separate objects; for example, $exp$ represents the product of $e$, $x$, and $p$.

A fixed-width font is used to represent computer input or output; for example,

$$a = bx + \sin(c).$$

In computer text, a string of letters or numerals with no intervening spaces or other characters, such as $bx$ above, represents a single object, and there is no distinction in the font to indicate the type of object.

Some important mathematical structures and other objects are:

- $\mathbb{R}$ : The field of reals, or the set over which that field is defined.
- $\mathbb{R}^d$ : The usual $d$-dimensional vector space over the reals, or the set of all $d$-tuples with elements in $\mathbb{R}$.
- $\mathbb{R}_+^d$ : The usual $d$-dimensional vector space over the reals, or the set of all $d$-tuples with positive real elements.
A Notation and Definitions

\[ \mathbb{C} \]  The field of complex numbers, or the set over which that field is defined.

\[ \mathbb{Z} \]  The ring of integers, or the set over which that ring is defined.

\[ \mathbb{G}(n) \]  A Galois field defined on a set with \( n \) elements.

\( C^0, C^1, C^2, \ldots \)  The set of continuous functions, the set of functions with continuous first derivatives, and so forth.

\( i \)  The imaginary unit, \( \sqrt{-1} \).

Computer Number Systems

Computer number systems are used to simulate the more commonly used number systems. It is important to realize that they have different properties, however. Some notation for computer number systems follows.

\[ \mathbb{F} \]  The set of floating-point numbers with a given precision, on a given computer system, or this set together with the the four operators, \(+\), \(-\), \(*\), and \(/\). In some useful ways, \( \mathbb{F} \) is similar to \( \mathbb{R} \); see page 30.

\[ \mathbb{I} \]  The set of fixed-point numbers with a given length, on a given computer system, or this set together with the the four operators, \(+\), \(-\), \(*\), and \(/\). In some useful ways, \( \mathbb{I} \) is similar to \( \mathbb{Z} \); see page 30.

\( \epsilon_{\text{min}} \) and \( \epsilon_{\text{max}} \)  The minimum and maximum values of the exponent in the set of floating-point numbers with a given length (see page 32).

\( \epsilon_{\text{min}} \) and \( \epsilon_{\text{max}} \)  The minimum and maximum spacings around 1 in the set of floating-point numbers with a given length (see page 35).

\( \epsilon \) or \( \epsilon_{\text{mach}} \)  The machine epsilon, the same as \( \epsilon_{\text{min}} \) (see page 35).

\( \left[ \cdot \right]_c \)  The computer version of the object \( \cdot \) (see page 45).

\text{NaN}  Not-a-Number (see page 38).

Notation Relating to Random Variables

A common function used with continuous random variables is a density function, and a common function used with discrete random variables is a probability function. The more fundamental function for either type of random
variable is the cumulative distribution function, or CDF. The CDF of a random variable \( X \), denoted by \( P_X(x) \) or just by \( P(x) \), is defined by

\[
P(x) = \Pr(X \leq x),
\]

where “\( \Pr \)”, or “probability”, can be taken here as a primitive (it is defined in terms of a measure). For vectors (of the same length), “\( X \leq x \)” means that each element of \( X \) is less than or equal to the corresponding element of \( x \). Both the CDF and the density or probability function for a \( d \)-dimensional random variable are defined over \( \mathbb{R}^d \). (It is unfortunately necessary to state that “\( P(x) \)” means the “function \( P \) evaluated at \( x \)”, and likewise “\( P(y) \)” means the same “function \( P \) evaluated at \( y \)”, unless \( P \) has been redefined. Using a different expression as the argument does not redefine the function, despite the sloppy convention adopted by some statisticians.)

The density for a continuous random variable is just the derivative of the CDF (if it exists). The CDF is therefore the integral. To keep the notation simple, we likewise consider the probability function for a discrete random variable to be a type of derivative (a Radon-Nikodym derivative) of the CDF. Instead of expressing the CDF of a discrete random variable as a sum over a countable set, we often also express it as an integral. (In this case, however, the integral is over a set whose ordinary Lebesgue measure is 0.)

A useful analog of the CDF for a random sample is the empirical cumulative distribution function, or ECDF. For a sample of size \( n \), the ECDF is

\[
P_n(x) = \frac{1}{n} \sum_{i=1}^{n} I_{(-\infty,x]}(x_i)
\]

or, equivalently,

\[
P_n(x) = \frac{1}{n} \sum_{i=1}^{n} I_{[x,\infty)}(x),
\]

for the indicator function \( I \).

Functions and operators such as Cov and E that are commonly associated with Latin letters or groups of Latin letters are generally represented by that letter in a Roman font.

\[
\Pr(A) \quad \text{The probability of the event } A.
\]

\[
p_X(\cdot) \quad \text{The probability density function (or probability function), or the cumulative probability function, of the random variable } X.
\]

\[
E(g(X)) \quad \text{The expected value of the function } g \text{ of the random variable } X. \text{ The notation } E_P(\cdot), \text{ where } P \text{ is a cumulative distribution function or some other identifier of a probability distribution, is sometimes used to indicate explicitly the distribution with respect to which the expectation is evaluated.}
\]
V(g(X)) The variance of the function $g$ of the random variable $X$. The notation $V_P(\cdot)$ is also often used.

Cov(X, Y) The covariance of the random variables $X$ and $Y$. The notation $\text{Cov}_P(\cdot, \cdot)$ is also often used.

Cov(X) The variance-covariance matrix of the vector random variable $X$.

Corr(X, Y) The correlation of the random variables $X$ and $Y$. The notation $\text{Corr}_P(\cdot, \cdot)$ is also often used.

Corr(X) The correlation matrix of the vector random variable $X$.

**General Mathematical Functions and Operators**

Functions such as sin, max, span, and so on that are commonly associated with groups of Latin letters are generally represented by those letters in a roman font.

Generally, the argument of a function is enclosed in parentheses, for example, sin($x$), but often for the very common functions, the parentheses are omitted: sin$x$. In expressions involving functions, parentheses are generally used for clarity, for example, $\langle E(X) \rangle^2$ instead of $E^2(X)$.

Operators such as $d$ (the differential operator) that are commonly associated with a Latin letter are generally represented by that letter in a roman font.

$\times$ Binary operator denoting multiplication of elements of a field or ring.

$\times$ Binary operator denoting the cartesian product of two sets. The result is the set of ordered pairs of elements from the operand sets. This product is also called the direct product and the cross product.

$\times$ Binary operator denoting the cross product of two vectors in $\mathbb{R}^3$.

The phrase “cross product” is also used to refer to elementwise multiplication of the values of a variable, but the symbol $\times$ is not used to represent this operation.

$|x|$ The modulus of the real or complex number $x$; if $x$ is real, $|x|$ is the absolute value of $x$. 

The ceiling function evaluated at the real number $x$: $\lceil x \rceil$ is the smallest integer greater than or equal to $x$.

The floor function evaluated at the real number $x$: $\lfloor x \rfloor$ is the largest integer less than or equal to $x$.

The cardinality of the set $S$.

The indicator function:

$$I_S(x) = \begin{cases} 1, & \text{if } x \in S; \\ 0, & \text{otherwise.} \end{cases}$$

If $x$ is a scalar, the set $S$ is often taken as the interval $(-\infty, y]$, and in this case, the indicator function is the Heaviside function, $H$, evaluated at the difference of the argument and the upper bound on the interval:

$$I_{(-\infty, y]}(x) = H(y - x).$$

(An alternative definition of the Heaviside function is the same as this, except that $H(0) = 0$.) In higher dimensions, the set $S$ is often taken as the product set,

$$A^d = (-\infty, y_1] \times (-\infty, y_2] \times \cdots \times (-\infty, y_d]$$

$$= A_1 \times A_2 \times \cdots \times A_d,$$

and in this case,

$$I_{A^d}(x) = I_{A_1}(x_1)I_{A_2}(x_2) \cdots I_{A_d}(x_d),$$

where $x = (x_1, x_2, \ldots, x_d)$. The derivative of the indicator function is the Dirac delta function, $\delta(\cdot)$,
The Dirac delta “function”, defined by
\[ \delta(x) = 0, \quad \text{for } x \neq 0, \]
\[ \int_{-\infty}^{\infty} \delta(t) \, dt = 1. \]
The Dirac delta function is not a function in the usual sense. We do, however, refer to it as a function. For any continuous function \( f \), we have the useful fact
\[
\int_{-\infty}^{\infty} f(y) \, dy \delta(y - x) \, dy = f(x).
\]

The minimum value of the real scalar-valued function \( f \), or the smallest element in the countable set of real numbers \( S \).

The value of the argument of the real scalar-valued function \( f \) that yields its minimum value.

Bitwise binary exclusive-or (see page 141). The operator also is used as the direct sum of vector spaces.

Big \( O \); \( g(n) = O(f(n)) \) means there exists a positive constant \( M \) such that \( |g(n)| \leq M|f(n)| \) as \( n \to \infty \). \( g(n) = O(1) \) means \( g(n) \) is bounded from above.

Little \( o \); \( g(n) = o(f(n)) \) means \( g(n)/f(n) \to 0 \) as \( n \to \infty \).

Convergent in probability; \( X(n) = o_p(f(n)) \) means that for any positive \( \epsilon \), \( \Pr(|X(n) - f(n)| > \epsilon) \to 0 \) as \( n \to \infty \).

Big \( \Omega \); \( g(n) = \Omega(f(n)) \) means there exists a positive constant \( m \) such that \( |g(n)| \geq m|f(n)| \) as \( n \to \infty \). \( g(n) = \Omega(1) \) means \( g(n) \) is bounded from below.

Little \( \omega \); \( g(n) = \omega(f(n)) \) means \( f(n)/g(n) \to 0 \) as \( n \to \infty \).

The differential operator. The derivative with respect to the variable \( x \) is denoted by \( \frac{d}{dx} \).

For the scalar-valued function \( f \) of a scalar variable, differentiation (with respect to an implied variable) taken on the function once, twice, \ldots, \( k \) times.
For the vector-valued function $f$, the transpose of $f$ (a row-vector).

For the scalar-valued function $f$ of a vector variable, the gradient (that is, the vector of partial derivatives), also often denoted as $g_f$.

For the vector-valued function $f$ of a vector variable, the transpose of the Jacobian, which is often denoted as $J_f$; so $\nabla f = J_f^T$ (see below).

For the vector-valued function $f$ of a vector variable, the transpose of the Jacobian. The element in position $(i, j)$ is

$$\frac{\partial f_i(x)}{\partial x_j}.$$

For the scalar-valued function $f$ of a vector variable, the Hessian. Except in pathological cases it is symmetric. The element in position $(i, j)$ is

$$\frac{\partial^2 f(x)}{\partial x_i \partial x_j}.$$

The symbol $\nabla^2 f$ is sometimes also used to denote the diagonal of the Hessian, in which case it is called the Laplacian.

The convolution of the functions $f$ and $g$,

$$(f \ast g)(t) = \int f(x)g(t-x) \, dx.$$  

The convolution is a function.

For the functions $f$ and $g$ whose integrals are zero, the covariance of $f$ and $g$ at lag $t$;

$$\text{Cov}(f, g)(t) = \int f(x)g(t+x) \, dx.$$  

The covariance is a function; its argument is called the lag. $\text{Cov}(f, f)(t)$ is called the autocovariance of $f$ at lag $t$, and $\text{Cov}(f, f)(0)$ is called the variance of $f$. 

Covariance of $f$ and $g$ at lag $t$; $\text{Cov}(f, g)(t)$ is called the autocovariance of $f$ at lag $t$, and $\text{Cov}(f, f)(0)$ is called the variance of $f$. 

The symbol $\nabla^2 f$ is sometimes also used to denote the diagonal of the Hessian, in which case it is called the Laplacian.
Corr$(f, g)$ For the functions $f$ and $g$ whose integrals are zero, the correlation of $f$ and $g$ at lag $t$;

$$\text{Corr}(f, g)(t) = \frac{\int f(x)g(t + x) \, dx}{\sqrt{\text{Cov}(f, f)(0)\text{Cov}(g, g)(0)}}$$

The correlation is a function; its argument is called the lag. $\text{Cov}(f, f)(t)$ is called the autocorrelation of $f$ at lag $t$.

$f \otimes g$ The tensor product of the functions $f$ and $g$.

$$(f \otimes g)(w) = f(x)g(y) \text{ for } w = (x, y).$$

The operator is also used for the tensor product of two function spaces, and for the Kronecker product of two matrices.

$f^T$ or $Tf$ The transform of the function $f$ by the functional $T$.

$f^F$ usually denotes the Fourier transform of $f$.

$f^L$ usually denotes the Laplace transform of $f$.

$f^W$ usually denotes a wavelet transform of $f$.

$\delta$ A perturbation operator; $\delta x$ represents a perturbation of $x$, and not a multiplication of $x$ by $\delta$, even if $x$ is a type of object for which a multiplication is defined.

$\Delta(\cdot, \cdot)$ A real-valued difference function; $\Delta(x, y)$ is a measure of the difference of $x$ and $y$; for simple objects, $\Delta(x, y) = |x - y|$; for more complicated objects, a subtraction operator may not be defined, and $\Delta$ is a generalized difference.

$\tilde{x}$ A perturbation of the object $x$; $\Delta(x, \tilde{x}) = \delta x$.

Ave$(S)$ An average (of some kind) of the elements in the set $S$.

$(f^r)_p$ The $r^{th}$ moment of the function $f$ with respect to the density $p$.

$\bar{x}$ The mean of a sample of objects generically denoted by $x$.

$\bar{x}$ The complex conjugate of the object $x$; that is, if $x = r + \text{i}c$, then $\bar{x} = r - \text{i}c$.

Special Functions
log $x$  

The natural logarithm evaluated at $x$.

sin $x$  

The sine evaluated at $x$ (in radians), and similarly for other trigonometric functions.

$x!$  

The factorial of $x$. If $x$ is a positive integer, $x! = x(x-1) \cdots 2 \cdot 1$. For other values of $x$, except negative integers, $x!$ is often defined as $x! = \Gamma(x+1)$.

$\Gamma(\alpha)$  

The complete gamma function. For $\alpha$ not equal to a nonpositive integer, 

$$\Gamma(\alpha) = \int_0^\infty t^{\alpha-1} e^{-t} \, dt.$$  

We have the useful relationship, $\Gamma(\alpha) = (\alpha - 1)!$. An important argument is $\frac{1}{2}$, and $\Gamma\left(\frac{1}{2}\right) = \sqrt{\pi}$.

$\Gamma_x(\alpha)$  

The incomplete gamma function: 

$$\Gamma_x(\alpha) = \int_0^x t^{\alpha-1} e^{-t} \, dt.$$  

$B(\alpha, \beta)$  

The complete beta function: 

$$B(\alpha, \beta) = \int_0^1 t^{\alpha-1} (1-t)^{\beta-1} \, dt,$$

where $\alpha > 0$ and $\beta > 0$. A useful relationship is 

$$B(\alpha, \beta) = \frac{\Gamma(\alpha) \Gamma(\beta)}{\Gamma(\alpha + \beta)}.$$  

$B_x(\alpha, \beta)$  

The incomplete beta function: 

$$B_x(\alpha, \beta) = \int_0^x t^{\alpha-1} (1-t)^{\beta-1} \, dt.$$  

Vectors, Vector Spaces, and Matrices

⊕  

Direct sum of vector spaces (see page 230).
<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{V}(G)$</td>
<td>For the set of vectors (all of the same order) $G$, the vector space generated by that set.</td>
</tr>
<tr>
<td>$\mathcal{V}(X)$</td>
<td>For the matrix $X$, the vector space generated by the columns of $X$.</td>
</tr>
<tr>
<td>$\text{span}(Y)$</td>
<td>For $Y$ either a set of vectors or a matrix, the vector space $\mathcal{V}(Y)$.</td>
</tr>
<tr>
<td>$\perp$</td>
<td>Orthogonality relationship (vectors, see page 233; vector spaces, see page 248).</td>
</tr>
<tr>
<td>$\mathcal{V}^\perp$</td>
<td>The orthogonal complement of the vector space $\mathcal{V}$ (see page 248).</td>
</tr>
<tr>
<td>$\mathcal{N}(X)$</td>
<td>The null space of the matrix $X$, that is, the orthogonal complement of the vector space generated by the columns of $X$. If $X$ has $n$ rows, $\mathcal{V}(X) \oplus \mathcal{N}(X) = \mathbb{R}^n$.</td>
</tr>
<tr>
<td>$\bar{A}$</td>
<td>The matrix whose elements are the complex conjugates of the elements of the matrix $A$.</td>
</tr>
<tr>
<td>$A^T$</td>
<td>For the matrix $A$, its transpose (also used for a vector to represent the corresponding row vector).</td>
</tr>
<tr>
<td>$A^H$</td>
<td>The conjugate transpose of the matrix $A$; $A^H = \bar{A}^T$.</td>
</tr>
<tr>
<td>$A^{-1}$</td>
<td>The inverse of the square, nonsingular matrix $A$.</td>
</tr>
<tr>
<td>$A^+$</td>
<td>The $g_4$ inverse, or the Moore-Penrose inverse, or the pseudoinverse, of the matrix $A$ (see page 247).</td>
</tr>
<tr>
<td>$A^*$</td>
<td>A $g_2$ inverse of the matrix $A$ (see page 247).</td>
</tr>
<tr>
<td>$A^-$</td>
<td>A $g_1$, or generalized, inverse of the matrix $A$ (see page 247).</td>
</tr>
<tr>
<td>$A^\dagger$</td>
<td>For the nonnegative definite matrix $A$, the Cholesky factor; that is, $(A^\dagger)^T A^\dagger = A$.</td>
</tr>
<tr>
<td>$\otimes$</td>
<td>The tensor product or the direct product; in particular, if the operands are matrices, the Kronecker multiplication (see page 240).</td>
</tr>
</tbody>
</table>
644 A Notation and Definitions

\text{sign}(x) \quad \text{For the vector } x, \text{ a vector of units corresponding to the signs:}

\begin{align*}
\text{sign}(x)_i &= 1 \text{ if } x_i > 0, \\
&= 0 \text{ if } x_i = 0, \\
&= -1 \text{ if } x_i < 0;
\end{align*}

with a similar meaning for a scalar. The sign function is also sometimes called the signum function, and denoted \text{sgn}(\cdot).

\text{L}_p \quad \text{For real } p \geq 1, \text{ a norm formed by accumulating the } p^{\text{th}} \text{ powers of the moduli of individual elements in an object and then taking the } (1/p)^{\text{th}} \text{ power of the result (see page 258).}

\| \cdot \| \quad \text{In general, the norm of the object } \cdot. \text{ Often, however, specifically either the } \text{L}_2 \text{ norm, or the norm defined by an inner product.}

\| x \|_p \quad \text{For the vector } x, \text{ the } \text{L}_p \text{ norm:}

\| x \|_p = \left( \sum |x_i|^p \right)^{\frac{1}{p}}

(see page 258).

\| X \|_p \quad \text{For the matrix } X, \text{ the } \text{L}_p \text{ norm:}

\| X \|_p = \max_{\| v \|_p = 1} \| Xv \|_p

(see page 260).

\| f \|_p \quad \text{For the function } f, \text{ the } \text{L}_p \text{ norm:}

\| f \|_p = \left( \int |f(x)|^p dx \right)^{\frac{1}{p}}

(see page 548).

\| X \|_F \quad \text{For the matrix } X, \text{ the Frobenius norm:}

\| X \|_F = \sqrt{\sum_{i,j} x_{ij}^2}

(see page 261).
\((x, y)\) The inner product of \(x\) and \(y\) (vectors, see page 230; functions, see page 547).

\(\kappa_p(A)\) The L\(_p\) condition number of the nonsingular square matrix \(A\) with respect to inversion (see page 264).

\(\text{diag}(v)\) For the vector \(v\), the diagonal matrix whose nonzero elements are those of \(v\); that is, the square matrix, \(A\), such that \(A_{ii} = v_i\) and for \(i \neq j\), \(A_{ij} = 0\).

\(\text{diag}(A_1, A_2, \ldots, A_k)\) The block diagonal matrix whose submatrices along the diagonal are \(A_1, A_2, \ldots, A_k\).

\(\text{vec}(A)\) The vector consisting of the columns of the matrix \(A\), all strung into one vector; if the column vectors of \(A\) are \(a_1, a_2, \ldots, a_m\) then 
\[
\text{vec}(A) = (a_1^T, a_2^T, \ldots, a_m^T).
\]

\(\text{vech}(A)\) For the symmetric the matrix \(A\), the vector consisting of the unique elements all strung into one vector:
\[
\text{vech}(A) = (a_{11}, a_{21}, a_{22}, a_{31}, \ldots, a_{m1}, \ldots, a_{mm}).
\]

\(\text{trace}(A)\) The trace of the square matrix \(A\), that is, the sum of the diagonal elements.

\(\text{rank}(A)\) The rank of the matrix \(A\), that is, the maximum number of independent rows (or columns) of \(A\).

\(\rho(A)\) The spectral radius of the matrix \(A\) (the maximum absolute value of its eigenvalues).

\(\det(A)\) The determinant of the square matrix \(A\), \(\det(A) = |A|\).

\(|A|\) The determinant of the square matrix \(A\), \(|A| = \det(A)\).

**Special Vectors and Matrices**

1 or \(1_n\) A vector (of length \(n\)) whose elements are all 1’s.

0 or \(0_n\) A vector (of length \(n\)) whose elements are all 0’s.
I or $I_n$  The \((n \times n)\) identity matrix.

e_i  The \(i^{th}\) unit vector (with implied length) (see page 232).

$E_{jk}$  The \((i,j)^{th}\) elementary permutation matrix (see page 249).

Models and Data

A form of model used often in statistics and applied mathematics has three parts: a left-hand side representing an object of primary interest; a function of another variable and a parameter, each of which is likely to be a vector; and an adjustment term to make the right-hand side equal the left-hand side. The notation varies depending on the meaning of the terms. One of the most common models used in statistics, the linear regression model with normal errors, is written as

\[ Y = \beta^T x + E. \tag{A.1} \]

The adjustment term is a random variable, denoted by an uppercase epsilon. The term on the left-hand side is also a random variable. This model does not represent observations or data. A slightly more general form is

\[ Y = f(x; \theta) + E. \tag{A.2} \]

A single observation or a single data item that corresponds to model (A.1) may be written as

\[ y = \beta^T x + \epsilon \]

or, if it is one of several,

\[ y_i = \beta^T x_i + \epsilon_i. \]

Similar expressions are used for a single data item that corresponds to model (A.2).

In these cases, rather than being a random variable, \(\epsilon\) or \(\epsilon_i\) may be a realization of a random variable, or it may just be an adjustment factor with no assumptions about its origin.

A set of \(n\) such observations is usually represented in an \(n\)-vector \(y\), a matrix \(X\) with \(n\) rows, and an \(n\)-vector \(\epsilon\):

\[ y = X\beta + \epsilon \]

or

\[ y = f(X; \theta) + \epsilon. \]

The model is not symmetric in \(y\) and \(x\). The error term is added to the systematic component that involves \(x\). The has implications in estimation and model fitting.
Solutions and Hints for Selected Exercises

Exercises Beginning on Page xxx
As might be expected, the literature in the interface of computer science, numerical analysis, and various areas of application is quite diverse. Relevant articles are likely to appear in journals devoted to quite different disciplines.

In addition to literature and learned societies in the traditional forms, an important source of communication and a repository of information are computer databases and forums. In some cases the databases duplicate what is available in some other form, but often the material and the communications facilities provided by the computer are not available elsewhere.

**Literature in Computational Science**

In the Library of Congress classification scheme, most books on numerical analysis are generally in QA279, and computer science in QA76. Many of the books in the interface of these disciplines are classified in these or other places within QA.

*Mathematical Reviews*, published by the American Mathematical Society (AMS), contains brief reviews of articles in all areas of mathematics. The areas of “Numerical Analysis” and “Computer Science” contain reviews of articles relevant to computational science. The papers reviewed in *Mathematical Reviews* are categorized according to a standard system that has slowly evolved over the years. In this taxonomy, called the AMS MR classification system, “Numerical Analysis”, including random number generation, is 65Xyy and “Computer Science” is 68Xyy. (“X” represents a letter and “yy” represents a two-digit number.) *Mathematical Reviews* is available to subscribers via the World Wide Web at MathSciNet:

http://www.ams.org/mathscinet/

*Current Index to Statistics*, published annually by the American Statistical Association and the Institute for Mathematical Statistics, contains both author and subject indexes that are useful in finding journal articles or books in statistical computing.
The Association for Computing Machinery (ACM) publishes an annual index, by author, title, and keyword, of the literature in the computing sciences. There are various handbooks of mathematical functions and formulas that are useful in numerical computations. Three that should be mentioned are Abramowitz and Stegun (1964), Spanier and Oldham (1987), and Thompson (1997). Anyone doing serious scientific computations should have ready access to at least one of these volumes.

Some of the more important journals in computational science and scientific computing include the following.

**ACM Transactions on Mathematical Software**, published quarterly by the ACM (Association for Computing Machinery). This journal publishes algorithms in Fortran and C. The ACM collection of algorithms is sometimes called CALGO. The algorithms published during the period 1975 through 1999 are available on a CR-ROM from ACM. Most of the algorithms are available through netlib at

http://www.netlib.org/liblist.html

**ACM Transactions on Modeling and Computer Simulation**, published quarterly by the ACM.

**Applied Statistics**, published quarterly by the Royal Statistical Society. (Until 1998, included algorithms in Fortran. Some of these algorithms, with corrections, were collected by Griffiths and Hill, 1985. Most of the algorithms are available through statlib at Carnegie Mellon University.)

**SIAM Journal on Numerical Analysis** published bimonthly by SIAM.

**SIAM Journal on Scientific Computing**, published bimonthly by SIAM. This journal was formerly **SIAM Journal on Scientific and Statistical Computing**.

### World Wide Web, News Groups, List Servers, and Bulletin Boards

The best way of storing information is in a digital format that can be accessed by computers. In some cases the best way for people to access information is by computers; in other cases the best way is via hard copy, which means that the information stored on the computer must go through a printing process resulting in books, journals, or loose pages.

A huge amount of information and raw data is available online. Much of it is in publicly accessible sites. Some of the repositories give space to ongoing discussions to which anyone can contribute.

There are various ways of remotely accessing the computer databases and discussion groups. The high-bandwidth wide-area network called the “Internet” is the most important way to access information. Early development of the Internet was due to initiatives within the United States Department of Defense and the National Science Foundation. The Internet is making fundamental changes to the way we store and access information.
The references that I have cited in this text are generally traditional books, journal articles, or compact disks. This usually means that the material has been reviewed by someone other than the author. It also means that the author possibly has newer thoughts on the same material. The Internet provides a mechanism for the dissemination of large volumes of information that can be updated readily. The ease of providing material electronically is also the source of the major problem with the material: it is often half-baked and has not been reviewed critically. Another reason that I have refrained from making frequent reference to material available over the Internet is the unreliability of some sites. It has been estimated that the average life of a Web site is 45 days (in early 1998).

A very useful site for scientific computing is netlib, which was established by research workers at AT&T (now Alcatel-Lucent) Bell Laboratories and national laboratories, primarily Oak Ridge National Laboratories. The URL is

http://www.netlib.org

The Collected Algorithms of the ACM (CALGO), which are the Fortran, C, and Algol programs published in ACM Transactions on Mathematical Software (or in Communications of the ACM prior to 1975), are available in netlib.

There is also an X Windows, socket-based system for accessing netlib, called Xnetlib; see Dongarra, Rowan, and Wade (1995).

The Guide to Available Mathematical Software (GAMS), to which I have referred several times in this book, can be accessed at

http://gams.nist.gov

A different interface, using Java, is available at

http://math.nist.gov/HotGAMS/

Another useful site is the electronic repository statlib, maintained at Carnegie Mellon University, which contains programs, datasets, and other items of interest. The URL is

http://lib.stat.cmu.edu

The collection of algorithms published in Applied Statistics is available in statlib. These algorithms are sometimes called the ApStat algorithms.

There are two major problems in using the WWW to gather information. One is the sheer quantity of information and the number of sites providing information. The other is the “kiosk problem”; anyone can put up material. Sadly, the average quality is affected by a very large denominator. The kiosk problem may be even worse than a random selection of material; the “fools in public places” syndrome is much in evidence.

There is not much that can be done about the second problem. It was not solved for traditional postings on uncontrolled kiosks, and it will not be solved on the WWW.
For the first problem, there are remarkable programs that automatically crawl through WWW links to build a database that can be searched for logical combinations of terms and phrases. Such systems and databases have been built by several people and companies. Three of the most useful are Northern Light at

http://www.northernlight.com

Alta Vista, at

http://www.altavista.com

and HotBot at

http://www.hotbot.com

In a study by Lawrence and Giles (1998), the Alta Vista and HotBot search engines provided more complete coverage of the scientific literature than four other search engines considered. The Lawrence and Giles study, however, indicated that use of all six search engines provided about 3.5 times as many documents on average as use of just a single engine. A later study by the same authors (Lawrence and Giles, 1999) found that Northern Light and Snap indexed more pages than HotBot, but many links were invalid on Northern Light and Snap. An important finding of the later study was that the number of pages on the Web is growing faster than the rate at which they are being indexed. They estimated that, of the eleven search engines studied, none indexed more than 16 percent of all Web pages. The eleven search engines together covered about 42 percent of the Web. A software tool such as MetaCrawler that harnesses together several search engines makes it more likely that relevant Web pages will be found. Of course, it also insures that many irrelevant ones will be found.

A very widely used search program is “Yahoo” at

http://www.yahoo.com

A neophyte can be quickly disabused of an exaggerated sense of the value of such search engines by doing a search on “Monte Carlo”. Aside from the large number of hits that relate to a car and to some place in Europe, the hits (in mid 1998) that relate to the interesting topic are dominated by references to some programs for random number generation put together by a group at a university somewhere. (Of course, “interesting” is in the eye of the beholder.)

It is not clear at this time what will be the media for the scientific literature within a few years. Many of the traditional journals will be converted to an electronic version of some kind. Journals will become Web sites. That is for certain; the details, however, are much less certain. Many bulletin boards and discussion groups have already evolved into “electronic journals”. A publisher of a standard commercial journal has stated that “we reject 80% of the articles submitted to our journal; those are the ones you can find on the Web”. Lesk (1997) discusses many of the issues that must be considered as the standard
repositories of knowledge change from paper books and journals to digital libraries.

**References for Software Packages**

There is a wide range of software used in the computational sciences. Some of the software is produced by a single individual who is happy to share the software, sometimes for a fee, but who has no interest in maintaining the software. At the other extreme is software produced by large commercial companies whose continued existence depends on a process of production, distribution, and maintenance of the software. Information on much of the software can be obtained from GAMS. Some of the free software can be obtained from [statlib](#) or [netlib](#).

**References to the Literature**

The following bibliography obviously covers a wide range of topics in scientific computing. Except for a few of the general references, all of these entries have been cited in the text.

The purpose of this bibliography is to help the reader get more information; hence I eschew “personal communications” and references to technical reports that may or may not exist. Those kinds of references are generally for the author rather than for the reader.

**A Note on the Names of Authors**

In these references, I have generally used the names of authors as they appear in the original sources. This may mean that the same author will appear with different forms of names, sometimes with given names spelled out, and sometimes abbreviated. In the author index, beginning on page ??, I use a single name for the same author. The name is generally the most unique (i.e., least abbreviated) of any of the names of that author in any of the references. This convention may occasionally result in an entry in the author index that does not occur exactly in any references. For example, a reference to J. Paul Jones together with one to John P. Jones, if I know that the two names refer to the same person, would result in an Author Index entry for John Paul Jones.

Abramowitz, Milton, and Irene A. Stegun (Editors) (1964), *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, National Bureau of Standards (NIST), Washington. (Reprinted by


Bentley, Jon Louis; Mary F. Fernandez; Brian W. Kernighan; and Norman L. Schryer (1993), Template-driven interfaces for numerical subroutines, ACM Transactions on Mathematical Software 19, 265–287.


Best, D. J. (1983), A note on gamma variate generators with shape parameter less than unity, Computing 30, 185–188.


Bratley, Paul; Bennett L. Fox; and Harald Niederreiter (1992), Implementation and tests of low-discrepancy sequences, ACM Transactions on Modeling and Computer Simulation 2, 195–213.

Bratley, Paul; Bennett L. Fox; and Harald Niederreiter (1994), Algorithm 738: Programs to generate Niederreiter’s low-discrepancy sequences, ACM Transactions on Mathematical Software 20, 494–495.


Carta, David G. (1990), Two fast implementations of the “minimal standard” random number generator, *Communications of the ACM* 33, Number 1 (January), 87–88.


Coleman, Dan; Xiaopeng Dong; Johanna Hardin; David M. Rocke; and David L. Woodruff (1999), Some computational issues in cluster analysis with no a priori metric, *Computational Statistics & Data Analysis* **31**, 1–11.


Corana, A.; M. Marchesi; C. Martin; and S. Ridella (1987), Minimizing multimodal function of continuous variables with the “simulated annealing” algorithm, *ACM Transactions on Mathematical Software* 13, 262–280.


Du, Qiang; Vance Faber; and Max Gunzburger (1999), Centroidal Voronoi tessellations: Applications and algorithms, SIAM Review 41, 637–676.


Eckart, Carl, and Gale Young (1936), The approximation of one matrix by another of lower rank, Psychometrika 1, 211–218.


Engquist, Bjorn; L. Johnsson; M. Hammill; and F. Short (Editors) (2000), *Simulation and Visualization on the Grid*, Springer-Verlag, Berlin.


Fayyad, Usama M.; Gregory Piatetsky-Shapiro; Padhraic Smyth; and Ramasamy Uthurusamy (Editors) (1996), Advances in Knowledge Discovery and Data Mining, American Association for Artificial Intelligence, Menlo Park, California.


Ferris, Michael; Michael P. Mesnier; and Jorge J. Moré (2000), NEOS and Condor: Solving optimization problems over the internet, ACM Transactions on Mathematical Software 26, 1–18.


Fushimi, Masanori (1990), Random number generation with the recursion $X_t = X_{t-3p} \oplus X_{t-3q}$, *Journal of Computational and Applied Mathematics* **31**, 105–118.


Glymour, Clark; David Madigan; Daryl Pregibon; and Padhraic Smyth (1996), Statistical inference and data mining, *Communications of the ACM* **39**(11), 35–41.


Jaulin, Luc; Michel Keiffer; Olivier Didrit; and Eric Walter (2001), *Applied Interval Analysis*, Springer-Verlag, New York.


Jöhnk, M. D. (1964), Erzeugung von Betaverteilter und Gammaverteilter Zu-
Sons, New York.
Jones, M. C., and Robin Sibson (1987), What is projection pursuit (with
Joy, Corwin; Phelim P. Boyle; and Ken Seng Tan (1996), Quasi-Monte Carlo
Jung, Joo Sung, and Bong Jin Yum (1996), Construction of exact D-optimal
designs by tabu search, *Computational Statistics & Data Analysis* 21,
Kachitvichyanukul, Voratas (1982), *Computer generation of Poisson, binom-
ial, and hypergeometric random variables*, unpublished Ph.D. disserta-
tion, Purdue University, West Lafayette, Indiana.
Kachitvichyanukul, Voratas, and Bruce W. Schmeiser (1988a), Binomial ran-
Kachitvichyanukul, Voratas, and Bruce W. Schmeiser (1988b), Algorithm 668:
H2PEC: Sampling from the hypergeometric distribution, *ACM Transac-
tions on Mathematical Software* 14, 397–398.
Kachitvichyanukul, Voratas, and Bruce W. Schmeiser (1990), BTPEC: Sam-
ping from the binomial distribution, *ACM Transactions on Mathematical
Software* 16, 394-397.
Kahn, H., and A. W. Marshall (1953), Methods of reducing sample size in
Monte Carlo computations, *Journal of Operations Research Society of
America* 1, 263-278.
Kamel, M. S.; W. H. Enright; and K. S. Ma (1993), ODEXPERT: an expert
system to select numerical solvers for initial value ODE systems, *ACM
Transactions on Mathematical Software* 19, 44–62.
Kao, David T.; R. Daniel Bergeron; and Ted M. Sparr (1998), Efficient prox-
imity search in multivariate data, *Proceedings of the Tenth International
Conference on Scientific and Statistical Database Management* (edited
by Maurizio Rafanelli and Matthias Jarke), IEEE Computer Society, Los
Alamitos, California, 145–154.
Kato, Takashi; Li-ming Wu; and Niro Yanagihara (1996a), On a nonlinear con-
gruential pseudorandom number generator, *Mathematics of Computation
Kato, Takashi; Li-ming Wu; and Niro Yanagihara (1996b), The serial test for a
nonlinear pseudorandom number generator, *Mathematics of Computation
65*, 761–769.
Kaufman, Leonard, and Peter J. Rousseeuw (1990), *Finding Groups in Data:
Kavvadias, Dimitris J., and Michael N. Vrahatis (1996), Locating and com-
puting all the simple roots and extrema of a function, *SIAM Journal on
Scientific Computing* 17, 1232–1248.


Kennedy, William J.; James E. Gentle; and V. A. Sposito (1977), A computer oriented method for generating test problems for $L_1$ regression, Communications in Statistics — Simulation and Computation B6, 21–27.


Kim, Choongrak; Woochul Kim; Byeong U. Park; Changkon Hong; and Meeseon Jeong (1999), Smoothing techniques via the Bezier curve, Communications in Statistics — Theory and Methods 28, 1577–1597.


Lawson, C. L.; R. J. Hanson; D. R. Kincaid; and F. T. Krogh (1979), Basic linear algebra subprograms for Fortran usage, *ACM Transactions on Mathematical Software* 5, 308–323.


L’Ecuyer, Pierre, and Richard Simard (1999), Beware of linear congruential generators with multipliers of the form $a = \pm 2^q \pm 2^r$, *ACM Transactions on Mathematical Software* 25, 367–374.


Lüscher, Martin (1994), A portable high-quality random number generator for lattice field theory simulations, *Computer Physics Communications* 79, 100–110.

Ma, Jun, and H. Malcolm Hudson (1997), Modified Fisher scoring algorithms using Jacobi or Gauss-Seidel subiterations, *Computational Statistics* 12, 467–479.


Available at [http://stat.fsu.edu/~geo/diehard.html](http://stat.fsu.edu/~geo/diehard.html).


Marsaglia, George; Arif Zaman; and John C. W. Marsaglia (1994), Rapid evaluation of the inverse normal distribution function, *Statistics and Probability Letters* 19, 259–266.


Särndal, Carl-Erik; Bengt Swensson; and Jan Wretman (1992), *Model Assisted Survey Sampling*, Springer-Verlag, New York.


Seaver, Bill; Konstantinos Triantis; and Chip Reeves (1999), The identification of influential subsets in regression using a fuzzy clustering strategy, *Technometrics* 41, 340–351.


Strassen, V. 1969, Gaussian elimination is not optimal, Numerische Mathematik 13, 354–356.


Wu, Pei-Chi (1997), Multiplicative, congruential random-number generators with multiplier $\pm 2^{k_1} \pm 2^{k_2}$ and modulus $2^p - 1$, *ACM Transactions on Mathematical Software* 23, 255–265.


Zeger, Kenneth; Jacques Vaisey; and Allen Gersho (1992), Globally optimal vector quantizer design by stochastic relaxation *IEEE Transactions on Signal Processing* 40, 310–322.


## Index

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-splines</td>
<td>536</td>
</tr>
<tr>
<td>LDU factorization</td>
<td>284</td>
</tr>
<tr>
<td>LU factorization</td>
<td>283, 284</td>
</tr>
<tr>
<td>QR factorization</td>
<td>287</td>
</tr>
<tr>
<td>QR method for eigenvalues</td>
<td>295, 310, 316</td>
</tr>
<tr>
<td>T convexity</td>
<td>351</td>
</tr>
<tr>
<td>$\pi$ as a source of random numbers</td>
<td>146</td>
</tr>
<tr>
<td>$g_1$ inverse</td>
<td>247</td>
</tr>
<tr>
<td>$g_2$ inverse</td>
<td>247</td>
</tr>
<tr>
<td>$g_4$ inverse (pseudoinverse)</td>
<td>247</td>
</tr>
<tr>
<td>$k$-d-tree</td>
<td>583</td>
</tr>
<tr>
<td>$L_p$ norm</td>
<td>258, 260</td>
</tr>
<tr>
<td>$\text{sign}()$</td>
<td>251</td>
</tr>
<tr>
<td>$\text{span}()$</td>
<td>230</td>
</tr>
<tr>
<td>“inverse” distribution</td>
<td>198</td>
</tr>
<tr>
<td>“minimal standard” generator</td>
<td>124, 131, 136, 155, 164</td>
</tr>
<tr>
<td>ACM Transactions on Mathematical Software</td>
<td>89, 214, 650, 651</td>
</tr>
<tr>
<td>ACM Transactions on Modeling and Computer Simulation</td>
<td>650</td>
</tr>
<tr>
<td>Applied Statistics</td>
<td>89, 214, 650, 651</td>
</tr>
<tr>
<td>CALGO (Collected Algorithms of the ACM)</td>
<td>89, 650, 651</td>
</tr>
<tr>
<td>Collected Algorithms of the ACM (CALGO)</td>
<td>89, 650, 651</td>
</tr>
<tr>
<td>Mathematical Reviews</td>
<td>649</td>
</tr>
<tr>
<td>Reliable Computing</td>
<td>52</td>
</tr>
<tr>
<td>SIAM Journal on Numerical Analysis</td>
<td>650</td>
</tr>
<tr>
<td>SIAM Journal on Scientific Computing</td>
<td>650</td>
</tr>
<tr>
<td>RANDU, 129, 165, 224</td>
<td></td>
</tr>
<tr>
<td>Xnetlib</td>
<td>651</td>
</tr>
<tr>
<td>daxpy, 229</td>
<td></td>
</tr>
<tr>
<td>isnan, 38</td>
<td></td>
</tr>
<tr>
<td>netlib, 100, 214, 430, 439, 650, 651</td>
<td></td>
</tr>
<tr>
<td>rand, 149, 215</td>
<td></td>
</tr>
<tr>
<td>saxpy, 229, 277</td>
<td></td>
</tr>
<tr>
<td>statlib, 214, 650, 651</td>
<td></td>
</tr>
<tr>
<td>3-D sphere test</td>
<td>162</td>
</tr>
<tr>
<td>absolute error</td>
<td>47, 53, 274</td>
</tr>
<tr>
<td>acceptance/complement method, for random number generation</td>
<td>182</td>
</tr>
<tr>
<td>acceptance/rejection method, for random number generation</td>
<td>178</td>
</tr>
<tr>
<td>ACORN congruential generator</td>
<td>147</td>
</tr>
<tr>
<td>Ada (programming language)</td>
<td>94</td>
</tr>
<tr>
<td>adaptive direction sampling</td>
<td>194</td>
</tr>
<tr>
<td>adaptive simulated annealing</td>
<td>411</td>
</tr>
<tr>
<td>additive congruential random number generator</td>
<td>122</td>
</tr>
<tr>
<td>admissibility</td>
<td>427</td>
</tr>
<tr>
<td>affine transformation</td>
<td>271</td>
</tr>
<tr>
<td>algorithm, definition</td>
<td>64, 359</td>
</tr>
<tr>
<td>Alta Vista (Web search engine)</td>
<td>652</td>
</tr>
<tr>
<td>alternating conditional sampling</td>
<td>193</td>
</tr>
<tr>
<td>Amdahl’s law</td>
<td>63, 85</td>
</tr>
<tr>
<td>AMPL (modeling system)</td>
<td>437, 439</td>
</tr>
<tr>
<td>AMS MR classification system</td>
<td>649</td>
</tr>
<tr>
<td>angle between vectors</td>
<td>232, 262</td>
</tr>
</tbody>
</table>


angular separation, 571
ANSI (standards), 39, 88
ANSYS, 505
antithetic variates, 13, 133, 454
API (applications programming interface), 88
approximation of a matrix, 308
ArcView (software), 107
array data structure, 71, 562
artificial ill-conditioning, 266
ASCII code, 27, 88
ATLAS (software), 325
autonomous models, 507
axpy, 229, 545
B-tree, 566
Bézier curve, 537
backward error analysis, 53, 58
Banach space, 549
Banachiewicz factorization, 285
base, 32
base point, 31, 77
basis, 230
basis Exercise 5.2, 342
batch algorithm, 67
batch means for variance estimation, 11
benchmarks, 102
Beowulf, 84
Bernoulli distribution, generating random deviates, 174
Bernstein polynomial, 537
Bessel's inequality, 465
beta distribution, generating random deviates, 208
beta function, 642
beta weight function, 469, 537
beta-binomial distribution, generating random deviates, 210
Beyer ratio, 157
BFGS (method), 389
bias, in exponent of floating-point number, 33
big endian, 43
big O (order), 56, 60
big omega (order), 56
bilinear form, 242, 248
bin smoother, 540
bin width, 539
binary difference, 571
binary matrix rank test, 160
binary random variables, generating random deviates, 174
binary search, 565
binary tree, 565
binding, computer language, 88
binning, 74
binomial distribution, generating random deviates, 209
bioinformatics, 107
birthday spacing test, 160
bisection method, 360
bit, 27
bit stripping, 122, 123, 134
bitmap, 29
bitstream test, 160
BLACS (software), 326, 338
BLAS (software), 93, 323, 326, 338
blocks, simulation experiments, 152
Boltzmann distribution, 19, 407
boundary value problem, 489
branch and bound, 407
Broyden update, 389
Bulirsch-Stoer method, 487
byte, 27
C (programming language), 39, 50, 213, 326, 430
C++ (programming language), 40, 94
C5.0, 73
cache, 76
Canberra distance, 571
cancellation error, 50, 58
CART (method and software), 73
cartesian product, 637
catastrophic cancellation, 48
categorical variable, 539
Cauchy-Schwarz inequality, 231, 241, 547
Cayley multiplication, 239
CDF (Common Data Format), 28, 88, 104, 107
CDF (cumulative distribution function), 636
cellular automata, 146
centered data, 533, 567, 587
centroidal tessellation, 631
CFD, 506
chaining of operations, 48
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>chaotic systems</td>
<td>147</td>
</tr>
<tr>
<td>character data</td>
<td>28</td>
</tr>
<tr>
<td>character string</td>
<td>28</td>
</tr>
<tr>
<td>characteristic equation</td>
<td>252, 310</td>
</tr>
<tr>
<td>characteristic function</td>
<td>553</td>
</tr>
<tr>
<td>chasing</td>
<td>316</td>
</tr>
<tr>
<td>Chebyshev approximation</td>
<td>459, 473</td>
</tr>
<tr>
<td>Chebyshev generator</td>
<td>147</td>
</tr>
<tr>
<td>Chebyshev norm</td>
<td>258, 548</td>
</tr>
<tr>
<td>Chebyshev polynomial</td>
<td>469</td>
</tr>
<tr>
<td>chi-squared test</td>
<td>158</td>
</tr>
<tr>
<td>Cholesky decomposition</td>
<td>285</td>
</tr>
<tr>
<td>Christoffel-Darboux formula</td>
<td>467</td>
</tr>
<tr>
<td>circular data</td>
<td>573</td>
</tr>
<tr>
<td>class (of software objects)</td>
<td>94</td>
</tr>
<tr>
<td>classification</td>
<td>615, 622</td>
</tr>
<tr>
<td>classification tree</td>
<td>73, 617,622</td>
</tr>
<tr>
<td>cluster algorithm</td>
<td>20</td>
</tr>
<tr>
<td>cluster of computers</td>
<td>83</td>
</tr>
<tr>
<td>clustering</td>
<td>615</td>
</tr>
<tr>
<td>column rank</td>
<td>242</td>
</tr>
<tr>
<td>column space</td>
<td>235, 242</td>
</tr>
<tr>
<td>column-major</td>
<td>275, 320,322</td>
</tr>
<tr>
<td>column-sum norm</td>
<td>260</td>
</tr>
<tr>
<td>combinatorial optimization</td>
<td>407</td>
</tr>
<tr>
<td>Common Data Format (CDF)</td>
<td>28, 107</td>
</tr>
<tr>
<td>common variates</td>
<td>16</td>
</tr>
<tr>
<td>companion matrix</td>
<td>310</td>
</tr>
<tr>
<td>compartmental model</td>
<td>510</td>
</tr>
<tr>
<td>complete pivoting</td>
<td>282</td>
</tr>
<tr>
<td>complete space</td>
<td>549</td>
</tr>
<tr>
<td>completeness, of system of functions</td>
<td>465</td>
</tr>
<tr>
<td>complex data type</td>
<td>39, 40, 51, 321</td>
</tr>
<tr>
<td>computational fluid dynamics</td>
<td>506</td>
</tr>
<tr>
<td>concave function</td>
<td>350</td>
</tr>
<tr>
<td>conceptual clustering</td>
<td>632</td>
</tr>
<tr>
<td>concavity</td>
<td>543</td>
</tr>
<tr>
<td>condition (problem or data)</td>
<td>57</td>
</tr>
<tr>
<td>condition number</td>
<td>57, 59,264, 299,308, 370</td>
</tr>
<tr>
<td>condition number of a function with</td>
<td></td>
</tr>
<tr>
<td>respect to finding a root</td>
<td>370</td>
</tr>
<tr>
<td>condition number with respect to</td>
<td></td>
</tr>
<tr>
<td>computing a sample standard deviation</td>
<td>59</td>
</tr>
<tr>
<td>condition number with respect to</td>
<td></td>
</tr>
<tr>
<td>inversion</td>
<td>264, 299</td>
</tr>
<tr>
<td>Condor (software system)</td>
<td>84, 429</td>
</tr>
<tr>
<td>congruential random number generator</td>
<td>122</td>
</tr>
<tr>
<td>conjugate gradient method</td>
<td>297, 391, 396</td>
</tr>
<tr>
<td>conjugate transpose</td>
<td>236, 248</td>
</tr>
<tr>
<td>conjugate vectors</td>
<td>248</td>
</tr>
<tr>
<td>consistency property for matrix norms</td>
<td>259</td>
</tr>
<tr>
<td>consistency test for software</td>
<td>101, 341</td>
</tr>
<tr>
<td>consistent system of equations</td>
<td>246, 372</td>
</tr>
<tr>
<td>constrained sampling</td>
<td>16</td>
</tr>
<tr>
<td>container hull</td>
<td>580</td>
</tr>
<tr>
<td>convergence criterion</td>
<td>64, 359</td>
</tr>
<tr>
<td>convergence ratio</td>
<td>64</td>
</tr>
<tr>
<td>convex function</td>
<td>350</td>
</tr>
<tr>
<td>convex hull</td>
<td>580</td>
</tr>
<tr>
<td>convex hull peeling</td>
<td>580</td>
</tr>
<tr>
<td>convolution</td>
<td>550</td>
</tr>
<tr>
<td>convolution theorem</td>
<td>556</td>
</tr>
<tr>
<td>correlation</td>
<td>568</td>
</tr>
<tr>
<td>correlation matrix</td>
<td>569</td>
</tr>
<tr>
<td>correlation theorem</td>
<td>556</td>
</tr>
<tr>
<td>covariance</td>
<td>568</td>
</tr>
<tr>
<td>Cplex (optimization software)</td>
<td>437</td>
</tr>
<tr>
<td>CPU</td>
<td>76</td>
</tr>
<tr>
<td>craps test</td>
<td>162</td>
</tr>
<tr>
<td>cross product</td>
<td>637</td>
</tr>
<tr>
<td>cross product of vectors</td>
<td>233</td>
</tr>
<tr>
<td>cross products, computing sum</td>
<td></td>
</tr>
<tr>
<td>Exercise 3.17c</td>
<td>113</td>
</tr>
<tr>
<td>Crout method</td>
<td>284</td>
</tr>
<tr>
<td>crude Monte Carlo</td>
<td>8, 456</td>
</tr>
<tr>
<td>cryptography</td>
<td>119</td>
</tr>
<tr>
<td>cumulative distribution function</td>
<td>636</td>
</tr>
<tr>
<td>curse of dimensionality</td>
<td>66, 457</td>
</tr>
<tr>
<td>cycle length of random number generator</td>
<td>118, 123, 134</td>
</tr>
<tr>
<td>data augmentation</td>
<td>193</td>
</tr>
<tr>
<td>data depth</td>
<td>581</td>
</tr>
<tr>
<td>data mining</td>
<td>73</td>
</tr>
<tr>
<td>data standards</td>
<td>88</td>
</tr>
<tr>
<td>data structure</td>
<td>103</td>
</tr>
<tr>
<td>data vector</td>
<td>562</td>
</tr>
<tr>
<td>data warehousing</td>
<td>73</td>
</tr>
<tr>
<td>data-based random number generation</td>
<td>217</td>
</tr>
<tr>
<td>database, scientific</td>
<td>103</td>
</tr>
<tr>
<td>dataframe</td>
<td>103</td>
</tr>
</tbody>
</table>
defective matrix, 253
deficient matrix, 253
Delaunay triangulation, 629
depth median, 582
depth of data, 581
derivative with respect to a vector or
matrix, 267
det(·), 238
determinant of a matrix, 238, 257, 308
DFP (method), 389
diag(·), 237, 245
diagonal factorization, 253
diagonalization, 253
DIEHARD tests for random number
generators, 160, 219
differentiation of vectors and matrices,
267
differentiation, symbolic, 437
Diffpack, 506
diffusion equation, 491
Dijkstra’s algorithm, 577
dimension of vector space, 229
dimension reduction, 72, 566, 628
Dirac delta function, 639
direct method, linear systems, 279, 310
direct product, 637
direct sum, 230
direction cosines, 232, 273
directional data, 573
Dirichlet boundary condition, 492
Dirichlet tessellation, 628
discrepancy, 157, 166
discrete Legendre polynomials, 468, 558
discrete uniform distribution, generating
random deviates, 174
discretization error, 57, 65
discriminant analysis, 611
discrimination, 615
dissimilarity measure, 573
distributed linear algebra machine, 338
divide and conquer, 62, 69, 86
DNA test for random numbers, 161
domain decomposition, 495
dominant eigenvalue, 311
Doolittle method, 284
dot product, 547
dot product of matrices, 241
dot product of vectors, 230, 242
double exponential distribution,
generating random deviates, 205
double precision, 37, 42
dud method for least squares, 401
dud secant for least squares, 401
ECDF (empirical cumulative distribu-
tion function), 159, 636
EDA, 73
Edelstein-Keshet, Leah, 508, 509
Edgeworth expansion, 462
Edgeworth series, 472
efficient point, 427
eigenfunction, 546
eigenvalue, 251, 260, 310, 546
eigenvalue of a polynomial Exercise 5.8;
342
eigenvector, 251, 546
EISPACK, 325
elemental subset, 542
elementary operator matrix, 280
ellipsoid, minimum volume, 581
elliptic norm, 259
elliptically contoured distribution, 211
EM algorithm, 423
empirical cumulative distribution
function, 159, 636
empirical test, 158
encapsulation, 94
Erlang distribution, generating random
deviates, 207
error bound, 9, 55
error of approximation, 56
error, absolute, 47, 53, 274
error, cancellation, 50, 58
error, discretization, 57
error, measures of, 55, 56, 300
error, relative, 47, 53, 275
error, rounding, 50, 53, 54
error, rounding, models of 54
error, rounding, models of Exercise 3.9;
111
error, truncation, 57
error-free computations, 50
errors-in-variables, 534
Euclidean distance, 570
Euclidean matrix norm, 261
Euclidean vector norm, 258
Euler’s constant Exercise 3.2; 110
Euler’s method, 485
even function, 472
exact computations, 50
exact sampling of Markov chain stationary distribution, 190
exception, in computer operations, 46, 49
expert system, 91
exploratory data analysis, 73
exponent, 32
exponential distribution, generating random deviates, 204
exponential order, 61
exponential power distribution, generating random deviates, 205
exponential smoothing, 542
extended precision, 37
extrapolation, 65, 451
factor analysis, 600
fan-in algorithm, 48, 63, 86
fast Givens rotation, 293
fathoming, 407
Faure sequence, 166, 167
feasible point, 357
feasible region, 357
feature space, 611
feedback shift register generator, 141
Fibonacci random number generator, 139
Fickian diffusion, 510
fixed-point method, 360
fixed-point representation, 31, 122
flat, 233, 586
floating-point representation, 31
FLOP, or flop, 62
FLOPS, 76
FLOPS, or flops, 62
Fokker-Planck diffusion, 510
folded distribution, 199
Fortran, 321, 326, 430
Fortran 2000, 90, 321, 329
Fortran 90, 90, 321
Fortran 95, 90, 321
Fortran 95, 40, 62, 94, 213
Fortran D, 90
Fourier coefficients, 464
Fourier transform, 553, 555
fractal dimension, 609
Frobenius norm, 261
FSQP (optimization software), 431
FSQP (software), 423
full precision, 42
full rank, 242
function space, 546
functional, 459
fuzzy clustering, 624
gamma distribution, generating random deviates, 206
gamma function, 642
gamma weight function, 470
GAMS (modeling optimization system), 437, 439
GAMS (Guide to Available Mathematical Software), 89, 215, 320, 474, 497, 651
GAMS, electronic access, 651
Gauss (software), 330
Gauss-Newton method, 393, 401
Gauss-Seidel method, 295
Gaussian elimination, 279, 317
Gaussian quadrature, 451
GenBank, 107
generalized gamma distributions, 208
generalized inverse of a matrix, 247, 288
generating set, 230
 genetic algorithm, 98, 412
 genetic programming, 98
goodness-of-fit test, 158, 159
golden section search, 382
Goldstein-Armijo method, 383
goodness-of-fit test, 158, 159
graceful underflow, 35
gradient of a function, 267, 354, 372
GrADS (Grid Analysis and Display System), 105
gradual underflow, 35, 49
Gram matrix, 304
Gram-Charlier series, 472
Gram-Schmidt procedure, 294
Gram-Schmidt transformation, 262, 294, 463, 466
Gramian, 304
grand tour (in graphics), 579
graph, 564
Gray code, 168, 170
greedy algorithm, 63, 384
GRG2 (optimization software), 431
Grid Analysis and Display System (GrADS), 105
grid, computational, 84
growth models, 510
GSL (GNU Scientific Library), 214
guard digit, 47
Hadamard multiplication, 240
half precision, 42
halfnormal distribution, generating random deviates, 203
halfspace location depth, 581
Halton sequence, 166
Hamiltonian circuit, 579
Hamming distance, 571
Hamming weight, 124
Hastings method, 186
hat matrix, 250
HDF (Hierarchical Data Format), 28, 88, 104
headbanging, 544
heap, 566
heat equation, 491
Heaviside function, 638
Hemes formula, 303
Hermite function, 471
Hermite polynomial, 470, 478, 483
Hermite’s equation, 478, 483
Hermitian matrix, 237
Hessenberg matrix, 316
Hessian, 356
heterogeneous network computing, 83
hidden bit, 33
hierarchical clustering, 618
Hierarchical Data Format (HDF), 28
Hilbert curve, 584
Hilbert matrix, 339
Hilbert space, 549
hit-and-run method, 193, 210
hit-or-miss Monte Carlo, 8, 15, 21, 179, 456
Hodges-Lehmann estimator, 70
homogeneous coordinates, 273
Horner’s method, 67
HotBot (Web search engine), 652
Hotelling transform, 593
Householder transformation (reflection), 288, 289, 317
HPF, 90
hull, 580
hybrid splines, 536
hypercube (PPU topology), 81
hyperplane, 233
idempotent matrix, 243
identity matrix, 240
IDL (software), 330
IEEE standards, 29, 37, 43, 49, 88
ill-conditioned (problem or data), 57
ill-conditioned data, 57, 262
Illinois method, 367
image data, 29
importance sampling, 14, 21, 454
improper integral, 457
IMSL Libraries, 89, 214, 216, 326, 430
incomplete gamma function, 642
independence sampler, 188
independent streams of random numbers, 152
indicator function, 638
induced matrix norm, 259
infinity, floating-point representation, 38, 49
infix operator, 50
informatics, 107
inheritance, 94
initial value problem, 484
inner product, 230, 284, 547
inner product of matrices, 241
inner product space, 231, 547
inner pseudoinverse, 247
insertion sort, 68
instrumental density, 178
integer programming, 424
integer representation, 31
integrating factor, 481
integration of vectors and matrices, 270
interior-point method, 424
Internet, 650
interpolation, 520
interval arithmetic, 51, 52
invariance property, 270
inverse CDF method, 172
inverse CDF method for truncated distributions, 198
inverse chi-squared distribution, 198
inverse gamma distribution, 198
inverse of a matrix, 243
inverse problem, 519
inverse Wishart distribution, 198
IRLS (iteratively reweighted least squares), 395, 396
Ising model, 18
ISO (standards), 39, 88
isometric matrix, 261, 572
isometric transformation, 271, 572, 575
isotropic transformation, 271
iterative method, 64
iterative method, linear systems, 279, 295, 310
iterative method, random number generation, 183, 192
iterative refinement, 301
iteratively reweighted least squares, 395, 396
Jacobi method for eigenvalues, 310, 313
Jacobi polynomial, 469, 537
Jacobi transformation (rotation), 291
Jacobian, 268, 372
Java (programming language), 102
Jensen’s inequality, 351, 375
K-means clustering, 617, 631
Karhunen-Loève transform, 593
Karmarkar algorithm, 424
Karush-Kuhn-Tucker conditions, 422
KDD (knowledge discovery in databases), 73
kernel (function), 543
kernel function, 611
kernel smoother, 543
Kiefer-Wolfowitz procedure, 404
kind (for data types), 40
knowledge discovery in databases (KDD), 73
kolmogorov distance, 159
kolmogorov-Smirnov test, 158, 159
kriging, 544
Kronecker multiplication, 240
Krylov space, 298
lagged Fibonacci generator, 139, 140
Lagrange multiplier, 420
Lagrange polynomial, 521
Lagrangian function, 420
Laguerre polynomial, 470
Laguerre-Fourier index, projection pursuit, 610
LANCELOT, (optimization software), 431
Lanczos method, 298
LAPACK, 228, 283, 309, 326
Laplace approximation, 458
Laplace distribution, generating random deviates, 205
Laplace equation, 491
Laplace transform, 553
Laplacian operator, 640
latent semantic indexing, 599
lattice test for random number generators, 130, 156
leading principal submatrix, 244
leaped Halton sequence, 167
leapfrogging, in random number generation, 132, 145, 154
learning, 611
least squares, 286, 304, 522
least squares regression, 391, 401
Lebesgue space-filling curve, 586
Legendre polynomial, 468, 483
Legendre’s differential equation, 468, 482
Legendre’s equation, 482
Lehmer congruential random number generator, 122
Lehmer sequence, 122
Lehmer tree, 152
length of a vector, 228, 232, 258
Levenberg-Marquardt algorithm, 394
limited-memory quasi-Newton method, 390
Lindo (optimization software), 437
line search, 381
linear congruential random number generator, 122
linear convergence, 64
linear independence, 229, 546
linear programming, 358, 423
linear programming software, 436
linear transform, 552
lining, 578
LINPACK, 228, 283, 309, 326
Linpack benchmarks, 102
Linux (operating system), 84
Lisp-Stat (software), 330
little endian, 43
little o (order), 56
little omega (order), 56
loess, 542
log concave function, 351
log convexity, 351
log order, 61
lognormal distribution, generating random deviates, 203
loop unrolling, 323
Lotka-Volterra differential equations, 507, 510
lowess, 542
LSGRG2 (optimization software), 431
M(RT)^2 algorithm, 20
MACHAR, 37
MACHAR Exercise 3.3d; 111
machine epsilon, 35, 387
machine learning, 611
Mahalanobis distance, 572, 616
majorizing density, 178
Manhattan distance, 570
Manhattan norm, 258
manifold of a matrix, 235
Maple (software), 51, 321, 497, 504
Markov chain, 183
Markov chain Monte Carlo, 188, 191, 192
MARS (multivariate adaptive regression splines), 536, 538
massive dataset, 72
massively parallel processing, 79
Mathematica (software), 497, 504
mathematical programming problem, 357
Matlab (software), 89, 214, 331
matrix, 234
matrix congruential generator, 140
matrix derivative, 267
matrix factorization, 253, 283
matrix inverse, 243
matrix multiplication, 239, 244, 276
matrix of type 2, 236
matrix polynomial Exercise 5.8.; 342
matrix storage mode, 326
Matrix Template Library, 330
max norm, 258
maximum difference, 570
MCMC (Markov chain Monte Carlo), 188, 191, 192
mean squared error, 540
mean squared error, of series expansion, 464
median, 70
median polish, 543
memory, 76
Mersenne prime, 123
mesh, 539
mesh (PPU topology), 80
message passing, 82, 93, 326
Message Passing Library, 93, 326
metadata, 104
metric, 568
Metropolis algorithm, 20
Metropolis-Hastings method, 186, 192
MIL-STD-1753 standard, 42
MIMD, 78
minimal spanning tree, 576
minimax approximation, 473
minimum distance test, 161
minimum volume ellipsoid, 581
Minkowski inequality, 258
Minkowski norm, 258
Minkowski reduced basis, 156
MINOS (optimization software), 437
Minskowski distance, 571
MIPS, 76
missing value, representation of, 38, 104
mixed integer programming, 424
mixture distribution, 176
mobile Jacobi scheme, 315
modified Gauss-Newton algorithm, 393
modified Gram-Schmidt procedure, 294
modified midpoint method, 487
modular arithmetic, 120
moment, 549
Monte Carlo evaluation of an integral, 7, 452
Moore-Penrose inverse, 247, 287, 288, 305
moving average, 541
MPI (message passing interface), 82, 84, 93, 326, 339
MPI (Message Passing Library), 93, 326
MPS format, 437, 438
MR classification system, 649
MSE (mean squared error), 540
MST (minimal spanning tree), 576
MT19937 (generator), 145
MTOPS, 76
multigrid method, 299, 495
multinomial distribution, generating random deviates, 212
multiple precision, 51
multiple recursive random number generator, 139, 141
multiple roots of a function, 369
multiple shooting method, 490
multiplicative congruential random number generator, 122
multipolar mapping, 628
multivariate adaptive regression splines, 536, 538
multivariate distributions, generating random deviates, 210
multivariate normal distribution, generating random deviates, 211
Nag Libraries, 89, 326, 430
NaN (“not-a-number”), 38, 49, 104
Nastran, 505
nearest neighbor, 568, 583
nearest neighbors, 539, 541, 583, 584
Nelder-Mead simplex method, 397, 417
NEOS, 429
NetCDF, 105
network computing, 83
Neumann boundary condition, 492
neural net, 417
Newton’s method, 362, 372, 384
Newton-Cotes formula, 449
Newton-Raphson, 384
Niederreiter sequence, 166, 170
NIST Test Suite, for random number generators, 162
NL2SOL (optimization software), 431
NLPQL (optimization software), 431
NLPQL (software), 422
noisy function optimization, 400, 411, 417
noncentral gamma distribution, evaluating probabilities Exercise 8.6, 478
nonlinear regression, 391, 395, 401
nonnegative definite matrix, 256, 285
nonnegative matrix factorization, 295, 600
nonparametric regression, 538
nonparametric smoothing, 525
nonsingular matrix, 242
norm of a functional, 460
norm of an operator, 460
norm, function, 8, 548
norm, vector and matrix, 257–259, 543
normal (normalized) vector, 233
normal distribution, generating random deviates, 200
normal equations, 286, 304
normal function, 549
normalized floating-point numbers, 33
normalized vector, 233
Northern Light (Web search engine), 652
not-a-number (“NaN”), 38
NP-complete problem, 61
NPSOL (optimization software), 431
null space, 248
nullcline, 490

Object Constraint Language (OCL), 97
object, software, 93, 103
object-oriented software, 93
oblique partitioning, 73
Occam’s razor, 522
Octave (software), 89, 334
odd function, 472
one-way function, 119
online algorithm, 67, 74
online processing, 67
open software, 88
open source software, 88, 89
OpenMP, 90
Profiler, software, 102
Program Development Language, 97
projection matrix, 250
projection pursuit, 130, 604
projective transformation, 271
proximity search, 568, 628
PSBLAS (parallel sparse BLAS), 326
projection matrix, 250
projection pursuit, 130, 604
projective transformation, 271
proximity search, 568, 628
PSBLAS (parallel sparse BLAS), 326
PVM (Parallel Virtual Machine), 83, 84
Q-convergence, 65
quadratic convergence, 65
quadratic form, 242, 259
quadratic programming, 422, 437
quadratic programming software, 436
quadratic Shepard method, 538
quadrature, 448
quasi Monte Carlo method, 457
quasi-Monte Carlo method, 166
quasi-Newton method, 388
quasirandom sequence, 119, 166
quicksort, 69
random number generation, Chapter 4; 117
random number generator, congruential, 122
random number generator, feedback shift method, 141
random number generator, parallel, 152
random number generator, testing, 157
range of a matrix, 235
rank correlation, 569
rank determination, 307
rank of a matrix, 242, 288, 308
rank of an array, 234
rank reduction, 308
rank transformation, 569
rank(-), 242
rank, linear independence, 242, 307
rank, number of dimensions, 234
rank-one update, 289, 302
rank-revealing QR, 288, 308
rank-two update, 389
raster image, 106
rate constant, 64
rate of convergence, 64
ratio-of-uniforms method, for random number generation, 206
rational approximation, 461
real numbers, 31
real-time algorithm, 67
rectangle/wedge/tail method, for random number generation, 202, 204
recursion, 66
recursion formula for orthogonal polynomials, 466
recursive partitioning, 73, 617, 623
reduced gradient methods, 431
reflection, 289
register, in computer processor, 48, 76
regression, nonlinear, 391, 395, 401
regular matrix, 253
relative error, 47, 53, 275
relative spacing, 35
Remez algorithm, 461
renewal, simulation experiments, 152
residue, 121
residue arithmetic, 50
Richardson extrapolation, 66, 451, 487
ridge regression, 393
Ripley, Brian D., 622
RISC, 76
Robbins-Munro procedure, 369, 374, 417
robust covariance matrix, 575
robust method, 575
robustness (algorithm or software), 58, 92
Romberg quadrature, 451
root of a function, 49, 359
root-free Cholesky, 285
Rosenbrock function, 440
Rosser test matrix, 340
rotation, 271, 291
roughness of a function, 8
rounding error, 50, 54
row rank, 242
row-major, 275, 320, 322
row-sum norm, 260
running line smoother, 541
running median, 541, 543
runs test, 162, 163
S-Plus (software), 89, 103, 214, 219, 334
saddlepoint approximation, 458
sample variance, computing, 58
SAS optimization software, 437
scalable (parallel) program, 87
scalable parallel processing, 79
ScaLAPACK, 93, 326, 338
scalar product, 230
scaling of a vector or matrix, 266
scaling of an algorithm, 61
Schrödinger’s equation, 483
Schur complement, 246
scree plot, 593
seacant method, 366
See5, 73
seed, 123, 131–133, 153, 213, 216, 218, 221
self-organizing maps, 622
self-validating method, 101
seminorm, 258
sequential quadratic programming, 422, 431
sequential unconstrained minimization techniques, 422
setup time, 195
shape of matrix, 234
shapefile, 107
shearing transformation, 271
Shepard method for interpolation, 538
Sherman-Morrison formula, 302, 390
shooting method, 489
shuffled random number generator, 134, 149
side effect, 215, 324
Sierpiński space-filling curve, 586
sign bit, 30
significand, 32
signum function, 644
SIMD, 78
similar canonical form, 253
similar matrices, 255
similarity measure, 568
similarity transformation, 255, 313, 317
SIMP, 78
simple matrix, 253
simplex, 350
simplex algorithm, linear programming, 423
simplex method, 397, 417
simplicial location depth, 582
Simpson’s rule, 449
Simula-67 (software), 94
simulated annealing, 183, 407
simulation, 117, 189
simultaneous perturbation stochastic approximation, 404, 426
single precision, 37, 42
singular matrix, 242
singular value, 255, 295, 308, 596
singular value decomposition, 255, 294, 308, 318, 596
skew-normal distribution, 199
skewed distribution, 199
SLATEC (software), 89
SmallTalk (software), 94
smooth (noun), 539
smoothing, 525, 538
smoothing parameter, 539
SMP (symmetric multiprocessing), 79
Sobol’ sequence, 166, 167, 170, 458
software development, 95
software engineering, 88, 95, 215
software parts, 92
software reuse, 87, 92, 93
software standards, 88
software testing, 98, 339
SOR (method), 296
sorting, 68
sorting of multivariate data, 71, 576
space-filling curve, 584
spanning set, 230
sparse matrix, 275, 299, 321, 328, 338
spectral decomposition, 254
spectral norm, 260
spectral radius, 260, 296
spectral test for random number generators, 130, 155
spectrum of a matrix, 252
sphered data, 567, 572, 610
spline, 536
splitting extrapolation, 66
SPRNG, software for parallel random number generation, 154
SQP (software), 423
square root, matrix, 285
squeeze test, 162
stability, 58, 282, 370
standard deviation, computing, 58
standard distribution, 330
standardization, multivariate (sphering), 567
standardization, univariate, 567
standardizing data, 567
standards, 88
statistical learning, 611
Statistical Reference Datasets (StRD), 101
steepest descent, 384
step length factor, 381
stiff data, 60
stochastic approximation, 368, 374, 403, 417
stochastic differential equation, 495
stochastic integration, 452
stopping criterion, 64, 359
storage mode, for matrices, 326
storage unit, 29, 32, 43
Strassen algorithm, 278
stratified sampling, 14, 454
StRD (Statistical Reference Datasets), 101
stride, 275, 320, 324
string, character, 28
submatrix, 244
substitution sampling, 193
successive overrelaxation, 296
successive quadratic programming, 423
sum of vector spaces, 230
SUMT, 422
Super-Duper (generator), 149
superlinear convergence, 64
support vector, 463
SVD (singular value decomposition), 255, 294, 308, 318, 596
Swendsen-Wang algorithm, 20
swindle, Monte Carlo, 12
symbolic differentiation, 437
symmetric matrix, 237
symmetric multiprocessing, 79
tabu search, 416
Tausworthe random number generator, 141
Template Numerical Toolkit, 330
tensor, 234
tensor product, 552
tessellation, 628
testbed, 100, 339
testing random number generators, 157
testing software, 98, 339
TIGER file, 106
timing program execution, 102
Toeplitz matrix, 236, 347
torus (PPU topology), 81
total least squares, 531
trace of a matrix, 237
trace(·), 237
transcendental numbers as a source of random numbers, 146
transform, 552
translation transformation, 273
trapezoid rule, 448
tree, 564, 565
truncated distribution, 198
truncated Newton method, 390
truncated normal distribution, 203, 212
truncation error, 57
trust region, 386, 394
twicing, 543
twisted GSFR generator, 145
twoing rule, 623
twos-complement representation, 30, 46, 122
type 2 matrix, 236
ulp (“unit in the last place”), 36
UML (Unified Modeling Language), 97
underdetermined linear system, 247
underflow, in computer operations, 35, 49
undetermined coefficients method, 482
Unicode, 28, 88
Unified Modeling Language, 97
uniform norm, 548
uniform time algorithm, 196
unit in the last place, 36
unit roundoff, 35
unit vector, 240
unit vectors, 232
unitary matrix, 248
unpredictable, 119
unrolling do-loop, 323
updating a solution, 302, 306
upper Hessenberg form, 316

van der Corput sequence, 166
Vandermonde matrix, 251
variable metric method, 388
variance estimation, 11
variance reduction, 12, 133
variance, computing, 58
variance-covariance matrix, 569
vec(-), 237
vech(-), 237
vector, 228
vector image, 106
vector processing, 77, 338
vector space, 229, 235, 248
VERI clustering, 632
virtual machine, 83
visual information retrieval, 108
Voronoi diagram, 629
Voronoi tessellation, 628

Ware’s law, 63, 85
wave equation, 492
wavelet, 556
Wavelet transform, 555
weight window, 15
weighted least squares, 395
weighted norm, 259
weighted regression, 531
well-posed problem, 492
Wichmann/Hill random number generator, 150, 225
Wiener-Khinchin theorem, 556
Wilkinson matrix, 340
window width, 539
Windows Metafile (WMF), 29
Windows 2000, 95
WMF (Windows Metafile), 29
Wolff algorithm, 20
Woodbury formula, 303, 390
word, computer, 29, 32, 43

XDR (external data representation), 44
Yahoo (Web search engine), 652

zero of a function, 49, 359
zero of a polynomial, 359
zigurat method, for random number generation, 203