BIO 248 cd
Advanced Statistical Computing

Course Notes

by

Robert Gray

Contents

1 Computer Arithmetic **6**
- 1.1 Integers . . . 6
- 1.2 Floating Point . . . 7
- 1.3 Machine Constants . . . 8
- 1.4 Accuracy of Arithmetic Operations . . . 11
- 1.5 Computing Sums . . . 14
- 1.6 Computing Sample Variances . . . 15
- 1.7 Error Analysis for Variance Algorithms . . . 19
- 1.8 References . . . 20

2 Numerical Linear Algebra **22**
- 2.1 Matrix Multiplication . . . 22
- 2.2 Systems of Linear Equations . . . 24
  - 2.2.1 Triangular Systems . . . 25
  - 2.2.2 Gaussian Elimination and the LU Decomposition . . . 26
  - 2.2.3 The Choleski Decomposition . . . 33
- 2.3 Matrix Norms, Condition Numbers, and Error Analysis . . . 37
- 2.4 Rotations and Orthogonal Matrices . . . 43
- 2.5 Linear Least Squares . . . 45
- 2.6 QR Decomposition . . . 46
  - 2.6.1 Constant Terms . . . 48
  - 2.6.2 Variances, Sums of Squares, and Leverage Values . . . 48
  - 2.6.3 Updating Models . . . 52
  - 2.6.4 Accuracy of Least Squares Estimators . . . 53
  - 2.6.5 Computational Efficiency of QR . . . 54
- 2.7 Singular-Value Decomposition . . . 54
  - 2.7.1 Computing SVDs . . . 56
  - 2.7.2 Solving Linear Systems . . . 56
  - 2.7.3 SVD and Least Squares . . . 60
  - 2.7.4 Some timing comparisons . . . 62
- 2.8 Some Iterative Methods . . . 63
- 2.9 Nonparametric Smoothing Splines . . . 64
  - 2.9.1 Splines . . . 65
  - 2.9.2 Solution to the Penalized Least Squares Problem . . . 68
  - 2.9.3 Cross Validation and Smoothing Parameters . . . 73
  - 2.9.4 Central Diagonals of the Inverse of a Band Symmetric PD Matrix . . . 76

2.10 Additional Exercises . . . 78
2.11 References . . . 81

3 Optimization and Nonlinear Equations 83
3.1 One-Dimensional Problems . . . 83
3.1.1 Solving a Single Nonlinear Equation . . . 83
3.1.2 Rates of Convergence . . . 86
3.1.3 One-Dimensional Minimization . . . 86
3.2 The Newton-Raphson Optimization Algorithm . . . 89
3.2.1 The Problem . . . 90
3.2.2 The Basic Algorithm . . . 91
3.2.3 Backtracking . . . 93
3.2.4 Positive Definite Modifications to the Hessian . . . 95
3.2.5 The Function nr() . . . 102
3.2.6 Example: Weibull Regression . . . 104
3.2.7 Computing Derivatives . . . 108
3.3 The BFGS Algorithm . . . 111
3.3.1 Rank 1 Updates . . . 120
3.4 The Nelder-Mead Simplex Method . . . 121
3.5 A Neural Network Classification Model . . . 125
3.5.1 Application to Fisher's Iris Data . . . 133
3.6 Newton's Method for Solving Nonlinear Equations . . . 136
3.6.1 Example: Estimating Equations with Missing Covariate Data . . . 139
3.7 Nonlinear Gauss-Seidel Iteration . . . 145
3.8 Exercises . . . 147
3.9 References . . . 148

4 The EM Algorithm 150
4.1 Introduction . . . 150
4.2 Some General Properties . . . 161
4.3 An Accelerated EM Algorithm . . . 163
4.4 Calculating the Information . . . 165
4.4.1 Louis' Formula . . . 166
4.4.2 The SEM Algorithm . . . 166
4.5 The ECM Algorithm . . . 174
4.6 Comments . . . 176
4.7 Exercises . . . 177
4.8 References . . . 178

5 Laplace Approximations 180
5.1 Introduction . . . 180
5.2 Definition and Theory . . . 182
5.3 Examples . . . 187
5.4 Exercises . . . 197
5.5 References . . . 199

6 Quadrature Methods 200

6.1 Newton-Cotes Rules . . . 200
6.1.1 Romberg Integration . . . 203
6.1.2 Singularities . . . 206
6.2 Gaussian Quadrature . . . 211
6.2.1 Gauss-Hermite Quadrature . . . 219
6.3 Multi-dimensional Integrals . . . 223
6.3.1 Gauss-Hermite Quadrature . . . 224
6.4 Exercises . . . 227
6.5 Appendix: Gaussian Quadrature Functions . . . 229
6.6 References . . . 231

7 Basic Simulation Methodology 232
7.1 Generating Pseudo-Random Numbers . . . 232
7.1.1 Uniform Deviates . . . 232
7.1.2 Transformation Methods . . . 240
7.1.3 Rejection Sampling . . . 242
7.1.4 Testing Random Number Generators . . . 248
7.1.5 Splus . . . 249
7.2 Statistical Simulations . . . 249
7.3 Improving Efficiency . . . 256
7.3.1 Control Variates . . . 256
7.3.2 Importance Sampling . . . 259
7.3.3 Antithetic Sampling . . . 263
7.4 Exercises . . . 266
7.5 References . . . 267

8 More on Importance Sampling 269
8.1 Introduction . . . 269
8.2 Importance Sampling in Bayesian Inference . . . 269
8.3 Bayesian Analysis For Logistic Regression . . . 271
8.4 Exercises . . . 277

9 Markov Chain Monte Carlo 279
9.1 Markov Chains . . . 279
9.2 The Metropolis-Hastings Algorithm . . . 281
9.2.1 Random Walk Chain . . . 283
9.2.2 Independence Chain . . . 283
9.2.3 Rejection Sampling Chain . . . 283
9.3 Block-at-a-Time Algorithms . . . 284
9.3.1 Gibbs Sampling . . . 285
9.4 Implementation Issues . . . 285
9.4.1 Precision of Estimates . . . 286
9.5 One-way Random Effects Example . . . 287
9.6 Logistic Regression Example . . . 293
9.6.1 Independence Chain Sampling . . . 294
9.7 Rejection Chain Sampling . . . 296
9.8 Exercises . . . 298

9.9 References 299

10 Bootstrap Methods **300**
10.1 Variance Estimation 300
10.1.1 Regression Models 306
10.1.2 How many resamples? 308
10.2 Bootstrap Bias Correction 309
10.3 Bootstrap Hypothesis Tests 312
10.3.1 Importance Sampling for Nonparametric Bootstrap Tail Probabilities 314
10.3.2 Antithetic Bootstrap Resampling 318
10.4 Bootstrap Confidence Intervals 319
10.5 Iterated Bootstrap 323
10.6 Exercises 328
10.7 References 329

Chapter 1

Computer Arithmetic

Because of the limitations of finite binary storage, computers do not store exact representations of most numbers or perform exact arithmetic. A standard computer will use 32 bits of storage for an integer or for a single precision floating point number, and 64 bits for a double precision number.

1.1 Integers

For many computers the 32 bits of a stored integer u can be thought of as the binary coefficients x_i in the representation

$$u = \sum_{i=1}^{32} x_i 2^{i-1} - 2^{31},$$

where each $x_i = 0$ or 1. (This may not be the exact way the values are stored, but this model gives the same results as the most common representation.) In this representation, if $x_{32} = 1$ and all the other $x_i = 0$, then $u = 0$. The largest possible integer (all $x_i = 1$) is $\sum_{i=1}^{31} 2^{i-1} = 2147483647$, and the largest (in magnitude) negative integer is $-2^{31} = -2147483648$, which has all $x_i = 0$. When results of an integer arithmetic operation go beyond this range, the result is usually the lower order bits from the representation of the exact result. That is, $2147483647 + 1 = 2^{32} - 2^{31}$, so its representation would have $x_i = 0$ for $i = 1, \ldots, 32$, and $x_{33} = 1$, if there were an x_{33}. Since there is no 33rd bit, only bits 1 to 32 are stored (all zeros), and the result is $-2^{31} = -2147483648$. This can easily be seen in any compiled program that does integer arithmetic. For example, in Splus (where it takes quite a bit of work to coerce integer arithmetic):

```
> u <- as.integer(0)
> b <- as.integer(1)
> two <- as.integer(2)
> for (i in 1:31) {u <- u+b; b <- b*two}
> u
[1] 2147483647
> u+as.integer(1)
```

```
[1] -2147483648
```

It is possible to get integer results unintentionally, though:

```
> a <- rep(c('a','b','c'),1000)
> b <- table(a)
> b
    a    b    c
 1000 1000 1000
> b[1]*b[2]*b[3]
          a
 1000000000
> b[1]*b[2]*b[3]*b[1]
          a
 -727379968
```

The output from the table command is stored in integer mode, and multiplication of integers produces an integer result.

1.2 Floating Point

A floating point number can be thought of as being represented in the form

$$(-1)^{x_0}(\sum_{i=1}^{t} x_i 2^{-i})2^k,$$

where k is an integer called the exponent and $x_i = 0$ or 1 for $i = 1, \ldots, t$. x_0 is the sign bit, with $x_0 = 0$ for positive numbers and $x_0 = 1$ for negative. The fractional part $\sum_{i=1}^{t} x_i 2^{-i}$ is called the mantissa. By shifting the digits in the mantissa and making a corresponding change in the exponent, it can be seen that this representation is not unique. By convention, the exponent is chosen so that the first digit of the mantissa is 1, except if that would result in the exponent being out of range.

A 32-bit single precision floating point number is usually represented as a sign bit, a 23 bit mantissa, and an 8 bit exponent. The exponent is usually shifted so that it takes the values -126 to 128. The remaining possible value of the exponent is probably reserved for a flag for a special value (eg underflow, overflow [which sometimes prints as `Inf`] or `NaN` [not a number]). When the exponent is in the allowed range, the first digit in the mantissa is always a 1, so it need not be stored. When the exponent is at the minimum value (eg -126), the first digit does need to be given, so there has been a loss of at least one digit of accuracy in the mantissa, and the program may report that underflow has occurred. The IEEE standard calls for underflow to occur gradually. That is 2^{-130} could be represented with $k = -126$, $x_4 = 1$ and the other $x_i = 0$. Note that $2^{-130} + 2^{-152}$ would be stored the same as 2^{-130}, since there are not enough digits in the mantissa to include the additional term. It is this loss of accuracy that is called underflow (note that 2^{-152} would be stored as 0).

(Note that since the above representation includes both a positive and negative 0, which is not necessary, one additional value could also be included in the range of representable values.)

A standard double precision representation uses a sign bit, an 11 bit exponent, and 52 bits for the mantissa. With 11 bits there are $2^{11} = 2048$ possible values for the exponent, which are usually shifted so that the allowed values range from -1022 to 1024, again with one special value.

The IEEE standard for floating point arithmetic calls for basic arithmetic operations to be performed to higher precision, and then rounded to the nearest representable floating point number. Sun SPARC architecture computers appear to be compliant with this standard.

1.3 Machine Constants

The representation for floating point numbers described above is only one possibility, and may or may not be used on any particular computer. However, any finite binary representation of floating point numbers will have restrictions on the accuracy of representations. For example, there will be a smallest possible positive number, a smallest number that added to 1 will give a result different than 1, and in general for any number a closest representable number. There are also the largest (in magnitude) positive and negative numbers that can be stored without producing overflow. Numbers such as these are called machine constants, and are determined by how floating point numbers are stored and how arithmetic operations are performed. If we knew exactly how numbers were stored and arithmetic performed, then we could determine exactly what these numbers should be. This information is generally not readily available to the average computer user, though, so some standard algorithms have been developed to diagnose these machine constants.

The LAPACK library of linear algebra routines contains FORTRAN subroutines `slamch` and `dlamch` for computing single and double precision machine constants. Corresponding C routines are available in the C version of the library. Source code for LAPACK can be obtained from Netlib (`http://www.netlib.org`). The Numerical Recipes library (Press *et. al.* (1992), Section 20.1) also has a program for computing machine constants, called `machar`. Individual sections of the *Numerical Recipes* books can be accessed at `http://www.nr.com/`.

Since Splus is essentially a compiled C and FORTRAN program, for the most part it stores floating point numbers and performs arithmetic the same as in C and FORTRAN, and it will be used here to examine some of these constants. First consider the smallest number that can be added to 1 that will give a value different from 1. This is often denoted ϵ_m. Recalling the basic floating point representation,

$$1 = (1/2 + \sum_{i=2}^{t} 0/2^i)2^1,$$

so the next largest representable number is $1 + 1/2^{t-1}$. It was claimed above that in standard double precision, $t = 52$. However, since the leading bit in the mantissa need not be stored, an extra digit can be gained, effectively making $t = 53$. Thus $1 + 1/2^{52}$ should be different than 1:

```
> options(digits=20)
> 1+1/2^53
[1] 1
> 1+1/2^52
[1] 1.000000000000000222
```

```
> 1/2^52
[1] 2.2204460492503130808e-16
```

While $1 + 1/2^{52}$ is the next largest representable number, ϵ_m may be smaller than $1/2^{52}$. That is, if addition is performed to higher accuracy and the result rounded to the nearest representable number, then the next representable number larger than $1/2^{53}$, when added to 1, should also round to this value. The next number larger than $1/2^{53}$ should be $(1 + 1/2^{52})/2^{53} = 1/2^{53} + 1/2^{105}$.

```
> 1/2^53
[1] 1.1102230246251565404e-16
> 1/2^53+1/2^106
[1] 1.1102230246251565404e-16
> 1/2^53+1/2^105
[1] 1.1102230246251567869e-16
> 1+(1/2^53+1/2^106)
[1] 1
> 1+(1/2^53+1/2^105)
[1] 1.000000000000000222
```

If instead of adding numbers to 1 we subtract, the results are slightly different. Since the result is < 1, the exponent shifts by 1, gaining an extra significant digit. Thus the next representable number smaller than 1 should be $1 - 1/2^{53}$, and $1 - (1/2^{54} + 1/2^{106})$ should round to this value.

Exercise 1.1 Verify the previous statement.

If x is the true value of a number and $f(x)$ is its floating point representation, then ϵ_m is an upper bound on the relative error of any stored number (except in the case of overflow or underflow). That is, if $f(x) = 2^k(1/2 + \sum_{i=2}^{t} x_i/2^i)$ with $x_i = 0, 1$, then using exact (not floating point) arithmetic, the relative error is

$$\frac{|x - f(x)|}{|x|} \leq \frac{2^k}{2^{t+1}|x|},$$

since otherwise $|x|$ would round to a different floating point number, and since

$$\frac{1}{|x|} \leq \frac{1}{|f(x)| - 2^k/2^{t+1}} \leq \frac{1}{2^{k-1}(1 - 1/2^t)} \leq 2^{1-k}(1 + 1/2^{t-1}),$$

where the last inequality follows because $1 \leq (1 - 1/2^t)(1 + 1/2^{t-1})$. Combining with the previous expression then gives

$$\frac{|x - f(x)|}{|x|} \leq 1/2^t + 1/2^{2t-1} = \epsilon_m.$$

The value ϵ_m thus plays an important role in error analysis. It is often called the machine precision or machine epsilon. From above, in double precision on a Sun, $\epsilon_m \doteq 1.11 \times 10^{-16}$, so double precision numbers are stored accurate to about 16 decimal digits. Note: there is not

complete agreement on the terminology. `dlamch` in LAPACK gives the value above for `eps = relative machine precision`, while Press *et. al.* (1992) use $1/2^{52}$ for `eps = floating point precision`. Splus includes an object `.Machine` that specifies a number of machine constants, which gives $1/2^{52}$ for `.Machine$double.eps`.)

Exercise 1.2 Determine ϵ_m as defined above for single precision floating point numbers. (In Splus this can be done by making extensive use of the `as.single()` function.)

The largest and smallest positive floating point numbers are given below.

```
> # largest possible mantissa
> u <- 0
> for (i in 1:53) u <- u+1/2^i
> u
[1] 0.99999999999999988898
> # and the largest possible exponent--note that calculating 2^1024
> # directly overflows
> u*2*2^1023
[1] 1.797693134862315708e+308
> # next largest floating point number overflows
> (u*2*2^1023)*(1+1/2^52)
[1] Inf
>
> # smallest possible mantissa and smallest possible exponent
> 1/2^52*2^(-1022)
[1] 4.9406564584124654418e-324
> 1/2^52*2^(-1022)/2
[1] 0
```

Although numbers less than 2^{-1022} can be represented (on machines compliant with the IEEE standard), generally smaller numbers can have larger relative errors than machine precision, due to the gradual loss of significant digits in the mantissa:

```
> (1/2+1/2^53)
[1] 0.50000000000000011102
> 2^(-1022)/2
[1] 1.1125369292536006915e-308
> (1/2+1/2^53)*2^(-1022)
[1] 1.1125369292536006915e-308
> (1/2+1/2^52)*2^(-1022)
[1] 1.1125369292536011856e-308
```

That is, the representation of $1/2$ and $1/2 + 1/2^{53}$ are distinct, but the representations of $(1/2) \times 2^{-1022}$ and $(1/2 + 1/2^{53}) \times 2^{-1022}$ are not, so the latter has a larger relative error.

Exercise 1.3 (a) Determine the smallest and largest positive representable numbers in single precision. (b) Determine the smallest and largest representable negative numbers in single precision. (c) Determine the smallest and largest representable negative numbers in double precision.

Exercise 1.4 Describe a general algorithm for determining the largest representable number without producing overflow.

1.4 Accuracy of Arithmetic Operations

Consider computing $x + y$, where x and y have the same sign. Let δ_x and δ_y be the relative error in the floating point representation of x and y ($\delta_x = [f(x) - x]/x$, so $f(x) = x(1 + \delta_x)$). What the computer actually calculates is the sum of the floating point representations $f(x) + f(y)$, which may not have an exact floating point representation, in which case it is rounded to the nearest representable number $f(f(x) + f(y))$. Let $\delta_s = [f(f(x) + f(y)) - f(x) - f(y)]/[f(x) + f(y)]$ be the relative error in this final representation. Note that $\max\{|\delta_x|, |\delta_y|, |\delta_s|\} \leq \epsilon_m$. Also,

$$|f(x) + f(y)| = |x(1 + \delta_x) + y(1 + \delta_y)| \leq |x + y|(1 + \epsilon_m),$$

where the fact that x and y have the same sign has been used. Thus

$$\begin{aligned} |f(f(x) + f(y)) - (x + y)| &= |f(f(x) + f(y)) - f(x) - f(y) + f(x) - x + f(y) - y| \\ &\leq |\delta_s[f(x) + f(y)]| + |\delta_x x| + |\delta_y y| \\ &\leq |x + y|(\epsilon_m + \epsilon_m^2) + |x + y|\epsilon_m \\ &\doteq 2\epsilon_m |x + y|, \end{aligned}$$

where the higher order terms in ϵ_m have been dropped, since they are usually negligible. Thus $2\epsilon_m$ is an (approximate) bound on the relative error in a single addition of two numbers with the same sign.

For multiplication, letting δ_m be the relative error in the floating point representation of $f(x)f(y)$,

$$f(f(x)f(y)) = f(x)f(y)(1 + \delta_m) = x(1 + \delta_x)y(1 + \delta_y)(1 + \delta_m),$$

so the relative error is bounded by $3\epsilon_m$ (to first order in ϵ_m). A similar bound holds for division, provided the denominator and its floating point representation are nonzero.

These errors in single binary operations accumulate as more and more calculations are done, but they are so small that a very large number of such operations must be done to introduce substantial inaccuracy in the results. However, there is one remaining basic binary operation: subtraction, or adding numbers of opposite sign. For subtraction, if the two numbers are similar in magnitude, then the leading digits in the mantissa will cancel, and there may only be a few significant digits left in the difference, leading to a large relative error. To illustrate with an extreme example, suppose the floating point representations are $f(x) = (1/2 + \sum_{i=1}^{t-1} x_i/2^i + 1/2^t)$ and $f(y) = (1/2 + \sum_{i=1}^{t-1} x_i/2^i + 0/2^t)$. Then the computed difference $f(x) - f(y) = 1/2^t$. But due to the error in the representation of x and y ($f(x)$ and $f(y)$ could each have absolute error as large as $1/2^{t+1}$), the true difference could be anywhere in the interval $(0, 1/2^{t-1})$. Thus in this extreme case there is no guarantee of any significant digits of accuracy in the result, and the relative error can be arbitrarily large.

Example 1.1 It is easy to find examples where subtraction results in catastrophic cancellation. Consider the problem of computing $\exp(x)$. If $|x|$ is not too large, the Taylor series $\exp(x) = \sum_{i \geq 0} x^i/i!$ converges rapidly. A straightforward implementation for evaluating the partial sums in this series is given below. Note that it works well for positive x but poorly for negative x.

```
> fexp <- function(x) {
+   i <- 0
+   expx <- 1
+   u <- 1
+   while(abs(u)>1.e-8*abs(expx)) {
+     i <- i+1
+     u <- u*x/i
+     expx <- expx+u
+   }
+   expx
+ }
> options(digits=10)
> c(exp(1),fexp(1))
[1] 2.718281828 2.718281826
> c(exp(10),fexp(10))
[1] 22026.46579 22026.46575
> c(exp(100),fexp(100))
[1] 2.688117142e+43 2.688117108e+43
> c(exp(-1),fexp(-1))
[1] 0.3678794412 0.3678794413
> c(exp(-10),fexp(-10))
[1] 4.539992976e-05 4.539992956e-05
> c(exp(-20),fexp(-20))
[1] 2.061153622e-09 5.621884467e-09
> c(exp(-30),fexp(-30))
[1]  9.357622969e-14 -3.066812359e-05
> (-20)^10/prod(2:10)
[1] 2821869.489
> (-20)^9/prod(2:9)
[1] -1410934.744
> (-20)^20/prod(20:2)
[1] 43099804.12
```

By $x = -20$ there are no significant digits of accuracy, and the algorithm gives a negative value for $\exp(-30)$. The terms in the series for $x = -20$, for $i = 9, 10, 20$, are also given above. The problem occurs because the terms alternate in sign, and some of the terms are much larger than the final solution. Since double precision numbers have about 16 significant digits, through the 20th term the accumulated sum would be expected to have an absolute error of order at least 10^{-8}, but $\exp(-20)$ is order 10^{-9}.

When $x > 0$, all terms are positive, and this problem does not occur. A solution is easily

implemented by noting that $\exp(-x) = 1/\exp(x)$, as follows.

```
> fexp <- function(x) {
+   xa <- abs(x)
+   i <- 0
+   expx <- 1
+   u <- 1
+   while(u>1.e-8*expx) {
+     i <- i+1
+     u <- u*xa/i
+     expx <- expx+u
+   }
+   if (x >= 0) expx else 1/expx
+ }
> c(exp(-1),fexp(-1))
[1] 0.3678794412 0.3678794415
> c(exp(-10),fexp(-10))
[1] 4.539992976e-05 4.539992986e-05
> c(exp(-20),fexp(-20))
[1] 2.061153622e-09 2.061153632e-09
> c(exp(-100),fexp(-100))
[1] 3.720075976e-44 3.720076023e-44
```

□

A computing problem is said to be *ill conditioned* if small perturbations in the problem give large perturbations in the *exact* solution. An example of an ill conditioned problem is solving the system of linear equations

$$A_{n\times n}x_{n\times 1} = b_{n\times 1}$$

for x when A is nearly singular. In this problem small changes (or errors) in A or b can give large differences in the exact solution x. Since storing A and b in a computer involves small errors in rounding to floating point numbers, it is essentially impossible to get precise solutions to ill-conditioned problems.

In the exponential example above, if exact computations were done, then a small change in x would give only a small change in the computed value of $\exp(x)$. The difficulty encountered was with the particular algorithm, and was not inherent in the problem itself.

Suppose $g(x)$ is the exact solution to a problem with inputs x, and $g^*(x)$ is the value computed by a particular algorithm. The computed solution $g^*(x)$ can often be thought of as the exact solution to a perturbed problem with inputs $\tilde{x}$, that is, $g^*(x) = g(\tilde{x})$. If there is always an $\tilde{x}$ close to x such that $g^*(x) = g(\tilde{x})$, then the algorithm is said to be *stable*. If not, then the algorithm is *unstable*. The first algorithm given above for computing $\exp(x)$ is unstable for large negative x, while the second version is stable.

This definition of stability is related to the concept of backward error analysis. The analysis of errors for binary arithmetic operations given above was a forward error analysis. In forward

analysis, the calculations are examined one step at a time from the beginning, and the errors that can occur at each step accumulated. This can generally only be done for fairly simple problems. In backward analysis, given a particular computed solution, it is determined how much the original problems would need to be modified for the computed solution to be the exact solution to the perturbed problem. If only a small perturbation is needed, then the solution is about as good as could be expected. In ill-conditioned problems, such a solution still might not be very accurate, but it is as about as good as can be done with the available tools. The topic of backward error analysis was extensively developed by Wilkinson (1963). There will be little formal error analysis in this course, but a few results will be given below for sums and variances, and some results will also be given later for solving systems of equations.

1.5 Computing Sums

The basic problem is to compute $S = \sum_{i=1}^n x_i$ from the stored floating point representations $f(x_i)$. Suppose first that $f(x_1)$ is added to $f(x_2)$ and the result stored as a floating point number, then $f(x_3)$ is added to the sum and the result again converted to a floating point number, and the sum computed by continuing in this fashion. Denote this computed sum by S^*. Let δ_i be the relative error in the representation $f(x_i)$, and let ϵ_i be the relative error in converting the result of the ith addition to a representable floating point number. For $n = 3$,

$$\begin{aligned} S^* &= \{[x_1(1+\delta_1) + x_2(1+\delta_2)](1+\epsilon_1) + x_3(1+\delta_3)\}(1+\epsilon_2) \\ &= [x_1(1+\delta_1) + x_2(1+\delta_2)](1+\epsilon_1)(1+\epsilon_2) + x_3(1+\delta_3)(1+\epsilon_2) \\ &\doteq \sum_{i=1}^3 x_i + \sum_{i=1}^3 x_i(\delta_i + \sum_{j=i-1}^2 \epsilon_j), \end{aligned}$$

dropping higher order terms in the δ's and ϵ's, and where $\epsilon_0 \equiv 0$. The generalization to n terms is easily seen to be

$$\begin{aligned} S^* &= [x_1(1+\delta_1) + x_2(1+\delta_2)] \prod_{j=1}^{n-1}(1+\epsilon_j) + \sum_{i=3}^n x_i(1+\delta_i) \prod_{j=i-1}^{n-1}(1+\epsilon_j) \\ &\doteq \sum_{i=1}^n x_i + \sum_{i=1}^n x_i \left(\delta_i + \sum_{j=i-1}^{n-1} \epsilon_j\right). \end{aligned}$$

Thus to first order in the errors, again denoting the machine precision by ϵ_m

$$|S^* - S| \leq |x_1| n\epsilon_m + \sum_{i=2}^n |x_i|(n-i+2)\epsilon_m \tag{1.1}$$

$$\leq \max_i |x_i| \epsilon_m (n + \sum_{i=2}^n i) = \max_i |x_i| \epsilon_m (n^2 + 3n - 2)/2, \tag{1.2}$$

since $\sum_{i=1}^n i = n(n+1)/2$. From (1.2), the error in this algorithm increases as the square of the number of terms in the sum. As before, if some elements are of opposite sign, and $|S|$ is much smaller than $\max_i |x_i|$, then the relative error can be very large. If all numbers have the same sign, then the error in this algorithm can be reduced by summing the values with the smallest magnitude first, since by (1.1) the first 2 terms are involved in $n-1$ additions and accumulate the largest potential errors, while the last number added is only involved in 1 sum.

Organizing the calculations differently can reduce the bound on the error. Suppose $n = 8$ and the calculations are performed by first adding adjacent pairs of values, then adding adjacent pairs of the sums from the first stage, etc., as implied by the parentheses in the following expression:

$$S = \{[(x_1 + x_2) + (x_3 + x_4)] + [(x_5 + x_6) + (x_7 + x_8)]\}.$$

In this pairwise summation algorithm, each value is only involved in 3 additions, instead of the maximum of 7 in the previous algorithm. In general, if $n = 2^k$, then each term is involved in only $k = \log_2(n)$ additions in the pairwise summation algorithm. Reasoning as before, it can be shown that the sum computed from the floating point representations using pairwise summation, say S_p^*, satisfies

$$|S_p^* - S| \le \sum_{i=1}^{n} |x_i|(k+1)\epsilon_m \le \max_i |x_i| n(k+1)\epsilon_m, \tag{1.3}$$

when $n = 2^k$. When n is not a power of 2, the bound holds with k equal to the smallest integer $\ge \log_2(n)$. Thus roughly speaking, the error is order $n \log_2(n)$ for pairwise summation versus n^2 for the standard algorithm. For $n = 1000$, $n/\log_2(n) \doteq 100$, and pairwise summation could be as much as 2 digits more accurate than the standard approach.

Exercise 1.5 Derive the error bound (1.3) for the pairwise summation algorithm.

Exercise 1.6 Write a function in C or FORTRAN to implement pairwise summation. Do not overwrite the original data with the partial sums. What is the minimal amount of storage needed?

Error bounds consider worst case scenarios that may not represent typical behavior. If the relative error in each binary floating point arithmetic operation was uniformly distributed on $(-\epsilon_m, \epsilon_m)$, and the errors were independent, then the error in a series of N computations would approximately be distributed $\mathrm{N}(0, N\epsilon_m^2/3)$, suggesting a typical error of $O(N^{1/2}\epsilon_m)$ (the standard deviation). However, errors tend not to be independent, and as has been seen above, subtractions can produce much larger relative errors, so this value probably does not represent typical behavior either.

1.6 Computing Sample Variances

The familiar formula for the sample variance,

$$S^2 = \frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left[\sum_{i=1}^{n} x_i\right]^2\right),$$

suggests a one-pass algorithm for computing S^2:

Algorithm 1.1

1. Initialize $U = 0$, $V = 0$.
2. For $i = 1$ to n, set $U = U + x_i$ and $V = V + x_i^2$.
3. Set $S^2 = (V - U^2/n)/(n-1)$.

Unfortunately, this simple algorithm is unstable, since the subtraction can leave few significant digits of accuracy.

It is much better to use a two pass algorithm:

Algorithm 1.2

1. In the first pass compute U as above, and set $\overline{U} = U/n$.
2. Initialize $V = 0$.
3. For $i = 1$ to n, set $V = V + (x_i - \overline{U})^2$.
4. Set $S^2 = V/(n-1)$.

Since there will be floating point errors in computing $\overline{U}$, it may sometimes be possible to get a further improvement by correcting the two pass algorithm:

Algorithm 1.3

1. In the first pass compute U as above, and set $\overline{U} = U/n$.
2. Initialize $V = 0$ and $W = 0$.
3. For $i = 1$ to n, set $V = V + (x_i - \overline{U})^2$ and $W = W + (x_i - \overline{U})$.
4. Set $S^2 = (V - W^2/n)/(n-1)$.

While Algorithm 1.3 corrects V for errors in $\overline{U}$, these errors are generally small, and so it usually results in little change and may not be worth the added computations.

In most situations the standard two-pass algorithm would be preferred. However, if for some reason the entire data vector cannot be stored in memory simultaneously, then a two pass algorithm might be substantially slower, because of the need to access the data twice. For example, in a simulation where a large number of values were generated from each simulated sample, and the variances of these values over the simulated samples were to be computed, it might be preferred to compute the variance contributions from each sample as they were generated, rather than storing all the output and computing the variances at the end from the stored output.

Another setting where the two-pass algorithm is inefficient is computing weighted variances with weights that require extra computations. This occurs in computing the information for the Cox partial likelihood. Let T_i be the failure/censoring time, δ_i the failure indicator (1 for failures, 0 for censored), and z_i the covariate value, for subject i, $i = 1, \ldots, n$. The proportional hazards model specifies that the failure hazard satisfies $\lambda(t|z_i) = \lambda_0(t)\exp(z_i\beta)$, where λ_0 is an unknown underlying hazard, and β is an unknown parameter. The partial likelihood information for β is given by the formula

$$\sum_i \delta_i \left[\frac{\sum_{j \in R(T_i)} w_j z_j^2}{\sum_{j \in R(T_i)} w_j} - \left(\frac{\sum_{j \in R(T_i)} w_j z_j}{\sum_{j \in R(T_i)} w_j} \right)^2 \right],$$

where $R(t) = \{j : T_j \geq t\}$ and $w_j = \exp(z_j\beta)$. The term inside square brackets is a weighted sample variance. If the two-pass algorithm is applied, then the weights w_j need to be recomputed

in each pass. Alternately, they could be computed once and stored, but that would require using extra memory (which may not be a problem unless n is very large).

In settings where a two-pass algorithm is inconvenient or inefficient, there are two ways to proceed. The first is to use a preliminary guess at the mean, and to use this preliminary guess in place of $\overline{U}$ in algorithm 3. For computing the partial likelihood information, the unweighted mean $\sum_{j\in R(T_i)} z_j / \sum_{j\in R(T_i)} 1$ would usually be adequate (and unless censoring depends heavily on the covariates, the overall unweighted mean $\sum_{j=1}^{n} z_j/n$ would also usually be acceptable). That is, the unweighted mean could be computed in one pass, which would not require computation of the w_j, and then a weighted version of steps 2–4 of algorithm 3 applied with the unweighted mean in place of $\overline{U}$.

The second (and generally more accurate) method is to use an updating formula. Let $\overline{x}_k = \sum_{i=1}^{k} x_i/k$ and $SS(k) = \sum_{i=1}^{k}(x_i - \overline{x}_k)^2$. Then it is easily verified that

$$\begin{aligned} SS(k) &= SS(k-1) + k(k-1)(\overline{x}_k - \overline{x}_{k-1})^2 \\ &= SS(k-1) + \frac{k}{k-1}(x_k - \overline{x}_k)^2 \\ &= SS(k-1) + \frac{k-1}{k}(x_k - \overline{x}_{k-1})^2, \end{aligned} \tag{1.4}$$

any of which could be used to compute the variance in a single pass over the data. The third formula can be implemented as follows:

Algorithm 1.4

1. Initialize $U_1 = x_1$ and $V_1 = 0$.
2. For $i = 2$ to n set $V_i = V_{i-1} + (i-1)(x_i - U_{i-1})^2/i$ and $U_i = U_{i-1} + (x_i - U_{i-1})/i$.
3. Set $S^2 = V_n/(n-1)$.

Here at each step U_i is the mean of the first i observations and V_i is their corrected sum of squares.

Exercise 1.7 Verify the updating formula (1.4).

Exercise 1.8 Derive an updating formula similar to (1.4) for computing a weighted variance.

Example 1.2 To illustrate these approaches to computing the sample variance, a simple example with 5 data points will be considered. Calculations are done in single precision. Note that the exact variance for the original data is 2.5×10^{-8}. However, the exact data values cannot be represented in floating point, so the stored data values have errors. The variance of the stored values is $2.50023369119 \times 10^{-8}$ (to 12 significant digits), and the best that could be expected of any computational algorithm is to give the variance of the values as stored in the computer.

```
> options(digits=12)
> x <- as.single(c(1.0,1.0001,1.0002,.9999,.9998))
> x
```

```
[1] 1.000000000000 1.000100016594 1.000200033188 0.999899983406 0.999800026417
> n <- as.integer(length(x))
> c(is.single(x^2),is.single(sum(x)),is.single(sum(x)/n))
[1] T T T
> # usual "one-pass" formula (although not actually computed in one pass)
> (sum(x^2)-sum(x)^2/n)/(n-as.integer(1))
[1] -2.38418579102e-07
> # standard "two-pass" algorithm
> y <- x-sum(x)/n
> sum(y^2)/(n-as.integer(1))
[1] 2.50023504123e-08
> sum(y)
[1] -5.36441802979e-07
> # "two-pass" corrected
> (sum(y^2)-sum(y)^2/n)/(n-as.integer(1))
[1] 2.50023362014e-08
> # shifted one-pass (using x[5] as a preliminary guess at the mean)
> y <- x-x[5]
> (sum(y^2)-sum(y)^2/n)/(n-as.integer(1))
[1] 2.50023326487e-08
> # updating correction by current mean
> u <- x[1]
> v <- as.single(0)
> for (i in 2:n) {
+   t1 <- x[i]-u
+   t2 <- t1/i
+   v <- v+(i-as.integer(1))*t1*t2
+   u <- u+t2
+ }
> v <- v/(n-as.integer(1))
> is.single(v)
[1] T
> v
[1] 2.50053169282e-08
```

As can be seen, all approaches except the first work reasonably well.

It is interesting to note that if the above calculations are repeated on the 5 numbers 9998, 9999, 10000, 10001, 10002, then the one pass algorithm gives 0, and all the other approaches give the exact result of 2.5. The difference is that in this case the numbers are integers that have exact floating point representations in single precision. However, the squares of several of these values cannot be represented exactly in single precision, and hence the one-pass algorithm still fails. □

1.7 Error Analysis for Variance Algorithms

Chan, Golub and LeVeque (1983) summarize results on error bounds for algorithms for computing sample variances. A key part of their analysis, and for error analysis of many problems, is a quantity called the condition number. Condition numbers generally measure how well (or ill) conditioned a problem is. That is, the condition number relates the relative change in the exact solution to a problem when the inputs are perturbed to the magnitude of the perturbation of the inputs. Suppose the exact value of the corrected sum of squares is $SS = \sum_{i=1}^{n}(x_i - \overline{x}_n)^2$. Suppose also each x_i is perturbed by a relative amount u_i, and set $x_i^* = x_i(1+u_i)$ and $SS^* = \sum_{i=1}^{n}(x_i^* - \overline{x}_n^*)^2$. Then to first order in $\|u\|$,

$$\frac{|SS^* - SS|}{|SS|} \leq \|u\| 2(1 + n\overline{x}_n^2/SS)^{1/2}, \tag{1.5}$$

where $\|u\| = (\sum_i u_i^2)^{1/2}$ is the usual Euclidean norm of the vector of perturbations u. The quantity $\kappa = 2(1 + n\overline{x}_n^2/SS)^{1/2}$ is the condition number, which is a function of the particular data set. If κ is large, then small perturbations in the data can give large changes in the sum of squares. Note that in the data used above, the condition number is 2.8×10^4, so computing the variance for this data is not a well conditioned problem. Computing the variance will be ill-conditioned whenever $|\overline{x}_n| >> (SS/n)^{1/2}$, that is, whenever the mean is much larger than the standard deviation.

To give a rough argument for (1.5), first note that (using exact arithmetic)

$$\begin{aligned}
SS^* &= \sum_i x_i^2(1+u_i)^2 - \frac{1}{n}\left[\sum_i x_i(1+u_i)\right]^2 \\
&= \sum_i x_i^2 + 2\sum_i x_i^2 u_i + \sum_i x_i^2 u_i^2 - n\overline{x}_n^2 - 2\overline{x}_n \sum_i x_i u_i - \left(\sum_i x_i u_i\right)^2 /n \\
&= SS + 2\sum_i u_i x_i (x_i - \overline{x}_n) + O(\|u\|^2),
\end{aligned}$$

so

$$\begin{aligned}
\frac{|SS^* - SS|}{SS} &\leq 2\left|\sum_i u_i x_i (x_i - \overline{x}_n)\right| /SS + O(\|u\|^2) \\
&\leq 2\|u\| \left[\sum_i x_i^2 (x_i - \overline{x}_n)^2\right]^{1/2} /SS + O(\|u\|^2) \\
&\leq 2\|u\| \left[\sum_i x_i^2 SS\right]^{1/2} /SS + O(\|u\|^2) \\
&= 2\|u\| \left[(SS + n\overline{x}_n^2)SS\right]^{1/2} /SS + O(\|u\|^2) \\
&= 2\|u\| (1 + n\overline{x}_n^2/SS)^{1/2} + O(\|u\|^2).
\end{aligned}$$

The second line follows from the Cauchy-Schwarz inequality, which states that $(\sum_i a_i b_i)^2 \leq (\sum_i a_i^2)(\sum_i b_i^2)$, and the third line because $\sum_i a_i^2 b_i^2 \leq (\sum_i a_i^2)(\sum_i b_i^2)$, for any real numbers.

The condition number given by Chan, Golub and LeVeque does not have the factor of 2, so there may be a better argument that gives a tighter bound.

The effect of shifting the data can be seen in the condition number. Shifting the data to have mean 0 reduces the condition number to 1. Shifting the data so that the mean is small relative to the SS will make the condition number small.

The condition number indicates how much the exact SS can be influenced by small errors in the data. It still remains to determine how much error can result from using a particular algorithm to compute SS. Chan, Golub and LeVeque give the following approximate bounds on the relative error in computing SS (with small constant multipliers and higher order terms dropped). The relative error in computing the variance is essentially the same as for SS.

Table 1. Error bounds on the relative error in computing SS, from Chan, Golub and LeVeque (1983).

Algorithm 1	$N\kappa^2\epsilon_m$
Algorithm 2	$N\epsilon_m + N^2\kappa^2\epsilon_m^2$
Algorithm 3	$N\epsilon_m + N^3\kappa^2\epsilon_m^3$
Algorithm 4	$N\kappa\epsilon_m$

Generally, if pairwise summation is used throughout, then N is replaced by $\log_2(N)$ in these formulas. From these approximate bounds, if the data are shifted so that the condition number is not too large, then the one-pass algorithm 1 could be used, while for large condition numbers a two-pass or updating algorithm is needed. If $N = 5$, $\kappa = 2.8 \times 10^4$ (as in the example earlier), and $\epsilon_m \doteq 6 \times 10^{-8}$ (fairly standard for single precision), then the bound on the precision of Algorithm 1 is about 235 (ie no precision), the bound on the precision of Algorithm 2 is about 7×10^{-5}. These bounds suggest that Algorithm 1 has no accuracy for this problem, while Algorithm 2 should be accurate to at least 4 significant digits, both of which were reflected in the numerical results above. The bound on the corrected two-pass algorithm (algorithm 3) is about 3×10^{-7}, suggesting it can give substantial additional gains over Algorithm 2. Note that in the example, the accuracy of Algorithm 3 compared to the true variance of the original data was less than this. However, these bounds are in the error of the computational algorithm, and do not take into account the initial errors in the floating point representations of the data. In the example, Algorithm 3 was within the error bound when compared to the variance of the data as stored. The updating algorithm has an error bound of 8×10^{-3}, but the numerical result was more accurate than this would suggest (the bounds need not be terribly tight in specific examples).

Exercise 1.9 Suppose $N = 1000$, $\kappa = 100$, and $\epsilon_m \doteq 6 \times 10^{-8}$. How many significant digits would be expected for each of the 4 algorithms, based on the error bounds in Table 1?

1.8 References

Chan TF, Golub GH and LeVeque RJ (1983). Algorithms for computing the sample variance: analysis and recommendations. *American Statistician*, 37:242–247.

Press WH, Teukolsky SA, Vetterling WT, and Flannery BP (1992). *Numerical Recipes in C: The Art of Scientific Computing. Second Edition*. Cambridge University Press.

Wilkinson JH (1963). *Rounding Errors in Algebraic Processes*. Prentice-Hall.

Chapter 2

Numerical Linear Algebra

Some Terminology and Notation:
Elements of a matrix A will usually be denoted a_{ij}, with the first subscript denoting the row and the second the column. Matrices will also be written as $A = (a_{ij})$.
A' denotes the transpose of a matrix or vector A. If $A = (a_{ij})$ and $(b_{ij}) = B = A'$, then $b_{ij} = a_{ji}$.
A vector x is a column vector, and x' is a row vector.
The main diagonal of a square matrix A consists of the elements a_{ii}.
Sub-diagonal elements are those elements below the main diagonal (which are the elements a_{ij} with $i > j$.)
Super-diagonal elements are those elements above the main diagonal ($j > i$).
An upper triangular matrix has all sub-diagonal elements equal to 0.
A lower triangular matrix has all super-diagonal elements equal to 0.
A diagonal matrix is a square matrix with all elements equal to 0 except those on the main diagonal.
An identity matrix is a square matrix I with 1's on the main diagonal and all other elements 0.
A symmetric matrix A is positive definite if $x'Ax > 0$ for all $x \neq 0$.

2.1 Matrix Multiplication

Interesting matrix operations on $p \times p$ matrices generally require $O(p^3)$ operations.

Consider the following FORTRAN fragment for computing

$$A_{m\times k}B_{k\times n} = C_{m\times n}.$$

```
      DO 90, J = 1, N
         DO 50, I = 1, M
            C( I, J ) = ZERO
50       CONTINUE
         DO 80, L = 1, K
            IF( B( L, J ).NE.ZERO )THEN
               TEMP = B( L, J )
               DO 70, I = 1, M
```

```
                  C( I, J ) = C( I, J ) + TEMP*A( I, L )
70              CONTINUE
             END IF
80        CONTINUE
90     CONTINUE
```

If none of the elements of B are 0, then the command

```
          C( I, J ) = C( I, J ) + TEMP*A( I, L )
```

is executed mkn times.

For square matrices, $m = k = n = p$, so there are exactly p^3 multiplications and exactly p^3 additions.

A FLOP is a measure of computation complexity equivalent to one step in an inner product calculation (1 floating point addition, 1 floating point multiplication, plus some array address calculations).

Floating point multiplication and division require similar amounts of processing time, and both require substantially more time than floating point addition or subtraction. Instead of formally counting FLOPs, a close approximation to computation complexity in numerical linear algebra problems can often be obtained by just counting the number of floating point multiplications and divisions. Here FLOP will be used to refer specifically to the number of multiplications and divisions in an algorithm, although this usage is not standard and could give somewhat different results than the formal definition. As already noted, there are exactly p^3 multiplications in the above algorithm, so in this terminology it requires p^3 FLOPS.

If matrix multiplication requires $O(p^3)$ operations, it would be expected that more complex problems, such as inverting matrices and solving systems of equations, would require at least this order of computational complexity.

The order of the loops in the above algorithm is important. Compare the algorithm above with the naive ordering of commands in the following code:

```
      do 10 i=1,m
         do 12 j=1,n
            c(i,j)=0
            do 14 l=1,k
               c(i,j)=c(i,j)+a(i,l)*b(l,j)
14          continue
12       continue
10    continue
```

Using separate programs for each algorithm, with $n = m = k = 500$, the first version took 62 CPU seconds while the second required 109 CPU seconds. Using optimization flags when compiling the code, ie `f77 -O`, the time is reduced to 12 CPU seconds for the first algorithm and

to 53 CPU seconds for the second. (Calculations were done on a SUN workstation using the SUN f77 compiler.) The reason for the difference is that in the first algorithm the inner loop is addressing consecutive storage locations in the A and C arrays, and the computer can access consecutive locations faster than arbitrary memory addresses. In FORTRAN, the elements in the column of a matrix are stored in consecutive memory addresses, while in C the elements in a row are stored in consecutive addresses. (One other difference is that in FORTRAN the entire array is stored in a consecutive block of memory, that is, the entire array can be thought of as one long vector subdivided into columns, while in C it is possible for different rows in an array to not be in adjacent memory locations, unless some extra work is done setting up the array; see the discussion of this point in Press *et. al.* (1992).) In Splus, arrays are stored in the same order as in FORTRAN. Quite possibly this was done because when the original version of old S was written, LINPACK FORTRAN libraries were used for matrix calculations.

It is possible to improve on the p^3 multiplications required in the algorithms above. Strassen's algorithm for matrix multiplication requires only $p^{\log_2(7)} \doteq p^{2.807}$ multiplications for square matrices, but requires many more additions, so the savings is likely to be small unless p is quite large. This algorithm is briefly discussed in Section 2.11 of Press *et. al.* (1992).

2.2 Systems of Linear Equations

Consider the problem of solving for $x_1, \ldots, x_p$ in the system of equations

$$\sum_{j=1}^{p} a_{ij} x_j = b_i, \quad i = 1, \ldots, p, \tag{2.1}$$

given values for the a_{ij} and b_i. In matrix terms, this system can be written

$$Ax = b,$$

where $A_{p\times p} = (a_{ij})$, and x and b are column vectors.

For example, suppose A is an information matrix. The delta method variance of the MLE is of the form

$$b'A^{-1}b. \tag{2.2}$$

It is generally better to calculate $A^{-1}b$ as the solution to $Ax = b$ than to calculate A^{-1} directly and multiply. Actually, in this application A should be positive definite, and it is usually even better to factor $A = U'U$, where U is upper triangular, so

$$b'A^{-1}b = b'(U'U)^{-1}b = [(U')^{-1}b]'[(U')^{-1}b].$$

Then $(U')^{-1}b$ can be calculated by solving the triangular system $U'x = b$, and $b'A^{-1}b$ computed by taking the inner product $x'x$.

2.2.1 Triangular Systems

Suppose A is upper triangular:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1p}x_p &= b_1 \\ a_{22}x_2 + \cdots + a_{2p}x_p &= b_2 \\ \vdots \quad \vdots & \quad \vdots \\ a_{pp}x_p &= b_p \end{aligned}$$

This system can be solved easily by starting with the last equation and working back up the system. That is

$$\begin{aligned} x_p &= b_p/a_{pp} \\ x_{p-1} &= (b_{p-1} - a_{p-1,p}x_p)/a_{p-1,p-1} \\ \vdots \quad \vdots & \quad \vdots \\ x_1 &= (b_1 - a_{12}x_2 - \cdots - a_{1p}x_p)/a_{11} \end{aligned}$$

The solution can overwrite b, since once x_j is computed b_j is no longer needed.

Solving for x_{p-j}, given the values for $x_{j+1}, \ldots, x_p$, requires j multiplications and one division, so the total number of FLOPS needed in computing the entire solution is

$$\sum_{j=0}^{p-1}(j+1) = p(p+1)/2.$$

This algorithm is called *backward substitution*.

There is an analogous *forward substitution* algorithm for lower triangular systems. That is, for the system

$$\sum_{j=1}^{i} a_{ij}x_j = b_i, \quad i = 1, \ldots, p,$$

the solution can be computed by

$$x_1 = b_1/a_{11},$$

and in general

$$x_j = (b_j - \sum_{k=1}^{j-1} a_{jk}x_k)/a_{jj}.$$

Again the total number of FLOPS required is $p(p+1)/2$.

Although it is usually unnecessary, consider the problem of solving for the inverse of a triangular matrix. Define vectors $e^{(i)}$ by $e_i^{(i)} = 1$ and $e_j^{(i)} = 0$ for $j \neq i$. Then the ith column of A^{-1} can be found by solving $Ax = e^{(i)}$. For finding all the columns of the inverse, either forward or backward

substitution as described above would require $p^2(p+1)/2$ FLOPS. However, there are substantial additional savings if special procedures are used. If the first k components of b are 0, then the first k components of the solution x of a lower triangular system will also be 0, so they need not be computed explicitly. Thus the forward substitution algorithm for solving for the inverse of a lower triangular matrix can be implemented with

$$\sum_{i=1}^{p} i(i+1)/2 = (p+2)(p+1)p/6$$

FLOPS. There is an analogous savings when solving for the inverse of an upper triangular matrix, since in this case, if the last k elements of b are 0, so are the last k elements of x.

2.2.2 Gaussian Elimination and the LU Decomposition

For a square nonsingular matrix A, the LU decomposition factors A into the product of a lower triangular matrix L and an upper triangular matrix U. This decomposition is essentially equivalent to the standard Gaussian elimination procedure for solving systems of equations.

The Gaussian elimination procedure for solving $Ax = b$ for a general nonsingular matrix A, starts by appending the right hand side b as an extra column to the matrix A, giving

$$C = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1p} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2p} & b_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{p1} & a_{p2} & \cdots & a_{pp} & b_p \end{pmatrix}.$$

Denote the ith row of C by c_i'. The next step is to reduce the A portion of this matrix to upper triangular form using elementary row operations. That is, first replace c_2' by $c_2' - (a_{21}/a_{11})c_1'$, replace c_3' by $c_3' - (a_{31}/a_{11})c_1'$, continuing in this manner through the pth row. The resulting matrix is

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1p} & b_1 \\ 0 & a_{22} - a_{12}a_{21}/a_{11} & \cdots & a_{2p} - a_{1p}a_{21}/a_{11} & b_2 - b_1a_{21}/a_{11} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & a_{p2} - a_{12}a_{p1}/a_{11} & \cdots & a_{pp} - a_{1p}a_{p1}/a_{11} & b_p - b_1a_{p1}/a_{11} \end{pmatrix}.$$

Then multiples of the second row of this modified matrix are subtracted from rows $3, \ldots, p$, to zero out the subdiagonal elements in the second column, and the process is continued until the A portion of the matrix has zeros for all subdiagonal matrices. (At each stage, the elements as modified by previous stages are used, so the formulas become progressively more complex.) These operations give an equivalent system of equations to (2.1), which still has the same solution vector x.

Once the matrix is reduced to upper triangular form, the process could be continued starting from the last row and working back up, eventually reducing the A portion of the matrix to an identity matrix. The modified column b at the end would then be the solution to (2.1). Alternately, once A is reduced to an upper triangular matrix, the backward substitution algorithm for upper triangular systems can be applied. It turns out these two approaches are identical.

The reduction of the system of equations to upper triangular form described above can also be thought of as factoring the matrix A into the product of a lower triangular matrix L and an upper triangular matrix U. The matrix U is precisely the matrix left in the A portion of the matrix above after it has been reduced to upper triangular form. The sub-diagonal elements of the matrix L simply record the multipliers used at each stage of the Gaussian elimination procedure, and the diagonal elements of L are 1.

To give explicit formulas for the matrices U and L, let $L = (l_{ij})$, with $l_{ij} = 0$ for $j > i$ and $l_{ii} = 1$, and $U = (u_{ij})$ with $u_{ij} = 0$ for $i > j$. Recursive formulas for the other elements of L and U are as follows. For $j = 1, 2, \ldots, p$ (in that order), compute both

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}, \quad i = 1, \ldots, j,$$

and then

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right), \quad i = j+1, \ldots, p.$$

The l_{ik} and u_{kj} needed at each step have already been computed. Also, once a component u_{ij} or l_{ij} has been computed, the corresponding a_{ij} is no longer required, so the upper triangle of U and the sub-diagonal elements of L can overwrite A. The resulting matrices then satisfy $LU = A$, which can be verified directly (although with some rather tedious algebra). This factorization is called the LU decomposition. The standard approach to Gaussian elimination is to first perform this decomposition on A, and then solve the system for x. The particular set of calculations above for the decomposition is known as Crout's algorithm.

Once the LU decomposition is computed, it is still necessary to solve the system $LUx = b$. First forward substitution can be used to solve for y in

$$Ly = b,$$

and then backward substitution can be used to solve for x in

$$Ux = y,$$

giving the solution to $Ax = b$.

Why perform the decomposition first and then solve the equations? First, no additional computations are needed, since applying the forward and backward substitution algorithms to b is the same as the calculations done when b is carried along in the standard Gaussian elimination procedure. Also, once the decomposition is computed, solutions for any number of right-hand side vectors b can be computed from the factored matrix. Further, b need not be available at the time A is factored. The factored form also provides easy ways to compute related quantities. For example, the determinant of A is given by the product of the diagonal elements of U, and each of the columns of A^{-1} can be calculated by taking b to be the corresponding column of the $p \times p$ identity matrix.

Small diagonal elements u_{ii} can cause numerical instability in the solutions obtained using this algorithm. If the order of the equations in (2.1) is permuted, the system of equations remains

exactly the same, and the solution remains the same, but the diagonal elements in the decomposition will change. For this reason the above algorithm is modified to permute the rows at each stage to make the diagonal element the largest of those possible for the remaining rows. That is, u_{ii} and the numerators of the l_{ki} for $k > i$ are computed, and if u_{ii} is not the largest of these then the ith row is swapped with the row that has the larges l_{ki}. This procedure is known as partial pivoting. The algorithm must keep track of the permutations, so they can be applied later to the right-hand side vectors b.

A (row) permutation matrix is a matrix P such that the product PA is the matrix A with its rows permuted. In general P is an identity matrix with its rows permuted. If A has 3 rows, then

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

is a permutation matrix that swaps the first and second rows of A. Permutation matrices can be stored as a vector giving the permutation of the row indices, so the full matrix need not be stored. When partial pivoting is used (as it should be), the LU decomposition represents A as $A = P^{-1}LU$, where P is the row permutation matrix recording the row interchanges due to partial pivoting. Although a literal interpretation would be that this is the product of 3 $p \times p$ matrices, as indicated above U and the subdiagonals of L (all that is needed since the diagonal of L consists of 1's) can be written in a single $p \times p$ matrix, and the permutations can be stored in an integer vector, so only slightly more than a single $p \times p$ matrix is needed for storage.

The number of FLOPS needed to compute the LU decomposition is approximately $p^3/3$ (the implementation in Press *et. al.* (1992) appears to use about $p^3/3 + 3p^2/2 + p/6$). Solving for a single right hand side then requires roughly an additional p^2 FLOPS ($p^2/2$ each for the forward and backward substitution parts of the algorithm), which as p gets larger is negligible. Solving for p right hand side vectors generally takes an additional p^3 FLOPS, but in the special case of solving for A^{-1}, if special procedures are used in the forward substitution algorithm for the leading zeros in the right hand side vectors, then the total count is reduced to approximately $p^3/6$ for forward substitution plus $p^3/2$ for backward substitution (plus $p^3/3$ for the decomposition). Thus a matrix inverse can be computed using roughly p^3 FLOPS (including those needed in the LU decomposition).

Splus has 2 types of matrix objects, as described in Venables and Ripley (1997), Section 2.9, and in the Splus on-line documentation. Standard matrix objects are created with the command `matrix()`. The functions for these matrices do not include an implementation of the LU decomposition (the `solve()` function uses a QR decomposition; see Section 2.6, below). The second type of matrix objects require attaching a separate library with the command `library(Matrix)`. This library includes a function `Matrix()` that creates a different type of Matrix object. The Matrix library creates special classes for diagonal, upper and lower triangular, row and column permutation, and Hermitian matrices (for real matrices, Hermitian is equivalent to symmetric). There are special methods written for these classes. For example, a `Diagonal` matrix will only have the vector of diagonal elements stored, and has a special method `solve.Diagonal()` for efficiently solving diagonal systems of equations. Unfortunately, the classes in the Matrix library do not have special methods for multiplication. Instead the matrix multiplication function expands a matrix with special storage structure (such as `Diagonal` or

`RowPermutation`) into the full $p \times p$ matrix, and performs the ordinary $O(p^3)$ matrix multiplication, even though there are much more efficient ways to perform the calculation. Many of the functions in the Matrix library rely on FORTRAN code from LAPACK for the main computations. However, due to the overhead in checking classes and so on in the Matrix functions, most are not particularly efficient, and there may be little advantage to using them over the ordinary matrix functions in many applications. They are used here to illustrate various computations that are not available in the regular functions.

(If `w` is a vector containing the diagonal elements of a diagonal matrix, to form the product of the diagonal matrix and another matrix `A`, do not use the Splus commands `diag(w) %*% A` (in the ordinary matrix commands) or `Diagonal(w) %*% A` in the Matrix library, since they require p^3 multiplications (assuming `A` is $p \times p$). Instead use `w*A`, which requires only p^2 multiplications. To perform the multiplication with the diagonal matrix on the right, the form `t(w*t(A))` can be used.)

The Matrix library includes a function `lu()` which will compute the LU decomposition of a general square matrix. (It also has a separate method for computing a factorization of a symmetric [Hermitian] matrix.) The function `solve()` can then be used to solve a system of equations using the factored matrix. (`solve()` is actually a generic function. It calls appropriate methods for different classes of Matrix objects.) If the factored matrix is only needed for computing one set of solutions, then `solve()` can be called with the unfactored matrix and it will perform the factorization internally. There are a variety of other functions for performing operations with the factored matrix, such as `facmul()`, with extracts one of the factors P, L, or U, and multiplies it times another vector or matrix. Here is a simple illustration.

Example 2.1
```
> A <- Matrix(c(2,1,1,-1),2,2)
> A.lu <- lu(A)
> expand(A.lu)
$l:
     [,1] [,2]
[1,]  1.0    0
[2,]  0.5    1
attr($l, "class"):
[1] "UnitLowerTriangular" "LowerTriangular"     "Matrix"

$u:
     [,1] [,2]
[1,]    2  1.0
[2,]    0 -1.5
attr($u, "class"):
[1] "UpperTriangular" "Matrix"

$permutation:
[1] 2
attr($permutation, "class"):
[1] "Identity" "Matrix"
```

```
attr(, "class"):
[1] "expand.lu.Matrix"
> facmul(A.lu,'L',c(1,2))
     [,1]
[1,]  1.0
[2,]  2.5
attr(, "class"):
[1] "Matrix"
> facmul(A.lu,'P',c(1,2))
     [,1]
[1,]    1
[2,]    2
attr(, "class"):
[1] "Matrix"
> A.s <- solve(A.lu,c(1,2))
> A.s
     [,1]
[1,]    1
[2,]   -1
attr(, "class"):
[1] "Matrix"

other attributes:
[1] "rcond" "call"
> attributes(A.s)
$dim:
[1] 2 1

$dimnames:
$dimnames[[1]]:
character(0)

$dimnames[[2]]:
character(0)

$class:
[1] "Matrix"

$rcond:
[1] 0.3333333

$call:
solve.lu.Matrix(a = A.lu, b = c(1, 2))
```

□

The attribute `rcond` of the solution above is an estimate of the reciprocal of the condition number of the matrix, which is a measure of how close to singular the matrix is. Values of `rcond` close to 1 are good, and values close to 0 indicate singular or nearly singular matrices. The value 0.33 above indicates a well-conditioned matrix. Condition numbers will be discussed in more detail below.

Example 2.2 Suppose a vector parameter $\theta = (\theta_1, \ldots, \theta_p)'$ is estimated by solving the (nonlinear) equations

$$U_i(\theta; \mathbf{X}) = 0, \quad i = 1, \ldots, p,$$

where $\mathbf{X}$ represents the data. Let $\hat{\theta}$ be the estimate, and set

$$g_{ij} = \partial U_i(\hat{\theta}; \mathbf{X})/\partial\theta_j, \quad G = (g_{ij}),$$

$$v_{ij} = \mathrm{Cov}[U_i(\theta; \mathbf{X}), U_j(\theta; \mathbf{X})]|_{\theta=\hat{\theta}}, \quad \text{and} \quad V = (v_{ij}).$$

Let $h(\theta)$ be a parametric function, and

$$b = \left(\frac{\partial h(\hat{\theta})}{\partial\theta_1}, \ldots, \frac{\partial h(\hat{\theta})}{\partial\theta_p}\right)'.$$

Subject to regularity conditions, a large sample estimator of the variance of the estimator $h(\hat{\theta})$ is given by

$$b'G^{-1}V(G^{-1})'b. \tag{2.3}$$

Note that G need not be symmetric. It might be necessary to compute (2.3) for many different values of b.

There are many different ways the calculations for computing (2.3) could be organized. Consider the following options.

1. Compute G^{-1} using the LU decomposition algorithm described above. Then use matrix multiplication to calculate $B = G^{-1}V$, and a second multiplication to compute $H = B(G^{-1})'$. Finally, compute (2.3) from

$$\sum_{i=1}^{p} b_i^2 h_{ii} + 2\sum_{j=2}^{p} b_j \left(\sum_{i=1}^{j-1} b_i h_{ij}\right). \tag{2.4}$$

2. First compute the LU decomposition of G. Then calculate $B = G^{-1}V$ by solving the systems $Gx = v_j$, where v_j is the jth column of V. The solution corresponding to v_j is the jth column of B. Next note that $GH = B'$, so the jth column of H can be calculated by solving the system $Gx = y_j$, where y_j is the jth column of B'. Once H has been calculated, $b'Hb$ is again computed from (2.4).

3. First compute the LU decomposition of G'. Then for each vector b, compute $(G')^{-1}b$ by solving $G'x = b$, and then compute (2.3) by calculating $x'Vx$, using a formula analogous to (2.4).

In option 1, computing the LU decomposition plus the calculations involved in solving for the inverse require about p^3 FLOPS. The product $B = G^{-1}V$ requires another p^3 multiplications. Since H is known to be symmetric, only $p(p+1)/2$ of its p^2 elements need to be computed, so the product $B(G^{-1})'$ requires only $p^2(p+1)/2$ multiplications. Thus calculating H in this way requires about $5p^3/2$ FLOPS. The matrix H only needs to be computed once. Formula (2.4) requires

$$2p + 1 + \sum_{j=2}^{p}(j - 1 + 1) = 2p + p(p+1)/2$$

multiplications for computing $b'Hb$. This takes advantage of the symmetry of H. If instead Hb were computed first, then that would require p^2 multiplications, and then an additional p would be required for the inner product of b and Hb, giving a total of $p^2 + p$ FLOPS, which is nearly twice as large as for (2.4). If $b'Hb$ needs to be computed for m vectors b, then given H, the number of multiplications needed is about $mp^2/2$. Thus the total computational complexity of this algorithm is roughly $5p^3/2 + mp^2/2$ FLOPS.

For option 2, the LU decomposition requires $p^3/3$ FLOPS, solving for B requires about p^3 FLOPS, and solving for H requires roughly $2p^3/3$ FLOPS, giving a total of about $2p^3$. The reason solving for H requires only $2p^3/3$, is that since H is known to be symmetric, solutions do not need to be calculated for all values in the final backward substitution algorithm (solving for the needed values can be done with about $p(p+1)(p+2)/6$ FLOPS, instead of the usual $p^2(p+1)/2$). Comparing with option 1, computing H by explicitly computing the inverse of G and multiplying requires about $p^3/2$ more FLOPS than directly solving for the matrix products from the factored version of G. This is true quite generally. When a matrix inverse appears in a formula, it is almost always better to factor the matrix and solve for the products, rather explicitly compute the inverse and multiply. The latter option also tends to be less numerically stable. Thus in this example option 2 is superior to option 1. Since the final step in option 2 is the same as for option 1, the total number of FLOPS required in option 2 is roughly $2p^3 + mp^2/2$.

For option 3, the LU decomposition of G' again requires about $p^3/3$ FLOPS. For each b, solving for x in $G'x = b$ then requires roughly p^2 FLOPS, and computing $x'Vx$ as in (2.4) requires roughly $p^2/2$ FLOPS. Thus the total number of FLOPS required is roughly $p^3/3 + 3mp^2/2$. This is smaller than for option 2 if $m < 5p/3$, and is larger than for option 2 when $m > 5p/3$, so the values of m and p determine which algorithm is faster. Another consideration could be that in option 2, once H is computed, the LU decomposition of G and V are no longer needed, and do not need to be retained, while in option 3 both V and the LU decomposition of G are needed throughout. □

To further emphasize the point that matrix inverses should not be explicitly computed, suppose as part of some computation it is necessary to compute $A^{-1}b_j$ for $j = 1, \ldots, m$. The b_j might be columns of a matrix, distinct vectors, or some combination. Computing A^{-1} requires roughly p^3 FLOPS, and each multiplication $A^{-1}b_j$ requires p^2 FLOPS, so the total is $p^3 + mp^2$. If the quantities $A^{-1}b_j$ are computed by repeatedly solving $Ax = b_j$, then the initial LU decomposition requires roughly $p^3/3$ FLOPS, and the forward and backward substitution algorithms for computing each solution together require roughly p^2 FLOPS, so the total is $p^3/3 + mp^2$. Again computing the inverse and multiplying tends to require more computation and is generally less stable than repeatedly solving for the $A^{-1}b_j$ from the factored form of A.

2.2.3 The Choleski Decomposition

The LU decomposition above can always be applied to square, non-singular matrices. However, for matrices with special features it is often possible to give alternate methods that perform better. If the coefficient matrix A is symmetric and positive definite, then A can be factored as

$$A = U'U,$$

where U is upper triangular. The additional condition that the diagonal elements of U be positive is sufficient to uniquely determine U. Recursive formulas for U are

$$u_{11} = a_{11}^{1/2}, \quad u_{1j} = a_{1j}/u_{11}, \quad j = 2, \ldots, p,$$

and proceeding row by row,

$$u_{ii} = \left(a_{ii} - \sum_{k=1}^{i-1} u_{ki}^2 \right)^{1/2}, \quad u_{ij} = \frac{1}{u_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} u_{ki} u_{kj} \right), \quad j = i+1, \ldots, p.$$

Done in the proper order, the elements of U can overwrite the upper triangle of A. The number of FLOPS required is

$$\sum_{i=1}^{p} [(i-1) + i(p-i)] = p(p+1)(p+2)/6 - p \doteq p^3/6,$$

plus the computations required for p square-root calculations. The square-root calculations are much slower than multiplications, but since the number of square-root calculations is $O(p)$, while the number of multiplications is $O(p^3/6)$, the square-root calculations are usually an insignificant part of the computational burden, unless p is small. Thus the number of computations required for the Choleski factorization is about half that for the LU decomposition.

Once the Choleski decomposition is calculated, the solution to $Ax = b$ can be obtained by applying forward substitution to solve $U'y = b$ followed by backward substitution to solve $Ux = y$. As before each of these steps requires roughly $p^2/2$ FLOPS.

Once the Choleski decomposition has been calculated, to compute $b'A^{-1}b$, as in (2.2), requires roughly $p^2/2$ FLOPS for the forward substitution algorithm to solve $U'x = b$, plus an additional p for the inner product calculation $x'x$. Computing $b'A^{-1}d$ requires roughly p^2 FLOPS, since either $A^{-1}b$ needs to be calculated using both forward and backward substitution (followed by the inner product calculation $b'(A^{-1}d)$), or both $U'x = d$ and $U'y = b$ need to be solved (followed by calculation of $x'y$).

In Splus, there is a function `chol()` for computing the Choleski factor for regular matrix objects (and curiously a separate function `choleski()`). Within the standard matrix functions there is also a function `backsolve()` that can be used to solve upper triangular systems. The Matrix library does not have a function for computing the Choleski decomposition of a Matrix object. This may be partly because it is difficult to determine whether a matrix is positive definite without computing some form of decomposition of the matrix. The `chol()` function appears to work correctly on Matrix objects, but to continue using Matrix library routines on the output of `chol()`, the class must be redefined appropriately. Also, there are no special methods in the

library for Choleski decomposition objects, which may make some computations slightly more cumbersome.

As already noted, the `lu()` function does have a special method for symmetric (Hermitian) indefinite matrices. A symmetric indefinite matrix can be factored in the form TBT', where T is lower triangular with 1's on the main diagonal, and B is block diagonal with block sizes of 1 or 2. For a positive definite matrix, B will be an ordinary diagonal matrix. The Choleski factor U above can then be obtained from the output of `lu()` through $U' = TB^{1/2}$. One way to do this is implemented in the functions `Choleski.lu()` and `Choleski.lu2()` in the example below (the only difference is in how they handle multiplication by a diagonal matrix). Unfortunately obtaining the Choleski factor from the output of `lu()` is not very efficient, even when multiplication by the diagonal factor is done properly (a safer but even slower variation is given in the function `Choleski()` on page 60 of Venables and Ripley). The symmetric indefinite factorization is more complicated, and details will not be given, since it arises less frequently in statistical applications (for more information on the output of `lu()` in this case see `help(lu.Hermitian)`). Since obtaining the Choleski decomposition from the output of `lu()` is substantially slower than other methods, it seems preferable to use `chol()`, or to just work with the symmetric indefinite decomposition from `lu()`, which for most purposes is as good as the Choleski decomposition.

Example 2.3
```
> library(Matrix)
> A <- Matrix(c(2,1,1,1),2,2)
> A.c <- chol(A)   # works, even though Matrix instead of matrix
> class(A.c) <- c('UpperTriangular','Matrix')  # define the class so
> ## solve() will use special methods
> A.c
         [,1]      [,2]
[1,] 1.414214 0.7071068
[2,] 0.000000 0.7071068
attr(, "class"):
[1] "UpperTriangular" "Matrix"

other attributes:
[1] "rank"
> x1 <- solve(t(A.c),c(1,2)) # forward substitution for lower triangular
> x1 <- solve(A.c,x1)   # backward substitution for upper triangular
> x1
     [,1]
[1,]   -1
[2,]    3
attr(, "class"):
[1] "Matrix"

other attributes:
[1] "rcond" "call"
> attributes(x1)
$dim:
```

```
[1] 2 1

$dimnames:
$dimnames[[1]]:
character(0)

$dimnames[[2]]:
character(0)


$class:
[1] "Matrix"

$rcond:
[1] 0.3333333



$call:
solve.UpperTriangular(a = A.c, b = x1)

> ## rcond above is the reciprocal condition number of A.c, not the reciprocal
> ## condition number of A (rcond(A)=rcond(A.c)^2)
> x2 <- solve(A,c(1,2)) # check the solution
> x2
     [,1]
[1,]   -1
[2,]    3
attr(, "class"):
[1] "Matrix"

other attributes:
[1] "rcond" "call"
> attr(x2,'rcond')
[1] 0.1111111
> rcond(A.c)^2
[1] 0.1111111
>
>
> Choleski.lu <- function(x) { # function to use lu.Hermitian()
+   if (!inherits(x,'Hermitian')) stop('x must be a Hermitian Matrix')
+   x <- lu(x)
+   x <- facmul(x,'T') %*% sqrt(facmul(x,'B'))
+   class(x) <- c('LowerTriangular','Matrix')
+   t(x)
+ }
```

```
>
> Choleski.lu2 <- function(x) { # faster function to use lu.Hermitian()
+   if (!inherits(x,'Hermitian')) stop('x must be a Hermitian Matrix')
+   x <- lu(x)
+   x <- t(facmul(x,'T'))*sqrt(c(facmul(x,'B')))
+   class(x) <- c('UpperTriangular','Matrix')
+   x
+ }
> # start with a large pd matrix and compare timings
> u <- Matrix(runif(40000),nrow=200)
> uu <- t(u) %*% u
> class(uu) <- c('Matrix')
> unix.time(lu(uu)) # standard LU decomposition
[1] 0.4900002 0.0000000 1.0000000 0.0000000 0.0000000
> unix.time(lu(uu)) # standard LU decomposition
[1] 0.5 0.0 1.0 0.0 0.0
> class(uu) <- c('Hermitian','Matrix')
> unix.time(lu(uu)) # symmetric indefinite (calls lu.Hermitian())
[1] 0.3099999 0.0000000 1.0000000 0.0000000 0.0000000
> unix.time(lu(uu)) # symmetric indefinite (calls lu.Hermitian())
[1] 0.3199997 0.0000000 1.0000000 0.0000000 0.0000000
> unix.time(chol(uu)) # regular Choleski (class need not be defined)
[1] 0.2200003 0.0200001 1.0000000 0.0000000 0.0000000
> unix.time(chol(uu)) # regular Choleski (class need not be defined)
[1] 0.2299995 0.0000000 0.0000000 0.0000000 0.0000000
> unix.time(Choleski.lu(uu)) #Choleski from symmetric indefinite
[1] 1.18000031 0.00999999 1.00000000 0.00000000 0.00000000
> unix.time(Choleski.lu(uu)) #Choleski from symmetric indefinite
[1] 1.17999935 0.00999999 2.00000000 0.00000000 0.00000000
> unix.time(Choleski.lu2(uu)) #faster Choleski from symmetric indefinite
[1] 0.46000004 0.01999998 1.00000000 0.00000000 0.00000000
> unix.time(Choleski.lu2(uu)) #faster Choleski from symmetric indefinite
[1] 0.4699993 0.0000000 0.0000000 0.0000000 0.0000000
> range(chol(uu)-Choleski.lu(uu)) #check if results are the same
[1] -3.667344e-13  2.220675e-12
> range(chol(uu)-Choleski.lu2(uu)) #check if results are the same
[1] -3.667344e-13  2.220675e-12
> ## almost 0
> range(chol(uu))
[1] -1.334690  7.851741
```

The first two values in the output of `unix.time()` are the user and system cpu times used by the Splus process while executing the command. The user time may be the most relevant to comparing computational algorithms. The LU decomposition should take about twice as long as the Choleski decomposition, and above it takes a little longer than that. The difference may in part be that `lu()` is doing some extra calculations, such as computing a norm for the Matrix.

The symmetric indefinite decomposition is substantially faster than the LU decomposition, but is still slower than direct calculation of the Choleski decomposition in the `chol()` function. `Choleski.lu()`, which performs an $O(p^3)$ matrix multiplication to rescale the triangular factor from `lu()`, is extremely slow. The faster version `Choleski.lu2()` is still noticeably slower than just computing the symmetric indefinite factorization. □

Next is an illustration of computing $b'A^{-1}b$ for a positive definite matrix A. Some effort is required in Splus to invoke a routine for performing forward substitution to solve a lower triangular system.

Example 2.4
```
> library(Matrix)
> u <- Matrix(runif(16),4,4)
> A <- crossprod(u)
> b <- runif(4)
> C <- chol(A)
> class(C) <- c('UpperTriangular','Matrix') # done so the next solve()
> # command invokes a routine for lower triangular matrices.
> sum(solve(t(C),b)^2) # b'A^{-1}b
[1] 1622.994
> #check
> sum(b*solve(A,b))
[1] 1622.994
> # if need to compute for many vectors b, stored as columns in B
> B <- Matrix(runif(40),nrow=4)
> t(c(1,1,1,1)) %*% (solve(t(C),B)^2)
         [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]     [,8]
[1,] 67.73769 8676.919 3522.208 72.83364 7164.785 5261.729 32.75261 395.1495
         [,9]    [,10]
[1,] 5394.538 6.435748
attr(, "class"):
[1] "Matrix"
```

□

2.3 Matrix Norms, Condition Numbers, and Error Analysis

Matrix norms play an important role in error analysis for solutions of linear systems. A norm is a function that in some sense measures the magnitude of its argument. For real numbers, the usual norm is the ordinary absolute value function. For vectors $x = (x_1, \ldots, x_p)'$, three common norms are the 1-norm (or L_1 norm) defined by

$$\|x\|_1 = \sum_{i=1}^{p} |x_i|,$$

the 2-norm (or L_2 norm or Euclidean norm) defined by

$$\|x\|_2 = \left(\sum_{i=1}^{p} x_i^2 \right)^{1/2},$$

and the ∞-norm (or L_∞ norm or sup norm) defined by

$$\|x\|_\infty = \max_i |x_i|.$$

There are many ways these norms could be generalized to matrices. The most useful turns out to be to define the corresponding matrix norms from the definitions of the vector norms through

$$\|A\|_j = \sup_{x \neq 0} \|Ax\|_j / \|x\|_j, \quad j = 1, 2, \infty.$$

(This definition is equivalent to $\sup_{\|x\|_j=1} \|Ax\|_j$, and since the set $\{x : \|x\|_j = 1\}$ is compact, the sup is attained for some vector x.)

For the 1-norm,

$$\|A\|_1 = \max_j \sum_i |a_{ij}|, \tag{2.5}$$

which is the maximum of the vector 1-norms of the columns of A. To see this, suppose A is $n \times p$, again let $e^{(i)}$ be defined by $e^{(i)}_i = 1$ and $e^{(i)}_j = 0$ for $j \neq i$, and let $\mathcal{E} = \{e^{(1)}, \ldots, e^{(p)}\}$. Note $\|e^{(i)}\|_1 = 1$. Thus

$$\sup_{x \neq 0} \|Ax\|_1 / \|x\|_1 \geq \max_{x \in \mathcal{E}} \|Ax\|_1 / \|x\|_1 = \max_j \sum_i |a_{ij}|. \tag{2.6}$$

Also, for any $x \neq 0$, setting $w_j = |x_j| / \sum_j |x_j|$,

$$\frac{\|Ax\|_1}{\|x\|_1} = \frac{\sum_i |\sum_j a_{ij} x_j|}{\sum_j |x_j|} \leq \sum_i \sum_j |a_{ij}| w_j \leq \sum_j w_j \max_k \sum_i |a_{ik}| = \max_k \sum_i |a_{ik}|,$$

since $\sum_j w_j = 1$. Since this holds for any $x \neq 0$,

$$\sup_{x \neq 0} \|Ax\|_1 / \|x\|_1 \leq \max_k \sum_i |a_{ik}|.$$

Combining this inequality with (2.6) gives the result.

Similarly, it can be shown that

$$\|A\|_\infty = \max_i \sum_j |a_{ij}| = \|A'\|_1. \tag{2.7}$$

Exercise 2.1 Derive (2.7).

It can also be shown that $\|A\|_2$ is the largest singular value of the matrix A; see equation (2.25), below. Since the singular values are more difficult to compute, error analysis of linear systems tends to focus more on the 1-norm and ∞-norm. The Splus Matrix library function `norm(A,type='1')` will compute the 1-norm of a Matrix, and `norm(A,type='I')` will compute the ∞-norm. (The function `norm()` is a generic function which calls the function `norm.Matrix()` for objects of class Matrix. The default is `type='M'`, which just computes $\max_{ij} |a_{ij}|$, which is less useful. There are many other definitions of matrix norms which could be considered, but the three given above are the most commonly used.)

From the definitions of the norms, it is clear that for any matrix A and vector x,

$$\|A\|_j\|x\|_j \geq \|Ax\|_j. \tag{2.8}$$

The *condition number* of a square matrix with respect to one of the norms $\|\cdot\|_j$ is defined to be

$$\kappa_j(A) = \|A^{-1}\|_j\|A\|_j,$$

with $\kappa_j(A) = \infty$ if A is singular. For the 2-norm it can be shown that the condition number is the ratio of the largest singular value to the smallest singular value; see Section 2.7. Another important relationship for estimating condition numbers is that

$$\|A^{-1}\|_j = \left(\inf_{x\neq 0} \|Ax\|_j/\|x\|_j\right)^{-1}. \tag{2.9}$$

For nonsingular A this follows because the set $\{y \neq 0\} = \{Ax : x \neq 0\}$, so

$$\|A^{-1}\|_j = \sup_{y\neq 0} \frac{\|A^{-1}y\|_j}{\|y\|_j} = \sup_{\{y=Ax:x\neq 0\}} \frac{\|A^{-1}y\|_j}{\|y\|_j} = \sup_{x\neq 0} \frac{\|A^{-1}Ax\|_j}{\|Ax\|_j} = \sup_{x\neq 0} \frac{\|x\|_j}{\|Ax\|_j} = \left(\inf_{x\neq 0} \frac{\|Ax\|_j}{\|x\|_j}\right)^{-1}.$$

From

$$\kappa_j(A) = \left(\sup_{x\neq 0} \|Ax\|_j/\|x\|_j\right)\left(\inf_{x\neq 0} \|Ax\|_j/\|x\|_j\right)^{-1}, \tag{2.10}$$

it follows that the condition number is ≥ 1. If A is singular then there is an $x \neq 0$ such that $Ax = 0$, so the denominator of the expression above is 0, and this formula is infinite, consistent with the original definition. The condition number of a permutation matrix P is 1, because for a permutation matrix $\|Px\|_j = \|x\|_j$. For orthogonal matrices the 2-norm condition number is 1 (see Section 2.4, below), but in general this is not true for other norms. Also, note that if $c \neq 0$ is a real number, then $\kappa_j(cA) = \kappa_j(A)$, so the condition number is not related to the overall magnitude of the elements of A.

Let $\hat{x}$ be the computed solution to $Ax = b$, and let x_0 be the exact solution. A measure of the relative error in the solution is

$$\|\hat{x} - x_0\|/\|x_0\|,$$

where the specific norm has not been indicated. In error analysis of algorithms, it is often most convenient to use $\|\cdot\|_\infty$, but it appears the `solve()` functions in the Splus Matrix library use $\|\cdot\|_1$. Let $\hat{b} = A\hat{x}$. Then

$$\|\hat{x} - x_0\| = \|A^{-1}(\hat{b} - b)\| \leq \|A^{-1}\|\|\hat{b} - b\| = \kappa(A)\frac{\|\hat{b} - b\|}{\|A\|},$$

using (2.8) and the definition of the condition number. Since $\|b\| = \|Ax_0\| \leq \|A\|\|x_0\|$, it follows that

$$\frac{\|\hat{x} - x_0\|}{\|x_0\|} \leq \kappa(A)\frac{\|\hat{b} - b\|}{\|A\|\|x_0\|} \leq \kappa(A)\frac{\|\hat{b} - b\|}{\|b\|}. \tag{2.11}$$

This relation bounds the relative error in the solution in terms of the condition number and the relative error of $A\hat{x}$ as an estimate of b. For most reasonable algorithms in most problems,

$\|\hat{b} - b\|/\|b\|$ is a small multiple of the relative machine precision ϵ_m (arguments to establish this for particular algorithms often involve a form of backward error analysis, often working with the ∞-norm). Alternately, although $\hat{b}$ cannot be calculated exactly, it could be estimated using a computed approximation to $A\hat{x}$. If $\|\hat{b} - b\|/\|b\| = O(\epsilon_m)$, and if $\kappa(A) = O(1/\epsilon_m)$, then there may be no significant digits in the computed solution (recall $1/\epsilon_m \doteq 10^{16}$ for double precision). If $\kappa(A) = O(1/\epsilon_m^{1/2})$, then the computed solution (in double precision) could have about 8 significant digits. More generally, if $\|\hat{b} - b\|/\|b\| \doteq 10^{-q}$, then the number of significant digits in the computed solution $\hat{x}$ is about $q - \log_{10}[\kappa(A)]$.

The value of the condition number is needed to make use of the bound (2.11). The norm of A is easily computed for the 1- and ∞-norms. Unfortunately calculating the 1- or ∞-norm of A^{-1} requires computing A^{-1} first. This requires an extra $2p^3/3$ FLOPS, so is not an attractive option. Algorithms have been developed for estimating the norm of A^{-1}, based on formula (2.9). The concept behind these algorithms is to find a vector x for which $\|Ax\|/\|x\|$ is close to its minimum, using only $O(p^2)$ FLOPS. Details are given in Golub and van Loan (1989), Section 3.5.4. The function `rcond()` in Splus calculates one such approximation to the reciprocal condition number (the reciprocal is used to avoid overflow for singular matrices). The LAPACK library also has routines for estimating condition numbers (and in fact the Splus function ultimately relys on these routines for its computations). Many of the `solve()` methods in the Splus Matrix library automatically return estimates of condition numbers.

The bound (2.11) expresses the relative error in the solution in terms of the relative error in the estimated $\hat{b}$. The relative error in $\hat{b}$ is usually small, but if it is not it can often be made very small using a technique called *iterative refinement*. The idea is to first compute the solution $\hat{x}$ as usual. Then the system

$$Ay = A\hat{x} - b$$

is solved giving a solution $\hat{y}$, and a new solution to the original problem, $\hat{x}^{(2)} = \hat{x} - \hat{y}$, computed. Then the system

$$Ay = A\hat{x}^{(2)} - b$$

is solved giving a new value $\hat{y}$, and a new solution to the original problem, $\hat{x}^{(3)} = \hat{x}^{(2)} - \hat{y}$, computed. This process is continued until the changes become small. The relative error in $A\hat{y}$ at each step should be similar to that in the original problem, but the magnitude of $A\hat{y}$ should be decreasing, so the precision of $A\hat{x}^{(j)}$ should be improving. More details on this procedure are given in Section 2.5 of Press *et. al.* (1992), and in Section 3.5.3 of Golub and van Loan (1989). Note that this procedure need not improve the accuracy of $\hat{x}$, but it does tend to reduce the computed difference $A\hat{x} - b$. This procedure could improve the precision of $\hat{x}$ if the intermediate calculations in computing $A\hat{x}^{(k)} - b$ were carried out in extended precision. LAPACK includes routines for iterative refinement of solutions. If both A and its decomposition have been stored, then each iteration requires roughly p^2 FLOPS to update $A\hat{x}^{(j)}$ and another p^2 for the forward and backward substitutions to compute the adjustment to the solution.

Example 2.5

```
> # norms and condition numbers
> a <- Matrix(rnorm(9),3,3)
> norm(a,'1') # one-norm
[1] 2.091627
> max(t(c(1,1,1)) %*% abs(a)) # max of column sums of abs(a)
```

```
[1] 2.091627
> norm(a,'I') # infinity-norm
[1] 1.991345
> max(abs(a) %*% c(1,1,1)) # max of row sums of abs(a)
[1] 1.991345
> norm(t(a),'1')
[1] 1.991345
> norm(t(a),'I')
[1] 2.091627
> au <- lu(a)
> ai <- solve(au)
> norm(ai,'1')*norm(a,'1') # one-norm condition number
[1] 5.780239
> 1/rcond(a)   # estimated one-norm condition number
[1] 4.106333
> 1/rcond(au)  # condition number estimate much more accurate when factored
[1] 5.780239
> 1/attr(ai,'rcond') # from one-norm
[1] 5.780239
> norm(ai,'I')*norm(a,'I') # infinity-norm condition number
[1] 4.371721
> 1/rcond(a,F) # estimated infinity-norm condition number
[1] 3.66198
> b <- c(1,3,5)
> x <- solve(au,b)
> 1/attr(x,'rcond')
[1] 5.780239
> x
          [,1]
[1,]  2.452791
[2,]  1.300757
[3,] -6.688391
attr(, "class"):
[1] "Matrix"

other attributes:
[1] "rcond" "call"
> # check accuracy of ax
> range(a%*%x-b)
[1] -4.440892e-16  8.881784e-16
> norm(a%*%x-b,'1')/sum(abs(b)) # norm(b,'1') gives an error
[1] 1.480297e-16
>
> # a more singular system
> options(digits=12)
> as <- Matrix(rep(runif(4),rep(4,4)),4,4)
```

```
> diag(as) <- diag(as)+1.e-9
> bs <- apply(as,1,sum) # if this sum were exact, true sol would=c(1,1,1,1)
> as
               [,1]           [,2]           [,3]          [,4]
[1,] 0.970216591445 0.346583154984 0.933693890925 0.45553969685
[2,] 0.970216590445 0.346583155984 0.933693890925 0.45553969685
[3,] 0.970216590445 0.346583154984 0.933693891925 0.45553969685
[4,] 0.970216590445 0.346583154984 0.933693890925 0.45553969785
attr(, "class"):
[1] "Matrix"
> bs
[1] 2.7060333342 2.7060333342 2.7060333342 2.7060333342
> asu <- lu(as)
> xs <- solve(asu,bs)
> 1/attr(xs,'rcond')
[1] 4874972927.57
> xs
               [,1]
[1,] 1.000000123279
[2,] 0.999999901235
[3,] 0.999999956746
[4,] 0.999999901235
attr(, "class"):
[1] "Matrix"

other attributes:
[1] "rcond" "call"
> range(as%*%xs-bs)
[1] -4.4408920985e-16  0.0000000000e+00
> norm(as%*%xs-bs,'1')/sum(abs(bs))
[1] 4.10276921053e-17
> range(as%*%rep(1,4)-bs)
[1] 0 0
>
> # a singular system
> as <- Matrix(rep(runif(4),rep(4,4)),4,4)
> as[,1] <- as[,1]+1:4
> bs <- apply(as,1,sum) # if this sum were exact, a sol would=c(1,1,1,1)
> asu <- lu(as)
> xs <- solve(asu,bs)
> 1/attr(xs,'rcond')
[1] 8.5522178561e+17
> xs
                [,1]
[1,]  1.000000000000
[2,]  0.249450634867
```

```
[3,] 16.000000000000
[4,]  0.000000000000
attr(, "class"):
[1] "Matrix"

other attributes:
[1] "rcond" "call"
> range(as%*%xs-bs)
[1] 0.0000000000e+00 4.4408920985e-16
> norm(as%*%xs-bs,'1')/sum(abs(bs))
[1] 2.1757641057e-17
> range(as%*%rep(1,4)-bs)
[1] 0 0
```

In the next to last example (nearly singular), the estimated condition number is approximately 5×10^9 so $\kappa_1(a)\epsilon_m \doteq 5 \times 10^{-7}$, suggesting at least 6 significant digits of accuracy. The computed solution does appear to be accurate to about 7 digits. In the final example (singular), the condition number is $> 1/\epsilon_m$, suggesting no accuracy in the solution. A more precise description is that since the matrix is singular, there are many solutions. The user must decide how small the value of `rcond` can be for the solution to still be acceptable. The default is essentially 0. Note that in all cases the computed solution gave a computed $\hat{b}$ that was very close to the true b. □

There are more refined versions of error analysis that allow approximate component-wise bounds on the solution to be computed. Some LAPACK routines will compute estimates of these bounds; see the *LAPACK Users' Guide* for details.

2.4 Rotations and Orthogonal Matrices

A rotation in R^p is a linear transformation $Q : R^p \to R^p$ such that

$$\|Qx\|_2 = \|x\|_2 \tag{2.12}$$

for all $x \in R^p$. A rotation does not affect the length of vectors, but changes their orientation. Rotations can be thought of as rotating the coordinate axes, but leaving the relative orientation and lengths of the axes unchanged.

From $\|Qx\|_2 = \|x\|_2$ for all x, it follows that $x'Q'Qx = x'x$ for all x, and hence that $x'[Q'Q - I]x = 0$ for all x, which can only be true if $Q'Q = I$.

Now for square matrices, $Q'Q = I$ implies $QQ' = I$ (caution: this is not true if Q is not square). That is, if Q is square and $Q'Q = I$, then Q must have rank p (and be nonsingular). Thus any $x \in R^p$ can be represented as $x = Qy$ for some $y \in R^p$. Then for any $x \in R^p$, $QQ'x = QQ'Qy = Q(Q'Q)y = Qy = x$, so $QQ' = I$.

When $Q'Q = I$, the columns of Q are mutually orthogonal and each has unit length, and when $QQ' = I$ the rows of Q are mutually orthogonal and each has unit length. If Q is square, from the previous result, either implies that the other is true. A square matrix satisfying either of these properties is said to be *orthogonal*.

From the above discussion, any rotation is given by an orthogonal matrix, and vice versa, so rotations are also often referred to as orthogonal transformations. Note that a rotation as defined here also includes the concept of a reflection, where the direction of one axis is reversed while the others are left unchanged.

From the definition, it is clear that if Q is a rotation, then $\|Q\|_2 = 1$. Since $Q^{-1} = Q'$ is also a rotation, $\|Q^{-1}\|_2 = 1$ as well, so the 2-norm condition number of any orthogonal matrix (or rotation) is 1.

If Q_1 and Q_2 are orthogonal matrices, then $(Q_1Q_2)'(Q_1Q_2) = Q_2'Q_1'Q_1Q_2 = Q_2'Q_2 = I$, so Q_1Q_2 is also orthogonal.

If Y is a random vector with $\mathrm{Var}(Y) = \sigma^2 I$, and Q is orthogonal, then $\mathrm{Var}(QY) = \sigma^2 QQ' = \sigma^2 I$, so orthogonal transformations can be used to simplify the mean structure of a random vector while leaving the variance structure unchanged, which turns out to be useful in various contexts, especially in linear regression analysis.

Two types of rotations are of special interest. The first is a plane rotation. In R^p, a plane rotation keeps $p-2$ of the axes fixed, and rotates the plane defined by the other 2 axes through a fixed angle about the others (these are also referred to as Jacobi or Givens rotations). Suppose for simplicity the first $p-2$ coordinates are held fixed, and the plane of the last two coordinates is rotated through an angle θ. Then the matrix of a plane rotation is

$$Q_p = \left(\begin{array}{c|cc} I_{p-2,p-2} & 0_{p-2} & 0_{p-2} \\ \hline 0'_{p-2} & \cos(\theta) & \sin(\theta) \\ 0'_{p-2} & -\sin(\theta) & \cos(\theta) \end{array}\right)$$

To rotate the plane defined by the ith and jth coordinates, the cosines are put in the ii the jj elements, and the sine terms in the ij and ji elements.

Suppose x is a vector. To rotate the plane of the $p-1$ and pth coordinates so that in the new coordinate system the pth coordinate is 0, set $\sin(\theta) = x_p/(x_{p-1}^2 + x_p^2)^{1/2}$ and $\cos(\theta) = x_{p-1}/(x_{p-1}^2 + x_p^2)^{1/2}$. Then the first $p-2$ components of $Q_p x$ are the same as those of x, the $(p-1)$st component is $(x_{p-1}^2 + x_p^2)^{1/2}$, and the pth component is 0. (Note that fortunately the angle θ need not be explicitly calculated.) A series of rotations of this type can be constructed in such a way that when applied to a matrix X the resulting matrix will be upper triangular (for example).

The other family of orthogonal transformations of special interest are the Householder transformations. A Householder transformation has the form

$$H = I - \frac{2}{u'u}uu',$$

where I is the identity matrix and u is any vector (of the proper length), interpreting $H = I$ when $u = 0$. Note that $H' = H$, and $HH = I - 4uu'/u'u + 4uu'uu'/(u'u)^2 = I$, so H is a symmetric orthogonal matrix.

An important application of Householder transformations is to transform matrices to upper

triangular form. If x is an n-dimensional vector, and u is defined by

$$u_i = \begin{cases} 0, & 0 < i < t, \\ x_t + s, & i = t, \\ x_i, & t < i \le n, \end{cases} \tag{2.13}$$

where

$$s = \text{sign}(x_t) \left(\sum_{j=t}^{n} x_j^2 \right)^{1/2},$$

then

$$Hx = x - 2u(u'x)/(u'u) = x - 2u \frac{x_t^2 + x_t s + \sum_{j>t} x_j^2}{x_t^2 + 2x_t s + s^2 + \sum_{j>t} x_j^2} = x - 2u \frac{x_t s + s^2}{2x_t s + 2s^2} = x - u.$$

Thus $(Hx)_i = x_i$ for $i < t$, $(Hx)_i = 0$ for $i > t$, and $(Hx)_t = -s$. (The sign of s is chosen so x_t and s will have the same sign.) Thus the last $n - t$ components have been set to 0 in the transformation Hx. A series of such transformations can be based on the columns of a matrix in such a way as to leave the transformed matrix in upper triangular form.

The Householder transformation defined above for x, applied to another vector y, gives

$$Hy = y - 2u(u'y)/(u'u)$$

so the first $t - 1$ components of Hy are the same as y, and the other components are of the form $y_i - fu_i$, where $f = 2 \sum_{j \ge t} y_j u_j / \sum_{j \ge t} u_j^2$.

Calculating the transformation and applying it to x, and to k other vectors y, requires computing $s^* = \sum_{j=t+1}^{n} x_j^2$ ($n - t$ multiplications), computing s from s^* and x_t (1 multiplication plus one square-root), computing $2/u'u = 2/[s^* + (s + x_t)^2]$ (1 multiplication plus one division), and for each y computing the corresponding f ($k(n - t + 2)$ multiplications) and finally the components of Hy ($k(n - t + 1)$ multiplications). This gives a total of

$$(n - t + 1)(2k + 1) + k + 1 \tag{2.14}$$

multiplications plus one division and one square-root.

2.5 Linear Least Squares

A very common statistical problem is to fit a regression model

$$y_i = \beta_1 + \sum_{j=2}^{p} x_{ij} \beta_j + \epsilon_i, \tag{2.15}$$

where the y_i are the responses, the x_{ij} are the covariates, the β_j are the unknown regression coefficients, and the ϵ_i are unobserved errors assumed to be independent and identically distributed. If the errors can be assumed to have mean 0 and finite variance σ^2, then least squares estimators are often used for the β_j.

The basic linear least squares problem is to determine the vector $\beta = (\beta_1, \ldots, \beta_p)'$ that minimizes

$$\|y - X\beta\|_2^2 = (y - X\beta)'(y - X\beta), \tag{2.16}$$

where $y = (y_1, \ldots, y_n)'$ and $X_{n\times p} = (x_{ij})$ (usually with $x_{i1} = 1$ for all i, although models without a constant term also fit in this framework). Thus least squares chooses the estimator $\hat{\beta}$ that gives the vector of fitted values $\hat{y} = X\hat{\beta}$ that is closest (in the Euclidean norm) to the actual responses.

In regression analysis often many different models are fit, especially in systematic model search algorithms such as stepwise regression, so it would be useful to have fast methods for updating the fitted model when covariates are added or dropped. Also, along with computing $\hat{\beta}$, usually it is desirable to calculate quantities such as residuals, fitted values, regression and error sums of squares, and diagonal elements of the projection operator $X(X'X)^{-1}X'$ (which are useful in assessing case influence). These factors should be kept in mind when comparing possible algorithms for computing least squares estimators.

The most obvious way to find the least squares solution is to set the gradient of (2.16) equal to 0, obtaining the normal equations

$$X'X\beta = X'y.$$

This is a system of linear equations, that can be solved using any of the methods discussed previously. In particular, if $\text{rank}(X) = p$, then $X'X$ is positive definite, and the system can be efficiently solved using the Choleski decomposition. This does not, however, give an easy way to update solutions for adding and dropping variables, or for computing other derived quantities. A different method for solving the normal equations is called the *sweep operator*. The sweep operator actually is a particular way of implementing a variation on Gaussian elimination called the Gauss-Jordan algorithm. By sequentially applying the sweep operator to the augmented matrix

$$\begin{pmatrix} X'X & X'y \\ y'X & y'y \end{pmatrix},$$

the least squares estimates and residual sums of squares corresponding to models with just the first k covariates included, $k = 1, \ldots, p$, are obtained. And the operator has an inverse, so from any partial model it is easy to update the fit for adding or deleting a covariate. Thus, although it is less efficient than the Choleski factorization for fitting a single model, the sweep operator is probably used more frequently in regression analysis. For details on the sweep operator see Section 3.4 of Thisted (1988) or Chapter 7 of Lange (1999).

It turns out, though, that it is possible to calculate the least squares estimates by applying a matrix factorization directly to X, and that it is not necessary to compute $X'X$ or form the normal equations. Applying a factorization directly to X tends to be a better conditioned problem than factoring $X'X$, so direct decomposition of X is usually preferred to calculating $X'X$ and solving the normal equations. Two such decompositions will be discussed below: the QR decomposition and the singular value decomposition.

2.6 QR Decomposition

Throughout this section it will be assumed that X has full column rank p.

The motivation for using the QR decomposition to solve least squares problems comes from the property (2.12) of orthogonal matrices. For any $n \times n$ orthogonal matrix Q,

$$\|Q'y - Q'X\beta\|_2^2 = \|y - X\beta\|_2^2,$$

so a β minimizing $\|Q'y - Q'X\beta\|_2^2$ solves the original least squares problem. Suppose a Q can be found such that

$$Q'X = \begin{pmatrix} R_{p\times p} \\ \mathbf{0}_{(n-p)\times p} \end{pmatrix}, \tag{2.17}$$

where R is upper triangular and $\mathbf{0}$ is a matrix with all elements 0. Partition $Q = (Q_1, Q_2)$ with Q_1 containing the first p columns of Q and Q_2 the other columns. Then

$$\|Q'y - Q'X\beta\|_2^2 = \left\| \begin{pmatrix} Q_1'y - R\beta \\ Q_2'y \end{pmatrix} \right\|_2^2 = \|Q_1'y - R\beta\|_2^2 + \|Q_2'y\|_2^2, \tag{2.18}$$

and $\|Q_1'y - R\beta\|_2^2$ is minimized by $\hat{\beta} = R^{-1}Q_1'y$ (since then $\|Q_1'y - R\hat{\beta}\|_2^2 = 0$), which is thus the least squares estimator.

Exercise 2.2 Use (2.17) to show directly that $\hat{\beta} = R^{-1}Q_1'y$ solves the normal equations.

A transformation Q satisfying (2.17) is easy to construct using the product of Householder transformations. Let X_j be the jth column of X, $j = 1, \ldots, p$. Let H_1 be the Householder transformation (2.13) with $x = X_1$ and $t = 1$. Let $X_j^{(1)}$ be the jth column of H_1X. Then $X_1^{(1)}$ has all elements except the first equal to 0. Next let H_2 be the householder transformation (2.13) with $x = X_2^{(1)}$ and $t = 2$, and let $X_j^{(2)}$ be the jth column of H_2H_1X. Then $X_2^{(2)}$ has all elements except possibly the first 2 equal to 0. Also, note that $X_1^{(2)} = X_1^{(1)}$; that is, H_2 did not change the first column, so now the first two columns of H_2H_1X are in upper triangular form. Continuing in this fashion, at the kth stage let H_k be the transformation (2.13) with $x = X_k^{(k-1)}$ and $t = k$, and let $X_j^{(k)}$ be the j column of the matrix $H_k \cdots H_1X$. Then $X_j^{(k)} = X_j^{(k-1)}$ for $j < k$, and the first k columns of the resulting matrix are in upper triangular form. After the pth step the matrix $H_p \cdots H_1X$ is in the form (2.17), and thus $Q' = H_p \cdots H_1$ is an orthogonal transformation that transforms X into the desired form. (This is not the only way a QR decomposition can be computed; see Golub and Van Loan, 1989, Section 5.2 for other options.)

To calculate the least squares estimates, $Q_1'y$ is also needed. It can be computed by applying each of the Householder transformations to y. Then the least squares estimates are computed by using the backward substitution algorithm to solve the upper triangular system

$$R\beta = Q_1'y.$$

The Householder transformations can either be applied to y at the same time as they are applied to X, or they can be stored and applied later. The latter is particularly appropriate if least squares estimates are to be computed for the same covariates for a number of response vectors, not all of which may be available at the same time. For example, this could occur in a simulation with X fixed and different response vectors y generated for each sample.

Note that storing the Householder transformations only requires storing the u vectors in (2.13), and does not require storing $n \times n$ matrices. The QR decomposition can be done "in place", overwriting the elements of X, with only one extra p-vector of storage needed. The elements of R can be stored in the upper triangle of the first p rows of X. The other elements of X will be 0 following the transformations, and can be used to store the nonzero elements of the u vectors for each transformation, except for the first nonzero element, which is stored in the extra vector. This in place algorithm can be done column by column, as described earlier.

2.6.1 Constant Terms

Suppose the regression model contains a constant term, as in (2.15). Then the model can be reparameterized to

$$y_i = \alpha + \sum_{j=2}^{p}(x_{ij} - \overline{x}_j)\beta_j + \epsilon_i, \tag{2.19}$$

where $\alpha = \beta_1 + \sum_{j=2}^{p} \beta_j \overline{x}_j$ and $\overline{x}_j = \sum_i x_{ij}/n$. In this model $\hat{\alpha} = \overline{y}$. Subtracting $\hat{\alpha}$ from both sides of (2.19), it follows that the least squares estimates for $\beta_2, \ldots, \beta_p$ can be computed by regressing $y_i - \overline{y}$ on the covariates $x_{ij} - \overline{x}_j$. This only involves a system with $p-1$ unknowns, and often gives a much better conditioned problem.

If the first column of X is a constant, then the first Householder transformation in the QR decomposition essentially mean corrects the other terms (the actual formulas are more complicated, though). Thus the first Householder transformation in the QR decomposition automatically transforms the model as described above. In practice this is usually done explicitly, rather than by applying the general Householder computations. (That is, it is not necessary, for example, to explicitly compute the sum of squares of the elements of the first column if they are known to all be equal.)

Exercise 2.3 Suppose all elements of x equal 1, and let H_1 be the Householder transformation (2.13) formed from x with $t = 1$. Give formulas for $y_i^{(1)}$, $i = 1, \ldots, n$, the elements of $H_1 y$, and show

$$\sum_{i=1}^{n}(y_i - \overline{y})^2 = \sum_{i=2}^{n}(y_i^{(1)})^2.$$

2.6.2 Variances, Sums of Squares, and Leverage Values

The error variance σ^2 is usually estimated by

$$\hat{\sigma}^2 = \|y - X\hat{\beta}\|_2^2/(n-p).$$

If adequate memory is available, then it is probably best to keep a copy of y and X instead of overwriting them during the QR decomposition algorithm. Then once $\hat{\beta}$ has been computed, the fitted values $\hat{y} = X\hat{\beta}$, the residuals $y - \hat{y}$, and the residual sum of squares $\|y - \hat{y}\|_2^2$ can be computed directly using the X and y values. If copies of X and y are not available, the residual sum of squares can still be computed directly from the transformed versions $Q'y$ and $Q'X$ computed in the QR decomposition, since

$$\|y - X\hat{\beta}\|_2^2 = \|Q'y - Q'X\hat{\beta}\|_2^2.$$

In fact, substituting $\hat{\beta}$ in (2.18), it follows that

$$\|Q'y - Q'X\hat{\beta}\|_2^2 = \|Q_2'y\|_2^2, \tag{2.20}$$

which is just the sum of squares of the last $n-p$ elements of $Q'y$.

The variance of any linear combination $a'\hat{\beta}$ is given by $\sigma^2 a'(X'X)^{-1}a$, and the covariance of $a'\hat{\beta}$ and $d'\hat{\beta}$ is $\sigma^2 a'(X'X)^{-1}d$. Since $X = Q_1 R$, it follows that $X'X = R'Q_1'Q_1R = R'R$. (Since Q_1 consists of the first p columns of Q, $Q_1'Q_1 = I_p$. However, Q_1Q_1' is not an identity matrix.) Thus

$$a'(X'X)^{-1}a = a'(R'R)^{-1}a = \|(R')^{-1}a\|_2^2. \tag{2.21}$$

Given the stored value of R, (2.21) is easily computed by using forward substitution to solve $R'w = a$, and then computing $w'w$. Covariances can be computed similarly, by also solving $R'v = d$ and computing $w'v$.

Since R is an upper triangular matrix with $R'R = X'X$, there is a close connection between R and the Choleski decomposition of $X'X$. The only difference between R and the Choleski factor is in the signs of the elements. The Choleski factor is constrained to always have positive diagonal elements, while the diagonal elements in R can be either positive or negative. Essentially, though, the QR decomposition computes the Choleski factorization of $X'X$, without actually computing $X'X$.

The matrix $H = X(X'X)^{-1}X'$ is called the "hat" matrix, since it transforms the observations to the fitted values $\hat{y} = Hy$. The diagonal elements h_{ii} of H are called *leverage values*, and are useful in case influence diagnostics. Large leverage values indicate that the corresponding covariate vector $x_i = (x_{i1}, \ldots, x_{ip})'$, by virtue of its location in the covariate space, has potentially large influence on the least squares estimators. Since $\text{trace}(H) = p$, the average value of the h_{ii} is p/n, and values substantially larger than this may indicate potentially troublesome data points.

Since $h_{ii} = x_i'(X'X)^{-1}x_i$, if a copy of X is stored, then the h_{ii} can be computed from R and the rows of X, as described above for computing variances. If a copy of X is not stored, it follows from $X = Q_1R$ that $H = Q_1Q_1'$, so $h_{ii} = \sum_{j=1}^p q_{ij}^2$. The elements of the jth column of Q_1 can be calculated by applying the Householder transformations in reverse order to the jth column of the $n \times n$ identity matrix (that is, $Q = QI_n = H_1 \cdots H_p I_n$). The contributions to the h_{ii} from each column can be accumulated, so the full Q_1 matrix need not be stored. Since $\hat{y} = Q_1Q_1'y$, $\hat{y}$ can also be computed during this process, using the previously computed values of $Q_1'y$.

Regression packages often display a sequential regression sum of squares table as part of their output. That is, the regression sum of squares from the full model is partitioned into $S(\beta_1)$, the regression sum of squares when only the first column of X is in the model, and $S(\beta_j|\beta_1, \ldots, \beta_{j-1})$, $j = 2, \ldots, p$, the change in the regression sum of squares when each column of X is added to the model. It turns out these sequential sums of squares are automatically computed in the QR decomposition algorithm. Let $y^{(j)} = H_j \cdots H_1 y$. First note that since the H_j are orthogonal transformations, $\|y^{(j)}\|_2^2 = \|y\|_2^2$. Then applying (2.20) to the model with just j covariates, it follows that the residual sum of squares for this model is $\sum_{i=j+1}^n (y_i^{(j)})^2$, so the regression sum of squares, which is the difference between the total sum of squares and the residual sum of squares, is $\sum_{i=1}^j (y_i^{(j)})^2$. Noting that $y_i^{(j)} = y_i^{(j-1)}$ for $i < j$, it follows that the regression sum of squares for the model with the first $j-1$ terms is $\sum_{i=1}^{j-1}(y_i^{(j)})^2$, and hence $S(\beta_j|\beta_1, \ldots, \beta_{j-1}) = (y_j^{(j)})^2$. Since

these terms are not affected by the H_k for $k > j$, it then follows that after completing all p transformations, $S(\beta_j|\beta_1, \ldots, \beta_{j-1}) = (y_j^{(p)})^2$.

If the model has been mean corrected before applying the QR decomposition, then the total sum of squares above is corrected for the mean. If the model has not been mean corrected and does contain a constant as the first term, then the residual sum of squares after adding the constant is the total sum of squares corrected for the mean, and $(y_1^{(j)})^2 = n\overline{y}^2$, the difference between the corrected and uncorrected total sum of squares.

Note that an alternate method to (2.20) for computing the residual sum of squares for the model with j terms is given by the formula $\|y\|_2^2 - \sum_{i=1}^{j}(y_i^{(j)})^2$. Although this formula is algebraically equivalent, it suffers from the same instability as the naive one-pass variance algorithm, and is not recommended.

The following example illustrates the use of the QR decomposition in regression analysis.

Example 2.6

```
> library(Matrix)
> # produce data set
> .Random.seed
 [1] 57 59 11 34 56  3 30 37 36 56 52  1
> n <- 100
> p <- 5
> X <- cbind(rep(1,n),Matrix(runif(n*(p-1)),nrow=n))
> beta <- c(10,1,1,-1,-1)
> Y <- X %*% beta +rnorm(n)
> data <- data.frame(Y,X=X[,-1])
> print(fit1 <- lm(Y~X,data))
Call:
lm(formula = Y ~ X, data = data)

Coefficients:
 (Intercept)       X1        X2        X3        X4
    10.16435 1.592054 0.7286286 -1.109259 -1.110602

Degrees of freedom: 100 total; 95 residual
Residual standard error: 1.029967
> anova(fit1)
Analysis of Variance Table

Response: Y

Terms added sequentially (first to last)
          Df Sum of Sq  Mean Sq  F Value        Pr(F)
X          4   41.5843 10.39607 9.799931 1.112035e-06
Residuals 95  100.7790  1.06083
> # QR decomposition
> X.qr <- qr(X)
```

```
> print(b1 <- solve(X.qr,Y))  # computes LS solution
            [,1]
[1,]  10.1643467
[2,]   1.5920543
[3,]   0.7286286
[4,]  -1.1092590
[5,]  -1.1106021
attr(, "class"):
[1] "Matrix"

other attributes:
[1] "rcond"     "workspace" "call"
> 1/attr(b1,'rcond')
[1] 6.385741
> QtY <- facmul(X.qr,'Q',Y,transpose=T,full=F) #t(Q_*) %*% Y
> # direct calculation of parameter estimates
> solve(facmul(X.qr,'R',full=F),c(QtY))
            [,1]
[1,]  10.1643467
[2,]   1.5920543
[3,]   0.7286286
[4,]  -1.1092590
[5,]  -1.1106021
attr(, "class"):
[1] "Matrix"

other attributes:
[1] "rcond" "call"
> c(QtY)^2
[1] 10488.602249    15.025345     7.563192     8.699584    10.296167
> sum(Y^2)-QtY[1]^2  # corrected total SS; can be inaccurate
[1] 142.3632
> sum(Y^2)-sum(c(QtY)^2) # residual SS; can be inaccurate
[1] 100.779
> sum(c(QtY[-1])^2)  # regression sum of squares (accuracy should be ok)
[1] 41.58429
> Yhat <- facmul(X.qr,'Q',c(QtY),F,full=F) # Q_* %*% t(Q_*) %*% Y
> range(Yhat-X%*%b1) # previous line gave fitted values
[1] 5.329071e-15 8.348877e-14
> sum((Y-Yhat)^2) # residual SS
[1] 100.779
> QtY2 <- facmul(X.qr,'Q',Y,transpose=T,full=T) #t(Q) %*% Y
> sum(c(QtY2[-(1:p)])^2)  # residual SS, another way
[1] 100.779
```

□

2.6.3 Updating Models

Adding a variable to a regression model fit using a stored QR decomposition is a simple process. Each of the Householder transformations is applied to the new covariate vector (recall that only the u vectors for the transformations need to be stored), and then the Householder transformation for the new covariate is formed and applied to the response vector. Backward substitution using the updated R matrix is then used to solve for the new estimates.

The QR decomposition is thus also easily extended to forward stepwise regression, where initially only the constant term is in the model and at each step the most significant of the remaining candidate variables is added. The Householder transformations for each variable added to the model can be applied to all the remaining candidate variables at each step. The t statistics for adding each remaining candidate variable to the model are also easily computed from the transformed response and candidate covariates.

Exercise 2.4 Suppose k covariates have been included in a forward stepwise algorithm by updating the QR decomposition as described above. Suppose z is another covariate being considered for entry in the model at the next step. Let $z^{(k)} = H_k \cdots H_1 z$ and $y^{(k)} = H_k \cdots H_1 y$ be the result of applying the Householder transformations used in fitting the current model to the new covariate vector and the response vector. Also, let H_z be the Householder transformation defined from (2.13) with $x = z^{(k)}$ and $t = k+1$, and let w be the $(k+1)$st element of $H_z y^{(k)}$. Show that the F-statistic for adding z to the model is given by $(n-k-1)w^2/(\sum_{j=k+1}^{n}(y_j^{(k)})^2 - w^2)$. Thus only the quantities needed to compute the $(k+1)$st element of $H_z y^{(k)}$ need to be computed to determine the significance level for adding z to the current model. □

Deleting variables is not quite as simple. The Householder transformations are symmetric, and hence each is its own inverse. Thus applying the transformation for the last covariate entered a second time updates the response vector and covariate matrix to delete this covariate from the model. By stepping through the transformations in order, the last k covariates entered can be removed from the model. To remove just the kth of p covariates entered in the model, all the variables added after the kth plus the kth could be removed in this fashion, and then the others added back in, but this is not necessarily a computationally efficient or numerically stable procedure. A better approach is to use Givens rotations to update the R portion of the decomposition (see Golub and van Loan, 1989, Section 12.6, and Seber, 1977, Section 11.9). Basically, the kth column is dropped, and Givens rotations used to zero out the elements r_{jj} for $j > k$ from the old R matrix. The Givens rotations can be applied to the transformed response, and then the least squares estimates can be computed as before. If this approach is used repeatedly, it can become increasingly difficult to keep track of the transformations forming the Q matrix and to apply them to new vectors. Also, in a stepwise procedure with many variables added and dropped during the model fitting process, the accuracy of the updated models will gradually deteriorate. Thus it can be appropriate to periodically refit a model from scratch.

There are also methods for updating the QR decomposition for adding and deleting cases from the data set. These methods also involve applying Givens rotations to the QR decomposition. For details see Golub and van Loan (1989), Section 12.6, and Seber (1977), Section 11.8.

2.6.4 Accuracy of Least Squares Estimators

To bound the accuracy of the solution of a least squares problem, a concept of a condition number is needed for general rectangular matrices. The 2-norm condition number of a square matrix is the ratio of the largest singular value to the smallest singular value of the matrix. For a general rectangular matrix, the 2-norm condition number is defined to be the ratio of the largest and smallest singular values. It can also be shown that for this definition, $\kappa_2(X) = \kappa_2(X'X)^{1/2}$.

Solving for the least squares estimates using the backward substitution algorithm breaks down if the matrix is singular. The numerical algorithms tend to break down (in the sense of producing no significant digits of accuracy) whenever the condition number approaches machine accuracy ϵ_m. In the case of the QR decomposition, it is the condition number of X that is relevant. For algorithms that first form the normal equations, it is the condition number of $X'X$ that determines this breakdown point. Since $\kappa_2(X'X) = \kappa_2(X)^2$, it follows that the QR decomposition can solve problems where normal equation based methods will fail. This might suggest that the QR decomposition is generally superior. However, this issue is not quite that clear cut. The following result, contained in Theorem 5.3.1 in Golub and van Loan (1989), shows that the accuracy of computed least squares estimates is inherently related to $\kappa_2(X)^2$, regardless of the algorithm used. Note that this result relates the exact solution to a least squares problem to the exact solution solution of a perturbed problem, and thus deals with the conditioning of the least squares problem, not with the accuracy of particular algorithms.

Theorem 2.1 *Suppose β_0 is the exact value minimizing $\|y - X\beta\|_2^2$, and $\hat{\beta}$ is the exact minimum of the perturbed system $\|y + \delta - (X + E)\beta\|_2^2$. Let $r = y - X\beta_0$, $\hat{r} = y + \delta - (X + E)\hat{\beta}$, and*

$$\epsilon = \max\left\{\|E\|_2/\|X\|_2, \|\delta\|_2/\|y\|_2\right\},$$

and suppose $\epsilon < \kappa_2(X)$ and $\|r\|_2 < \|y\|_2$. Then

$$\frac{\|\hat{\beta} - \beta_0\|_2}{\|\beta_0\|_2} \leq \epsilon\kappa_2(X)\left\{2 + \kappa_2(X)\|r\|_2/\|y\|_2\right\}/(1 - \|r\|_2^2/\|y\|_2^2)^{1/2} + O(\epsilon^2), \tag{2.22}$$

and

$$\frac{\|\hat{r} - r\|_2}{\|\hat{y}\|_2} \leq \epsilon\left\{1 + 2\kappa_2(X)\right\}\min\{1, n - p\} + O(\epsilon^2). \tag{2.23}$$

Generally in statistical problems $n > p$ and $\|r\|_2 >> 0$, so (2.22) shows that relative $O(\epsilon)$ errors in storing X and y will alter the exact solution by $O(\epsilon\kappa_2(X)^2)$, and (2.23) shows this perturbation gives a relative error in the fitted values of only $O(\epsilon\kappa_2(X))$.

Golub and van Loan (1989, p. 226) also state that for the QR algorithm, the computed $\hat{\beta}$ exactly minimizes a perturbed problem $\|y + \delta - (X + E)\beta\|_2^2$ with $\|E\|_2/\|A\|_2 \leq (6n - 3p + 41)p^{3/2}\epsilon_m + O(\epsilon_m^2)$ and $\|\delta\|_2/\|y\|_2 \leq (6n - 3p + 40)p\epsilon_m + O(\epsilon_m^2)$. Thus for this algorithm, ϵ in (2.22) and (2.23) is approximately $6np^{3/2}\epsilon_m$.

For the normal equation approach, both the errors in calculating $X'X$ and the errors in solving the system need to be taken into account. This will tend to produce an error of order of some factor involving n and p times $\epsilon_m\kappa_2(X'X)$, even for problems where $\|r\|_2$ is very small. When $\|r\|_2$ is small, Theorem 2.1 indicates that the QR decomposition may be much more accurate.

However, in large residual problems where neither approach breaks down, the accuracy of the two methods can be similar. As will be seen below, the QR algorithm is slower than normal equation methods, so choice of algorithm may not always be simple.

2.6.5 Computational Efficiency of QR

From (2.14), with $k = p$ and $t = 1$, applying H_1 to the other columns of X and to a single response vector y requires $(2p + 1)n + p + 1$ multiplications, one division and one square root. In general for the jth transformation H_j, (2.14) applies with $k = p - j + 1$ and $t = j$. Summing these gives a total operation count for the QR decomposition of

$$\begin{aligned}
&\sum_{j=1}^{p}[(2p - 2j + 3)(n - j + 1) + p - j + 2] \\
&\quad = \quad p[(2p + 3)(n + 1) + p + 2] + 2\sum_j j^2 - 2(p + n + 3)\sum_j j \\
&\quad = \quad 2np^2 + 3np + p(p + 1)(2p + 1)/3 - (p + n + 3)p(p + 1) + O(p^2) \\
&\quad = \quad np^2 + 2np - p^3/3 + O(p^2),
\end{aligned}$$

multiplications plus p divisions and square roots. Solving for $\hat{\beta}$ once the QR decomposition is computed then adds another $O(p^2)$ operations. If n is much larger then p, then the total operation count is approximately np^2.

If the QR decomposition is used to solve a $p \times p$ linear system $Ax = b$, then using the above formula with $n = p$ gives on operation count of $2p^3/3 + O(p^2)$. The LU decomposition required only $p^3/3 + O(p^2)$ multiplications, and so requires only about half as much computation as the QR decomposition.

In the least squares problem, if the normal equations are solved using the Choleski decomposition, then $np(p + 3)/2$ multiplications are required to compute $X'X$ and $X'y$, and $p^3/6 + O(p^2)$ are required for the Choleski factorization, with an additional $O(p^2)$ to solve for the least squares estimators. Thus this approach requires about half the computations of the QR decomposition. The QR decomposition is preferred for its numerical stability, not its computational efficiency at fitting a single model (the QR decomposition also has advantages for calculating related quantities needed in regression analysis, and for updating models, as described above).

2.7 Singular-Value Decomposition

The most stable method for calculating solutions to systems of linear equations and least squares estimators is generally the singular value decomposition.

As before, the matrix of covariates X is $n \times p$ with $n \geq p$. The singular value decomposition (SVD) of X is of the form

$$X = UDV',$$

where $U_{n\times p}$ has orthonormal columns, $D_{p\times p}$ is diagonal with $d_{ii} \geq 0$, and $V_{p\times p}$ is orthogonal. The d_{ii} are called the singular values of X. For convenience, it will be assumed throughout that $d_{11} \geq \cdots \geq d_{pp}$. (There is an alternate form, where U is an $n \times n$ orthogonal matrix, and D is

extended to an $n \times p$ matrix by appending $n - p$ rows with all elements 0, but only the reduced version will be considered here.)

Note that since the columns of U are orthonormal, $U'U = I_p$ (however, $UU' \neq I_n$ unless $n = p$). Since

$$X'X = VDU'UDV' = VD^2V',$$

it follows that the columns of V are eigenvectors of $X'X$, and that the d_{ii}^2 are the corresponding eigenvalues.

If X is a square $(p \times p)$ nonsingular matrix, then both U and V are orthogonal matrices, and $X^{-1} = (V')^{-1}D^{-1}U^{-1} = VD^{-1}U'$, so once the SVD is computed, inverting the matrix requires only inverting a diagonal matrix and computing a matrix product. Also note that $VD^{-1}U'$ is the SVD of X^{-1}, so the singular values of X^{-1} are the inverse of the singular values of X.

For a general $n \times p$ matrix X with SVD UDV', the rank of X is the number of nonzero d_{ii}. A generalized inverse of X is any matrix G satisfying $XGX = X$. Let D^+ be the diagonal matrix with diagonal elements

$$d_{ii}^+ = \begin{cases} 1/d_{ii} & d_{ii} > 0 \\ 0 & d_{ii} = 0, \end{cases}$$

Then a particular generalized inverse for X is given by

$$X^+ = VD^+U'. \tag{2.24}$$

This generalized inverse is called the Moore-Penrose generalized inverse.

Exercise 2.5 Verify that X^+ is a generalized inverse for X.

If d_{11} is the largest singular value of X, then

$$\begin{aligned} \|X\|_2 &= \sup_a \|Xa\|_2/\|a\|_2 \\ &= \sup_a (a'X'Xa/a'a)^{1/2} \\ &= \sup_a (a'VD^2V'a/a'a)^{1/2} \\ &= \sup_b (b'V'VD^2V'Vb/b'V'Vb)^{1/2} \\ &= \sup_b (b'D^2b/b'b)^{1/2} \\ &= \sup_b (\sum_i b_i^2 d_{ii}^2 / \sum_i b_i^2)^{1/2} \\ &= d_{11}, \end{aligned} \tag{2.25}$$

where the 4th line follows because $V'V = VV' = I_p$, so any $a \in R^p$ can be mapped to $b = V'a$ and any $b \in R^p$ to $a = Vb$, and the final line follows because the previous line is maximized by choosing b so that as much weight as possible is placed on the largest singular value.

2.7.1 Computing SVDs

Computing the SVD of $X_{n\times p}$ can be thought of in terms of finding orthogonal matrices U_e and V such that the upper $p \times p$ block of $U_e'XV$ is a diagonal matrix, with the rest of the matrix 0's. The Householder transformation (2.13) can be used to zero out elements in a column, and applying analogous transformations to the columns of the transpose of the matrix can be used to zero out elements in a row. It is not possible to reduce X to a diagonal form in this way. Suppose that first all elements in the first column (except x_{11}) have been set to 0 by H_1. Then the transformation to set all elements in the first row (except the first) to 0 would overwrite the first column with nonzero elements. However, if instead the transformation which sets all elements in the first row to 0 except for the first 2 is used, then the first column remains unchanged. Proceeding in this way (alternating columns and rows), transformations U_h and V_h can be built up as the product of Householder transformations so that $U_h'XV_h = B$ is in bidiagonal form, with the only nonzero elements b_{ii} and $b_{i,i+1}$, $i = 1, \ldots, p$. Computing the SVD then uses an iterative algorithm to find the singular values and transformations U_b and V_b such that $U_b'BV_b = D$. The details, which are a little complicated, are given in Section 8.3 of Golub and van Loan (1989); see also Section 2.6 of Press *et. al.* (1992). The transformations for the full SVD are then given by $U_e = U_hU_b$ and $V = V_hV_b$ (and U is given by the first p columns of U_e).

An alternate algorithm for reducing to bidiagonal form is to first use the QR decomposition to reduce to an upper triangular matrix, and then to apply the above approach just to the upper triangular matrix. This tends to be more efficient when n is substantially larger than p.

Since the columns of U and V have to be built up from a number of Householder transformations and the construction of the U_b and V_b matrices, the computational efficiency of the SVD depends on how many of these terms are actually needed. That is, if the SVD will only be used to solve a linear system or a least squares problem, then U need never be explicitly computed (only applied to the right hand side or response vector), resulting in considerable savings. D and V can be computed in $2np^2 + 4p^3$ FLOPS using the first algorithm above, and in $np^2 + 11p^3/2$ FLOPS by first reducing to an upper triangular form (Golub and van Loan, 1989, p.239—note that they define a FLOP to be any floating point operation, so their values are twice as large as those reported here). If $n >> p$, then the second approach is comparable to the computations in the QR decomposition. If the $n \times p$ matrix U is also needed, then the computation time increases to $7np^2 + 4p^3$ for the first approach and $3np^2 + 10p^3$ for the second.

In practice SVD algorithms do not return exact 0's for 0 singular values, and the user must decide when to treat singular values as 0. A rough guide is regard d_{jj} as 0 whenever d_{jj}/d_{11} approaches machine precision ϵ_m (where d_{11} is the largest singular value). Thus numerically, the rank of the matrix is the number of d_{ii} with $d_{ii}/d_{11} > \epsilon$, for some $\epsilon \geq \epsilon_m$.

2.7.2 Solving Linear Systems

Solving the linear system $Ax = b$ using the SVD is straightforward when A is nonsingular. Since $A = UDV'$, the solution $\hat{x}$ is given formally by $VD^{-1}U'b$. As noted above, it is often not necessary to explicitly form U and multiply; instead, the individual transformations can be applied to b as they are constructed. Since D is diagonal, multiplying $U'b$ by D^{-1} just requires p divisions, and multiplying by V requires p^2 multiplications.

A more interesting problem occurs if A is singular. Then there are 2 cases to consider. If b is not in the range space of A, then there is no solution to the system. If b is in the range space, then there are an infinite number of solutions. The quantity A^+b, where A^+ is the Moore-Penrose g-inverse, is one solution, since if b is in the range of A, then $b = Af$ for some f, so

$$AA^+b = AA^+Af = Af = b, \tag{2.26}$$

since A^+ is a generalized inverse of A (this result is also easily verified by using the SVD formulas for A and A^+). To characterize all solutions, suppose $d_{11}, \ldots, d_{kk} > 0$ and $d_{k+1,k+1}, \ldots, d_{pp} = 0$, and let V_j be the jth column of V. Then all solutions are of the form

$$A^+b + \sum_{j=k+1}^{p} \alpha_j V_j \tag{2.27}$$

for any real α_j, and any vector of this form is a solution.

It can be determined whether b is numerically in the range space of A by using SVDs to determine the ranks of A and $(A; b)_{p\times(p+1)}$ (A with b appended as an additional column). If these ranks are equal, then b is in the range of A, and otherwise it is not. This test, as any numerical test, is not foolproof, but only determines ranks to the accuracy of the calculations. (When the matrix has more columns than rows, the SVD is still of the form UDV', with now U orthogonal and $V_{(p+1)\times p}$ having orthonormal columns.)

Exercise 2.6 Suppose b is in the range space of $A_{p\times p}$, and that $\text{rank}(A) < p$. Show that A^+b has the minimum 2-norm of all solutions to $Ax = b$.

Formally, when b is not in the range space of A, then A^+b is a solution to the least squares problem $\min_x \|b - Ax\|_2^2$, as is any x of the form (2.27). This follows from the general development given in the next section.

The following examples illustrate the use of the SVD in solving systems of equations.

Example 2.7

```
> # SVD with a square singular matrix
> A
     [,1] [,2] [,3] [,4]
[1,]    1    2    1    2
[2,]    1    2    3    4
[3,]    1    2   -1   -1
[4,]   -1   -2    3    4
attr(, "class"):
[1] "Matrix"
> b
     [,1]
[1,]    5
[2,]    9
[3,]    1
[4,]    3
attr(, "class"):
```

```
[1] "Matrix"
> A.svd <- svd(A)
> A.svd$val
[1] 7.542847e+00 4.472136e+00 3.247434e-01 1.722363e-16
> svd(cbind(A,b))$val  # is b in the range space of A?
[1] 1.287793e+01 5.193904e+00 4.270761e-01 1.845964e-16
> svd(cbind(A,c(1,1,1,1)))$val
[1] 7.6883690 4.5584515 1.0233832 0.2493763
> x1 <- solve(A.svd,b)
> x1
              [,1]
[1,]  5.212331e+00
[2,] -1.106166e+00
[3,]  2.000000e+00
[4,] -3.682369e-15
attr(, "class"):
[1] "Matrix"

other attributes:
[1] "rcond" "rank"  "call"
> attr(x1,'rcond')
[1] 2.283439e-17
> attr(x1,'rank')
[1] 4
> x2 <- solve(A.svd,b,tol=1e-15)
Warning messages:
  singular solve in: solve(A.svd, b, tol = 1e-15)
> x2
              [,1]
[1,]  6.000000e-01
[2,]  1.200000e+00
[3,]  2.000000e+00
[4,] -4.440892e-16
attr(, "class"):
[1] "Matrix"

other attributes:
[1] "rcond" "rank"  "call"
> attr(x2,'rcond')
[1] 0.04305316
> attr(x2,'rank')
[1] 3
> U <- A.svd$vectors$left
> V <- A.svd$vectors$right
> x3 <- V[,1:3] %*% ((1/A.svd$values[1:3]) * (t(U[,1:3]) %*% b))
> x3
```

```
              [,1]
[1,]  6.000000e-01
[2,]  1.200000e+00
[3,]  2.000000e+00
[4,] -4.440892e-16
attr(, "class"):
[1] "Matrix"
> V[,4]   # basis for null space
              [,1]
[1,] -8.944272e-01
[2,]  4.472136e-01
[3,] -8.361367e-16
[4,]  6.279699e-16
attr(, "class"):
[1] "Matrix"
> c(-2,1)/sqrt(5)
[1] -0.8944272  0.4472136
> A %*% V[,4]
              [,1]
[1,]  7.528700e-16
[2,]  3.365364e-16
[3,]  5.412337e-16
[4,] -3.295975e-16
attr(, "class"):
[1] "Matrix"
> x4 <- x3+10*V[,4]  # also a solution (can replace 10 by any real number)
> A %*% x4
     [,1]
[1,]    5
[2,]    9
[3,]    1
[4,]    3
attr(, "class"):
[1] "Matrix"
>

> xx <- solve(A.svd,c(1,1,1,1),tol=1e-15)
Warning messages:
  singular solve in: solve(A.svd, c(1, 1, 1, 1), tol = 1e-15)
> xx
            [,1]
[1,]  0.03333333
[2,]  0.06666667
[3,] -1.50000000
[4,]  1.33333333
attr(, "class"):
```

```
[1] "Matrix"

other attributes:
[1] "rcond" "rank"  "call"
> A %*% xx
          [,1]
[1,] 1.3333333
[2,] 1.0000000
[3,] 0.3333333
[4,] 0.6666667
attr(, "class"):
[1] "Matrix"
> ## Moore-Penrose least squares solution
> V[,1:3] %*% ((1/A.svd$values[1:3]) * (t(U[,1:3]) %*% c(1,1,1,1)))
            [,1]
[1,]  0.03333333
[2,]  0.06666667
[3,] -1.50000000
[4,]  1.33333333
attr(, "class"):
[1] "Matrix"
```

□

2.7.3 SVD and Least Squares

Again consider the least squares problem (2.16). Suppose first $\text{rank}(X) = p$, and let UDV' be the SVD of X. Then

$$X'X = VD^2V', \tag{2.28}$$

and the only solution to (2.16) is

$$\hat{\beta} = (X'X)^{-1}X'y = VD^{-2}V'VDU'y = VD^{-1}U'y.$$

Thus computing the least squares estimators requires applying the orthogonal transformations used to form U to y, dividing by the diagonal elements of D, and multiplying by V. From (2.28), it also follows that variances of linear functions of the least squares estimators are easily computed from V and D. Generally, easy computation of other quantities of interest in regression analysis will require keeping a copy of X and y in addition to the SVD. For example, the leverage values $h_{ii} = x_i'VD^{-2}V'x_i$, where x_i' is the ith row of X.

For the general case where $\text{rank}(X) \le p$, $X^+y = VD^+U'y$ is a solution to the least squares problem. To see this, note that even when $\text{rank}(X) < p$, a least squares estimate still satisfies the normal equations

$$(X'X)\beta = X'y. \tag{2.29}$$

This is a linear system of the form discussed in the previous section, and $b = X'y$ is in the range space of $A = X'X$. From (2.28), it follows that the SVD of $X'X$ is VD^2V' where UDV' is the

SVD of X. That is, the "U", "D" and "V" matrices of the SVD of $X'X$ are V, D^2 and V, respectively. Thus the Moore-Penrose generalized inverse is $(X'X)^+ = V(D^+)^2V'$. From (2.27), all solutions to (2.29) are therefore of the form

$$(X'X)^+X'y + \sum_{j:d_{jj}=0} \alpha_j V_j = V(D^+)^2V'VDU'y + \sum_{j:d_{jj}=0} \alpha_j V_j = VD^+U'y + \sum_{j:d_{jj}=0} \alpha_j V_j, \quad (2.30)$$

where the α_j are any real numbers, and the V_j are the columns of V. In particular, setting all $\alpha_j = 0$, a solution is X^+y.

Recall that a parametric function $l'\beta$ is estimable if l is in the range space of X'. Since the range space of X' is the same as the range space of VD, it follows that the columns of V corresponding to nonzero singular values are a basis for the space

$$\{l \,:\, l'\beta \text{ is estimable}\}.$$

Also note that the columns of V corresponding to singular values of 0 are a basis for the null space of X.

Exercise 2.7 Use (2.30) to show that the least squares estimate of of any estimable parametric function $l'\beta$ is unique.

Thisted (1988), p. 99, gives the following interpretation of the singular values. If the ϵ_i in the regression model (2.15) are $N(0,1)$, then the Fisher information for β is $\sigma^{-2}X'X = \sigma^{-2}VD^2V'$, and in a sense the information for a parametric function $l'\beta$ is $\sigma^{-2}l'VD^2V'l$. If V_j is a column of V, then since the columns of V are orthonormal, the information about $V_j'\beta$ is d_{jj}^2/σ^2. Thus the d_{jj}^2 are proportional to the information in the sample about the parametric functions $V_j'\beta$. $V_1'\beta$ is the linear combination of the parameters about which the data is most informative, and $V_p'\beta$ the linear combination about which the data are least informative.

Below the SVD calculations for least squares are illustrated using the same example considered for the QR decomposition.

Example 2.6 (continued)

```
> # SVD
> X.svd <- svd(X)
> print(b2 <- solve(X.svd,Y)) # solves for LS parameter estimates
            [,1]
[1,]  10.1643467
[2,]   1.5920543
[3,]   0.7286286
[4,]  -1.1092590
[5,]  -1.1106021
attr(, "class"):
[1] "Matrix"

other attributes:
```

```
[1] "rcond" "rank"  "call"
> 1/attr(b2,'rcond')
[1] 7.847564
> names(X.svd)
[1] "values"  "vectors"
> names(X.svd$vectors)
[1] "left"  "right"
> dim(X.svd$vectors$left)  # reduced form
[1] 100   5
> dim(X.svd$vectors$right)
[1] 5 5
> X.svd$values
[1] 14.561409  3.159465  2.963923  2.672461  1.855532
> max(X.svd$values)/min(X.svd$values) # 2-norm condition number
[1] 7.847564
> U <- X.svd$vectors$left
> V <- X.svd$vectors$right
> range(U %*% (X.svd$values*t(V))-X)  # can recover X
[1] -2.442491e-15  5.440093e-15
> w <- t(U) %*% Y
> V %*% ((1/X.svd$values)*w) # parameter estimates
            [,1]
[1,]  10.1643467
[2,]   1.5920543
[3,]   0.7286286
[4,]  -1.1092590
[5,]  -1.1106021
attr(, "class"):
[1] "Matrix"
> Yhat2 <- U %*% w  #fitted values without using original X
> range(Yhat-Yhat2)
[1] 3.552714e-15 7.105427e-14
> sum((Y-Yhat2)^2)  # Residual SS
[1] 100.779
```

□

2.7.4 Some timing comparisons

The speed of the QR, SVD and Choleski approaches to computing least squares estimates in a single model were compared in a Fortran program with all matrix operations performed using LAPACK routines. The calculations were run on a SUN workstations (several years ago). Only the least squares estimates were computed, with no associated calculations, so from the discussion in Section 2.7.1, the computation time for the SVD and QR approaches should be similar. The QR estimates were computed using the routine `dgels`, the SVD using `dgelss`, and for the Choleski approach the normal equations were formed using `dsyrk` for $X'X$, `dgemm` for $X'y$, and

`dposv` to solve the positive definite system of equations $X'X\beta = X'y$. The results are given in the following table. (Entries are the average of two repeat runs.)

Timing comparison (in CPU seconds) of the QR, SVD and Choleski approaches to computing least squares estimates.

n	p	np^2	QR	SVD	Choleski
5000	50	1.25×10^7	4.88	5.08	2.27
10000	50	2.5×10^7	9.66	9.82	4.42
5000	100	5.0×10^7	15.66	17.42	7.06
10000	100	10.0×10^7	31.30	32.97	13.90

As expected, the QR and SVD are similar, and both take over twice the time of the normal equation/Choleski method. Also note how the CPU times for different configurations are nearly proportional to np^2.

2.8 Some Iterative Methods

A linear system $Ax = b$ can sometimes be solved using simple iteration schemes. If the matrix A is sparse, these iterations can sometimes be substantially faster than the methods discussed above.

Simple iterations are of the following form. A is decomposed into two pieces M and N, with $A = M - N$. At each step, the $k+1$ iterate $x^{(k+1)}$ is calculated from the kth by solving

$$Mx = Nx^{(k)} + b. \tag{2.31}$$

Clearly if $x^{(k)} \to x_0$, then $Ax_0 = b$, so x_0 is a solution. General theory states that such an iteration converges if all eigenvalues λ_j of $M^{-1}N$ satisfy $|\lambda_j| < 1$ (see Theorem 10.1.1 of Golub and van Loan, 1989, and Section 6.4 of Lange, 1999). Clearly for such an iteration to be reasonably fast, the matrix M must be chosen so that $x^{(k+1)}$ can be easily computed at each iteration.

Let D be the diagonal matrix with the a_{ii} on the diagonal, U the matrix with elements $u_{ij} = a_{ij}$ for $i < j$ and 0 otherwise (the elements above the main diagonal), and L the matrix with $l_{ij} = a_{ij}$ for $i > j$ and 0 otherwise (the elements below the main diagonal), so $A = L + D + U$. The Gauss-Seidel iteration is defined by setting $M = D + L$ and $N = -U$ in (2.31). The iterates $x_j^{(k+1)}$, $j = 1, \ldots, p$, are then easily computed by

$$x_j^{(k+1)} = a_{jj}^{-1} \left(b_j - \sum_{l<j} a_{jl} x_l^{(k+1)} - \sum_{l>j} a_{jl} x_l^{(k)} \right). \tag{2.32}$$

Exercise 2.8 Show (2.32) solves (2.31) when $M = D + L$ and $N = -U$.

It can be shown that if A is symmetric and positive definite, then the Gauss-Seidel iteration converges (Golub and van Loan, 1989, Theorem 10.1.2).

If $\max |\lambda_j|$ is close to 1, the rate of convergence of the Gauss-Seidel iteration can be very slow. Successive over-relaxation (SOR) is a variation that attempts to increase the rate of convergence

by varying the step size. That is, if $\gamma_j^{(k+1)}$ is defined by the right hand side of (2.32), then SOR updates are given by $x_j^{(k+1)} = \omega\gamma_j^{(k+1)} + (1-\omega)x_j^{(k)}$. In general determining an appropriate value for the relaxation parameter ω requires careful analysis of the eigenvalues of $M^{-1}N$. For references on this topic see Section 10.1 of Golub and van Loan (1989). Additional variations on SOR are also discussed there.

Solving the system $Ax = b$ can also be reformulated as finding the value of x that minimizes $x'Ax - x'b$. Since this is a nonlinear minimization problem, iterative methods developed for such problems can be applied to solving the linear system. Most do not lead to anything useful (Newton's method, for example, just gives that the minimization problem is solved by computing the solution to $Ax = b$), but conjugate gradient algorithms can usefully be applied in some problems. As with the iterative algorithms discussed above, conjugate gradient algorithms are often most useful in sparse systems where the product Ax can be computed quickly. Conjugate gradient algorithms attempt to construct a series of search directions $d^{(j)}$ for line minimizations, with the directions satisfying $(d^{(j)})'Ad^{(k)} = 0$. Application of conjugate gradient algorithms to solving linear systems is discussed in Sections 10.2 and 10.3 of Golub and van Loan (1989), and a simple implementation is given in Section 2.7 of Press *et. al.* (1992). An interesting statistical application is given in O'Sullivan (1990).

2.9 Nonparametric Smoothing Splines

The nonparametric smoothing spline is a problem that at first glance would appear to require $O(n^3)$ computations, where n is the number of data points. However, by taking advantage of the special structure of the problem, it is possible to compute all quantities needed in $O(n)$ computations. An excellent reference on smoothing splines is Green and Silverman (1994). Hastie and Tibshirani (1990) give an elementary nonrigorous treatment of the basics. Also useful are the books by Eubank (1988) and Wahba (1990), and the review articles of Wegman and Wright (1983) and Silverman (1985).

Suppose (x_i, y_i), $i = 1, \ldots, n$, are independent bivariate observations, with

$$y_i = g(x_i) + \epsilon_i, \quad \mathrm{E}(\epsilon_i) = 0, \quad \mathrm{Var}(\epsilon_i) = \sigma^2 < \infty. \tag{2.33}$$

The x_i are restricted to a finite interval $[a, b]$. The problem considered here is estimating the unknown function g, without specifying a parametric form for g. Without any restriction on the class of functions, it is impossible to construct useful estimates except at x_i where there are multiple observations. However, if g can be assumed to lie in a class of suitably smooth functions, then a general solution can be given. In particular, let $S_2[a, b]$ be the space of functions $f(x)$ on $[a, b]$ such that $f(x)$ and $f'(x)$ are absolutely continuous on $[a, b]$, such that $f''(x)$ exists, in the sense that there exists an integrable function f'' satisfying $f'(x) - f'(a) = \int_a^x f''(u)\,du$ for all $x \in [a, b]$, and such that $f''(x)$ is square integrable. Note that any function with a continuous second derivative on $[a, b]$ is in $S_2[a, b]$.

Consider the problem of estimating g in (2.33) with the function in $S_2[a, b]$ that minimizes the penalized least squares criterion

$$\sum_{i=1}^{n}[y_i - g(x_i)]^2 + \lambda \int_a^b [g''(u)]^2\,du. \tag{2.34}$$

The integrated squared second derivative term is a penalty term to force some smoothness on the estimate. If the least squares criterion with $\lambda = 0$ in (2.34) is minimized, the estimate would interpolate the observed data points in some fashion. For a linear function $g(x) = \beta_0 + \beta_1 x$, the second derivative $g''(x) \equiv 0$, so the penalty term does not restrict the slope of a linear function. The penalty does restrict how quickly the slope can change (the second derivative is the rate of change in the slope), so the penalty term penalizes functions whose slopes are changing quickly, and forces the estimate towards smoother functions. Here λ plays the role of a smoothing parameter. If λ is increased to a very large value, then any departure of g from linearity is given a large penalty, so the estimate becomes the ordinary least squares line. As λ is decreased towards 0, the estimate becomes closer to a noisy interpolant of the observed data points.

For $\lambda > 0$, there is a general solution to (2.34). The solution takes the form of a spline function with knots at the observed x_i. In the following subsection, spline functions will be considered. Then in Section 2.9.2, the solution to the penalized least squares problem will be given. Section 2.9.3 considers used of cross validation for choosing the value of the smoothing parameter λ.

2.9.1 Splines

Splines are piecewise polynomials between breakpoints or *knots*, that satisfy certain continuity constraints at the knots. Specifying a spline model requires specifying the knot locations, the degree of the polynomials, and the constraints at the knots. To define a spline on an interval $[a, b]$, let $a \le t_1 < t_2 < \cdots < t_k \le b$ be the knot locations. A spline of order $q + 1$ is a qth degree polynomial on each of the intervals (t_l, t_{l+1}). That is, let

$$P_l(x) = \beta_{0l} + \beta_{1l}x + \cdots + \beta_{ql}x^q.$$

Then the spline $Q(x) = P_l(x)$ if $x \in (t_l, t_{l+1})$. For a spline of order $q + 1$, usually the continuity constraints at the knots specify that the function is continuous, and that the derivatives through order $q - 1$ are also continuous. These are the only types of constraints that will be considered here. For a cubic spline (order 4, the most popular choice), the constraints are

$$\begin{aligned} \beta_{0,l-1} + \beta_{1,l-1}t_l + \beta_{2,l-1}t_l^2 + \beta_{3,l-1}t_l^3 &= \beta_{0l} + \beta_{1l}t_l + \beta_{2l}t_l^2 + \beta_{3l}t_l^3 \\ \beta_{1,l-1} + 2\beta_{2,l-1}t_l + 3\beta_{3,l-1}t_l^2 &= \beta_{1l} + 2\beta_{2l}t_l + 3\beta_{3l}t_l^2 \\ 2\beta_{2,l-1} + 6\beta_{3,l-1}t_l &= 2\beta_{2l} + 6\beta_{3l}t_l. \end{aligned}$$

When a statistical model is specified in terms of a spline, the β_{jl} are unknown parameters to be estimated.

If the knots are on the interior of the interval, then there is a polynomial piece P_0 for the region $a \le x < t_1$, $k - 1$ polynomials P_l for the intervals (t_l, t_{l+1}), $l = 1, \ldots, k - 1$, and an additional polynomial P_k for the region $b \ge x > t_k$. For qth degree polynomials there would thus be $(q + 1) \times (k + 1)$ parameters in the polynomials, and at each of the k knots there would be q constraints, so the total dimension of (or degrees of freedom in) this model is $(q+1) \times (k+1) - qk = q+1+k$, which is the order of the spline plus the number of interior knots.

When working with splines, it is usually easier to first reparameterize to eliminate the constraints.

One unconstrained reparameterization is

$$Q(x) = \alpha_0 + \alpha_1 x + \cdots + \alpha_q x^q + \sum_{l=1}^{k} \alpha_{q+l} I(x \geq t_l)(x - t_l)^q. \tag{2.35}$$

This is a reparameterization because (a) (2.35) is a qth degree polynomial between each of the knots; (b) (2.35) satisfies all the continuity constraints of the original model, since $I(x \geq t_l)(x - t_l)^q$ has $q - 1$ continuous derivatives; and (c) (2.35) has $q + 1 + k$ free parameters, so the dimensions of the models are the same. The functions $I(x \geq t_l)(x - t_l)^q$ are often written $(x - t_l)_+^q$, where the '+' subscript denotes that the function is 0 if the argument $x - t_l < 0$. For this reason the parameterization (2.35) is often said to use the *plus function* basis.

Although the terms in the plus function basis are fairly simple and reasonably interpretable, they often lead to numerical problems because of near collinearity in the design matrix. More numerically stable bases are usually used for computations. One of these is the B-spline basis, which requires some additional notation to describe. Extend the knot sequence by letting $t_i = a$ for $i < 1$, and $t_i = b$ for $i > k$. Set

$$B_{l1}(x) = I(t_l < x \leq t_{l+1}),\ l = 0, \ldots, k,$$

and set $B_{l1}(x) \equiv 0$ for $l < 0$ or $l > k$. Note that $\sum_{l=0}^{k} \alpha_l B_{l1}(x)$ is a piecewise constant function that could be thought of as a 1st order (0th degree) spline. B-splines of higher order are related to the B_{l1} through the recursion

$$B_{lr}(x) = \frac{x - t_l}{t_{l+r-1} - t_l} B_{l,r-1}(x) + \frac{t_{l+r} - x}{t_{l+r} - t_{l+1}} B_{l+1,r-1}(x),$$

$l = 1 - r, \ldots, k$, where $r = q + 1$ is the order of the spline (and q is the degree of the polynomials). The rth order spline is then

$$Q(x) = \sum_{l=1-r}^{k} \alpha_l B_{lr}(x).$$

For a proof that this is a reparameterization of the plus function basis, and discussion of many other properties of B-splines, see de Boor (1978). Two important properties of the B-splines are that $\sum_l B_{lr}(x) \equiv 1$, and that they have a local support property, in that only r of the basis functions can be non-zero at any value of x. The first property is important because it is often convenient to explicitly include a constant term in a model. The constant plus the $r + k$ terms in the full B-spline basis would be over parametrized. Because $\sum_l B_{lr}(x) \equiv 1$, a constant plus any $k + r - 1$ of the B_{lr} give a basis for the space of rth order splines. The local support property is important for numerical stability, and in some applications for computational speed.

There is a function `bs()` in Splus that will compute B-spline basis functions at specified values, as given in the following. Specifying `intercept=T` tells the function to include all $r + k$ basis terms.

```
> x_1:19
> u_bs(x,knots=c(4,8,12,16),degree=3,intercept=T)
> u
          1       2       3       4       5       6       7       8
```

```
 [1,] 1.000 0.0000 0.00000 0.00000 0.00000 0.00000 0.0000 0.000
 [2,] 0.296 0.5835 0.11585 0.00433 0.00000 0.00000 0.0000 0.000
 [3,] 0.037 0.5729 0.35539 0.03463 0.00000 0.00000 0.0000 0.000
 [4,] 0.000 0.3265 0.55659 0.11688 0.00000 0.00000 0.0000 0.000
 [5,] 0.000 0.1378 0.59561 0.26404 0.00260 0.00000 0.0000 0.000
 [6,] 0.000 0.0408 0.50139 0.43696 0.02083 0.00000 0.0000 0.000
 [7,] 0.000 0.0051 0.34108 0.58350 0.07031 0.00000 0.0000 0.000
 [8,] 0.000 0.0000 0.18182 0.65152 0.16667 0.00000 0.0000 0.000
 [9,] 0.000 0.0000 0.07670 0.60559 0.31487 0.00284 0.0000 0.000
[10,] 0.000 0.0000 0.02273 0.47727 0.47727 0.02273 0.0000 0.000
[11,] 0.000 0.0000 0.00284 0.31487 0.60559 0.07670 0.0000 0.000
[12,] 0.000 0.0000 0.00000 0.16667 0.65152 0.18182 0.0000 0.000
[13,] 0.000 0.0000 0.00000 0.07031 0.58350 0.34108 0.0051 0.000
[14,] 0.000 0.0000 0.00000 0.02083 0.43696 0.50139 0.0408 0.000
[15,] 0.000 0.0000 0.00000 0.00260 0.26404 0.59561 0.1378 0.000
[16,] 0.000 0.0000 0.00000 0.00000 0.11688 0.55659 0.3265 0.000
[17,] 0.000 0.0000 0.00000 0.00000 0.03463 0.35539 0.5729 0.037
[18,] 0.000 0.0000 0.00000 0.00000 0.00433 0.11585 0.5835 0.296
[19,] 0.000 0.0000 0.00000 0.00000 0.00000 0.00000 0.0000 1.000
attr(, "knots"):
 [1]  1  1  1  1  4  8 12 16 19 19 19 19
> apply(u,1,sum)
 [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Note the local support, seen in the large numbers of 0's in the design matrix. Also note that from the output of `apply()`, the rows do sum to 1 as claimed.

It is also interesting to look at plots of basis functions. Figure 2.1 plots the odd numbered basis functions (1,3,5,7) from the following.

```
> x1 _ 0:100/100
> x1.bs _ bs(x1,df=8,degree=3,intercept=T)
> attributes(x1.bs)$knots
 [1] 0.0 0.0 0.0 0.0 0.2 0.4 0.6 0.8 1.0 1.0 1.0 1.0
```

Note that here specifying 8 degrees of freedom gave 4 interior knots.

A cubic spline $Q(x)$ on an interval $[a, b]$ with knots $t_1, \ldots, t_k$ ($a < t_1$, $t_k < b$) is a *natural spline* if $Q''(a) = Q'''(a) = Q''(b) = Q'''(b) = 0$. If

$$Q(x) = \beta_{00} + \beta_{10}x + \beta_{20}x^2 + \beta_{30}x^3, \quad a \leq x \leq t_1,$$

then clearly $Q''(a) = Q'''(a) = 0$ implies that $\beta_{30} = \beta_{20} = 0$, so that $Q(x)$ is a straight line on the interval $[a, t_1]$, and by a similar argument, on the interval $[t_k, b]$ as well. The continuity of the second derivative then implies that $Q''(t_1) = Q''(t_k) = 0$. Since the natural spline has 4 more constraints than the B-spline, it has 4 fewer terms in a basis, and thus there are k basis functions in a natural spline with k interior knots.

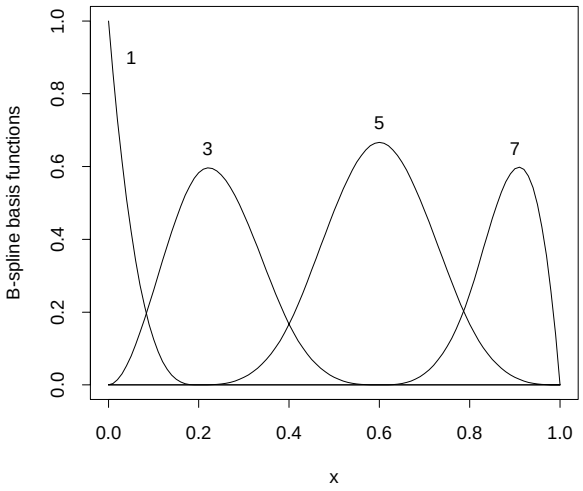

Figure 2.1: Cubic B-spline basis functions. The odd numbered functions are plotted, from a basis with a total of 8 functions.

The function `ns()` in Splus will generate a natural spline basis at a specified set of points. However, it implicitly treats the smallest and largest points as interior knots in the sense used above, so the number of basis functions is only 2 fewer than for a call to `bs()` with the same interior knots.

The second derivative of a cubic spline is a continuous piecewise linear function. That is

$$\frac{d^2}{dx^2}(\beta_{0l} + \beta_{1l}x + \beta_{2l}x^2 + \beta_{3l}x^3) = 2\beta_{2l} + 6\beta_{3l}x.$$

Also, the third derivative is constant on each of the intervals $[a, t_1), (t_1, t_2), \ldots, (t_{k-1}, t_k), (t_k, b]$, and generally does not exist at the knots. For a natural spline as defined formally above, the second and third derivatives are 0 on $[a, t_1)$ and $(t_k, b]$.

2.9.2 Solution to the Penalized Least Squares Problem

For $\lambda > 0$, the unique minimizer of (2.34) is a natural cubic spline with knots at all the observed x_i. This follows from the following lemma.

Lemma: Let $g(x)$ be a natural cubic spline on $[a, b]$ with knots $t_1 < \cdots < t_n$ ($n > 1$, $a < t_1$, $t_n < b$), and let $f(x)$ be any other function in $S_2[a, b]$ with $f(t_i) = g(t_i)$, $i = 1, \ldots, n$. Then

$$\int_a^b [f''(x)]^2\, dx \geq \int_a^b [g''(x)]^2\, dx,$$

with equality only if $f(x) \equiv g(x)$.

Proof: First note that $g \in S_2[a, b]$, since it is a natural cubic spline. Then $h(x)$ defined by $h(x) = f(x) - g(x)$ is also in $S_2[a, b]$. Using integration by parts and the facts that $g''(a) = g''(b) = 0$, $g'''(x) = 0$ for $x < t_1$ and $x > t_n$, and $g'''(x)$ is constant between the knots, it

follows that

$$\begin{aligned}
\int_a^b g''(x)h''(x)\,dx &= g''(b)h'(b) - g''(a)h'(a) - \int_a^b g'''(x)h'(x)\,dx \\
&= -\int_a^b g'''(x)h'(x)\,dx \\
&= -\int_a^{t_1} g'''(x)h'(x)\,dx + \sum_{j=2}^{n}\int_{t_{j-1}}^{t_j} g'''(x)h'(x)\,dx + \int_{t_n}^{b} g'''(x)h'(x)\,dx \\
&= \sum_{j=2}^{n} g'''(t_j-)\int_{t_{j-1}}^{t_j} h'(x)\,dx \\
&= \sum_{j=2}^{n} g'''(t_j-)[h(t_j) - h(t_{j-1})] \\
&= 0,
\end{aligned}$$

where the last equality is because $h(x) = f(x) - g(x)$ is 0 at the knots, because f and g are assumed to be equal there. Using this result, it then follows that

$$\begin{aligned}
\int_a^b [f''(x)]^2\,dx &= \int_a^b [g''(x) + h''(x)]^2\,dx \\
&= \int_a^b [g''(x)]^2\,dx + 2\int_a^b g''(x)h''(x)\,dx + \int_a^b [h''(x)]^2\,dx \\
&= \int_a^b [g''(x)]^2\,dx + \int_a^b [h''(x)]^2\,dx \\
&\geq \int_a^b [g''(x)]^2\,dx,
\end{aligned}$$

as required. Equality can hold only if $\int_a^b [h''(x)]^2\,dx = 0$, which means $h''(x) = 0$ (almost everywhere) on $[a, b]$, which together with the continuity of $h'(x)$ means that $h'(x)$ is constant, and thus that $h(x)$ is linear on $[a, b]$. But $h(t_j) = 0$ for all j, so $h(x) \equiv 0$ on $[a, b]$. □

Using this lemma, it is straightforward to argue that (2.34) is minimized by a natural spline. Let f be any function in $S_2[a, b]$, and let g be a natural spline with with knots at all the x_i, with $g(x_i) = f(x_i)$, $i = 1, \ldots, n$. Then clearly

$$\sum_i [y_i - f(x_i)]^2 = \sum_i [y_i - g(x_i)]^2,$$

and from the lemma,

$$\int_a^b [f''(x)]^2\,dx \geq \int_a^b [g''(x)]^2\,dx,$$

with equality only if $f \equiv g$, so if $\lambda > 0$, for any $f \in S_2[a, b]$ there is a natural spline g which has a smaller value of (2.34), unless $f \equiv g$, in which case f is already a natural spline. Thus to minimize (2.34), only the family of natural splines needs to be considered.

Knowing this, the estimate can then be calculated by specifying a basis for the natural spline, and solving for the coefficients of the basis functions to minimize (2.34). The algorithm discussed below actually starts with a slightly larger family, the class of cubic splines with knots at all the

data points, which can be represented using a B-spline basis. It turns out that the penalty still forces the solution to be a natural spline, so the same solution is obtained, regardless. For a slightly more efficient algorithm based directly on natural splines, see Green and Silverman (1994).

To specify the solution in matrix notation, let $B_j(x)$ be the B-spline basis functions for a cubic spline with knots at all the data points (the min and max of the x_i are not regarded as interior knots, so there are $n+2$ such functions if there are n distinct values x_i), and let X be the matrix whose ijth element is $b_{ij} = B_j(x_i)$. Also let β be the vector of coefficients for the basis functions. For this parameterization, the penalty function in (2.34) can be written $\lambda\beta^t P\beta$, for some non-negative definite matrix P that is just a function of the knot locations. Specifically, writing the spline as $g(u) = \sum_j \beta_j B_j(u)$, then

$$\int [g''(u)]^2 \, du = \sum_j \sum_k \beta_j \beta_k \int B_j''(u) B_k''(u) \, du,$$

so

$$P_{jk} = \int B_j''(u) B_k''(u) \, du. \tag{2.36}$$

Thus (2.34) becomes

$$(y - X\beta)^t (y - X\beta) + \lambda\beta^t P\beta, \tag{2.37}$$

where $y^t = (y_1, \ldots, y_n)$. Setting the gradient of (2.37) to 0 then gives

$$-2X^t(y - X\beta) + 2\lambda P\beta = \mathbf{0},$$

which is equivalent to

$$(X^t X + \lambda P)\beta = X^t y.$$

The advantage to using B-splines is their local support property. Since at most 4 cubic B-spline basis functions are positive at any point, this is a narrowly banded system of equations (because the matrices $X^t X$ and P are banded). Thus even though the number of parameters and equations is large (and grows with the sample size), the estimates can be calculated in $O(n)$ floating point operations, for example by using a Cholesky decomposition for banded matrices and backsubstitution to find the solution to the equations. Formally, the solution for β may be written

$$\hat{\beta} = (X^t X + \lambda P)^{-1} X^t y,$$

and the smoothed y's by

$$\hat{y} = X\hat{\beta} = X(X^t X + \lambda P)^{-1} X^t y, \tag{2.38}$$

although in practice one would not want to explicitly calculate the matrix inverse given in these formulas.

The steps in this algorithm to compute the smoothing spline estimate for a fixed value of λ are thus as follows. Methods for choosing λ will be discussed below, especially in Section 2.9.3.

1. Determine the knot locations. For the true nonparametric smoothing spline this would put knots at all unique values of the covariate. In practice, it is often adequate to use a much smaller number of knots, say at every 10th data point for sample sizes < 500, and using 50 knots for samples > 500. However, the algorithm is very fast even with knots at all data points. In general suppose there are k interior knots.

2. Evaluate the B-spline basis functions corresponding to the chosen knot sequence at the data points. Since at most 4 basis functions are positive at any point, only a $4 \times n$ array is needed to store these. De Boor's (1978) Fortran routines for calculating B-splines are available at `netlib`.

3. Calculate the penalty matrix. Deriving closed form formulas is a bit of work. The FORTRAN subroutine `pencbc.f`, given below, can perform the computations. Since P is a symmetric band matrix it only requires a $4 \times (k+4)$ array for storage.

4. Calculate X^tX. Because each row of X has at most 4 nonzero elements, this can be done with $10 \times n$ multiplications and additions, and again only requires an array of dimension $4 \times (k+4)$ for storage.

5. Calculate X^ty. This requires $4 \times n$ multiplications and additions, (and gives a vector of length $k+4$).

6. Form $X^tX + \lambda P$, and compute its Cholesky factorization. The Cholesky factor of a band matrix is again a band matrix (and can be done in place in the band storage array used for $X^tX + \lambda P$; see the LAPACK subroutine `dpbtrf`). This requires $O(k)$ operations.

7. Use forward and back substitution to solve for $\hat{\beta}$ (again $O(k)$ operations because of the band structure).

8. Compute $\hat{y} = X\hat{\beta}$ (again about $4n$ multiplications and additions).

Note that even with knots at all the data points, the entire algorithm takes $O(n)$ operations, and requires $O(n)$ storage.

The following FORTRAN subrouting computes the components of P.

```
      subroutine pencbc(nk,wk,penm)
c This subroutine calculates the penalty matrix for integrated squared
c second derivative penalty. On output the first row of penm will have the main
c diagonal, the 2nd row the first sub diag ... the 4th row the 3rd sub diag
c stored in lapack style symmetric band storage (lower triangle)
c actual row dim of penm is 4 (must have at least nk+4 cols)
c nk is # interior knots,
c wk is augmented knot sequence.  That is, wk(-2), wk(-1),
c  wk(0) all equal the lower limit of integration in the penalty function,
c  wk(1) to wk(nk) are the interior knots in ascending order, and wk(nk+1),
c  wk(nk+2), wk(nk+3) all equal the upper limit of integration.
      double precision wk(-2:nk+3),penm(4,nk+4),tl(4),tu(4)
      double precision u1,u2,u3
      integer i,nk,l,j,ll,j2,k
      do 10 i=1,nk+4
         do 11 j=1,4
            penm(j,i)=0
 11      continue
```

```
 10   continue
      tl(1)=6/((wk(1)-wk(-2))*(wk(1)-wk(-1)))
      tl(2)=-6/(wk(1)-wk(-1))*(1/(wk(1)-wk(-2))+1/(wk(2)-wk(-1)))
      tl(3)=6/((wk(2)-wk(-1))*(wk(1)-wk(-1)))
      tl(4)=0
      do 20 l=1,nk+1
c tl has the value of the 2nd deriv at the lower endpoint of the lth
c interval, tu at the upper (2nd derivs of basis fcns are linear on
c each interval
         tu(1)=0
         tu(2)=6/((wk(l+1)-wk(l-2))*(wk(l+1)-wk(l-1)))
         tu(3)=-6/(wk(l+1)-wk(l-1))*(1/(wk(l+1)-wk(l-2))+
     &        1/(wk(l+2)-wk(l-1)))
         tu(4)=6/((wk(l+2)-wk(l-1))*(wk(l+1)-wk(l-1)))
c since integrating a quadratic, use Simpson's rule:
c u1 and u2 are the integrand at the endpoints, u3 at the midpoint
         do 22 j=1,4
            ll=l+j-1
            j2=j-1
            do 23 k=j,4
               u1=tl(j)*tl(k)
               u2=tu(j)*tu(k)
               u3=(tl(j)+tu(j))*(tl(k)+tu(k))/4
               penm(k-j2,ll)=penm(k-j2,ll)+(wk(l)-wk(l-1))*
     $              (u1+u2+4*u3)/3
 23         continue
 22      continue
         tl(1)=tu(2)
         tl(2)=tu(3)
         tl(3)=tu(4)
         tl(4)=0
 20   continue
      return
      end
```

The matrix $S_\lambda = X(X^tX + \lambda P)^{-1}X^t$ is called the *smoother matrix*, since it maps the observed y to the smoothed values. Sometimes it is also called the "hat" matrix, since it maps y to $\hat{y}$. The matrix S_λ is thus the analog of the projection operator $U(U^tU)^{-1}U^t$ from ordinary least squares regression with a design matrix U. However, S_λ is generally not a projection operator.

The smoother matrix S_λ satisfies $S\mathbf{1}_n = \mathbf{1}_n$, where $\mathbf{1}_n$ is the n-dimensional vector of 1's, and $S_\lambda x = x$, where x is the vector of the x_i. To see this, let $g(u, \beta) = \sum_j \beta_j B_j(u)$, and note that as β varies this gives the full space of cubic splines with this set of knots (that is what it means for the $B_j(u)$ to be a basis for this space). Now constants and linear functions are subsets of the space of cubic splines, so there is a β_1 such that $g(u, \beta_1) \equiv 1$, and a β_2 such that $g(u, \beta_2) \equiv u$. Then

$X\beta_1 = \mathbf{1}_n$ and $X\beta_2 = x$. Also, $g''(u, \beta_j) \equiv 0$, $j = 1, 2$. Thus

$$0 = \int_a^b [g''(u, \beta_j)]^2 \, du = \beta_j^t P \beta_j,$$

so $P\beta_j = \mathbf{0}$, $j = 1, 2$. Then

$$\begin{aligned} S_\lambda X\beta_j &= X(X^tX + \lambda P)^{-1}X^tX\beta_j \\ &= X(X^tX + \lambda P)^{-1}(X^tX + \lambda P)\beta_j \\ &= X\beta_j, \end{aligned}$$

where the third equality follows because $P\beta_j = \mathbf{0}$, and the result follows by recalling $X\beta_1 = \mathbf{1}_n$ and $X\beta_2 = x$.

Since the expected value of the smoothing spline is

$$\mathrm{E}(\hat{y}) = S_\lambda \mathrm{E}(y)$$

it follows that smoothing splines are unbiased when the true g is a straight line.

The connection between the smoother matrix and the projection operator in ordinary parametric regression can be used to motivate a definition of "degrees of freedom" or "equivalent number of parameters" for smoothing splines. For the projection operator in least squares, $\mathrm{trace}\{U(U^tU)^{-1}U^t\} = \mathrm{rank}(U)$, which is the number of parameters in the model. For smoothing splines,

$$d_f = \mathrm{trace}(S_\lambda) \tag{2.39}$$

also gives a useful measure of model complexity. $d_f \to 2$ as $\lambda \to \infty$ (and the fit is forced toward a 2 parameter linear model), and increases towards the number of parameters as λ gets small. Choosing λ to give a target degrees of freedom is one useful method for specifying the smoothing parameter (at any rate the degrees of freedom is easier to interpret than λ itself). In many applications "several" degrees of freedom would be a reasonable starting point, although different values should be examined to see the effect on the fit. A more automated method for choosing a smoothing parameter is discussed in Section 2.9.3. There are 2 other related definitions of degrees of freedom for smoothers; see Section 3.5 of Hastie and Tibshirani (1990) for details.

Smoothing splines can be fit in Splus using the function `smooth.spline()`. The smoothing parameter can be specified in one of three ways. One is to explicitly specify the value of the smoothing parameter λ, using the argument `spar`. Since λ does not have a clear intuitive interpretation, it is usually difficult to specify a reasonable value without substantial trial and error. The second way is to specify an approximate degrees of freedom, as defined in (2.39), using the argument `df`. The third method (the default) is to automatically choose the smoothing parameter using generalized cross validation (there is also an option to specify ordinary cross validation, instead). The meaning of cross validation in this setting will be discussed in the following section.

2.9.3 Cross Validation and Smoothing Parameters

Ideally, the smoothing parameter would be chosen to maximize the accuracy of the estimated curve. Often it is reasonable to focus on the accuracy at the observed data points, although this

is not the only possibility. The average mean squared error (MSE) of an estimator $\hat{g}(x)$ of $g(x)$ is

$$\sum_i \mathrm{E}\{\hat{g}(x_i) - g(x_i)\}^2/n.$$

The average predictive squared error (PSE) is

$$\sum_i \mathrm{E}[Y_i^* - \hat{g}(x_i)]^2/n,$$

where the Y_i^* are new observations taken at the same x_i, independent of the original observations. Since

$$\begin{aligned}\sum_i \mathrm{E}[Y_i^* - \hat{g}(x_i)]^2/n &= \sum_i \mathrm{E}[Y_i^* - g(x_i)]^2/n + \sum_i \mathrm{E}[\hat{g}(x_i) - g(x_i)]^2/n - \\ &\quad 2\sum_i \mathrm{E}\{[\hat{g}(x_i) - g(x_i)][Y_i^* - g(x_i)]\}/n \\ &= \sigma^2 + \sum_i \mathrm{E}[\hat{g}(x_i) - g(x_i)]^2/n \qquad (2.40)\end{aligned}$$

(the expectation of the cross product term is 0 because Y_i^* is independent of $\hat{g}(x_i)$), it follows that minimizing the average predictive squared error is equivalent to minimizing the average MSE over the data points.

The MSE and the PSE both involve the unknown function g, which makes them difficult to estimate directly. Cross validation is a method for estimating the PSE. Essentially, the estimate of $g(x_i)$ is calculated with the ith observation omitted, and then compared to the observed y_i. This is done in turn for each observation, and the error averaged over the observations. Denote the smoothed estimate at x_i with the ith observation omitted by $\hat{g}_{-i}(x_i;\lambda)$, explicitly indicating that this quantity depends on the smoothing parameter λ. Then the cross validated estimator of the PSE is

$$\mathrm{CV}(\lambda) = \sum_i [y_i - \hat{g}_{-i}(x_i;\lambda)]^2/n.$$

Proceeding as in (2.40), it follows that

$$\mathrm{E}[\mathrm{CV}(\lambda)] = \sigma^2 + \sum_i \mathrm{E}[\hat{g}_{-i}(x_i;\lambda) - g(x_i)]^2/n.$$

Thus if the difference between $\hat{g}(x_i;\lambda)$ and $\hat{g}_{-i}(x_i;\lambda)$ is small, then $\mathrm{CV}(\lambda)$ should be a good estimator of the PSE.

Direct computation of $\mathrm{CV}(\lambda)$ by deleting each point and recomputing the estimate would be very time consuming. It turns out that there is a simple formula, though, based on the following lemma.

Lemma: For fixed λ and i, let g_{-i} be the vector with components $\hat{g}_{-i}(x_j;\lambda)$, $j = 1,\ldots,n$, and let V^* be the vector with components $V_j^* = y_j$ for $j \neq i$ and $V_i^* = \hat{g}_{-i}(x_i;\lambda)$. Then

$$g_{-i} = S_\lambda V^*.$$

Proof: For any smooth curve g,

$$\begin{aligned}\sum_{j=1}^{n}[V_j^* - g(x_j)]^2 + \lambda\int g''(u)\,du &\geq \sum_{j\neq i}^{n}[V_j^* - g(x_j)]^2 + \lambda\int g''(u)\,du \\ &\geq \sum_{j\neq i}^{n}[V_j^* - \hat{g}_{-i}(x_j;\lambda)]^2 + \lambda\int \hat{g}''_{-i}(u;\lambda)\,du \\ &= \sum_{j=1}^{n}[V_j^* - \hat{g}_{-i}(x_j;\lambda)]^2 + \lambda\int \hat{g}''_{-i}(u;\lambda)\,du,\end{aligned}$$

since $\hat{g}_{-i}(x;\lambda)$ minimizes the right hand side of the first line, and because of the definition of V_i^*. Thus $\hat{g}_{-i}(x;\lambda)$ also minimizes the left hand side of the first line. But this has the same number of observations and covariate values as the original data set (but a slightly different response vector), and thus has the same smoother matrix S_λ as the original penalized least squares problem, and so the minimizer of the left hand side of the first line (g_{-i}) is given by $S_\lambda V^*$, giving the result. □

Writing $s_{ij}(\lambda)$ for the elements of S_λ, it follows from this lemma and the definition of V_i^* that

$$\begin{aligned}\hat{g}_{-i}(x_i;\lambda) - y_i &= \sum_{j=1}^{n} s_{ij}(\lambda)V_j^* - y_i \\ &= \sum_{j=1}^{n} s_{ij}(\lambda)y_j - y_i + s_{ii}(\lambda)[\hat{g}_{-i}(x_i;\lambda) - y_i] \\ &= \hat{g}(x_i;\lambda) - y_i + s_{ii}(\lambda)[\hat{g}_{-i}(x_i;\lambda) - y_i].\end{aligned}$$

Thus

$$y_i - \hat{g}_{-i}(x_i;\lambda) = [y_i - \hat{g}(x_i;\lambda)]/[1 - s_{ii}(\lambda)],$$

and

$$\mathrm{CV}(\lambda) = \frac{1}{n}\sum_i \left[\frac{y_i - \hat{g}(x_i;\lambda)}{1 - s_{ii}(\lambda)}\right]^2. \tag{2.41}$$

Thus the key to fast computation of $\mathrm{CV}(\lambda)$ is access to the diagonal elements of $S_\lambda = X(X^tX + \lambda P)^{-1}X^t$. Set $A = X^tX + \lambda P$. Although A is a band matrix, so its Cholesky factorization is banded, A^{-1} is generally not a band matrix, and solving for $A^{-1}X^t$ by factoring and using forward and backward substitution would take $O(n^2)$ operations, which would not always be acceptable (recall estimating the curve itself only requires $O(n)$. However, it turns out a rather modest reorganization of the calculations leads to an $O(n)$ algorithm. In general, for a B-spline basis for a spline of order $q+1$, the only nonzero elements of the ith row of X are $b_{ij}, b_{i,j+1}, \ldots, b_{i,j+q}$, for some j. Denoting the elements of A^{-1} by $w(i,j)$, it follows that

$$s_{ii}(\lambda) = \sum_{u=j}^{q+j} b_{iu}\sum_{v=j}^{q+j} b_{iv}w(u,v),$$

so only the $2q+1$ central diagonals of A^{-1} are needed, or recalling that A is symmetric, only the main diagonal and the next q above the main diagonal are needed. An $O(n)$ algorithm for calculating these diagonals is given in Section 2.9.4.

Ordinary cross validation as described above is not invariant to certain types of transformations of the problem, see Section 4.3 of Wahba (1990). This coupled with some computational issues led to a modified cross validation criteria where s_{ii} in (2.41) is replaced by $\text{trace}(S_\lambda)/n$, the average of the s_{ii}. This procedure is usually called *generalized* cross validation, although it is not more general than ordinary cross validation, it is just a different procedure. This gives the GCV function

$$\text{GCV}(\lambda) = \frac{1}{n}\sum_i \left[\frac{y_i - \hat{g}(x_i;\lambda)}{1 - \text{trace}(S_\lambda)/n}\right]^2.$$

Smoothing parameters chosen to minimize GCV and/or CV have been shown to have good asymptotic properties, see Section 4.4 of Wahba (1990). Basically, the relative difference between the estimated smoothing parameter and a suitably defined optimal value goes to 0 as $n \to \infty$. Unfortunately, the rate at which the difference goes to 0 tends to be very slow. Both ordinary and generalized cross validation are thus not entirely reliable for choosing the smoothing parameter, and sometimes have a tendency to undersmooth.

There has been considerable recent interest in fitting smoothing splines with correlated data. Simulations in that setting have shown that a different criterion, called Generalized Maximum Likelihood (GML), tends to perform better, although the extent to which that applies to independent data is less clear. In the GML procedure, the smoothing parameter is treated as a variance component in a mixed model, and its value estimated using a REML type procedure. See Wang (1998), for example, for a description.

The algorithm given in Section 2.9 to fit a spline for fixed λ is easily extended to find the value of λ that minimizes CV or GCV. The search can be done over a prespecified set of values (the minimum need not be determined to high precision), or searched for using a standard line search algorithm (it is usually recommended that a conservative algorithm, like the golden sections search, be used). At each λ in the search, the spline is estimated as described in Section 2.9. Then the diagonal elements of the smoother matrix are computed, as described above and in Section 2.9.4, and the CV or GCV score computed. Again everything can be done in $O(n)$ operations.

The Splus function `smooth.spline()` has options for cross validation and generalized cross validation. Either can be specified by setting `spar=0` and omitting `df`, with the choice between the two specified using the parameter `cv`.

2.9.4 Calculating the Central Diagonals of the Inverse of a Band Symmetric Positive Definite Matrix

Let $A_{p\times p}$ be a band symmetric positive definite matrix with q nonzero diagonals above the main diagonal. The goal is to calculate the main diagonal and first q diagonals above the main diagonal for the matrix A^{-1}. Note that A^{-1} is generally not a band matrix. The algorithm described below is due to Hutchinson and de Hoog (1985); see also Green and Silverman (1994).

Let $U^tU = A$ be the Cholesky factorization of A. To solve for the components of A^{-1}, we usually consider the system of equations

$$U^tUW_{p\times p} = I_{p\times p},$$

where I is the identity matrix, and the solution is $W = A^{-1}$. Instead of first solving for $(U^t)^{-1}$,

which would usually be the first step in solving this system, consider modifying the equations to

$$UW = (U^t)^{-1}. \tag{2.42}$$

Why does this help? Since W is symmetric, we only need to consider the equations in the upper triangle of this $p \times p$ array of equations. And since U^t is lower triangular, $(U^t)^{-1}$ is also lower triangular, so the only nonzero elements in the upper triangle of $(U^t)^{-1}$ are on the main diagonal, and these elements are just $1/u_{ii}$, where u_{ij} is the element in the ith row and jth column of U. Thus the advantage in working with (2.42) is that we never need to calculate the rest of $(U^t)^{-1}$.

Since A is banded, so is its Cholesky factor, so the only nonzero elements in U are on the main diagonal and on the next q diagonals above the main diagonal. Denoting the element of W in the lth row and mth column by $w(l, m)$, then the iith equation in the array of equations can be written

$$\sum_{l=i}^{\min(i+q,p)} u_{il} w(l, i) = 1/u_{ii}.$$

Noting that W is symmetric, this can be rewritten

$$\sum_{l=i}^{\min(i+q,p)} u_{il} w(i, l) = 1/u_{ii},$$

to express it entirely in terms of the elements of W of interest (the bands in the upper triangle). Also, for $i < j \le \min(i + q, p)$ the ijth equation can be written

$$\sum_{l=i}^{\min(i+q,p)} u_{il} w(l, j) = 0,$$

or again using the symmetry of W as

$$\sum_{l=i}^{j} u_{il} w(l, j) + \sum_{l=j+1}^{\min(i+q,p)} u_{il} w(j, l) = 0.$$

The algorithm then starts from the lower right corner of the array, and solves these equations in the order (p, p), $(p - 1, p)$, $(p - 1, p - 1)$, $(p - 2, p)$, $(p - 2, p - 1)$, $(p - 2, p - 2)$, $\ldots$, $(1, 1)$. The (p, p) equation gives

$$u_{pp} w(p, p) = 1/u_{p,p}, \quad \text{so} \quad w(p, p) = 1/u_{pp}^2.$$

The $(p - 1, p)$ equation gives

$$u_{p-1,p-1} w(p - 1, p) + u_{p-1,p} w(p, p) = 0, \quad \text{so} \quad w(p - 1, p) = -u_{p-1,p} w(p, p)/u_{p-1,p-1}.$$

The $(p - 1, p - 1)$ equation gives

$$u_{p-1,p-1} w(p - 1, p - 1) + u_{p-1,p} w(p - 1, p) = 1/u_{p-1,p-1},$$

so

$$w(p - 1, p - 1) = 1/u_{p-1,p-1}^2 - u_{p-1,p} w(p - 1, p)/u_{p-1,p-1}.$$

Continuing in this fashion, from the equations in the ith row we find

$$\begin{aligned} w(i, \min(i+q,p)) &= -\frac{1}{u_{ii}} \sum_{l=i+1}^{\min(i+q,p)} u_{il} w(l, \min(i+q,p)), \\ w(i,j) &= -\frac{1}{u_{ii}} \left(\sum_{l=i+1}^{j} u_{il} w(l,j) + \sum_{l=j+1}^{\min(i+q,p)} u_{il} w(j,l) \right), \quad i < j < \min(i+q,p), \\ w(i,i) &= 1/u_{ii}^2 - \sum_{l=i+1}^{\min(i+q,p)} u_{il} w(i,l)/u_{ii}. \end{aligned}$$

The values $w(\cdot,\cdot)$ needed at each stage in the recursion have already been determined at earlier steps.

2.10 Additional Exercises

Exercise 2.9 Updating Choleski factorizations.

1. Suppose $A_{p\times p}$ is a positive definite matrix, and let $U = (u_{ij})$ be the Choleski factorization, $U'U = A$. Suppose A is augmented to a $(p+1)\times(p+1)$ matrix by adding a new row and column with elements $a_{p+1,j} = a_{j,p+1}$, $j = 1,\ldots,p+1$ (assume this matrix is also positive definite.) Let $V = (v_{ij})$ be the Choleski factor of this augmented matrix. Give formulas for the v_{ij} in terms of the u_{ij} and $a_{j,p+1}$.

   This could arise, for example, if A were the $X'X$ matrix for a regression model, and a new covariate were added to the model.

2. Suppose

$$A = \begin{pmatrix} 3 & 1 & 1 & 1 \\ 1 & 5 & 3 & 2 \\ 1 & 3 & 8 & 3 \\ 1 & 2 & 3 & 9 \end{pmatrix}.$$

   (a) Compute the Choleski factor U of $A = U'U$.

   (b) Suppose that the second row and column are deleted from A, giving $A^*_{3\times 3}$. Then U can be updated to be the Choleski factor of A^* by dropping the second column of U, giving $U^*_{4\times 3} = (u^*_{ij})$, and using plane rotations (see Section 2.4) to zero-out the subdiagonal elements u^*_{32} and u^*_{43}.

      i. First a rotation can be applied to 2nd and 3rd coordinates to zero-out u^*_{32}. Give the matrix of this rotation G, and the result of applying this transformation to U^*.

      ii. Next a rotation H of the 3rd and 4th coordinates can be applied to GU^* to zero-out $(GU^*)_{43}$. Give the matrix H and the result of applying this transformation to GU^*. The upper 3×3 portion of the result should be the updated Choleski factor.

      iii. Verify the result of the previous part by directly computing the Choleski factor of A^*.

Exercise 2.10 Solve the system of equations

$$Ax = b,$$

where

$$A = \begin{pmatrix} 5 & 2 & 3 & 5 & 3 \\ 3 & 2 & 6 & 7 & 4 \\ 5 & 2 & 7 & 9 & 1 \\ 9 & 4 & 8 & 3 & 10 \\ 3 & 2 & 2 & 3 & a_{55} \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} 4 \\ -3 \\ -4 \\ 9 \\ 5 \end{pmatrix},$$

for $a_{55} = 6 - 10^{-k}$, $k = 2, 4, 6, 8, 10$. In each case estimate the accuracy of the solution $\hat{x}$, assuming the relative error in $A\hat{x}$ is approximately relative machine precision.

Exercise 2.11 Now consider the system of equations from Exercise 2.10 with $a_{55} = 6$. What is the rank of A? Identify all solutions to the system of equations in this case. Also identify all solutions if instead $b = (5, 9, 3, 10, 3)'$.

Exercise 2.12 The constrained least squares problem is to find β to minimize

$$(y - X\beta)'(y - X\beta)$$

subject to $C\beta = b$. Here y is $n \times 1$, X is $n \times p$, β is $p \times 1$, C is $m \times p$ ($m < p$), and b is $m \times 1$. Assume $\text{rank}(X) = p$ and $\text{rank}(C) = m$. Develop an algorithm to solve this problem. There are a variety of ways this can be done. Discuss the speed and accuracy of possible approaches, and explain why you make the particular choices you do in your algorithm.

Use your algorithm to solve for $\hat{\beta}$ if

$$y = \begin{pmatrix} 5 \\ 5 \\ 1 \\ 6 \\ 6 \\ 1 \\ 10 \\ 2 \\ 12 \\ 7 \\ 5 \\ 1 \end{pmatrix}, \quad X = \begin{pmatrix} 2 & 11 & 12 & 4 \\ 14 & 10 & 12 & 6 \\ 15 & 14 & 10 & 9 \\ 4 & 7 & 12 & 2 \\ 8 & 8 & 9 & 1 \\ 7 & 13 & 6 & 8 \\ 4 & 7 & 9 & 2 \\ 6 & 9 & 7 & 3 \\ 8 & 2 & 4 & 1 \\ 6 & 10 & 6 & 13 \\ 4 & 11 & 2 & 3 \\ 4 & 11 & 7 & 4 \end{pmatrix},$$

$$C = \begin{pmatrix} 1.80 & 2.3 & 0.6 & 2.1 \\ 0.35 & 0.8 & 1.6 & 0.6 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 2 \end{pmatrix}.$$

(Note that if this is thought of as a linear model, then the model does not contain a constant term.)

Exercise 2.13 The solution to a linear least squares problem with design matrix X can be computed by (a) using the Choleski factorization of $X'X$, (b) using the QR decomposition of X, or (c) using the singular value decomposition of X. In each case, once the decomposition is formed, it is still necessary to solve for the parameter estimates. For each method, determine the approximate number of FLOPS required to solve for the parameter estimates, once the decomposition has been computed. Include all additional calculations needed, including any operations on the response vector, such as computing $X'y$ in method (a).

Exercise 2.14 It is easy to construct examples where methods based on the normal equations fail, but direct decomposition of the design matrix X can be performed. Suppose the components of the design matrix are defined by

$$x_{ij} = \begin{cases} 1 & i \geq j, \\ 1 - 10^{-7} & i < j, \end{cases}$$

for $i = 1, \ldots, 100$ and $j = 1, \ldots, 5$, and suppose the response vector is `1:100` (in Splus notation).

1. Calculate both the QR and SVD estimates of the parameter values, and note the 1-norm and 2-norm estimated condition numbers of X. Also compute the residual sum of squares and R^2 for this problem.

2. Calculate $X'X$, and attempt to estimate the regression parameters from the Choleski factorization of this matrix. Also use the SVD of $X'X$ (using both `tol=0` and `tol=1e-15`) to solve the normal equations. Calculate the residual sum of squares for both the SVD solutions.

3. The model for the above problem can be written

   $$y_i = \beta_1 + \beta_2 x_{i2} + \cdots + \beta_5 x_{i5} + \epsilon_i.$$

   As discussed in Section 2.6.1, when the model contains a constant term (as is the case here), it is often numerically advantageous to subtract off the means of the covariates, giving the reparameterization

   $$y_i = \alpha + \beta_2(x_{i2} - \overline{x}_2) + \cdots + \beta_5(x_{i5} - \overline{x}_5) + \epsilon_i,$$

   where $\overline{x}_j = \sum_i x_{ij}/n$. For the reduced model with responses $y_i - \overline{y}$ and covariates $x_{i2} - \overline{x}_2, \ldots, x_{i5} - \overline{x}_5$ (and no constant term), form the normal equations and estimate the parameters using the Choleski factorization. How do the results compare with those from the original QR decomposition? What is the condition number of $X'X$ in the reduced model?

Exercise 2.15 In this problem, simulations are used to study the performance of spline smoothing with the smoothing parameter selected using cross validation. All calculations can be done in Splus, with the splines fit using the `smooth.spline()` function.

Let the covariate values x_i be equally spaced over the interval $(0, 1)$. Generate samples from the model

$$y_i = \exp[-20(x_i - .5)^2] + \epsilon_i, \quad i = 1, \ldots, 100,$$

with $\epsilon_i \sim \mathrm{N}(0, .25)$. For each generated sample, fit a smoothing spline using ordinary cross validation, and compute the quantities needed to answer the following questions. (This will need to be repeated for many samples, with the answers given by summaries of the results from the repeated samples. Some justification of the number of samples used should be given. In particular you may want to calculate standard errors of estimated quantities.)

1. What is the expected value of the "degrees of freedom" of the spline smooth? How much sample to sample variation is there in the degrees of freedom?
2. What are the bias, variance, and MSE of the spline estimate at $x = .1, .25, .5, .8$? (Note that there is a function `predict.smooth.spline()`.)

2.11 References

Anderson E, Bai Z, Bischof C *et. al.* (1995). *LAPACK Users' Guide, 2nd Edition*. SIAM.

De Boor C (1978). *A Practical Guide to Splines*. New York: Springer-Verlag.

Eubank RL (1988). *Spline Smoothing and Nonparametric Regression*. New York: Marcel-Dekker. (Statistics: textbooks and monographs, vol. 90.)

Golub GH and van Loan CF (1989). *Matrix Computations, 2nd Edition* Johns Hopkins University Press. (a third edition was published in 1996)

Green PJ and Silverman BW (1994). *Nonparametric Regression and Generalized Linear Models*. Chapman and Hall.

Hastie TJ and Tibshirani RJ (1990). *Generalized Additive Models*. London: Chapman and Hall. (Monographs on Statistics and Applied Probability, vol. 43.)

Hutchinson MF and de Hoog FR (1985). Smoothing noisy data with spline functions. *Numer. Math.*, 47:99–106.

Lange K (1999). *Numerical Analysis for Statisticians*. Springer.

O'Sullivan F (1990). An iterative approach to two-dimensional Laplacian smoothing with application to image restoration. *Journal of the American Statistical Association*, 85:213-219.

Press WH, Teukolsky SA, Vetterling WT, and Flannery BP (1992). *Numerical Recipes in C: The Art of Scientific Computing. Second Edition*. Cambridge University Press.

Seber GAF (1977). *Linear Regression Analysis*. Wiley.

Silverman BW (1985). Some aspects of the spline smoothing approach to non-parametric regression curve fitting (with discussion). *Journal of the Royal Statistical Society, Series B*, 47:1–52.

Thisted RA (1988). *Elements of Statistical Computing*. Chapman & Hall.

Venables WN and Ripley BD (1997) *Modern Applied Statistics with S-Plus, 2nd Edition*. Springer. (the third edition is now available)

Wahba G (1990). *Spline Models for Observational Data.* Philadelphia: SIAM.

Wang Y (1998). Smoothing spline models with correlated random errors. *Journal of the American Statistical Association*, 93:341–348.

Wegman EJ and Wright IW (1983). Splines in statistics. *Journal of the American Statistical Association*, 78:351–365.

Chapter 3

Unconstrained Optimization and Solving Nonlinear Equations

3.1 One-Dimensional Problems

3.1.1 Solving a Single Nonlinear Equation

Problems that require solving a single nonlinear equation occur frequently in statistics.

Example 3.1 Let X be the observed number of responders out of n patients entered on a phase II cancer clinical trial. Suppose $X \sim \text{Binomial}(n, p)$. Having observed $X = r$, the 'exact' $1 - \alpha$ upper confidence limit on p is defined as the value p_u satisfying

$$\alpha = P(X \leq r|p_u) = \sum_{i=0}^{r} \left(\begin{array}{c} n \\ i \end{array} \right) p_u^i (1 - p_u)^{n-i},$$

and the $1 - \alpha$ exact lower confidence limit is defined as the value p_l of p satisfying

$$\alpha = P(X \geq r|p_l) = 1 - P(X < r|p_l),$$

(with p_l defined to be 0 if $r = 0$ and p_u defined to be 1 if $r = n$). Both cases give a nonlinear equation to solve. □

The simplest algorithm for solving a single equation is the bisection search. Implementation of a bisection search requires specifying an initial bracketing interval. For a continuous function f, an interval $[a, b]$ brackets a solution to $f(x) = c$ if $f(a) - c$ and $f(b) - c$ have opposite signs. The following function implements a bisection search to solve for the lower confidence limit. This function assumes that α will be a fairly small positive value, such as .05 or .10, so the root will be $< r/n$, and the initial bracketing interval is thus chosen to be $[0, r/n]$. At each step in the bisection search the midpoint of the current bracketing interval is determined, and one of the two resulting subintervals must still bracket a solution. This subinterval becomes the new bracketing interval, which is then split on the midpoint, and so on. The process continues until the width of the bracketing interval has determined the solution to sufficient accuracy. In the function, the

quantities `pll` and `plu` bracket the solution. At each step the midpoint of the current bracketing interval is computed, and either `pll` or `plu` is replaced by the midpoint, keeping a solution bracketed between the two values. The iteration terminates when the relative difference `(plu-pll)/plu < eps`. Typically 2 or 3 significant digits are adequate for statistical purposes.

```
> blci <- function(r,n,alpha=.05,eps=1e-3) {
+   if(r <= 0) pl <- 0 else {
+     plu <- r/n
+     pll <- 0
+     pl <- (pll + plu)/2
+     u <- 1 - pbinom(r - 1, n, pl)
+     i <- 1
+     while ((plu-pll)/plu > eps) {
+       i <- i+1
+       if(u > alpha) {
+         plu <- pl
+         pl <- (pll + plu)/2
+         u <- 1 - pbinom(r - 1, n, pl)
+       } else {
+         pll <- pl
+         pl <- (pll + plu)/2
+         u <- 1 - pbinom(r - 1, n, pl)
+       }
+     }
+   }
+   c(lcl=pl,neval=i)
+ }
> blci(5,15,eps=1e-2)
      lcl neval
 0.141276     9
> blci(5,15,eps=1e-3)
       lcl neval
 0.1416423    13
> blci(5,15,eps=1e-4)
       lcl neval
 0.1416677    16
> blci(5,15,eps=1e-6)
      lcl neval
 0.141664    23
> blci(5,15,eps=1e-8)
      lcl neval
 0.141664    29
>
```

Note how the number of evaluations of the binomial CDF is nearly linear in the values of $\log_{10}$(`eps`) (which is approximately the number of significant digits). This is typical of the

bisection search, which is said to have *linear* convergence.

The bisection search used no information on the value of the function beyond whether it is positive or negative. Splus and R have a function `uniroot()` which implements a more sophisticated search algorithm (Brent's method, discussed in Section 9.3 of Press *et. al.* (1992)). This method uses local quadratic interpolation, which will converge much faster once the iteration gets close to the root. To guarantee convergence, the method still keeps a bracket on the solution, and if the quadratic interpolation does not produce an acceptable point it rejects it and uses a bisection step instead.

To use `uniroot()`, the user must first create a function evaluating the equation to be solved. The required arguments to `uniroot()` are the function evaluating the equation and an interval bracketing a solution. The argument `tol` can be used to change the convergence criterion. The description in the help file states that the iteration stops when the length of the bracketing interval is <`tol`, which from the output apparently means the absolute length (this cannot be verified from the code as the test is apparently done within a call to a FORTRAN routine). In the calls below `tol=eps*.142` to make the precision similar to that of the previous bisection algorithm. Only part of the output is displayed (`root` giving the root, `f.root` giving the value of `f` at the root, `nf` giving the number of function evaluations, and `neg` and `pos` giving the range of the bracketing interval at termination. The `unlist()` command is used to collapse the list returned by `uniroot()` into a vector. (See the help file for more details on `uniroot()`.)

```
> # solve f(p)=0 for lower binomial confidence limit
> f <- function(p,r,n,alpha=.05) 1-pbinom(r-1,n,p) - alpha
> unlist(uniroot(f,c(0,5/15),r=5,n=15,tol=.142*1e-2)[c(1:4,6)])
     root        f.root nf      neg      pos
 0.141602 -8.135718e-05  9 0.141602 0.142312
> unlist(uniroot(f,c(0,5/15),r=5,n=15,tol=.142*1e-3)[c(1:4,6)])
     root      f.root nf      neg      pos
 0.141673 1.17973e-05  9 0.141602 0.141673
> unlist(uniroot(f,c(0,5/15),r=5,n=15,tol=.142*1e-4)[c(1:4,6)])
      root        f.root nf       neg       pos
 0.1416632 -9.648463e-07 10 0.1416632 0.1416703
> unlist(uniroot(f,c(0,5/15),r=5,n=15,tol=.142*1e-6)[c(1:4,6)])
     root       f.root nf       neg      pos
 0.141664 1.083704e-11 11 0.1416639 0.141664
> unlist(uniroot(f,c(0,5/15),r=5,n=15,tol=.142*1e-8)[c(1:4,6)])
     root       f.root nf      neg      pos
 0.141664 1.083704e-11 11 0.141664 0.141664
```

Note that bisection and `uniroot()` need a similar number of function evaluations to reach about 3 significant digits. However, once `uniroot()` gets close to a solution, it gives much better accuracy with only a little additional work, while the bisection method continues the slow (but safe) linear rate from the early part of the algorithm.

Another widely used approach is Newton's method, which in its basic form repeatedly solves the tangent line approximation to the nonlinear problem. That is, given the current iterate or initial

value $x^{(i)}$, Newton's method approximates the equation $f(x) = 0$ with $f(x^{(i)}) + f'(x^{(i)})(x - x^{(i)}) = 0$. The solution to the linear approximation is $x^{(i+1)} = x^{(i)} - f(x^{(i)})/f'(x^{(i)})$, which becomes the new guess at the solution. This iteration is repeated until convergence. Unfortunately, this algorithm often diverges unless the starting value is close to the solution. A modified version first starts with a bracketing interval and computes new candidate points using Newton's method, but only uses them if they fall within the interval. If not, then a bisection step could be used instead. Regardless of how the new point is chosen, the bracketing interval is updated at each iteration as before. See Section 9.4 of Press *et. al.* (1992) for details. An alternate approach, which does not require specifying a bracketing interval, will be described for the multiparameter setting in Section 3.6.

3.1.2 Rates of Convergence

Let $x_1, x_2, \ldots$ be a sequence converging to a value x^*. In particular, think of the values of p_l given at each iteration of one of the above algorithms. If there is a c, $0 \leq c < 1$, such that for all i sufficiently large,

$$|x_{i+1} - x^*| < c|x_i - x^*|,$$

then the sequence converges linearly. If there is a sequence $c_i \to 0$ such that

$$|x_{i+1} - x^*| < c_i|x_i - x^*|$$

(for i sufficiently large), then the sequence $\{x_i\}$ converges superlinearly. If there is a c, $0 \leq c < 1$, and a $p > 1$ such that

$$|x_{i+1} - x^*| < c|x_i - x^*|^p,$$

then the sequence $\{x_i\}$ converges with order at least p. If this holds for $p = 2$ then the sequence has quadratic convergence.

Since the width of the bounding interval is reduced by 1/2 at each iteration in a bisection search, the sequence of values generated is guaranteed to converge linearly (with $c = 1/2$). Newton's method has quadratic convergence, when it converges.

Thisted (1988, Section 4.2) gives a number of other algorithms. One of these, the Illinois method, is safe and improves substantially on the bisection search, having a convergence order of 1.442.

3.1.3 One-Dimensional Minimization

Given a function $f(x)$, where x is a scalar, the one-dimensional minimization problem is to determine $\hat{x}$ such that $f(\hat{x}) \leq f(x)$ for all x. If f has a continuous derivative, then a solution to the minimization problem will satisfy $f'(x) = 0$, and so methods described above for solving equations can be applied to this problem as well, but this approach will not be discussed further here. The focus in this section is on directly minimizing f without making use of derivative information. Newton's method and some other derivative based methods will be considered later for multi-parameter problems.

Bracketing a minimum is more complicated than bracketing a solution to an equation, because to be sure that an interval contains a minimum, the function must also be evaluated at an interior

point that has a smaller function value than the endpoints. Thus keeping track of a bracketing interval requires tracking 3 points. The safe, simple approach to locating a minimum, given a bracketing interval, is the golden section search, which will be described next.

Golden Section Search

At any stage in the golden section search, there are 3 points $x_1 < x_2 < x_3$, with the interval thought to contain a (local) minimum. A new point $x_0 \in (x_1, x_3)$ is chosen, and a new bracketing interval formed based on the values of $f(x_0)$ and $f(x_2)$. For example, if $x_0 < x_2$, then the new bracketing interval is (x_0, x_3) if $f(x_0) > f(x_2)$, and is (x_1, x_2) if $f(x_0) < f(x_2)$.

The main question is how to choose the new point x_0. If the point x_2 had been chosen appropriately, it is possible to make the proportionate reduction in the size of the interval the same at every iteration. Let α be the proportion of the interval eliminated at each step. If $x_0 < x_2$, then for this to happen regardless of the value of $f(x_0)$, the points must satisfy $x_0 - x_1 = x_3 - x_2 = \alpha(x_3 - x_1)$, and to be able to get the same proportionate reduction at the next iteration, the points must also satisfy $x_2 - x_0 = \alpha(x_3 - x_0) = \alpha[\alpha(x_3 - x_1) + (x_2 - x_0)]$, so $x_2 - x_0 = (x_3 - x_1)\alpha^2/(1 - \alpha)$. Since $(x_0 - x_1) + (x_2 - x_0) + (x_3 - x_2) = x_3 - x_1$, it follows that $2\alpha + \alpha^2/(1 - \alpha) = 1$, which is a quadratic equation whose only solution satisfying $0 < \alpha < 1$ is $\alpha = (3 - \sqrt{5})/2$. The proportion of the interval retained at each iteration, $1 - \alpha = (\sqrt{5} - 1)/2 \doteq .618$, is known as the golden mean.

In the golden section search, given an interval $[x_1, x_3]$ thought to contain a minimum, initially interior points are located at $x_0 = x_1 + \alpha(x_3 - x_1)$ and $x_2 = x_3 - \alpha(x_3 - x_1)$. Then $f(x_0)$ and $f(x_2)$ are evaluated. If $f(x_0) < f(x_2)$ then the new interval is $[x_1, x_2]$ and the next point added is $x_1 + \alpha(x_2 - x_1)$, and if $f(x_0) > f(x_2)$ then the new interval is $[x_0, x_3]$ and the next point added is $x_3 - \alpha(x_3 - x_0)$. The algorithm continues in this fashion until the width of the interval determines the solution to sufficient precision. Note that it is not necessary to evaluate f at the endpoints of the initial interval, and if this interval turns out not to contain a local minimum, then the algorithm will converge to one of the endpoints.

A function implementing the golden section search is given below. Here `f` is a function whose first argument is the variable of the minimization (x above), `brack.int` is the bracketing interval (a vector of length 2), and '`...`' are additional arguments to `f`. The iteration terminates when the relative length of the interval is `< eps`.

```
golden <- function(f,brack.int,eps=1.e-4,...) {
  g <- (3-sqrt(5))/2
  xl <- min(brack.int)
  xu <- max(brack.int)
  tmp <- g*(xu-xl)
  xmu <- xu-tmp
  xml <- xl+tmp
  fl <- f(xml,...)
  fu <- f(xmu,...)
  while(abs(xu-xl)>(1.e-5+abs(xl))*eps) {
```

```
    if (fl<fu) {
      xu <- xmu
      xmu <- xml
      fu <- fl
      xml <- xl+g*(xu-xl)
      fl <- f(xml,...)
    } else {
      xl <- xml
      xml <- xmu
      fl <- fu
      xmu <- xu-g*(xu-xl)
      fu <- f(xmu,...)
    }
  }
  if (fl<fu) xml else xmu
}
```

To illustrate the use of this function, the multinomial log likelihood

$$1997\log(2+\theta) + 1810\log(1-\theta) + 32\log(\theta), \tag{3.1}$$

where $0 < \theta < 1$, will be maximized. The probabilities of the three categories are actually $p_1 = (2+\theta)/4$, $p_2 = (1-\theta)/2$ and $p_3 = \theta/4$, but the constant divisors have been dropped from the likelihood. This likelihood is discussed by Thisted (1988, Section 4.2.6.1). Since the function is designed to locate a minimum, the function `f` evaluates the negative of the log likelihood.

```
> f <- function(theta) -1997*log(2+theta)-1810*log(1-theta)-32*log(theta)
> golden(f,c(.001,.999))
[1] 0.03571247
```

Brent's Method

The golden section search is safe, but has a slow linear rate of convergence. Methods based on fitting quadratic interpolants to the evaluated points can converge much faster once they get close to the solution. Details of implementation in a way that maintains a bracketing interval and guarantees convergence are a little involved, and there is substantial bookkeeping involved in keeping track of the points. A fairly standard algorithm, known as Brent's method, is described in Section 10.2 of Press *et. al.* (1992).

The Splus (and R) function `optimize()` will perform one-dimensional minimization (or optionally maximization) using Brent's method (although it is not clear how similar the algorithm is to that in Press *et. al.* (1992)). The `optimize()` function does maintain a bracket on the minimum (the initial bracket is supplied by the user as the `interval` argument), and uses a minimum of a local quadratic approximation to choose the next point. If the new point does not meet certain criteria, the algorithm reverts to a golden section step. As with the function `uniroot()` above, `optimize()` has an argument `tol` that controls the precision needed at convergence. Again as in

`uniroot()`, it appears the iteration stops when the absolute width of the bracketing interval is `<tol`.

As an example, again consider the multinomial likelihood given in Section 4.2.6.1 (page 175) of Thisted (1988), with log likelihood given by (3.1).

In the call to `optimize()` below, the function `f` returns the negative of the log likelihood, as before. The initial lower and upper limits on the bracketing interval need to be > 0 and < 1. In the output, `nf` gives the number of function evaluations used, and the `interval` values give the bracketing interval at termination. (See the help file for more details on `optimize()`.)

```
> f <- function(theta)
+   -1997*log(2+theta)-1810*log(1-theta)-32*log(theta)
> unlist(optimize(f,c(.001,.999),tol=1e-4)[1:4])
    minimum objective nf  interval1  interval2
 0.03571547 -1247.105 12 0.03568214 0.03577505
> unlist(optimize(f,c(.001,.999),tol=1e-8)[1:4])
   minimum objective nf interval1 interval2
 0.0357123 -1247.105 15 0.0357123 0.0357123
```

The precision with 15 function evaluations is similar to that for the Newton-Raphson and scoring algorithms given by Thisted (1988, p. 176) after 6–7 iterations. However, those algorithms require both first and second derivative calculations, so the overall computational burden is not very different.

If the initial bracketing interval does not contain a local minimum, the algorithm still works correctly, in that it converges to the minimum within the specified interval, which will be one of the endpoints. (If the function is decreasing as an endpoint of the initial interval is approached from within the interval, it is possible for the algorithm to converge to the endpoint even if the interval does contain a local minimum. In that case, though, the endpoint is still a local minimum within the interval, even though the function may continue to decrease outside the interval, so the algorithm is technically still converging to a local minimum within the interval.)

```
> unlist(optimize(f,c(.2,.999),tol=1e-8)[1:4])
 minimum objective nf interval1 interval2
     0.2 -1119.158 39       0.2       0.2
```

3.2 The Newton-Raphson Optimization Algorithm

Some Notation and Definitions:
For a function $f : R^p \to R^1$, the gradient is the vector

$$\nabla f(x) = \left(\frac{\partial f(x)}{\partial x_1}, \cdots, \frac{\partial f(x)}{\partial x_p} \right)',$$

and the Hessian matrix (or just Hessian) is the matrix of second partial derivatives

$$\nabla^2 f(x) = \left(\frac{\partial^2 f(x)}{\partial x_i \partial x_j} \right).$$

For a transformation $G : R^p \to R^p$, defined by $G(x) = (g_1(x), \ldots, g_p(x))'$, the Jacobian is the matrix of partial derivatives

$$J_G(x) = \left(\frac{\partial g_i(x)}{\partial x_j} \right).$$

(The ith row of J_G is $[\nabla g_i(x)]'$.). For $f : R^p \to R^1$, the Hessian is the Jacobian of the gradient; that is, $\nabla^2 f(x) = J_{\nabla f}(x)$.

The directional derivative of a function $f : R^p \to R^1$ at x in the direction d is defined by

$$\lim_{\epsilon \to 0} \frac{f(x + \epsilon d) - f(x)}{\epsilon} = \frac{\partial}{\partial \epsilon} f(x + \epsilon d) \Big|_{\epsilon = 0} = d' \nabla f(x).$$

A function $f : R^p \to R^1$ is convex on a set A if $\lambda f(a) + (1 - \lambda) f(b) \geq f[\lambda a + (1 - \lambda) b]$ for all $a, b \in A$ and $0 < \lambda < 1$. The function is strictly convex if equality holds only when $a = b$. For smooth functions, f is convex on A if $\nabla^2 f(x)$ is nonnegative definite (nnd) for all $x \in A$. If $\nabla^2 f(x)$ is positive definite (pd) for all $x \in A$, then f is strictly convex on A.

3.2.1 The Problem

The general unconstrained minimization problem for a smooth function $f : R^p \to R^1$ is to find an x^* such that

$$f(x^*) = \min_x f(x), \tag{3.2}$$

where the minimum is over all $x \in R^p$. In general such an x^* need not exist (the function could be decreasing as x becomes infinite in some direction). Another problem is that there may be multiple local minima. A local minimum is a point x^* such that

$$f(x^*) \leq f(x) \tag{3.3}$$

for all x in some neighborhood of x^*. Generally it is impossible to guarantee convergence of a numerical algorithm to a global minimum, unless the function is known to be everywhere convex. For this reason, the problem considered will be that of trying to find a local minimum x^*.

If x^* is a local minimum of $f(x)$, then x^* is a local maximum of $c - f(x)$ for any constant c. Thus there is no loss of generality in restricting attention to minimization. In particular, maximum likelihood estimates for a log likelihood $l(\theta)$ can be found by minimizing $-l(\theta)$.

For a smooth function f, if x^* is a local minimum, then $\nabla f(x^*) = 0$. If $\nabla f(x^*) = 0$ and $\nabla^2 f(x^*)$ is nnd, then x^* is a local minimum. Thus the search for a minimum can also focus on finding points x^* satisfying $\nabla f(x^*) = 0$. Such points need not be local minima, since in general they could also be local maxima or saddle points.

Many algorithms for searching for a local minimum are similar in structure to the following general outline.

1. Given the current point x_0, choose a direction d in which to move next.
2. Find a point $x_1 = x_0 + \lambda d$ such that $f(x_1) < f(x_0)$.

3. Set $x_0 = x_1$, and repeat (until convergence).

For algorithms of this type to be successful, it is important that the direction d chosen at each stage be a descent direction for f; that is, a direction in which f is decreasing. Formally, a direction d is a descent direction for f at x_0 if

$$f(x_0 + \lambda d) < f(x_0) \quad \text{for } 0 < \lambda < \epsilon,$$

for some $\epsilon > 0$. From the definition of a directional derivative above, it is clear that d is a descent direction for f at x_0 if and only if $d'\nabla f(x_0) < 0$.

From the Cauchy-Schwarz inequality,

$$|d'\nabla f(x)|/\|d\|_2 \leq \|\nabla f(x)\|_2,$$

and taking $d = \nabla f(x)$ gives equality, so $\nabla f(x)$ (normalized) is the direction of most rapid increase in f at x, and $-\nabla f(x)$ is the direction of most rapid decrease. This suggests considering $d = -\nabla f(x)$ in the above algorithm. This is called the method of steepest descent. It turns out to perform poorly for most functions, though; see Exercise 3.1.

3.2.2 The Basic Algorithm

Newton-Raphson is a method for iteratively searching for a local minimum, that can perhaps most easily be thought of as searching for solutions to the system of nonlinear equations $\nabla f(x) = 0$. In this method, given the current point x_0, the gradient $\nabla f(x)$ is approximated by

$$\nabla f(x_0) + \nabla^2 f(x_0)(x - x_0),$$

the plane tangent to $\nabla f(x)$ at x_0. The solution to the system of linear equations

$$\nabla f(x_0) + \nabla^2 f(x_0)(x - x_0) = 0$$

is then the next guess at the solution of the minimization problem. Formally, this solution can be written

$$x_1 = x_0 - [\nabla^2 f(x_0)]^{-1}\nabla f(x_0).$$

The algorithm continues computing updates from this formula until convergence is reached, or since it often does not converge, until numerical errors occur or a maximum number of iterations is exceeded.

Example 3.2 Consider the negative log likelihood for the logistic regression model for a binary response y_i and a single covariate z_i,

$$f(\beta) = -\sum_{i=1}^{n}(y_i[\beta_0 + \beta_1 z_i] - \log[1 + \exp(\beta_0 + \beta_1 z_i)]),$$

which has gradient

$$\nabla f(\beta) = -\sum_i \begin{pmatrix} y_i - p_i \\ z_i[y_i - p_i] \end{pmatrix},$$

where $p_i = P(y_i = 1|z_i) = \exp(\beta_0 + \beta_1 z_i)/[1 + \exp(\beta_0 + \beta_1 z_i)]$, and Hessian

$$\nabla^2 f(\beta) = \sum_i \begin{pmatrix} 1 \\ z_i \end{pmatrix} (1, z_i) p_i (1 - p_i).$$

Provided the covariate is not everywhere constant, the Hessian is positive definite at all parameter values, which follows from

$$(a_1, a_2)\nabla^2 f(\beta)(a_1, a_2)' = \sum_i (a_1 + a_2 z_i)^2 p_i (1 - p_i) > 0$$

unless $a_1 = a_2 = 0$. Thus this is a particularly well behaved problem. In spite of this the Newton-Raphson iteration diverges unless the starting point is fairly close to the solution, as the following calculations show.

```
> f1 <- function(b,r,z) {
+ ### function to compute logistic regression -log likelihood, -score and
+ ### information. b=parameters, r=binary response, z=covariate
+   u <- b[1]+b[2]*z
+   u2 <- exp(u)
+   l <- -sum(u*r-log(1+u2))
+   p <- u2/(1+u2)
+   s <- -c(sum(r-p),sum(z*(r-p)))
+   v <- matrix(c(sum(p*(1-p)),sum(z*p*(1-p)),0,sum(z*z*p*(1-p))),2,2)
+   v[1,2] <- v[2,1]
+   list(loglik=l,score=s,inf=v)
+ }
> n <- 100; beta <- c(0,4) # true beta
> # generate some data
> > z <- runif(n)>.1
> p <- exp(beta[1]+z*beta[2])
> p <- p/(1+p)
> r <- runif(n)<p
> table(r,z)
      FALSE TRUE
FALSE     8    6
 TRUE     6   80
> b <- c(1,2) # initial value
> for (i in 1:7) { # iteration
+   q <- f1(b,r,z)
+ ## since q$inf is pd, the following, written for general matrices, is not ideal
+   b <- b-solve(q$inf,q$score)
+   print(c(q$l,b))
+ }
[1]  34.5641759  -0.5384984   3.0439606
[1]  31.4487584  -0.2805269   2.8677997
[1]  31.3220062  -0.2876785   2.8779418
```

```
[1]  31.3218934  -0.2876821   2.8779492
[1]  31.3218934  -0.2876821   2.8779492
[1]  31.3218934  -0.2876821   2.8779492
[1]  31.3218934  -0.2876821   2.8779492
> # converged after a small number of iterations
>
> b <- c(-1,-1) # initial value
> for (i in 1:7) { # iteration
+   q <- f1(b,r,z)
+   b <- b-solve(q$inf,q$score)
+   print(c(q$l,b))
+ }
[1]  181.301473   -0.188096    5.912660
[1]  44.2054842  -0.2871378 -14.4945353
[1]  1.192095e+03 -2.876821e-01  2.444489e+06
Error in .Fortran(if(!cmplx) "dqr" else "zqr",: subroutine dqr: 4 missing
        value(s) in argument 1
Dumped
> f1(c(-.2876821,2.444489e+06),r,z)
$loglik:
[1] Inf

$score:
[1] NA NA

$inf:
     [,1] [,2]
[1,]   NA   NA
[2,]   NA   NA

> # diverged until numerical errors occurred
```

In the second starting point above, the second variable keeps overshooting the minimum, and the algorithm diverges. Also note that with a single binary covariate, the model is saturated, and it is possible to solve for the MLEs directly from the tabulation of `r` by `z`. In particular, $\hat{\beta}_1 = \log(80 * 8/6^2) \doteq 2.877949$. □

3.2.3 Backtracking

The divergence problem with the Newton-Raphson iteration in Example 3.2 is easily fixed. Instead of taking fixed steps at each iteration, the general outline for search algorithms given in Section 3.2.1 can be applied. In particular, regard the Newton-Raphson step $-[\nabla^2 f(x_0)]^{-1}\nabla f(x_0)$ as a direction in which to search for a better point. The modified Newton-Raphson iteration then takes steps of the form

$$x_1(\lambda) = x_0 - \lambda[\nabla^2 f(x_0)]^{-1}\nabla f(x_0), \tag{3.4}$$

where the multiplier λ to be used at each iteration remains to be determined.

As seen in Example 3.2, the unmodified Newton-Raphson iteration tends to converge very quickly once it gets close to a solution. In fact, it has quadratic convergence in multi-dimensional problems, just as it does in the univariate case. It would be be unfortunate if modifications to improve convergence from distant points decreased the rate of convergence once the algorithm got close to a solution, or made the calculations in the later stages substantially less efficient. These considerations lead to the following strategy.

1. First compute the full Newton-Raphson step, corresponding to $\lambda = 1$ in (3.4). If $f(x_1(1)) < f(x_0)$, then keep the new point, and repeat.
2. (Backtracking step). If $f(x_1(1)) \geq f(x_0)$, then reject the new point and backtrack towards x_0 by computing $x_1(\lambda)$ for values $\lambda < 1$, until a better point is found.

It is technically possible to construct sequences where $f(x_1) < f(x_0)$ at each step but where the sequence never converges. For this reason a slightly stronger condition is usually used. Dennis and Schnabel (1983) recommend requiring that λ satisfy

$$f[x_1(\lambda)] < f(x_0) + 10^{-4}[x_1(\lambda) - x_0]'\nabla f(x_0). \tag{3.5}$$

It can be shown that if $x_1 - x_0$ is a descent direction, then this condition can always be satisfied. (If $x_1 - x_0$ is a descent direction, then $[x_1 - x_0]'\nabla f(x_0) < 0$, so this condition requires at least a small decrease in f at each iteration.)

The remaining problem is how to choose the sequence of λ values in the backtracking step (step 2 above). A crude but usually effective strategy is simply to reduce the step length by a fixed fraction δ each time. That is, if $f(x_1(1))$ does not improve on $f(x_0)$, try $x_1(\delta)$. If $f(x_1(\delta))$ does not improve on $f(x_0)$, try $x_1(\delta^2)$, etc. When $\delta = 1/2$ (a common choice), this procedure is called *step halving*. In poorly behaved problems, smaller values of δ will sometimes lead to faster convergence. A more sophisticated backtracking algorithm would use quadratic and cubic interpolation based on previously computed function values and derivatives. An implementation of backtracking using polynomial interpolation is given in the function `lnsrch` in Section 9.7 of Press *et. al.* (1992). Note that even the `lnsrch` backtracking algorithm stops as soon as it finds a point satisfying (3.5); it is not thought to be worthwhile to look for a precise minimum in this direction. This is because the backtracking steps are only being used if the algorithm is far enough away from a local minimum of $f(x)$ for the full Newton-Raphson step to fail. But in this case it is unlikely that the minimum in the search direction will be a very good point in a global sense, either.

The following code implements step halving for the logistic regression example.

Example 3.2 continued.

```
> f2 <- function(b,r,z) { #like f1, except only returns -log(likelihood)
+   u <- b[1]+b[2]*z
+   u2 <- exp(u)
+   -sum(u*r-log(1+u2))
```

```
+ }
> b <- c(-1,-1) # initial value
> for (i in 1:7) { # iteration
+   q <- f1(b,r,z)
+   print(c(q$l,b))
+   db <- -solve(q$inf,q$score)
+   bn <- b+db
+   ln <- f2(bn,r,z)
+   print(ln)
+   while (ln > q$l+1.e-4*sum(db*q$score)) {
+     db <- db/2  # cut the step in half
+     bn <- b+db
+     ln <- f2(bn,r,z)
+     print(ln)
+   }
+   b <- bn
+ }
[1]  181.3015   -1.0000   -1.0000
[1] 44.20548
[1]  44.205484  -0.188096   5.912660
[1] 1192.095
[1] 372.7729
[1] 50.84318
[1] 32.11035
[1]  32.1103513  -0.2004763   3.3617603
[1] 31.4125
[1]  31.4124967  -0.2872491   2.7018403
[1] 31.32232
[1]  31.3223245  -0.2876821   2.8655422
[1] 31.32189
[1]  31.3218934  -0.2876821   2.8778833
[1] 31.32189
[1]  31.3218934  -0.2876821   2.8779492
[1] 31.32189
```

In this example, backtracking was only needed at one iteration, and a better point was found at the third backtracking step. This algorithm (with a few minor modifications) converges almost universally for logistic regression problems with finite MLEs. □

3.2.4 Positive Definite Modifications to the Hessian

There is still a problem with the modified Newton-Raphson iteration given above. In general, there is nothing to guarantee that the Newton-Raphson step direction is a descent direction for f at x_0, and if it is not, the backtracking algorithm will not be able to find a better point satisfying (3.5). Recall that d is a descent direction for f at x_0 if $d'\nabla f(x_0) < 0$. The Newton-Raphson

iteration takes $d = -[\nabla^2 f(x_0)]^{-1}\nabla f(x_0)$, so

$$d'\nabla f(x_0) = -[\nabla f(x_0)]'[\nabla^2 f(x_0)]^{-1}\nabla f(x_0). \tag{3.6}$$

If $\nabla^2 f(x_0)$ is positive definite, then (3.6) is negative at any x_0 for which $\nabla f(x_0) \neq 0$. However, if $\nabla^2 f(x_0)$ is indefinite (or even negative definite), then there is no guarantee that (3.6) will be negative, and positive values are possible. (An example of this will be seen in a Weibull regression model, later.) Since $d = -A^{-1}\nabla f(x_0)$ is a descent direction for any positive definite matrix A (that is, $d'\nabla f(x_0) < 0$ in this case), the usual approach when $\nabla^2 f(x_0)$ is not positive definite is to use some other matrix A which is positive definite. Since $\nabla^2 f(x)$ is positive definite at a local minimum, it will usually be pd in a neighborhood of a local minimum, and once the Newton-Raphson iteration with backtracking gets close enough to a local minimum, the iteration will converge quickly. Again it is not desirable to do anything which would disrupt this property. The usual approach is to first check whether $\nabla^2 f(x_0)$ is pd, and if it is, just use it. If not, then modify it as little as possible to get a matrix which is pd. One such modification is to replace $\nabla^2 f(x_0)$ with

$$\nabla^2 f(x_0) + \alpha I,$$

with α chosen large enough to make the matrix positive definite. (This is similar to the Levenberg-Marquardt algorithm for nonlinear least squares, discussed in Section 4.5.3.3 of Thisted, 1988.)

The function `schol()` below, which calls the FORTRAN subroutine `dschol`, gives a crude implementation of this idea. The goal is to make sure the smallest diagonal element in the factorization is roughly at least as large as a fraction `ratm` of the largest. Let $a_m = \max_{ij}\{\nabla^2 f(x_0)\}_{ij}$. $\sqrt{a_m}$ is an upper bound on the largest element on the diagonal of the factorization. First, $\nabla^2 f(x_0)$ is checked to make sure that no off diagonal elements are larger than a_m, and that the smallest diagonal element is larger than $a_m(\texttt{ratm})^2$. If either is not true, $\nabla^2 f(x_0)$ is replaced by $H = \nabla^2 f(x_0) + cI$, with c large enough so that both conditions are satisfied. Then the Choleski factorization of H is attempted. If any diagonal element in the factorization (prior to taking the $\sqrt{\ }$) is less than $(a_m + c)(\texttt{ratm})^2$, then H is replaced by $H + (a_m + c)(\texttt{frac})I$, where `frac` is a small positive value that can be specified by the user. If the factorization still fails, the process repeated, doubling the magnitude of the value added to the diagonal each time the factorization fails. The purpose is to get a safely positive definite matrix, while perturbing the Hessian as little as possible. There is an efficiency tradeoff, since adding a value that is too small will require many attempts at the Choleski factorization before the matrix becomes positive definite, while adding a value that is too large will make the Newton-Raphson updates behave like steepest descent updates, and many iterations of the Newton-Raphson algorithm might be required to get close enough to the solution for $\nabla^2 f(x_0)$ to become positive definite. This strategy is also likely to be more successful if the variables have been defined on similar scales. The values used as defaults for `ratm` and `frac` are arbitrary. The performance is much more sensitive to `frac` than to `ratm`.

The code for the function `schol()` follows. If the `rhs` argument is supplied, then the solution to the system of equations with coefficient matrix `a`, as modified by the algorithm above, and right hand side given by the `rhs` vector, is computed. `rhs` can be either a vector or a matrix of multiple right hand side vectors.

```
schol <- function(a,rhs=NULL,frac=.005,ratm=1.e-4) {
```

```
## computes the upper triangle Choleski factor of a, or if a is not pd
##  by a 'safe' margin, computes the Choleski factor of a+cI for a
##  scalar c chosen to make this matrix pd, and optionally computes the
##  solution to (a+cI) %*% x = rhs
## a=symmetric matrix -- only the upper triangle of a is needed
## rhs: if given, the system of equations a %*% x = rhs will be solved
##  for x.  rhs should be a vector of length n or a matrix with n rows
## frac: the minimum value added to the diagonal of a when a does not
##  meet the ratm criterion is roughly  2*frac*max(abs(a)).
##  This is repeatedly doubled until the matrix is pd
## ratm: the minimum value allowed for any diagonal element of the
##  factorization of a is roughly ratm*sqrt(max(abs(a))
## output: a list, with components
##  $chol = Choleski factor of a + dinc I
##  $sol  = solution to (a+dinc I ) %*% x = b
##  $dinc = amount added to diagonal of a
##  $info: if 0, factorization successfully completed, if > 0 factorization
##    failed, if =-2, a was replaced by an identity matrix.
  da <- dim(a)
  if (da[1] != da[2]) stop()
  if (!is.null(rhs)) {
    rhs <- as.matrix(rhs)
    nrhs <- ncol(rhs)
  } else {
    rhs <- nrhs <- 0
  }
  u <- .C('dschol_',as.integer(da[1]), as.double(a),
          as.double(ratm), as.double(frac), a=as.double(a),
          rhs=as.double(rhs),as.integer(nrhs), dinc=double(1),
          info=integer(1))[c(5,6,8,9)]
  if (nrhs>0) list(chol=matrix(u$a,nrow=da[1]),
                   sol=matrix(u$rhs,ncol=nrhs),dinc=u$dinc,info=u$info)
  else list(chol=matrix(u$a,nrow=da[1]),
                   sol=NULL,dinc=u$dinc,info=u$info)
}
```

Following is the source code for the FORTRAN subroutine `dschol`.

```
      subroutine dschol( n, a, ratm, frac, wa, rhs, nrhs, dinc, info)
      integer           info, n, nrhs
      double precision  a(n,*), ratm, dinc, frac, wa(n,*), rhs(n,*)
*
*  n       (input) INTEGER The order of the matrix A.
*
*  A       (input/output) DOUBLE PRECISION array, dimension (N,N)
*          On entry, the symmetric matrix A.  On output, the diagonal
```

```
*          elements of A may have been increased to make A pd.  Only the
*          upper triangle is needed on input.
*
*  ratm   min ratio allowed of smallest to largest diag elements of
*          U (the Choleski factor)
*  frac   fraction of largest element to add to diag when not pd
*  wa     (output) if INFO = 0, wa is the factor U  from the Choleski
*          factorization of the output matrix A, A = U'U (the lower
*          triangle is completed with 0's)
*  dinc   (output), the amount added to the diag of A
*  INFO   (output) INTEGER
*         <= 0: wa=cholsky factor of output A
*          -2 : all diagonal elements of A were <=0, so A was replaced by
*               an identity matrix
*          > 0: factorization could not be performed on the final A matrix.
*
*  =====================================================================
*
      double precision   one, zero
      parameter          ( one = 1.0d+0, zero = 0.0d+0 )
      integer            i, j
      double precision   dma,dm2,dm3,dmi,doff
      double precision   ddot
      external           ddot
      intrinsic          max, min, sqrt
      dinc=zero
*     Quick return if possible
      if( n.le.0 ) return
      if (n.eq.1) then
         if (a(1,1).gt.zero) then
            wa(1,1)=sqrt(a(1,1))
            go to 75
         else
            dinc=1-a(1,1)
            wa(1,1)=1
            a(1,1)=1
            info=-2
            go to 75
         endif
      endif
*     determine max and min diag and max abs off-diag elements
      dma=a(1,1)
      dmi=a(1,1)
      doff=zero
      do 5 i=1,n
         dma=max(dma,a(i,i))
```

```
         dmi=min(dmi,a(i,i))
         do 6 j=1,i-1
            doff=max(doff,abs(a(j,i)))
 6       continue
 5    continue
      if (dma.le.zero) then
         info=-2
         do 7 i=1,n
            do 8 j=1,n
               wa(j,i)=zero
               a(j,i)=zero
 8          continue
            wa(i,i)=one
            a(i,i)=one
 7       continue
         go to 75
      endif
c make sure dma > doff, and dmi >= dma*ratm*ratm
      dinc=max(dma*ratm**2-dmi,doff-dma)/(1-ratm**2)
      if (dinc.gt.zero) then
         do 9 i=1,n
            a(i,i)=a(i,i)+dinc
 9       continue
         dma=dma+dinc
      else
         dinc=zero
      endif
c dm3=base amount to add to diagonal if not pd
      dm3=dma*frac
c in ochol, diagonal elements of factorization required to be > sqrt(dm2)
c assuming largest diag element is approx sqrt(dma), and smallest
c should be > largest*ratm, need dm2=dma*ratm*ratm
 988  dm2=dma*ratm*ratm
c since # rows = n ...
      do 35 i=1,n*n
         wa(i,1)=a(i,1)
 35   continue
      call ochol(wa,n,dm2,info)
      if (info.gt.0) then
c not pd -- double dm3 and add it to diagonal of A
c adjust dma and dinc accordingly
         dm3=dm3*2
         dma=dma+dm3
         dinc=dinc+dm3
         do 50 i=1,n
            a(i,i)=a(i,i)+dm3
```

```
 50      continue
         go to 988
      endif
 75   if (nrhs.gt.0) then
c solve system of equations
         do 80 j=1,nrhs
c forward substitution to solve U'y=b
            do 82 i=1,n
               rhs(i,j)=(rhs(i,j)-
     $              ddot(i-1,wa(1,i),1,rhs(1,j),1))/wa(i,i)
 82         continue
c backward substitution to solve Ux=y
            do 84 i=n,1,-1
               rhs(i,j)=(rhs(i,j)-
     $              ddot(n-i,wa(i,i+1),n,rhs(i+1,j),1))/wa(i,i)
 84         continue
 80      continue
      endif
      return
      end

c ordinary Choleski decomposition -- return info=j if jth diagonal element
c of factorization (prior to sqrt) is < damin
      subroutine ochol(a,n,damin,info)
      double precision a(n,n),ajj,ddot,damin
      integer n,info,j,i
      info=0
      do 10 j = 1, n
*  Update jth column
         do 15 i=1,j-1
            a(i,j)=(a(i,j)-ddot(i-1,a(1,i),1,a(1,j),1))/a(i,i)
            a(j,i)=0
 15      continue
*  Compute U(J,J) and test for non-positive-definiteness.
         ajj = a(j,j)-ddot(j-1,a(1,j),1,a(1,j),1)
         if(ajj.le.damin) then
            info=j
            a(j,j)=ajj
            return
         endif
         a(j,j)=sqrt(ajj)
 10   continue
      return
      end
```

The function `ddot` used in the FORTRAN code is part of the Basic Linear Algebra Subroutine

(BLAS) library, which is available at `http://www.netlib.org`. It is already included in the compiled code of the current versions of Splus and R, and so does not need to be added to use this function from within these programs. The first argument of `ddot` is the number of terms in the inner product, the 2nd and 4th are the first elements of the two vectors, and the 3rd and 5th are the increments in the indices between elements of the vectors. Note that in the backward substitution algorithm, with an increment of `n`, the elements used in the inner product are from the `i`th row of `wa`.

Here is an example.

```
> a
     [,1] [,2] [,3]   [,4]
[1,] -1.0  1.0  0.5   -1.0
[2,]  1.0  2.0  0.3    0.4
[3,]  0.5  0.3 -1.0    3.0
[4,] -1.0  0.4  3.0  100.0
> b <- schol(a)
> b
$chol:
         [,1]      [,2]       [,3]       [,4]
[1,] 1.004988 0.9950367  0.4975183 -0.9950367
[2,] 0.000000 1.7377868 -0.1122399  0.7999244
[3,] 0.000000 0.0000000  0.8659554  4.1397427
[4,] 0.000000 0.0000000  0.0000000  9.1237358

$sol:
NULL

$dinc:
[1] 2.010001

$info:
[1] 0

> t(b$chol) %*% b$chol
          [,1]     [,2]     [,3]    [,4]
[1,]  1.010001 1.000000 0.500000   -1.00
[2,]  1.000000 4.010001 0.300000    0.40
[3,]  0.500000 0.300000 1.010001    3.00
[4,] -1.000000 0.400000 3.000000  102.01
> eigen(a)$value
[1]  100.1002448    2.3568880   -0.8000132   -1.6571196
> eigen(t(b$chol) %*% b$chol)$value
[1] 102.1102458   4.3668890   1.2099878   0.3528814
> schol(a,1:ncol(a))$sol
            [,1]
[1,] -1.64092900
```

```
[2,]  0.62749724
[3,]  3.87320301
[4,] -0.09324122
> solve(t(b$chol) %*% b$chol,1:ncol(a))
[1] -1.64092900  0.62749724  3.87320301 -0.09324122
```

Since a matrix is positive definite if and only if all of its eigenvalues are positive, it can be seen that the amount 2.01 added to the diagonal is slightly larger than the minimal value needed of about 1.66, but not very much larger.

3.2.5 The Function nr()

The function `nr()` below gives a full modified Newton-Raphson algorithm using the modified Choleski function `schol()` given above, in conjunction with proportional step reduction for backtracking.

The convergence criterion stops the iteration when $\max_i |\partial f(x)/\partial x_i| <$ `gtol`. Multiplying the objective function by arbitrary constants will change the magnitude of the gradient, without changing the location of the minimum, so the default value of `gtol` will be appropriate for all problems. The user needs to choose an appropriate value based on the scaling of the particular problem.

```
# Modified Newton's method for minimization
# Arguments
# b=initial parameter values
# fn=function to calculate function being minimized, called as fn(b,...)
#   must return the value of the function as a scalar value
# fhn=function to calc function, gradient, and hessian of function
#   being minimized, called as fhn(b,...)  fhn
#   must return a list with the function value as the first component,
#   the gradient as the second, and the hessian as the third.
# gtol= convergence criterion on gradient components (see below)
# iter=max # iterations (input),
# frac: the amount added to the diagonal of the hessian when it is not
#   pd is roughly 2*frac*max(abs(hessian))
# ratm: the minimum value allowed for any diagonal element of the
#   factored hessian is roughly ratm*sqrt(max(abs(hessian)))
# stepf=the fraction by which the step size is reduced in each step of
#   the backtracking algorithm
# ... additional arguments to fn and fhn
# returns a list with components 'b'=minimizing parameter values,
#   'value', 'score' and 'hessian' giving the value of these quantities
#   at b, and 'comp'=a vector with components named
#   'iter', giving the number of iterations used, an error
#   code ('error'=0 no errors, =1 if error in directional search, =2 if
#   max iterations exceeded), 'notpd' giving the number of iterations
```

```
#   where the hessian was not pd, and 'steph' giving the number of
#   times the step length was reduced
# Should invoke the Matrix library before using
nr <- function(b,fn,fhn,gtol=1e-6,iter=30,frac=.005,ratm=.0001,stepf=.5,...) {
  n <- length(b)
  error <- 0
  steph <- notpd <- 0
  for (ll in 1:iter) {
    z <- fhn(b,...)
    if (max(abs(z[[2]])) < gtol) return(list(b=b,
          value=z[[1]],score=z[[2]],hessian=z[[3]],
          comp=c(iter=ll,error=0,notpd=notpd,steph=steph)))
# if true, iteration converged
    h <- z[[3]]
    hc <- schol(h,z[[2]],frac,ratm)
    ut <- hc$info
    if (ut>0) stop('factorization failed')
    if (ut<0 | hc$dinc>0) notpd <- notpd+1
    sc <- -hc$sol
    bn <- b+sc
    fbn <- fn(bn,...)
# backtracking loop
    i <- 0
    while (is.na(fbn) || fbn>z[[1]]+(1e-4)*sum(sc*z[[2]])) {
      i <- i+1
      steph <- steph+1
      sc <- sc*stepf
      bn <- b+sc
      fbn <- fn(bn,...)
      if (i>20) return(list(b=b,value=z[[1]],score=z[[2]],
        comp=c(iter=ll,error=1,notpd=notpd,steph=steph)))
    }
    b <- c(bn)
  }
  z <- fhn(b,...)
# max number of iterations, but check for convergence again
  if (max(abs(z[[2]])) < gtol)
    list(b=b,value=z[[1]],score=z[[2]],hessian=z[[3]],
 comp=c(iter=iter,error=0,notpd=notpd,steph=steph)) else
    list(b=b,value=z[[1]],score=z[[2]],comp=c(iter=iter,error=2,
 notpd=notpd,steph=steph))
}
```

Another point to note is the use of the `is.na()` function in the backtracking loop. This is included so the algorithm can backtrack even if it took such a large step it resulted in serious errors such as overflow in the calculations.

3.2.6 Example: Weibull Regression

Consider the Weibull regression model with survivor function

$$S(t|z) = \exp(-[t/\exp(\beta_0 + \beta_1 z)]^{\exp(\alpha)}),$$

where z is a covariate. The log of the shape parameter (α) is used to avoid restrictions on the parameter space. If the observed data consist of independent observations (t_i, δ_i, z_i), $i = 1, \ldots, n$, where t_i is the failure/censoring time and δ_i is the failure indicator, then the negative log likelihood is

$$f(\theta) = -\sum_i \{\delta_i[\alpha + \log(v_i)] - v_i\},$$

where $v_i = -\log[S(t_i|z_i)]$ and $\theta = (\alpha, \beta_0, \beta_1)'$. Then $\partial v_i/\partial\alpha = v_i \log(v_i)$, $\partial v_i/\partial\beta_0 = -v_i \exp(\alpha)$, and $\partial v_i/\partial\beta_1 = z_i \partial v_i/\partial\beta_0$. From these facts it is easily established that

$$\begin{aligned}
\partial f(\theta)/\partial\alpha &= -\sum_i [\delta_i + (\delta_i - v_i)\log(v_i)] \\
\partial f(\theta)/\partial\beta_0 &= \exp(\alpha)\sum_i (\delta_i - v_i) \\
\partial f(\theta)/\partial\beta_1 &= \exp(\alpha)\sum_i z_i(\delta_i - v_i) \\
\partial^2 f(\theta)/\partial\alpha^2 &= -\sum_i [(\delta_i - v_i)\log(v_i) - v_i \log(v_i)^2] \\
\partial^2 f(\theta)/\partial\alpha\partial\beta_0 &= \exp(\alpha)\sum_i [\delta_i - v_i - v_i \log(v_i)] \\
\partial^2 f(\theta)/\partial\alpha\partial\beta_1 &= \exp(\alpha)\sum_i z_i[\delta_i - v_i - v_i \log(v_i)] \\
\partial^2 f(\theta)/\partial\beta_0^2 &= \exp(2\alpha)\sum_i v_i \\
\partial^2 f(\theta)/\partial\beta_0\partial\beta_1 &= \exp(2\alpha)\sum_i z_i v_i \\
\partial^2 f(\theta)/\partial\beta_1^2 &= \exp(2\alpha)\sum_i z_i^2 v_i.
\end{aligned}$$

Below are functions to calculate these quantities, and the call to the `nr()` function to fit the model. A copy of the data is in the file `weibreg.dat`. Following the initial fit, several additional calls are made to explore the sensitivity of the algorithm to the the value of `frac` and `stepf`.

```
> d <- read.table("../data/weibreg.dat",col.names=c("ti","fi","z"))
> fw <- function(b,time,fi,z) { # -log likelihood for Weibull
+   # regression with single covariate
+   # b[1]=log(shape param), b[2]=constant term, b[3]=reg coef
+   v <- (time/exp(b[2]+b[3]*z))^exp(b[1])
+   -sum(fi*(b[1]+log(v))-v)
+ }
> wh <- function(b,time,fi,z) { # - log likelihood, gradient, and hessian
+   t4 <- exp(b[1])
```

```
+   t6 <- t4*t4
+   v <- (time/exp(b[2]+b[3]*z))^t4
+   logv <- log(v)
+   f <- -sum(fi*(b[1]+logv)-v)
+   s <- -c(sum(fi+(fi-v)*logv),sum((v-fi))*t4,sum(z*(v-fi))*t4)
+   h <- matrix(0,3,3)
+   h[1,1] <- -sum((fi-v-v*logv)*logv)
+   h[2,1] <- h[1,2] <- -sum((-fi+v+v*logv))*t4
+   h[3,1] <- h[1,3] <- -sum(z*(-fi+v+v*logv))*t4
+   h[2,2] <- sum(v)*t6
+   h[3,2] <- h[2,3] <- sum(z*v)*t6
+   h[3,3] <- sum(z^2*v)*t6
+   list(f,s,h)
+ }
> survreg(Surv(ti,fi)~z,d,dist = "extreme") # the easy way
Call:
survreg(formula = Surv(ti, fi) ~ z, data = d, dist = "extreme")

Coefficients:
 (Intercept)           z
   2.308685 0.02375108

Dispersion (scale) = 0.3575806
Degrees of Freedom: 200 Total; 197 Residual
-2*Log-Likelihood: 270.2396
> unix.time(u <- nr(c(0,0,0),fw,wh,time=d$ti,fi=d$fi,z=d$z))
[1] 0.24 0.00 0.00 0.00 0.00
> u
$b:
[1] 1.02839444 2.30868513 0.02375108

$value:
[1] 135.1198

$score:
[1]  1.969536e-13  4.160457e-13 -2.142325e-14

$hessian:
           [,1]        [,2]       [,3]
[1,]  260.66437   -84.04584   16.47120
[2,]  -84.04584  1329.53871   58.06808
[3,]   16.47120    58.06808 1327.63250

$comp:
 iter error notpd steph
    9     0     3     4
```

```
> exp(-1.02839444)
[1] 0.3575806
> unix.time(u <- nr(c(0,0,0),fw,wh,frac=.0005,time=d$ti,fi=d$fi,z=d$z))
[1] 0.22 0.00 1.00 0.00 0.00
> u$comp
 iter error notpd steph
    8     0     3     3
> unix.time(u <- nr(c(0,0,0),fw,wh,frac=.05,time=d$ti,fi=d$fi,z=d$z))
[1] 0.3199999 0.0000000 1.0000000 0.0000000 0.0000000
> u$comp
 iter error notpd steph
   11     0     4     7
> unix.time(u <- nr(c(0,0,0),fw,wh,frac=.00005,time=d$ti,fi=d$fi,z=d$z))
[1] 0.21 0.00 0.00 0.00 0.00
> u$comp
 iter error notpd steph
    7     0     2     4
> unix.time(u <- nr(c(0,0,0),fw,wh,frac=.000005,time=d$ti,fi=d$fi,z=d$z))
[1] 0.23 0.00 0.00 0.00 0.00
> u$comp
 iter error notpd steph
    7     0     2     7
>
```

The results from `nr()` agree with the `survreg()` function. In the initial call, 9 iterations are needed. The Hessian fails to be positive definite in 3 of the 9 iterations. The step size is reduced 4 times (but some of those could have been within the same iteration). As `frac` is varied, the performance varies. The optimum value for this problem appears to be about .00005, but the performance is reasonable over a considerable range of values.

Maximizing the Weibull likelihood is a deceptively difficult problem where the Newton-Raphson iteration is very likely to fail without modification. The modified Newton-Raphson above required only 9 iterations to successfully fit the model with the default settings, and so performed quite well here.

An alternate approach to the problem of overshooting the solution in the Newton-Raphson iteration is to maintain a *model trust region*. The steps in the Newton-Raphson iteration can be thought of as based on a local quadratic approximation to $f(x)$. The model trust region is a region about the current point where the quadratic approximation should give a reasonable approximation. In this approach the full Newton-Raphson step is again computed first (using a positive definite modification of the Hessian if necessary), and if the new point falls within the current model trust region then it is accepted and the algorithm proceeds, but if the new point falls outside the region then an approximation to the minimum of the quadratic approximation over the model trust region is used as the next update. In either case, the algorithm then checks whether the new point gives an improved function value, and if not, the size of the trust region is reduced. There are also options for expanding the trust region under certain conditions. The

details are fairly involved, but can be found, for example, in Section 6.4 of Dennis and Schnabel (1983).

There is a built-in Splus function `nlminb()`, which implements both a modified Newton-Raphson algorithm and a BFGS algorithm (described in the next section), using a model trust region approach. If the Hessian matrix is provided, the Newton-Raphson iteration is used. Below `nlminb()` is applied to the Weibull regression example. The format of the functions needed is slightly different than for the `nr()` function above. `fw()` is the same as before, but `wgh()` computes just the gradient and Hessian, and to minimize storage, only the lower triangle of the Hessian is needed, but with its components arranged in a vector in a particular order (row by row). See the help file for additional details. By default `nlminb()` stops whenever either the relative change in the values of x are small (`<.Machine$double.eps^(1/2)`)or the relative change in the function value is small (`<1e-10`). It is difficult to compare this directly with the criteria used in `nr()`, but it appears it should give substantial precision in the results. (R does not include the `nlminb()` function, but has a related function `nlm()`.)

```
> wgh <- function(b,time,fi,z) { #gradient and hessian
+ #  assign('ng',ng+1,frame=0)
+   t4 <- exp(b[1])
+   t6 <- t4*t4
+   v <- (time/exp(b[2]+b[3]*z))^t4
+   logv <- log(v)
+   s <- -c(sum(fi+(fi-v)*logv),sum((v-fi))*t4,sum(z*(v-fi))*t4)
+   h <- c(-sum((fi-v-v*logv)*logv),-sum((-fi+v+v*logv))*t4,
+     sum(v)*t6,-sum(z*(-fi+v+v*logv))*t4,sum(z*v)*t6,sum(z^2*v)*t6)
+   list(gradient=s,hessian=h)
+ }
> unix.time(u <- nlminb(c(0,0,0),fw,gradient=wgh,hessian=T,
+                       time=d$ti,fi=d$fi,z=d$z)[1:8])
[1] 0.33999991 0.00999999 0.00000000 0.00000000 0.00000000
> u
$parameters:
[1] 1.02839444 2.30868513 0.02375108

$objective:
[1] 135.1198

$message:
[1] "RELATIVE FUNCTION CONVERGENCE"

$grad.norm:
[1] 8.633842e-09

$iterations:
[1] 6

$f.evals:
```

```
[1] 9

$g.evals:
[1] 7

$hessian:
           [,1]        [,2]       [,3]
[1,]  260.66437   -84.04584   16.47120
[2,]  -84.04584  1329.53871   58.06808
[3,]   16.47120    58.06808 1327.63250
```

The values `f.evals` and `g.evals` give the number of times the routines `fw()` and `wgh()` were called. These are slightly smaller than in `nr()` above, and the time needed is only a little longer than most of the `nr()` calls.

Analytic formulas were used for the second derivatives above. The Hessian can also be approximated with finite differences of the gradient, and this often gives reasonable performance (generally using analytic gradient calculations is substantially better than using finite differences for the gradient and second differences for the Hessian, though). Finite difference approximations to derivatives are discussed in the following subsection.

3.2.7 Computing Derivatives

While development of symbolic mathematics packages has made it easier to derive and program formulas for analytic derivatives, there can still be considerable effort involved, and there is always the possibility of errors in the resulting code. Correctly evaluated analytic derivatives can generally always give faster and more accurate results, but adequate performance can often be obtained using other methods.

One relatively recent development is automatic differentiation. An automatic differentiation program takes as input code for evaluating a function (generally either C or FORTRAN code), and automatically generates code for evaluating the derivatives. This is possible because the code for evaluating a function generally consists of a long sequence of binary operations together with calls to intrinsic functions (that have known derivatives), so the derivatives can be evaluated via recursive application of the chain rule. The resulting code is not necessarily numerically efficient, and there can be problems at points where intermediate calculations in the derivatives lead to singularities, and it is necessary to have the source code for any external routines that are called (so the derivatives of those routines can also be evaluated), but automatic differentiation has been used successfully in many applications. One source for more information and software is `http://www-unix.mcs.anl.gov/autodiff/index.html`.

The remainder of this section considers use of finite difference approximations to derivatives. These often give adequate, if somewhat slower, performance compared to use of analytic formulas. It is possible to attain the same rates of convergence using a finite difference approximation as when exact derivatives are used, but only if the step size in the finite difference $\to 0$ at an appropriate rate as the algorithm proceeds. In practice there are limits to how small the step size can be made in floating point arithmetic, which are related to the precision with which f can be

computed, so an arbitrarily small step size is not feasible. But often the modified Newton algorithm will converge to sufficient accuracy before the step size in the finite difference approximation needs to be reduced to a very small value.

For a function $g(x)$, the forward difference approximation to $\partial g(x)/\partial x_j$ is

$$[g(x + \epsilon e^{(j)}) - g(x)]/\epsilon$$

where $e^{(j)}$ is the unit vector in the jth coordinate direction, and the central difference approximation is

$$[g(x + \epsilon e^{(j)}) - g(x - \epsilon e^{(j)})]/(2\epsilon).$$

The central difference approximation can be more accurate, but requires roughly twice as many function evaluations to evaluate the gradient of g.

Since the derivatives are obtained in the limit as $\epsilon \to 0$, better approximations should result for small ϵ. However, due to errors in floating point computations, as ϵ decreases the number of accurate digits in $g(x + \epsilon e^{(j)}) - g(x)$ also decreases. For example, if the algorithm for computing $g(x)$ is accurate to 10 significant digits, and ϵ is small enough so that the true value of

$$|g(x + \epsilon e^{(j)}) - g(x)|/|g_i(x)| < 10^{-10},$$

then the computed difference $g(x + \epsilon e^{(j)}) - g(x)$ will have no accurate significant digits, so the forward difference approximation based on this ϵ would be useless.

A formal error bound can be given. To simplify notation, consider $g : R^1 \to R^1$. If exact calculations are done, by Taylor's theorem,

$$[g(x + \epsilon) - g(x)]/\epsilon = g'(x) + g''(x^*)\epsilon/2,$$

for some x^* between x and $x + \epsilon$, so the absolute error in a forward difference approximation is $|g''(x^*)|\epsilon/2$. However, as mentioned above, since the calculations cannot be done exactly, the error in the computed forward difference approximation is larger. If δ is a bound in the relative error in computing $g(x)$, then the total error can be approximately bounded by

$$|[g(x + \epsilon) - g(x)]/\epsilon - g'(x)| \leq |g''(x^*)|\epsilon/2 + 2\delta|g(x)|/\epsilon.$$

This expression is minimized by

$$\epsilon = 2\sqrt{\delta|g(x)|/|g''(x^*)|}.$$

In the absence of other information, it could be assumed that δ is close to relative machine precision ϵ_m, and that $2\sqrt{|g(x)|/|g''(x^*)|}$ is approximately 1 (in an order of magnitude sense), giving the rule of thumb

$$\epsilon = \sqrt{\epsilon_m}.$$

If the error in computing g is substantially larger than ϵ_m, then a larger value of ϵ should be used. This analysis also ignored errors in computing $x + \epsilon$, which could be substantial if x is large, and differences in the scale of different components of x (in the vector case). For these reasons, the value $\epsilon = |x|\sqrt{\epsilon_m}$ is also often recommended ($|x_j|\sqrt{\epsilon_m}$ for approximating the jth partial derivative of a function of several variables).

Below is a function `fdjac()` that will compute a forward difference approximation to a Jacobian matrix. The argument `G` is a possibly vector valued function.

```
fdjac <- function(PARAM,G,...,eps=sqrt(.Machine$double.neg.eps)) {
# Computes a finite difference approximation to the Jacobian of G
# G is a (possibly vector valued) function
# PARAM is the point where the Jacobian is approximated
# ... = additional arguments to G
# eps = step size in finite difference approximation
#   (scaled by max(abs(PARAM[j]),1))
  N <- length(PARAM)
  G0 <- G(PARAM,...)
  JAC <- matrix(0,length(G0),N)
  for (j in 1:N) {
    X1 <- PARAM
    X1[j] <- PARAM[j] + eps*max(abs(PARAM[j]),1)
    JAC[,j] <- (G(X1,...)-G0)/(X1[j]-PARAM[j]) #divide by actual difference
  }
  JAC
}
```

Since there is roundoff error in computing `X1[j]`, it is generally true that `X1[j]-PARAM[j]` will not be exactly equal to `eps*max(abs(PARAM[j]),1)`. It is better in this case to divide by the exact difference used, rather than the intended difference.

To illustrate, again consider the Weibull regression example. The functions `fw()` and `wh()` are the same as before. The function `wg()` is similar to `wh()`, but it only computes the gradient. Note that this provides a useful way to check functions that compute analytic derivatives.

```
> wg <- function(b,time,fi,z) { # just the gradient
+   t4 <- exp(b[1])
+   v <- (time/exp(b[2]+b[3]*z))^t4
+   logv <- log(v)
+   -c(sum(fi+(fi-v)*logv),sum((v-fi))*t4,sum(z*(v-fi))*t4)
+ }
> rbind(fdjac(c(.1,1.3,-.5),fw,time=d$ti,fi=d$fi,z=d$z),
+       wg(c(.1,1.3,-.5),time=d$ti,fi=d$fi,z=d$z))
         [,1]      [,2]      [,3]
[1,] 600.0353 -522.8105 -478.8034
[2,] 600.0353 -522.8105 -478.8034
> rbind(fdjac(c(.1,1.3,-.5),wg,time=d$ti,fi=d$fi,z=d$z),
+       wh(c(.1,1.3,-.5),time=d$ti,fi=d$fi,z=d$z)[[3]])
          [,1]       [,2]       [,3]
[1,]  2347.886 -1526.9009 -1592.4742
[2,] -1526.901   785.4334   538.2282
[3,] -1592.474   538.2282  1237.8493
[4,]  2347.886 -1526.9009 -1592.4742
[5,] -1526.901   785.4334   538.2282
[6,] -1592.474   538.2282  1237.8493
```

Below the Newton-Raphson algorithm is applied to the Weibull regression problem using a finite difference approximation to the Hessian matrix. This is slower, but still gives reasonable performance.

```
> wh2 <- function(b,time,fi,z) { # - log likelihood, gradient, and hessian
+   t4 <- exp(b[1])
+   t6 <- t4*t4
+   v <- (time/exp(b[2]+b[3]*z))^t4
+   logv <- log(v)
+   f <- -sum(fi*(b[1]+logv)-v)
+   s <- wg(b,time,fi,z)
+   h <- fdjac(b,wg,time=time,fi=fi,z=z)
+   list(f,s,h)
+ }
> unix.time(u <- nr(c(0,0,0),fw,wh2,time=d$ti,fi=d$fi,z=d$z))
[1] 0.4900002 0.0000000 1.0000000 0.0000000 0.0000000
> u
$b:
[1] 1.02839444 2.30868513 0.02375108

$value:
[1] 135.1198

$score:
[1] 6.205036e-13 8.516518e-13 1.319548e-13

$hessian:
           [,1]        [,2]       [,3]
[1,]  260.66437   -84.04581   16.47122
[2,]  -84.04584  1329.53866   58.06806
[3,]   16.47120    58.06808 1327.63249

$comp:
 iter error notpd steph
    9     0     3     4
```

3.3 The BFGS Algorithm

The modified Newton-Raphson algorithms discussed above work well on many problems. Their chief drawback is the need to compute the Hessian matrix at each iteration, and the need to solve for $[\nabla^2 f(x_0)]^{-1}\nabla f(x_0)$ at each iteration. As discussed above (following (3.6)), $d = -A^{-1}\nabla f(x_0)$ is a descent direction for any pd matrix A. The desire to avoid computing the Hessian has led to development of algorithms that use matrices other than the inverse Hessian in generating search directions. These methods generally are referred to as quasi-Newton methods, in that they try to mimic Newton's method without directly calculating the Hessian matrix. The methods discussed below are also referred to as variable metric methods. They are discussed in Section 4.3.3.4 and

4.5.3.4 of Thisted (1988), and Section 10.7 of Press *et. al.* (1992).

One class of quasi-Newton algorithms are those based on a method proposed independently by Broyden, Fletcher, Goldfarb and Shanno in 1970, which is a variant of a method due to Davidon, Fletcher and Powell. These methods are based on the idea of starting with a positive definite matrix A_0, and at each iteration performing a low rank update of A_0 that preserves positive definiteness. Some variations update A and solve for $A^{-1}\nabla f(x_0)$, while others update A^{-1} directly and compute the search direction using multiplication.

The general BFGS algorithm is as follows: (a) given the current point x_0 and approximate Hessian A_0, compute the new search direction $-A_0^{-1}\nabla f(x_0)$, (b) find a better point x_1 in this direction, and (c) update A_0 to a new approximation A_1, repeating until convergence. The algebraic form of the BFGS update of A_0 is

$$A_1 = A_0 + \frac{1}{y's}yy' - \frac{1}{s'A_0s}A_0ss'A_0, \tag{3.7}$$

where $y = \nabla f(x_1) - \nabla f(x_0)$ and $s = x_1 - x_0$. The difference $A_1 - A_0$ is the sum of two rank 1 matrices, and is thus generally a rank 2 matrix. If A_0 is positive definite, and if $s'y > 0$, then (3.7) will guarantee that A_1 is also positive definite.

It is easily verified by direct calculation that A_1 in (3.7) satisfies

$$\nabla f(x_1) - \nabla f(x_0) = A_1(x_1 - x_0). \tag{3.8}$$

The Hessian $\nabla^2 f(x_1)$ also satisfies (3.8) to first order, and will satisfy it exactly in the limit when the algorithm converges.

If A_1 is updated directly as in (3.7), it is necessary to solve for $A_1^{-1}\nabla f(x_1)$ to get the next search direction, which is an $O(p^3)$ operation, and part of the purpose of not using the Hessian was to avoid such a calculation. However, it is possible to give a similar update for the Choleski factorization of A_0. The update does not lead to a triangular matrix, but the Choleski factorization of A_1 can be recovered from the updated factor using plane rotations, in only $O(p^2)$ operations, so both the update and solving for the new search direction can be done in $O(p^2)$ operations. Details are given in Dennis and Schnabel (1983). This is the preferred version of the BFGS algorithm.

It is also possible to show that an algebraically equivalent update to (3.7) is given by

$$A_1^{-1} = A_0^{-1} + \frac{1}{y's}[(s - A_0^{-1}y)s' + s(s - A_0^{-1}y)'] - \frac{y'(s - A_0^{-1}y)}{(y's)^2}ss',$$

and with a bit more algebra this can be expressed

$$A_1^{-1} = A_0^{-1} + \frac{1}{y's}ss' - \frac{1}{y'A_0^{-1}y}A_0^{-1}yy'A_0^{-1} + (y'A_0^{-1}y)uu', \tag{3.9}$$

where

$$u = \frac{1}{y's}s - \frac{1}{y'A_0^{-1}y}A_0^{-1}y,$$

which is the form given in Section 10.7 of Press *et. al.* (1992). Using either of these last two versions, given an initial positive definite approximation to the inverse Hessian, this approximation can be updated at each iteration, and the new search direction calculated by multiplying. This is easier to program than the direct update of the Choleski factorization of A_0, and is used in the function `dfpmin` of Press *et. al.* (1992), and in the function `bfgs()` given below. The update (3.9) may not be as stable as updating the Choleski factorization of A_0, though.

Because of the good convergence properties of the Newton-Raphson iteration once it gets close to the solution, it would be desirable for the BFGS updates A_1 to closely approximate $\nabla^2 f(x_1)$ as the algorithm converged. In fact, it can be shown that if the true function is a positive definite quadratic, then the BFGS updates above converge to the Hessian in p iterations (this should be believable from the relation (3.8)). Since smooth functions tend to be nearly quadratic in the neighborhood of a local minimum, the BFGS updates (3.7) tend to converge to the Hessian as the algorithm converges. However, accuracy of the A or A^{-1} matrix at convergence cannot be guaranteed, and if the Hessian is needed for other purposes, such as estimating variances of parameter estimates, it is recommended the Hessian be computed separately after convergence.

The two remaining problems are what to use for the initial positive definite matrix, and how to choose the new point x_1 once the search direction has been determined in each iteration. For the initial matrix, usually a diagonal matrix is used. The most critical aspect of the choice is that the matrix be scaled appropriately. If it differs by several orders of magnitude from the correct Hessian, it will tend to create difficulties in the search for a better point at each iteration, since the initial step based on this matrix will be scaled inappropriately. Also, it may take many iterations of the BFGS updates to correct the poor initial scaling. In the function `bfgs()` below, the initial A_0 is set to a diagonal matrix with $\max(|f(x_0)|, 1)$ on the diagonal. This supposes that the magnitude of f is similar to that of $\nabla^2 f$, which may not be true (but is more likely to be true if irrelevant constants are dropped from f). There is also an implicit assumption that the scaling of the variables is such that the magnitude of the curvature of f will be similar in different directions. More advanced implementations allowing the user to specify different scaling factors for different variables can be given. In statistical applications such as fitting regression models, it is usually appropriate to rescale covariates to have similar length or variance before fitting the model.

For the search at each iteration, the function `bfgs()` below simply starts by computing $x_1 = x_0 - A_0^{-1}\nabla f(x_0)$, and if that point does not satisfy (3.5), the algorithm backtracks towards x_0 until a point satisfying (3.5) is found. Again this is done using the crude proportional step reduction used in the `nr()` function. The function `dfpmin` of Press *et. al.* (1992) uses a similar strategy, but with a more sophisticated backtracking algorithm using polynomial interpolation. The virtue of this general approach is that as the algorithm approaches a solution and the A^{-1} matrix approaches the inverse Hessian, the behavior of the algorithm becomes nearly identical to the Newton-Raphson iteration, and converges rapidly. As with the modified Newton methods discussed above, experience has indicated that more precise line searches early in the algorithm are not worth the added computational expense.

One requirement for the update formula to give a positive definite matrix is that $s'y > 0$. The strategy described in the previous paragraph is not guaranteed to produce new points that satisfy this condition. This condition could be added to the convergence criterion for the backtracking

algorithm, and it can be shown that points satisfying both (3.5) and $s'y > 0$ generally exist. Checking $s'y > 0$ does involve some additional expense, though, and it turns out that $s'y$ is usually > 0 at points satisfying (3.5), so most implementations of the BFGS algorithm skip this step. Instead, before updating A_0 or A_0^{-1}, the programs check whether $s'y$ is large enough, and if it is not they skip the update and use the old matrix to compute the next search direction. This approach is used in the function `bfgs()` below, and in the Press *et. al.* (1992) routine `dfpmin`. However, earlier versions of `dfpmin` contained an error, in that the condition checked was

```
if (fac*fac > EPS*sumdg*sumxi) {
```

where `fac` $= s'y$, `sumdg` $= \|y\|_2^2$, and `sumxi` $= \|s\|_2^2$. This condition ignores the sign of $s'y$, and will compute the update if $s'y$ is either sufficiently negative or sufficiently positive. If $s'y$ is sufficiently negative the update can fail to be positive definite, and the algorithm will probably fail. This has been corrected in the current online versions (`http://www.nr.com`).

A fairly simple function implementing the update (3.9), together with the proportional reduction backtracking algorithm, is as follows (this function is based on `dfpmin` from Press *et. al.* (1992)). The arguments are similar to `nr()`, except that `gn` only computes the gradient, and returns it as a vector. The convergence criterion used is the same as for `nr()`.

```
# bfgs method for minimization
# b=initial parameter values (input),
# fn=function to calculate function being minimized, called as fn(b,...)
#   must return the value of the function as a scalar value
# gn=function to calc gradient, called as gn(b,...)
#   gn must return the gradient as a vector
# gtol convergence on gradient components (see below)
# iter=max # iterations (input),
# stepf=the fraction by which the step size is reduced in each step of
#   the backtracking algorithm
# h = initial approximation to the inverse hessian (see below for default)
# ... additional arguments to fn and gn
# returns a list with components b=minimizing parameter values,
#   value=minimum value of fn, and comp=a vector with components named
#   iter and error, giving the number of iterations used, an error
#   code (error=0 no errors, =1 if error in directional search, =2 if
#   max iterations exceeded), steph giving the number of times the step
#   length was reduced, and nskip=# iterations where h was not updated
bfgs <- function(b,fn,gn,gtol=1e-6,iter=50,stepf=.5,h,...) {
  n <- length(b)
  steph <- nskip <- 0
  eps <- .Machine$double.neg.eps
  f1 <- fn(b,...)
# initial h (h=approx to the inverse hessian)
  if (missing(h)) h <- diag(rep(1/max(abs(f1),1),n))
  g1 <- gn(b,...)
```

```
  sc <- -c(h%*%g1)
  for (ll in 1:iter) {
    bn <- b+sc
    f2 <- fn(bn,...)
    i <- 0
    while (is.na(f2) || f2>f1+(1e-4)*sum(sc*g1)) { #backtrack
      i <- i+1
      steph <- steph+1
      sc <- sc*stepf
      bn <- c(b+sc)
      f2 <- fn(bn,...)
      if (i>20) return(list(b=b,value=f1,score=g1,
        comp=c(iter=ll,error=1,nskip=nskip,steph=steph)))
    }
    g2 <- gn(bn,...)
    if (max(abs(g2)) < gtol)
          return(list(b=b,value=f2,score=g2,hessinv=h,
          comp=c(iter=ll,error=0,nskip=nskip,steph=steph)))
# if true, iteration converged
# if not, update inverse hessian
#
#    b <- bn-b
#    g1 <- g2-g1
#    hg1 <- c(h %*% g1)
#    t1 <- sum(g1*b)
#    t2 <- sum(g1*hg1)
#    if (t1 > 0 && t1*t1 > eps*sum(g1*g1)*sum(b*b)) {
#      g1 <- b/t1-hg1/t2
#      h <- h+outer(b/t1,b)-outer(hg1/t2,hg1)+outer(t2*g1,g1)
#    } else nskip <- nskip+1
#    sc <- -c(h%*%g2)
# if the FORTRAN routine is not available, replace the lines from here
#  to the next comment with those above
    sc <- .C('bfgsup_',as.integer(n),as.double(b),as.double(bn),
            as.double(g1),as.double(g2),as.double(h),as.double(eps),
            as.integer(nskip),double(n))[c(6,8:9)]
    h <- matrix(sc[[1]],n,n)
    nskip <- sc[[2]]
    sc <- sc[[3]]
# end of lines to replace
    g1 <- g2
    b <- bn
    f1 <- f2
  }
  return(list(b=b,value=f2,score=g2,hessinv=h,
     comp=c(iter=iter+1,error=2,nskip=nskip,steph=steph)))# too many iterations
```

```
}
```

Using a compiled routine for updating the approximation to the inverse hessian can substantially speed up the calculations. The FORTRAN subroutine `bfgsup` is as follows.

```
c inputs are n, b, bn, g1, g2, h, eps, and nskip, as defined in the
c S function bfgs().  outputs are the updated approximation to the
c inverse hessian h, the new search direction h %*% g2, and nskip
c (updated if the update of h is skipped)
c assumes availability of the standard BLAS ddot for computing inner
c products
      subroutine bfgsup(n,b,bn,g1,g2,h,eps,nskip,sc)
      double precision b(n),bn(n),g1(n),g2(n),h(n,n),eps,sc(n)
      integer n,nskip,i,j
      double precision t1,t2,t3,t4,ddot
      do 5 i=1,n
         g1(i)=g2(i)-g1(i)
 5    continue
      do 10 i=1,n
         bn(i)=bn(i)-b(i)
         b(i)=ddot(n,g1,1,h(1,i),1)
 10   continue
      t1=ddot(n,g1,1,bn,1)
      t2=ddot(n,g1,1,b,1)
      t3=ddot(n,g1,1,g1,1)
      t4=ddot(n,bn,1,bn,1)
      if (t1.lt.0.or.t1*t1.le.eps*t3*t4) then
         nskip=nskip+1
         go to 50
      endif
      t3=sqrt(t1)
      t2=sqrt(t2)
      t1=t2/t3
      do 15 i=1,n
         bn(i)=bn(i)/t3
         b(i)=b(i)/t2
         g1(i)=bn(i)*t1-b(i)
 15   continue
      do 20 i=1,n
         do 21 j=i,n
            h(j,i)=h(j,i)+bn(i)*bn(j)-b(i)*b(j)+g1(i)*g1(j)
            h(i,j)=h(j,i)
 21      continue
 20   continue
 50   do 30 i=1,n
         sc(i)=-ddot(n,g2,1,h(1,i),1)
```

```
 30   continue
      return
      end
```

Below `bfgs()` is applied to the Weibull regression example. `fw()` and `wg()` are the same as before.

```
> unix.time(u <- bfgs(c(0,0,0),fw,wg,time=d$ti,fi=d$fi,z=d$z))
[1] 0.25999999 0.01000001 0.00000000 0.00000000 0.00000000
> u
$b:
[1] 1.02839444 2.30868513 0.02375108

$value:
[1] 135.1198

$score:
[1] -1.915766e-08  9.339495e-08  1.010147e-07

$hessinv:
              [,1]          [,2]          [,3]
[1,]  3.886858e-03  2.506445e-04 -5.531859e-05
[2,]  2.506445e-04  7.582680e-04 -4.876691e-05
[3,] -5.531859e-05 -4.876691e-05  7.423467e-04

$comp:
 iter error nskip steph
   16     0     0     9

> solve(u$hessinv)
           [,1]        [,2]       [,3]
[1,]  263.02438   -86.04535   13.94762
[2,]  -86.04535  1352.53909   82.44025
[3,]   13.94762    82.44025 1353.53457
> wh(u$b,time=d$ti,fi=d$fi,z=d$z)[[3]]
           [,1]        [,2]       [,3]
[1,]  260.66437   -84.04584   16.47122
[2,]  -84.04584  1329.53870   58.06806
[3,]   16.47122    58.06806 1327.63249
```

This algorithm also converges quickly to the correct results. The number of gradient evaluations is `iter+1`, and the number of function evaluations is `iter+1+steph`. While more iterations and function evaluations were needed than in `nr()`, this algorithm is nearly as fast and as accurate as the Newton-Raphson algorithm on this problem. The approximation to the inverse Hessian is fairly close, but not necessarily adequate for statistical inference. Next a finite difference approximation to the gradient will be used instead of `wg()`.

```
> # finite difference gradient
> unix.time(u2 <- bfgs(c(0,0,0),fw,function(b,...) c(fdjac(b,fw,...)),
+   time=d$ti,fi=d$fi,z=d$z))
[1] 0.75999999 0.00999999 1.00000000 0.00000000 0.00000000
> u2
$b:
[1] 1.02839445 2.30868511 0.02375108

$value:
[1] 135.1198

$score:
[1] 1.311461e-05 0.000000e+00 1.348699e-05

$comp:
 iter error nskip steph
   18     1     1    48

> unix.time(u2 <- bfgs(c(0,0,0),fw,function(b,...) c(fdjac(b,fw,...)),
+   gtol=.0001,time=d$ti,fi=d$fi,z=d$z))
[1] 0.50999999 0.00999999 1.00000000 0.00000000 0.00000000
> u2
$b:
[1] 1.02839441 2.30868495 0.02375122

$value:
[1] 135.1198

$score:
[1]  7.868766e-06 -5.841850e-06  1.348699e-05

$hessinv:
              [,1]          [,2]          [,3]
[1,]  0.0038843275  2.394380e-04 -4.666820e-05
[2,]  0.0002394380  7.606390e-04 -4.195437e-05
[3,] -0.0000466682 -4.195437e-05  7.273147e-04

$comp:
 iter error nskip steph
   15     0     0     9

> rbind(fdjac(u2$b,fw,time=d$ti,fi=d$fi,z=d$z),
+       wg(u2$b,time=d$ti,fi=d$fi,z=d$z))
             [,1]          [,2]         [,3]
[1,] 5.245844e-06 -0.0002056331 0.0001888179
[2,] 8.976966e-06 -0.0002250105 0.0001803243
```

In the first call, the algorithm failed to find a better point in the search direction in the final iteration. With exact arithmetic, this should not be possible. However, as can be seen in the comparison of the output of `fdjac` and `wg` following the second run, near the solution there is a large relative error in the finite difference approximation to the gradient, and consequently the search directions may or may not be descent directions. Also, the approximation to the Hessian tends to break down with large relative errors in the gradient computations. For this reason, it is not possible to iterate to as accurate a solution as with an analytic gradient.

The model trust region approach discussed in the previous section can also be used in BFGS algorithms. The Splus function `nlminb()`, when not provided with a routine to calculate the Hessian, uses an algorithm of this type. The following code applies this algorithm for the Weibull regression problem. The functions `fw()` and `wg()` are the same as before. When the gradient is not supplied, a finite difference approximation is used (possibly more reliable than `fdjac()`).

```
> unix.time(u <- nlminb(c(0,0,0),fw,gradient=wg,time=d$ti,fi=d$fi,z=d$z)[1:7])
[1] 0.71000004 0.00999999 0.00000000 0.00000000 0.00000000
> u
$parameters:
[1] 1.02839444 2.30868513 0.02375108

$objective:
[1] 135.1198

$message:
[1] "BOTH X AND RELATIVE FUNCTION CONVERGENCE"

$grad.norm:
[1] 3.541069e-06

$iterations:
[1] 21

$f.evals:
[1] 37

$g.evals:
[1] 22

> unix.time(u <- nlminb(c(0,0,0),fw,time=d$ti,fi=d$fi,z=d$z)[1:7])
[1] 1.73000002 0.00999999 2.00000000 0.00000000 0.00000000
> u
$parameters:
[1] 1.02839110 2.30868471 0.02375124

$objective:
[1] 135.1198
```

```
$message:
[1] "RELATIVE FUNCTION CONVERGENCE"

$grad.norm:
[1] 0.0008823412

$iterations:
[1] 27

$f.evals:
[1] 42

$g.evals:
[1] 111
```

This also converged quickly, but required a few more gradient and function evaluations than `bfgs()`. However, it required substantially more cpu time. This may have to do with the particular way the algorithm is coded in this function, and inferences cannot really be made on the relative performance of the underlying algorithms.

Using the finite difference gradient is again slower and less accurate (although the accuracy is certainly adequate for most purposes). In this case, `g.evals` is the number of function evaluations used in calculating the finite difference approximations to the gradient, so the total number of function evaluations used is `g.evals+f.evals`=153.

3.3.1 Rank 1 Updates

The BFGS algorithm above uses a rank 2 update to the Hessian approximation at each iteration. That is, the matrix added to the Hessian or Hessian inverse approximation at each step has rank 2. It has also been known for many years that rank 1 updates could be used, but these were thought to be less stable numerically. As discussed by Lange (1999, Section 11.6), rank 1 updates are coming back into vogue.

Given a current positive definite approximation to the Hessian A_0, it can be shown that the unique symmetric rank 1 update A_1 satisfying the secant condition (3.8) is

$$A_1 = A_0 - [(y + A_0 s)'s]^{-1}(y + A_0 s)(y + A_0 s)',$$

where again $y = \nabla f(x_1) - \nabla f(x_0)$ and $s = x_1 - x_0$.

For A_1 as above, it is easily verified that

$$A_1^{-1} = A_0^{-1} - [(A_0^{-1}y + s)'y]^{-1}(A_0^{-1}y + s)(A_0^{-1}y + s)',$$

so that as with the BFGS algorithm, the inverse Hessian can be updated directly as well. These updates to initial guesses to the Hessian and inverse Hessian can be used in place of the rank 2 updates in the BFGS algorithm. It is very important to monitor that these rank 1 updates stay

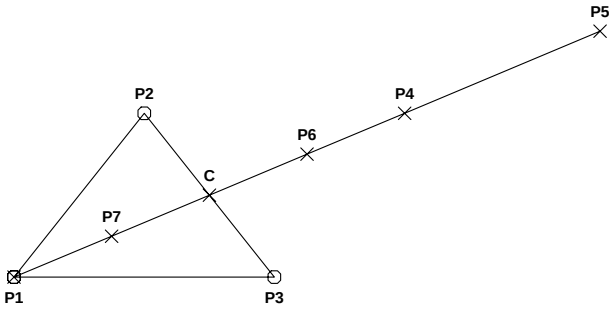

Figure 3.1: Possible steps in the simplex algorithm. $P1$, $P2$ and $P3$ are the initial simplex.

positive definite. Lange (1999, p. 137) shows that if A_0 is positive definite, and

$$1 - [(y + A_0 s)'s]^{-1}(y + A_0 s)A_0^{-1}(y + A_0 s) > 0,$$

then A_1 (and hence A_1^{-1}) is positive definite. He suggests carrying along the updates to both the approximate Hessian and its inverse to make checking this condition easier.

For quadratic objective functions, Lange (1999, Proposition 11.6.1) also gives sufficient conditions for the rank 1 update to the inverse Hessian to converge to the true inverse Hessian in p steps.

3.4 The Nelder-Mead Simplex Method

The Nelder-Mead simplex algorithm for unconstrained minimization is a pattern search algorithm that only uses information on the function values themselves. No information on derivatives is used, so no derivative calculations (analytical or numerical) are needed. (The simplex method discussed here should not be confused with the simplex method for linear programming, which is unrelated.)

To minimize a function f of p variables, the simplex method starts with $p + 1$ rather arbitrarily chosen points. The only requirement is that the span of the set of $p + 1$ points be all of R^p (the region bounded by this collection of points is called a simplex). Figure 3.1 illustrates this idea for $p = 2$. The points $P1$, $P2$, and $P3$ are the initial simplex. The function is evaluated at each of the $p + 1$ points. The point with the highest function value (say $P1$ if Figure 3.1) is then reflected through the center of mass C of the other points, and the function evaluated at the new point. This new point is $P4$ in Figure 3.1. There are then 4 cases.

1. If $f(P4)$ is somewhere between the best and worst of the values at the other points in the simplex, then the new point is retained (the simplex becomes $\{P2, P3, P4\}$), and the process is repeated.

2. If $P4$ is better than all of the other points in the simplex, then a longer step (twice as far from C) in the same direction is attempted. In Figure 3.1, this give $P5$. The better of $P4$ and $P5$ becomes the new point.

3. If the new point $P4$ is worse than the other p remaining points in the simplex, but better than $P1$, then a shorter step (half as far from C) in this direction is attempted. In

Figure 3.1, this gives $P6$. If $P6$ is still worse than the rest of the points in the simplex, then the entire simplex is shrunk towards the current best point by a factor of 1/2, and the process repeated.

4. If the new point $P4$ is worse than $P1$, then the point half way between C and $P1$ ($P7$ in Figure 3.1) is tried. If $P7$ is worse than the rest of the points in the simplex, then the entire simplex is again shrunk towards the current best point, and the process repeated.

This process continues until the relative difference between the highest and lowest function values in the current set of points is small (less than the argument `ftol` in the function `simplex()` below).

The simplex method is quite robust, and can successfully minimize functions that are not very smooth, where methods based on derivatives may fail. It can also be useful if there is some noise in the function evaluations, such as when evaluating the function requires Monte-Carlo integration, or when derivatives are difficult to compute. It is not particularly fast in most applications, though.

The main difficulty that can occur is that as the simplex stretches and contracts in different directions, it can collapse into a lower dimensional subspace, and will then only be able to find the minimum within that subspace. For this reason, it is often advised to restart the algorithm after it terminates, using a new full rank simplex, to make sure false convergence did not occur.

The following function is based on the C function `amoeba` of Press *et. al.* (1992).

```
simplex <- function(b,fun,del=1,ftol=1e-8,itmax=1000,...) {
# minimization using the Nelder-Mead simplex method, based on num-rec amoeba
# output gives the minimizing parameter value (b), the minimum value
# of fun, and comp giving the number of function evaluations and an
# error code (1=did not converge)
# b=initial values, fun=function to be minimized, called as fn(b,...)
# del=the step used from b in each coordinate direction to set up the
# initial simplex, ftol=convergence criterion--max relative difference
# between highest and lowest point in the simplex
# if ... is used to pass arguments to fun, then the arguments of fun must
# not use any of the names of the arguments to smtry (p,y,psum,fun,ihi,fac)
# setup
  iter <- 0
  np <- length(b)
  p <- rep(b,np)
  dim(p) <- c(np,np)
  p <- t(p)
  diag(p) <- diag(p)+del
  p <- rbind(b,p)
  y <- rep(0,np+1)
  for (i in 1:(np+1)) y[i] <- fun(p[i,],...)
#
```

```
  psum <- apply(p,2,sum)
  while (iter <= itmax) {
    o <- order(y) # don't need a full sort, only smallest and two largest
    p <- p[o,]    # so could be done more efficiently
    y <- y[o]
    ilo <- 1
    ihi <- np+1
    inhi <- np
    rtol <- 2*abs(y[ihi]-y[ilo])/(abs(y[ihi])+abs(y[ilo])+1e-8)
    if (rtol < ftol) return(list(b=p[ilo,],value=y[ilo],comp=
 c(iter=iter,error=0)))
    if (iter >= itmax) return(list(b=p[ilo,],value=y[ilo],comp=
                         c(iter=iter,error=1)))
    iter <- iter+2
# new point chosen by reflecting the worst current through the plane
# of the others
    z <- smptry(p,y,psum,fun,ihi,-1,...)
    if (z[[1]] <= y[ilo]) { # new point is best--try going further
      z <- smptry(z[[4]],z[[2]],z[[3]],fun,ihi,2,...)
      y <- z[[2]]; psum <- z[[3]]; p <- z[[4]]
    } else if (z[[1]] >= y[inhi]) {
      ysave <- z[[2]][ihi] #new point is still worst, try smaller step
      z <- smptry(z[[4]],z[[2]],z[[3]],fun,ihi,0.5,...)
      y <- z[[2]]; psum <- z[[3]]; p <- z[[4]]
      if (z[[1]] >= ysave) { # still bad, shrink simplex
for (i in (1:(np+1))[-ilo]) {
  psum <- (p[i,]+p[ilo,])/2
  p[i,] <- psum
  y[i] <- fun(psum,...)
}
iter <- iter+np
psum <- apply(p,2,sum)
      }
    } else {
      y <- z[[2]]; psum <- z[[3]]; p <- z[[4]]
      iter <- iter-1
    }
  }
  return(list(b=p[ilo,],value=y[ilo],comp=c(iter=iter,error=1)))
}

smptry <- function(p,y,psum,fun,ihi,fac,...) {
  ndim <- ncol(p)
  fac1 <- (1-fac)/ndim
  fac2 <- fac1-fac
  ptry <- psum*fac1-p[ihi,]*fac2
```

```
  ytry <- fun(ptry,...)
  if (ytry < y[ihi]) {
    y[ihi] <- ytry
    psum <- psum-p[ihi,]+ptry
    p[ihi,] <- ptry
  }
  list(ytry,y,psum,p)
}
```

Applying this to the Weibull regression example gives the following. `fw()` is the same as before.

```
> unix.time(u <- simplex(c(0,0,0),fw,time=d$ti,fi=d$fi,z=d$z))
[1] 1.139999 0.000000 2.000000 0.000000 0.000000
> u
$b:
[1] 1.02834543 2.30864089 0.02376035

$value:
[1] 135.1198

$comp:
 iter error
  132     0

> unix.time(u <- simplex(c(0,0,0),fw,ftol=1e-12,time=d$ti,fi=d$fi,z=d$z))
[1] 1.46 0.00 2.00 0.00 0.00
> u
$b:
[1] 1.02839421 2.30868513 0.02375152

$value:
[1] 135.1198

$comp:
 iter error
  176     0
```

Unlike modified Newton-Raphson, and methods which attempt to mimic it such as BFGS, the simplex method continues to converge slowly as it approaches the solution. For a few digits of accuracy the simplex method can perform nearly as well as other methods, but if high accuracy is needed it can be quite slow, and if the minimum is located in a region where the function is fairly flat, high accuracy may be impossible.

3.5 A Neural Network Classification Model

Consider the problem of classifying objects into K classes based on an observed p-dimensional feature vector x. Given a training set of m cases, where both the feature vectors $x_i = (x_{i1}, \ldots, x_{ip})'$ and the class indicators y_i, with $y_i = k$ if object i is in class k, are observed, the problem is to build a rule for determining the class of future objects based only on their feature vectors. For example, in diagnosing medical problems, the classes would be different possible diagnoses, and the feature vector would consist of the patient's symptoms and history, lab test results, etc. In the example to be analyzed below (a classic classification data set), the classes are 3 species of iris, and the feature vector consists of measurements on the 4 variables sepal length, sepal width, petal length, and petal width. The problem is to develop a rule for determining the species based on these measurements.

There are many methods for developing classification rules, including linear discriminant analysis, logistic regression (and polytomous extensions), nearest neighbor methods, recent extensions of linear discriminant analysis, such as flexible discriminant analysis and penalized discriminant analysis (see Hastie, Tibshirani and Buja, 1994), and many others. Given a model for the conditional distribution of x given the class, the optimal Bayes rule when all errors have equal costs is to classify cases with features x in class k if the posterior class probabilities satisfy $P(y = k|x) > P(y = j|x)$ for all $j \neq k$.

Neural networks have also been widely applied to classification problems. Neural networks have mostly been developed outside the field of statistics, and because of their connections with cognitive science, some very picturesque terminology has been developed to describe these models and the process of fitting them. Brief introductions to neural networks are given by Cheng and Titterington (1994), Warner and Misra (1996), and Venables and Ripley (1997, Sections 11.4 and 17.2), and a more extensive treatment in Ripley (1996). A fictional (ie false) account of neural network applications was given by Powers (1995).

The classic feed-forward neural network consists of a set of input nodes, a set of outputs, and one or more hidden layers of nodes. In the classification problem, there are p input nodes, each corresponding to one of the p features. Each of these is passed on to each of the hidden nodes in the next layer, where they are combined into new values using node specific weights, w_{jn} for node n; that is, the new value computed at node n from an input vector x_i is

$$z_{in} = w_{0n} + \sum_{j=1}^{p} w_{jn} x_{ij}.$$

These are then transformed by an activation function, which is almost always the logistic function $g(z) = 1/\{1 + \exp(-z)\}$, and then are sent on to each of the nodes in the next layer. The role of the activation function is to facilitate approximation of nonlinear effects, and to keep rescaling all the outputs to a common scale (the range of $g(z)$ is $(0, 1)$). In a similar vein, there is some advantage to scaling the input features to lie in this interval, too. At the next layer of nodes, the outputs from the previous layer are again comined using node specific weights, transformed, and sent on to the nodes in the next layer. The outputs from the final layer consist of one or more different weighted combinations of the outputs from the previous layer, which are then transformed by an output function. The most common output functions are the linear function (no transformation), the logistic function $g(z)$, and a threshold function, which equals 1 if its

argument is greater than some cutoff, and equals 0 otherwise.

It is known that a single layer of hidden nodes is sufficient to uniformly approximate continuous functions over compact sets, with the degree of approximation improving as the number of hidden nodes is increased. In the classification problem, if K output nodes are used to represent the probabilities of each of the K classes, a single hidden layer with N nodes is used, and the logistic function is applied to the outputs, then the resulting model for the class probabilities is

$$P(y_i = k|x_i) = p(k, x_i, v, w) = g\left\{v_{0k} + \sum_{n=1}^{N} v_{nk} g\left(w_{0n} + \sum_{j=1}^{p} w_{jn} x_{ij}\right)\right\},$$

where the weights w_{jn} for each hidden node and the output weights v_{nk} can be thought of as unknown parameters to be estimated from the training data set. There are thus $N(p+1) + K(N+1)$ unknown parameters. For a problem with $p = 4$ and $K = 3$ (as in the iris data), $N(p+1) + K(N+1) = 8N + 3$, so each hidden node adds 8 additional parameters. Neural networks thus provide very flexible models for classification problems. One additional extension sometimes used is to replace v_{0k} by $v_{0k} + x'\beta_k$ at the output layer, so the model includes the linear logistic model as a special case. Then the hidden layer models nonlinearities in the effects. Generally the probabilities in this model do not satisfy the constraint that $\sum_k P(y = k|x) = 1$, but they can be renormalized when this is important.

The process of fitting the model is often referred to as training the network, or more imaginatively, as the machine learning how to classify the data. Powers (1995, p. 30), writes of a neural network trained to produce verbal output from text input:

> No one told it how. No one helped it plough through the dough. The cell connections, like the gaps they emulated, taught themselves, with the aid of iterated reinforcement. Sounds that coincided with mother speech were praised. The bonds behind them tightened, closing in on remembrance like a grade-schooler approximating a square root. All other combinations died away in loneliness and neglect.

What this refers to in the traditional approach to training a neural network, is that the features from the training set are fed into the network one at a time. The outputs are then compared with the true classification, and information sent back through the network that allows the weights at each node to be updated in a way that should increase the probability of the correct class and decrease the probabilities of the other classes. This is usually done through a procedure called the back propagation algorithm.

The back propagation algorithm requires an objective function that can be used to compare the fit of different weights. Often the least squares function

$$\sum_i \sum_{k=1}^{K} \{I(y_i = k) - p(y_i, x_i, v, w)\}^2 \tag{3.10}$$

is used, although criteria based on likelihood functions can also be given. In the usual process of training the network, the weights are updated by computing an update to the gradient of the objective function, and moving the weights at each node a small step in the direction of the

negative of the gradient. The back propagation algorithm for training a neural network classifier is thus essentially a steepest descent algorithm for minimizing (3.10).

With the large number of parameters in a typical neural network model, overfitting the training data is a serious problem. That is, weights can be found that fit the training data very well, but are fitting random variation in the training data, and so do poorly at classifying new observations. Of an overtrained network, Powers (1995, p. 79) writes

> 'Autistic,' I remember saying. Particulars overwhelmed it. Its world consisted of this plus this plus this. Order would not striate out. Implementation A had sat paralyzed, a hoary, infantile widow in a house packed with undiscardable mementos, no more room to turn around. Overassociating, overextending, creating infinitesimal, worthless categories in which everything belonged always and only to itself.

Because of the problem of overfitting the training data, the standard paradigm for training a neural network splits the training data into two parts, and uses one part for training, and the second part (referred to as the 'test data') for estimating the error rate of the network at each step in the training. When the error rate on the test data stops decreasing, then the training is stopped. When very large data sets are not available (which is usually the case), the error rate can be estimated from cross validation, instead of using a separate test data set. In leave one out cross validation, the fitting process is conducted with one observation omitted, and the fitted model used to predict the omitted observation. This is repeated with each observation omitted in turn, and the error rate estimated by averaging over the omitted observations. (Instead of just omitting one observation, small randomly selected subsets can be omitted.) The main drawback to cross validation is the need to repeat the fitting process many times. Thus the traditional process of training the network does not attempt to minimize (3.10), but tries to find arbitrary v and w that give good predictions in test data set or under cross validation. The actual values of v and w will depend heavily on the starting values, although similar classification may result from quite different wets of weights. Even if the iteration is continued to a local optimum, neural network models tend to have many local optima, not all of which would give equivalent performance.

Another way to control the problem of overfitting is through penalty functions. In particular, an objective function of the form

$$F(v,w) = \sum_{i=1}^{m}\sum_{k=1}^{K}\{I(y_i=k) - p(y_i,x_i,v,w)\}^2 + \lambda\sum_{n=1}^{N}\left(\sum_{j=0}^{p} w_{jn}^2 + \sum_{k=1}^{K} v_{nk}^2\right) \tag{3.11}$$

could be used, where λ is a smoothing parameter, with larger values of λ forcing smaller weights, and hence smoother fits. Note that the constants in the output units (v_{0k}) have not been penalized. For (3.11), a search for local minima can be conducted using any of the techniques discussed in this chapter. The problem of multiple local minima still remains, and repeat runs from different starting values should always be done. Venables and Ripley (1997, p. 491) suggest averaging the probabilities over different local optima. In the penalized version, there is also the problem of choosing the penalty parameter λ. This could be based on minimizing classification errors in a test data set, or on cross validation.

Setting $r_{ik} = I(y_i = k) - p(y_i, x_i, v, w)$, $u_{in} = g(z_{in})$, $u_{i0} = 1$, and $x_{i0} = 1$, the components of the

gradient of (3.11) are easily seen to be

$$\frac{\partial F(v,w)}{\partial v_{hl}} = -2\sum_{i=1}^{m} r_{il} g'\left(v_{0l} + \sum_{n=1}^{N} v_{nl}u_{in}\right) u_{ih} + 2\lambda v_{hl} I(h > 0),$$

$h = 0, \ldots N$, $l = 1, \ldots, K$, and

$$\frac{\partial F(v,w)}{\partial w_{jh}} = -2\sum_{i=1}^{m}\sum_{k=1}^{K} r_{ik}\, g'\left(v_{0k} + \sum_{n=1}^{N} v_{nk}u_{in}\right) v_{hk}\, g'(z_{ih}) x_{ij} + 2\lambda w_{jh},$$

$j = 0, \ldots, p$ and $h = 1, \ldots, N$. Since there is no intrinsic interest in the v's and w's, and hence in their variances, the second derivative matrix is not of independent interest, and a BFGS algorithm seems like an appropriate choice for minimizing (3.11) for a fixed value of λ. The function `nnfit()` given below does this. (There is a function `nnet()` in the library distributed by Venables and Ripley, that is superior for practical work.) `nnfit()` uses a sample from a $U(0,1)$ distribution as the default initial values of the weights. The functions `nnf()` and `nng()` compute the objective function (3.11) and its gradient, the function `nnp()` computes the estimated classification probabilities, and `nn.classif()` computes the estimated probabilities and determines which class has the highest estimated probability. Classification based on the highest estimated class probability is motivated by the optimal Bayes rule. However, since a point estimate of the parameters is used, rather than an average over priors, these are not Bayes estimates, and this is not an optimal rule (although it should be asymptotically consistent for the optimal rule). As discussed in Hastie, Tibshirani and Buja (1994), a classification rule which takes the output from the neural network (usually without the final logistic transformation), and applies a post processor, such as linear discriminant analysis, to determine the final classification, can perform better.

```
nng <- function(b,y,x,lambda) {
# computes the gradient of the least squares classification neural
#  network objective function.
# b is the vector of weights (first the output unit weights, then
#  the weights for the hidden units)
# y[i]=k if the ith case is in class k
# x is the m x p matrix of feature vectors
# lambda=the penalty parameter
  x <- as.matrix(x)
  p <- ncol(x)
  k <- max(y)
  n <- (length(b)-k)/(p+k+1)
  gpen <- 2*lambda*b
  gpen[seq(1,(n+1)*k,by=n+1)] <- 0
  unlist(.C('nng_',as.integer(nrow(x)),as.integer(p),as.integer(n),
            as.integer(k),as.double(t(x)),as.integer(y),
            as.double(b[1:((n+1)*k)]),
            as.double(b[(((n+1)*k)+1):length(b)]),
            obj=double(1),gv=double(n*k+k),gw=double(n*p+n),
```

```
            double(n))[c(10,11)])+gpen
}
nnf <- function(b,y,x,lambda) {
# computes the least squares classification neural
#  network objective function.
# b is the vector of weights (first the output unit weights,
#  then the weights for the hidden units)
# y[i]=k if the ith case is in class k
# x is the m x p matrix of feature vectors
# lambda=the penalty parameter
  x <- as.matrix(x)
  p <- ncol(x)
  k <- max(y)
  n <- (length(b)-k)/(p+k+1)
  pen <- lambda*sum(b[-seq(1,(n+1)*k,by=n+1)]^2)
  .C('nnf_',as.integer(nrow(x)),as.integer(p),
     as.integer(n),as.integer(k),as.double(t(x)),
     as.integer(y),as.double(b[1:((n+1)*k)]),
     as.double(b[(((n+1)*k)+1):length(b)]),
     obj=double(1),double(n))$obj+pen
}
nnp <- function(nnobj,x) {
# computes the estimated class probabilities for a neural
#  network classification model fit
# nnobj=output from nnobj
# x=matrix of feature vectors where probabilities are to be calculated
# output = K x nrow(x) matrix whose ith column gives the estimated
#  probabilities that the feature vector in the ith row of x is in
#  each of the K classes
  x <- as.matrix(x)
  p <- nnobj$dims[1]
  if (ncol(x) != nnobj$dims[1]) stop('wrong # covs')
  for (i in 1:p) {
    x[,i] <- (x[,i]-nnobj$covtran[i,1])/
      (nnobj$covtran[i,2]-nnobj$covtran[i,1])
  }
  k <- nnobj$dims[2]
  n <- nnobj$dims[3]
  i1 <- (n+1)*k
  nwt <- length(nnobj$weights)
  probs <- .C('nnp_',as.integer(nrow(x)),as.integer(p),
              as.integer(n),as.integer(k),as.double(t(x)),
              as.double(nnobj$weights[1:i1]),
              as.double(nnobj$weights[(i1+1):nwt]),
              probs=double(k*nrow(x)),double(n))$probs
  dim(probs) <- c(k,nrow(x))
```

```
  probs
}
nn.classif <- function(nnobj,xx) {
# determine class with highest estimated class probability
#   (not necessarily the best classification rule)
# nnobj = output from nnfit
# xx = matrix of feature vectors (rows) for cases to be classified
# output is a list with components
#  $probs = the matrix of estimated class probabilities
#           (K x nrow(x) -- see nnp())
#  $predclass=the vector of class indicators (=k if predicted to be in class k)
  ap <- nnp(nnobj,xx)
  u <- apply(ap,2,function(u) (1:length(u))[u == max(u)][1])
  list(probs=ap,predclass=u)
}
nnfit <- function(y,x,lambda=0,N=5,maxiter=2000,init=NULL) {
# fits a neural network classification model by minimizing a
#  penalized least squares objective function
# y[i]=k if the ith case is in class k
# x is the m x p matrix of feature vectors
# lambda is the penalty parameter
# N= # nodes in the hidden layer
# maxiter=maximum iterations in the BFGS algorithm
# init=initial values of weights.  If not given, initial values
#  are generated from a U(0,1) distribution
# note that features are transformed to the interval [0,1]
# output = list with components $weight giving the estimated weights
#  (first from the output units, then from the hidden units), $dims
#  giving the values of the number of features p, the number of classes
#  K, the number of hidden units N, and the number of observations m,
#  $covtran giving the values needed for transforming the feature
#  vectors, and $perf giving the number of iterations and other info.
  x <- as.matrix(x)
  rr <- NULL
  for (i in 1:ncol(x)) {
    u <- range(x[,i])
    rr <- rbind(rr,u)
    x[,i] <- (x[,i]-u[1])/(u[2]-u[1])
  }
  p <- ncol(x)
  m <- nrow(x)
  K <- max(y)
  u <- table(y)
  if (length(u) != K | min(u)<2 | length(y) != m) stop('problems with y')
  nwt <- N*(p+K+1)+K
  if (is.null(init) || length(init) != nwt) init <- runif(nwt)
```

```
# use the following 4 lines in place of the last 3
#   for nlminb (sometimes faster than bfgs)
#  u <- nlminb(init,nnf,nng,control=nlminb.control(maxiter,maxiter),
#              y=y,x=x,lambda=lambda)
#  list(weights=u[[1]],dims=c(p=p,K=K,N=N,m=m),message=u[[3]],
#       covtran=rr,perf=unlist(u[c(2,4:7)]))
#
# use the following 3 lines for bfgs()
  u <- bfgs(init,nnf,nng,iter=maxiter,y=y,x=x,lambda=lambda)
  list(weights=u[[1]],dims=c(p=p,K=K,N=N,m=m),message=NULL,
       covtran=rr,perf=c(u$comp,score=sqrt(sum(u$score*u$score))))
}
```

Most of the work in `nnf()`, `nng()` and `nnp()` is done in the following FORTRAN routines, which substantially speeds up these computations.

```
c m  = (input) # obs
c np = (input) # covs (not including constant)
c n  = (input) # nodes in hidden layer
c k  = (input) # classes
c x  = (input) covariate matrix, covariates in rows, cases in columns
c iy = (input) iy(i)=kk if ith case in class kk
c v  = (input) v weights
c w  = (input) w weights
c obj= (output) objective function value at input weights
c gv = (output) gradient of f wrt v
c gw = (output) gradient of f wrt w
c z  = working vector of length n
c assumes standard BLAS ddot is available (computes inner product)
c note: if g(u)=1/(1+exp(-u)), then g'(u)=g(u)(1-g(u))
      subroutine nng(m,np,n,k,x,iy,v,w,obj,gv,gw,z)
      double precision x(np,m),v(0:n,k),w(0:np,n),gv(0:n,k),gw(0:np,n)
      double precision obj,z(n)
      integer m,np,n,k,iy(m),i,j,l,j2
      double precision ddot,rl,ql,t1
      obj=0
      do 10 i=1,n
         do 11 j=0,k
            gv(i,j)=0
 11      continue
         do 12 j=0,np
            gw(j,i)=0
 12      continue
 10   continue
      do 20 i=1,m
         do 21 j=1,n
```

```
            z(j)=w(0,j)+ddot(np,w(1,j),1,x(1,i),1)
            z(j)=1/(1+exp(-z(j)))
 21      continue
         do 30 l=1,k
            ql=v(0,l)+ddot(n,v(1,l),1,z,1)
            ql=1/(1+exp(-ql))
            if (iy(i).eq.l) then
               rl=1-ql
            else
               rl=-ql
            endif
            obj=obj+rl*rl
            ql=ql*(1-ql)
            ql=-2*ql*rl
            gv(0,l)=gv(0,l)+ql
            do 40 j=1,n
               gv(j,l)=gv(j,l)+ql*z(j)
               t1=ql*v(j,l)*z(j)*(1-z(j))
               gw(0,j)=gw(0,j)+t1
               do 42 j2=1,np
                  gw(j2,j)=gw(j2,j)+t1*x(j2,i)
 42            continue
 40         continue
 30      continue
 20   continue
      return
      end

c like above, but only evaluates obj fcn
      subroutine nnf(m,np,n,k,x,iy,v,w,obj,z)
      double precision x(np,m),v(0:n,k),w(0:np,n)
      double precision obj,z(n)
      integer m,np,n,k,iy(m),i,j,l
      double precision ddot,rl,ql
      obj=0
      do 20 i=1,m
         do 21 j=1,n
            z(j)=w(0,j)+ddot(np,w(1,j),1,x(1,i),1)
            z(j)=1/(1+exp(-z(j)))
 21      continue
         do 30 l=1,k
            ql=v(0,l)+ddot(n,v(1,l),1,z,1)
            ql=1/(1+exp(-ql))
            if (iy(i).eq.l) then
               rl=1-ql
            else
```

```
               rl=-ql
            endif
            obj=obj+rl*rl
 30      continue
 20   continue
      return
      end

c calculates estimated probabilities at arbitrary covariates x
      subroutine nnp(m,np,n,k,x,v,w,prob,z)
      double precision x(np,m),v(0:n,k),w(0:np,n)
      double precision z(n),prob(k,m)
      integer m,np,n,k,i,j,l
      double precision ddot,ql
      do 20 i=1,m
         do 21 j=1,n
            z(j)=w(0,j)+ddot(np,w(1,j),1,x(1,i),1)
            z(j)=1/(1+exp(-z(j)))
 21      continue
         do 30 l=1,k
            ql=v(0,l)+ddot(n,v(1,l),1,z,1)
            prob(l,i)=1/(1+exp(-ql))
 30      continue
 20   continue
      return
      end
```

3.5.1 Application to Fisher's Iris Data

The iris data set (available in the `s/.Datasets` subdirectory in the Splus search path) consists of 50 observations on the sepal length, sepal width, petal length, and petal width from each of 3 species of iris (Setosa, Versicolor, Virginica). Here the first 33 observations from each species will be used as the training set (`xx1` and `yy1` below) and the other 17 observations as a test set (`xx2` and `yy2` below). The following commands set up the data and fit the neural network model with 5 nodes in the hidden layer (and hence 43 unknown parameters).

```
> dim(iris)
[1] 50  4  3
> xx <- rbind(iris[,,1],iris[,,2],iris[,,3])
> xx1 <- rbind(iris[1:33,,1],iris[1:33,,2],iris[1:33,,3])
> xx2 <- rbind(iris[34:50,,1],iris[34:50,,2],iris[34:50,,3])
> dim(xx1)
[1] 99  4
> yy <- c(rep(1,50),rep(2,50),rep(3,50))
> yy1 <- c(rep(1,33),rep(2,33),rep(3,33))
> yy2 <- c(rep(1,17),rep(2,17),rep(3,17))
```

```
> a <- nnfit(yy1,xx1,lambda=.1)
> a[[5]]
 iter error nskip steph        score
  303     0     9     2 3.124937e-05
> a[[2]]
 p K N  m
 4 3 5 99
```

The BFGS algorithm needed 303 iterations to meet the convergence criterion. Generally in these examples, fewer iterations were needed if `nlminb()` was used instead of `bfgs()`. The following looks at the classification error in the training set (left side) and test set (right side).

```
> tmp1 <- nn.classif(a,rbind(xx1,xx2))
> cbind(table(tmp1$predclass[1:99],yy1),table(tmp1$predclass[100:150],yy2))
   1  2  3  1  2  3
1 33  0  0 17  0  0
2  0 31  0  0 16  2
3  0  2 33  0  1 15
```

Only 2/99 cases in the training set and 3/51 in the test set were misclassified. Decreasing λ can give a perfect fit on the training data, but not on the test data, as follows.

```
> a <- nnfit(yy1,xx1,lambda=.001)
> a[[5]]
 iter error nskip steph        score
  984     0   414    24 4.433492e-07
> tmp1 <- nn.classif(a,rbind(xx1,xx2))
> cbind(table(tmp1$predclass[1:99],yy1),table(tmp1$predclass[100:150],yy2))
   1  2  3  1  2  3
1 33  0  0 17  0  0
2  0 33  0  0 16  1
3  0  0 33  0  1 16
```

Repeating this last run with different (random) starting values (but the same λ) converged to superficially quite different weights, but give similar estimated probabilities (there is no guarantee that all local minima will give similar probabilities, though). In fact the w's in the example below can be seen to be nearly the same in the two runs (with different random starting values), except for permuting the order of the nodes in the hidden layer, and sign changes. The differences in the v_{0k} then are explained by the fact that $g(-z_{in}) = 1 - g(z_{in})$, so when the sign of z changes in a hidden node, there needs to be a corresponding shift in the constant terms in the output layer.

```
> # weights from the previous run
> a[[1]]
 [1]   7.0936628  -3.3158273  -0.2511496  -3.3158794  -3.3157866  -3.3157193
 [7]  -7.6710601   4.0802830 -16.2833373   4.0803637   4.0803526   4.0802773
```

```
[13] -10.2328227   0.5611847  16.6704100   0.5611336   0.5612029   0.5611675
[19]  -1.0412576   1.0914919  -1.7996024   3.0516115   3.1463228 -14.4668813
[25]  -1.8711734  -5.4136442  12.6672366  13.3805759  -1.0413502   1.0915204
[31]  -1.7995113   3.0516819   3.1463718  -1.0412965   1.0913711  -1.7995869
[37]   3.0516824   3.1464454  -1.0412297   1.0913732  -1.7996240   3.0516300
[43]   3.1463600
> a <- nnfit(yy1,xx1,lambda=.001)
> a[[5]]
 iter error nskip steph        score
  571     0     6    26 1.375081e-06
> tmp2 <- nn.classif(a,rbind(xx1,xx2))
> summary(tmp1$probs-tmp2$probs)
       Min.    1st Qu.    Median       Mean   3rd Qu.      Max.
 -1.406e-06 -3.897e-08 3.524e-09 -1.835e-08 1.005e-08 8.572e-07
> cbind(table(tmp2$predclass[1:99],yy1),table(tmp2$predclass[100:150],yy2))
   1  2  3  1  2  3
1 33  0  0 17  0  0
2  0 33  0  0 16  1
3  0  0 33  0  1 16
> a[[1]]
 [1]   3.7776828  -0.2510077  -3.3157990  -3.3155459   3.3159455  -3.3159144
 [7]  -3.5906640 -16.2833610   4.0803057   4.0802354  -4.0803482   4.0803571
[13]  -9.6717878  16.6704108   0.5612308   0.5609241  -0.5612391   0.5615104
[19] -14.4668706  -1.8711746  -5.4136404  12.6672279  13.3805672  -1.0412896
[25]   1.0914676  -1.7995930   3.0517861   3.1462479  -1.0411609   1.0914274
[31]  -1.7996984   3.0515710   3.1463002   1.0413374  -1.0913078   1.7995014
[37]  -3.0516416  -3.1464935  -1.0413625   1.0914902  -1.7995685   3.0515236
[43]   3.1465673
```

Using a larger value of λ than in the original fit gives the following.

```
> a <- nnfit(yy1,xx1,lambda=.3)
> a[[5]]
 iter error nskip steph        score
  200     0     8     5 0.0001279832
> tmp1 <- nn.classif(a,rbind(xx1,xx2))
> cbind(table(tmp1$predclass[1:99],yy1),table(tmp1$predclass[100:150],yy2))
   1  2  3  1  2  3
1 33  1  0 17  1  0
2  0  9  0  0  3  0
3  0 23 33  0 13 17
```

The fact that there is so much improvement between $\lambda = .3$ and $\lambda = .1$ in both the training and test data sets, but little improvement beyond $\lambda = .1$, gives some indication that the model with $\lambda = .1$ might give a reasonable classification rule. However, the test set is probably too small to be certain of this.

Interestingly, when $\lambda = 0$, the algorithm can converge to very different local minima, as shown below.

```
> a <- nnfit(yy1,xx1,lambda=0)
> a[[5]]
 iter error nskip steph       score
 1580     0  1493     9 3.12263e-07
> tmp1 <- nn.classif(a,rbind(xx1,xx2))
> cbind(table(tmp1$predclass[1:99],yy1),table(tmp1$predclass[100:150],yy2))
   1  2  3  1  2  3
1 33 20  0 17 12  0
2  0  4  0  0  2  0
3  0  9 33  0  3 17
> a <- nnfit(yy1,xx1,lambda=0)
> a[[5]]
 iter error nskip steph        score
   85     0    10     5 4.142313e-09
> tmp2 <- nn.classif(a,rbind(xx1,xx2))
> summary(tmp1$probs-tmp2$probs)
 Min.    1st Qu.     Median    Mean   3rd Qu.      Max.
   -1 -2.949e-20 -2.939e-39 -0.1133 6.446e-83 5.836e-08
> cbind(table(tmp2$predclass[1:99],yy1),table(tmp2$predclass[100:150],yy2))
   1  2  3  1  2  3
1 33  0  0 17  0  0
2  0 33  0  0 16  1
3  0  0 33  0  1 16
```

3.6 Newton's Method for Solving Nonlinear Equations

Suppose $G(x) = (g_1(x), \ldots, g_p(x))'$, where each $g_j : R^p \to R^1$ is a smooth function. Consider the problem of finding a solution x^* to the equations

$$G(x) = 0.$$

As discussed in Section 3.2.2, Newton's method for solving nonlinear equations is based on repeatedly solving the linear system given by the tangent plane approximation to G at the current point. The tangent plane approximation is

$$G(x) \doteq G(x_0) + J(x_0)(x - x_0),$$

and the Newton updates are of the form

$$x_1 = x_0 - J(x_0)^{-1} G(x_0),$$

where $J(x)$ is the Jacobian matrix of G at x,

$$J(x) = \left(\frac{\partial g_i(x)}{\partial x_j} \right)$$

(ith row, jth column).

As in the minimization problem, Newton's method usually has quadratic convergence when started close enough to a solution, but often diverges otherwise. Unlike the minimization problem, here there is not an objective function to monitor to check if the new point is better. The usual approach is similar, though, in that the Newton directions are used, and the magnitude of the components of G are monitored to see if they are getting closer to 0. In particular, the value of

$$f(x) = G(x)'G(x)/2$$

can be monitored. The Newton direction is

$$d = -J(x_0)^{-1}G(x_0).$$

Since $\nabla f(x) = J(x)'G(x)$, it follows that

$$d'\nabla f(x_0) = -G(x_0)'[J(x_0)^{-1}]'J(x_0)'G(x_0) = -G(x_0)'G(x_0) < 0,$$

so d is a descent direction for f at x_0.

As in the minimization problem, to preserve the good convergence rate of Newton's method, at each iteration the full Newton step is tried first. If the full step sufficiently improves on $f(x_0)$, then the new point is retained, and the iteration proceeds. If not, then backtracking is used to locate a better point in the same direction. Such a point exists because d is a descent direction for f. The new point then is

$$x_1(\lambda) = x_0 - \lambda J(x_0)^{-1}G(x_0),$$

for some $0 < \lambda \leq 1$, where x_0 is the current point (and where $\lambda = 1$ corresponds to the full Newton step). To guarantee minimal progress, the condition

$$f[x_1(\lambda)] < f(x_0) + 10^{-4}[x_1(\lambda) - x_0]'\nabla f(x_0) = f(x_0) - 10^{-4}\lambda G(x_0)'G(x_0),$$

(or something similar) should again be required to declare a new point to be better. Alternately, a model trust region approach could be implemented (see Dennis and Schnabel, 1983, for details).

Below is a function `neq()` which implements a backtracking version of Newton's method, using proportional step reduction for backtracking. Polynomial interpolation could also be used instead. `neq()` requires separate functions for evaluating $G(x)$ (`gn`) and the Jacobian of G (`jn`). The convergence criterion used stops when $\max_i |g_i(x_1)| <$ `gtol`. Of course, appropriate values of `gtol` depend on the scaling of the equations. It is the user's responsibility to choose a value that makes sense for the scaling used (and to use appropriate scaling). The iteration also terminates when the relative change in x between successive iterations is $<$ `steptol`. This can happen for example if $f(x) = G(x)'G(x)/2$ has a local minimum which is not at a solution to $G(x) = 0$. If this happens the function returns an error code of 3.

```
# Modified Newton's method for solving nonlinear equations
# Arguments
# b=initial parameter values
# gn=function to calculate vector of nonlinear equations, called as
```

```
#   gn(b,...)
# jn=function to calc Jacobian of the nonlinear equations,
#   called as jn(b,...)  should return a matrix
# gtol solution identified if max(abs(gn(b,...)))<gtol
# iter=max # iterations (input),
# stepf=the fraction by which the step size is reduced in each step of
#   the backtracking algorithm
# steptol--if step size is smaller than this then algorithm has stalled
# ... additional arguments to gn
# returns a list with components b=approx solution, f=||gn(b)||^2/2,
#   and comp=a vector with components named
#   iter, giving the number of iterations used, an error
#   code (error=0 no errors, =1 if error in directional search, =2 if
#   max iterations exceeded, error=3 if iteration has stalled at a
#   point that is not a solution), and steph giving the number of
#   times the step length was reduced

neq <- function(b,gn,jn,gtol=1e-6,iter=50,stepf=.5,steptol=1e-8,...) {
  n <- length(b)
  steph <- 0
  g0 <- gn(b,...)
  f0 <- sum(g0*g0)/2
  for (ll in 1:iter) {
# new direction
    j <- jn(b,...)
    sc <- -c(solve(j,g0))
    bn <- b+sc
    g1 <- gn(bn,...)
    f1 <- sum(g1*g1)/2
# backtracking loop
    i <- 0
    lam <- -2*f0
    while (is.na(f1) || f1>f0+(1e-4)*lam) {
      i <- i+1
      steph <- steph+1
      sc <- sc*stepf
      lam <- lam*stepf
      bn <- b+sc
      g1 <- gn(bn,...)
      f1 <- sum(g1*g1)/2
      if (i>20) return(list(b=b,f=f0,comp=c(iter=ll,error=1,steph=steph)))
    }
    if (max(abs(g1))<gtol) # if true, iteration converged
      return(list(b=bn,f=f1,comp=c(iter=ll,error=0,steph=steph)))
    if (max(abs(b-bn)/pmax(abs(b),1))<steptol)
      return(list(b=bn,f=f1,comp=c(iter=ll,error=3,steph=steph)))
```

```
    b <- bn
    g0 <- g1
    f0 <- f1
  }
# max number of iterations exceeded
  list(b=bn,f=f1,comp=c(iter=ll,error=2,steph=steph))
}
```

Since `solve()` is a generic function, the algorithm used to compute $J^{-1}G$ in the `solve()` command is under the control of the user. If the Matrix library has been invoked, then the method for whatever class J is defined as in the routine `jn()` will be invoked. Since in general J is not symmetric, the LU decomposition would often be appropriate.

Problems can result if points are encountered where the Jacobian matrix is poorly conditioned. A more sophisticated version of the algorithm would check for a poorly conditioned Jacobian matrix, and make some modification if necessary. One such modification is to use the direction

$$d^*(\alpha) = -[J(x_0)'J(x_0) + \alpha I]^{-1} J(x_0)'G(x_0),$$

where α is a small positive number, instead of the Newton direction; see Section 6.5 of Dennis and Schnabel (1983). (Note that $d^*(\alpha)$ reduces to the Newton direction at $\alpha = 0$.)

As in optimization problems, finite difference approximations to the Jacobian, such as given by the `fdjac()` function, can be used. In addition to using finite difference approximations, there are also methods based on updates to secant approximations, which do not directly require information on the Jacobian. Broyden's method in Section 9.7 of Press *et. al.* (1992) is an example of such a method. Additional details on such methods can be found in Chapter 8 of Dennis and Schnabel (1983).

3.6.1 Example: Estimating Equations with Missing Covariate Data

Consider logistic regression of a binary response y_i on vectors of covariates x_i and z_i. The true model is assumed to be

$$\frac{P(y_i = 1|x_i, z_i)}{1 - P(y_i = 1|x_i, z_i)} = \exp(\alpha + x_i'\beta + z_i'\gamma),$$

for unknown parameters $\theta' = (\alpha, \beta', \gamma')$. Suppose θ has length m. The components of z_i are not always observed. Let $R_i = 1$ if all components of z_i are observed, $= 0$ otherwise. Cases with $R_i = 1$ are called complete cases. Set $\pi = P(R_i = 1)$ marginally, and suppose

$$P(R_i = 1|y_i, x_i, z_i) = P(R_i = 1) = \pi, \tag{3.12}$$

that is, the data are missing completely at random. In this case the scores for the likelihood using only the complete cases form unbiased estimating equations for θ. Although it is unbiased, the complete case analysis ignores the information in the partially complete cases, and so is inefficient. There is considerable interest in how best to include information from the partially complete cases in the analysis.

Define $p(x_i, z_i, \theta) = P(y_i = 1|x_i, z_i)$. The complete data scores are $\sum_i R_i U(y_i, x_i, z_i, \theta)$, where

$$U(y, x, z, \theta) = \begin{pmatrix} 1 \\ x \\ z \end{pmatrix} [y - p(x, z, \theta)].$$

As already mentioned, the scores for the observed complete cases are unbiased, because

$$\mathrm{E}_\theta[U(y_i, x_i, z_i, \theta)|R_i = 1] = \mathrm{E}_\theta[U(y_i, x_i, z_i, \theta)] = 0,$$

by (3.12) and the fact that $U(y_i, x_i, z_i, \theta)$ is the usual logistic regression score vector contribution, which has marginal mean 0.

Any equation of the form

$$\sum_i \frac{R_i}{\pi} U(y_i, x_i, z_i, \theta) + \left(1 - \frac{R_i}{\pi}\right) \phi(y_i, x_i, \theta) = 0$$

is also an unbiased estimating equation for θ, where ϕ is any m-vector of functions of y_i and x_i. This follows since

$$\mathrm{E}\left[\left(1 - \frac{R_i}{\pi}\right) \phi(y_i, x_i, \theta)\right] = \mathrm{E}_{y_i, x_i}\left[\phi(y_i, x_i, \theta) \mathrm{E}_{R_i|y_i, x_i}\left(1 - \frac{R_i}{\pi}\right)\right] = 0.$$

For certain patterns of missingness, Robins, Rotnitzky and Zhao (1994) have shown the optimal choice of ϕ is

$$\phi^*(y_i, x_i, \theta) = \mathrm{E}_{z_i|y_i, x_i}\left[U(y_i, x_i, z_i, \theta)\right],$$

the conditional expectation of the complete data score.

Calculating this conditional expectation requires knowledge of the joint distribution of (y_i, x_i, z_i), and if z_i is continuous it would likely require numerical integration to compute. Thus simpler approximations are of interest. The choice of ϕ will only effect the efficiency, though, since the estimating equations remain unbiased regardless.

One simple approximation that could be considered is to set

$$\phi(y_i, x_i, \theta) = U(y_i, x_i, \mathrm{E}[z_i|y_i, x_i], \theta).$$

The conditional expectation $\mathrm{E}[z_i|y_i, x_i]$ could be estimated from the complete cases using any convenient approximation, such as linear regression. This crude approximation will be used below.

For the linear regression to estimate $\mathrm{E}[z_i|y_i, x_i]$, the model

$$z_{ij} = \mu_{0j} + \mu_{1j} y_i + \mu'_{2j} x_i + e_i$$

can be used, with the parameters estimated using only the complete cases. Ordinary least squares might be appropriate for calculating the estimates. More complex models incorporating interactions and nonlinear effects could also be considered. Whatever model is used, the estimates of $\mathrm{E}[z_i|y_i, x_i]$ are just the predicted values from the regression. These are needed for both the complete cases used in fitting the regression model, and for the incomplete cases which were not

included. Denote the predicted value for case i by z_i^*. The estimating equations for θ are then given by

$$\sum_i \frac{R_i}{\hat{\pi}} U(y_i, x_i, z_i, \theta) + \left(1 - \frac{R_i}{\hat{\pi}}\right) U(y_i, x_i, z_i^*, \theta) = 0, \tag{3.13}$$

with $\hat{\pi}$ set to the observed proportion of complete cases.

For arbitrary functions ϕ, the estimating equations would generally not be representable as the derivatives of an objective function. In this particular case, the functions on the left hand side of (3.13) are the partial derivatives of a common function. However, minimizing this turns out not to be very useful (it is **NOT** a likelihood), probably because it is sometimes minimized in the limit as combinations of parameters become infinite. For this reason it appears better to directly solve the system of nonlinear equations without utilizing an objective function.

Below the function `neq()` is used to compute the solution to these estimating equations for some simulated data.

```
> mlrsc <- function(b,resp,comp,miss,cov.pred,mi) {
+ # resp=response (y), comp=matrix of always observed covs (x)
+ # miss=matrix of incomplete covs (z), cov.pred=predicted values of miss
+ # from linear regressions, mi=1 if any data missing (1-r)
+ # compute estimating equations
+   n <- length(resp)
+   pihat <- table(mi)[1]/n #prob not missing
+   z1 <- cbind(rep(1,n)[!mi],comp[!mi,],miss[!mi,])
+   pr1 <- exp(z1 %*% b)
+   pr1 <- pr1/(1+pr1)
+   u <- apply(c(resp[!mi]-pr1)*z1,2,sum)/pihat
+   z1 <- cbind(rep(1,n),comp,cov.pred)
+   pr1 <- exp(z1 %*% b)
+   pr1 <- pr1/(1+pr1)
+   u+apply((pihat-1+mi)*c(resp-pr1)*z1,2,sum)/pihat
+ }
> mlrjac <- function(b,resp,comp,miss,cov.pred,mi) {
+ # compute Jacobian of estimating equations
+   n <- length(resp)
+   pihat <- table(mi)[1]/n #prob not missing
+   z1 <- cbind(rep(1,n)[!mi],comp[!mi,],miss[!mi,])
+   pr1 <- exp(c(z1 %*% b))
+   pr1 <- pr1/(1+pr1)^2
+   u <- -t(z1)%*%(pr1*z1)/pihat
+   z1 <- cbind(rep(1,n),comp,cov.pred)
+   pr1 <- exp(c(z1 %*% b))
+   pr1 <- pr1/(1+pr1)^2
+   u - t(z1) %*% ((pihat-1+mi)*pr1*z1)/pihat
+ }
>
```

```
> # generate some data: 4 covariates, pairwise correlation rho
> .Random.seed
 [1] 17 12  2 22 23  0 55 14 41  4 41  1
> rho <- .5
> n <- 200
> p <- 4
> nmis <- 2  # # covariates with some values missing
> alpha <- -1  #constant term
> beta <- c(1,-1,1,-1)  # true regression coefficients
> miss.prob <- .5  # prob cov value is missing for each case
> a <- matrix(rho,p,p)
> diag(a) <- 1
> a <- chol(a)
> Z <- t(a) %*% matrix(rnorm(n*p),nrow=p)
> cor(t(Z))
          [,1]      [,2]      [,3]      [,4]
[1,] 1.0000000 0.5058296 0.5109325 0.4887334
[2,] 0.5058296 1.0000000 0.5157523 0.5055901
[3,] 0.5109325 0.5157523 0.9999999 0.4875875
[4,] 0.4887334 0.5055901 0.4875875 1.0000000
> rp <- exp(alpha+c(beta%*%Z))
> rp <- rp/(1+rp)
> resp <- ifelse(runif(n)<rp,1,0)
> comp <- t(Z[1:(p-nmis),])
> miss <- t(Z[(p-nmis+1):p,])
> miss[runif(nmis*n)<miss.prob] <- NA
>
> # identify missing values
> mi <- is.na(apply(miss,1,sum))
> table(mi)
 FALSE TRUE
    42  158
> # regression to estimate conditional expectations
> newd <- data.frame(comp,resp)
> names(newd) <- c('comp1','comp2','resp')
> cov.pred <- predict(lm(miss~comp+resp,subset=!mi),newdata=newd)
> # (computed predicted values for all data points)
> # estimate logistic parameters
> neq(rep(0,p+1),mlrsc,mlrjac,resp=resp,comp=comp,miss=miss,
+   cov.pred=cov.pred,mi=mi)
$b:
[1] -1.937627  1.970884 -2.131761  2.421421 -2.943655

$f:
[1] 2.396976e-24
```

```
$comp:
 iter error steph
    7     0     0

> # different starting value
> neq(rnorm(p+1),mlrsc,mlrjac,resp=resp,comp=comp,miss=miss,
+   cov.pred=cov.pred,mi=mi)
$b:
[1] -1.937627  1.970884 -2.131761  2.421421 -2.943655

$f:
[1] 2.122109e-13

$comp:
 iter error steph
    6     0     0

> # complete cases analysis
> glm(resp~comp+miss,subset=!mi, family=binomial)
Call:
glm(formula = resp ~ comp + miss, family = binomial, subset = !mi)

Coefficients:
 (Intercept)    comp1     comp2    miss1     miss2
   -2.224338 1.667093 -1.875632 2.210221 -2.696877

Degrees of Freedom: 42 Total; 37 Residual
Residual Deviance: 27.41509
> neq(rep(0,p+1),mlrsc,mlrjac,resp=resp[!mi],comp=comp[!mi,],
+   miss=miss[!mi,],cov.pred=cov.pred[!mi,],mi=mi[!mi])
$b:
[1] -2.224705  1.667337 -1.875895  2.210749 -2.697571

$f:
[1] 5.119976e-15

$comp:
 iter error steph
    6     0     0

> # finite difference approximation to Jacobian
> mlrfdjac <- function(b,resp,comp,miss,cov.pred,mi)
+   fdjac(b,mlrsc,resp,comp,miss,cov.pred,mi)
> mlrjac(c(alpha,beta),resp,comp,miss,cov.pred,mi)
            [,1]        [,2]       [,3]       [,4]       [,5]
[1,] -34.3292050  -0.5134052   1.833713  -6.846105   6.644404
```

```
[2,]  -0.5134052 -32.8026178 -20.856771 -16.139398 -13.589402
[3,]   1.8337132 -20.8567711 -33.751099 -15.542784  -8.124899
[4,]  -6.8461052 -16.1393975 -15.542784 -20.417583  -8.416443
[5,]   6.6444042 -13.5894017  -8.124899  -8.416443 -20.769636
> mlrfdjac(c(alpha,beta),resp,comp,miss,cov.pred,mi)
            [,1]        [,2]       [,3]       [,4]       [,5]
[1,] -34.3292051  -0.5134054   1.833712  -6.846105   6.644404
[2,]  -0.5134057 -32.8026179 -20.856772 -16.139397 -13.589402
[3,]   1.8337133 -20.8567709 -33.751099 -15.542784  -8.124900
[4,]  -6.8461052 -16.1393975 -15.542784 -20.417583  -8.416443
[5,]   6.6444046 -13.5894014  -8.124899  -8.416443 -20.769636
> neq(rep(0,p+1),mlrsc,mlrfdjac,resp=resp,comp=comp,miss=miss,
+   cov.pred=cov.pred,mi=mi)
$b:
[1] -1.937627  1.970884 -2.131761  2.421421 -2.943655

$f:
[1] 2.221429e-24

$comp:
 iter error steph
    7     0     0

> unix.time(neq(rep(0,p+1),mlrsc,mlrfdjac,resp=resp,comp=comp,miss=miss,
+   cov.pred=cov.pred,mi=mi))
[1] 3.78 0.00 4.00 0.00 0.00
> unix.time(neq(rep(0,p+1),mlrsc,mlrjac,resp=resp,comp=comp,miss=miss,
+   cov.pred=cov.pred,mi=mi))
[1] 0.9799995 0.0000000 1.0000000 0.0000000 0.0000000
```

`neq()` converged quite rapidly in this example, even though there was a substantial proportion of cases with incomplete data, with moderate correlation among the covariates. In roughly a dozen similar data sets, no convergence problems were encountered. There is some sensitivity to the starting point in the performance of the algorithm.

The results of the complete cases analysis were not very close to those from the estimating equations, but that is not very surprising since only 21% of the cases were complete.

Performance using the finite difference approximation to the Jacobian is nearly identical to that for the analytical Jacobian, except with respect to cpu time. The difference in cpu time may be in part due to the extra overhead in the loops and nested function calls, which might give a larger difference in Splus than in other languages. The difference may be smaller in a pure FORTRAN or C implementation.

Since the `Matrix` library was not invoked, the default QR decomposition was used in the `solve()` commands. When the `Matrix` library was used, and the class of the analytical Jacobian was set to `c('Hermitian','Matrix')`, to invoke the symmetric indefinite version of `solve()`, the cpu

time was slightly longer than for the version above. In general, the `Matrix` library functions are not a particularly efficient implementation of the algorithms used.

3.7 Nonlinear Gauss-Seidel Iteration

The basic idea of the Gauss-Seidel iteration for solving linear equations can also be applied to non-linear problems (see eg Section 4.3.4 of Thisted, 1988). In this method, to solve

$$G(x) = (g_1(x), \ldots, g_p(x))' = (0, \ldots, 0)',$$

given an initial point $x^{(0)} = (x_1^{(0)}, \ldots, x_p^{(0)})'$, first the equation

$$g_1(x_1, x_2^{(0)}, \ldots, x_p^{(0)}) = 0$$

is solved for $x_1 = x_1^{(1)}$, keeping the other components fixed. Then

$$g_2(x_1^{(1)}, x_2, x_3^{(0)}, \ldots, x_p^{(0)}) = 0$$

is solved for $x_2 = x_2^{(1)}$, again keeping the other components fixed. This process is continued until finally

$$g_p(x_1^{(1)}, \ldots, x_{p-1}^{(1)}, x_p) = 0$$

is solved for $x_p = x_p^{(1)}$. Then $x^{(0)}$ is replaced by $x^{(1)}$, and the entire process repeated, until convergence. The exact order the equations are processed is arbitrary, and can be varied from one pass to the next.

This method can be quite useful in some problems, and is used routinely in problems like fitting generalized additive models. However, the rate of convergence depends heavily on how strong the dependencies among the equations are. If the jth equation is nearly independent of the other variables, then convergence will be very fast, while otherwise it can be quite slow. The method also only converges at a linear rate near the solution, so may be more useful when high precision is not needed.

Below is a modification of `mlrsc()` to return just the jth equation. This is called in combination with `uniroot()` to solve the jth equation, giving a crude implementation of the Gauss-Seidel iteration. This particular implementation is quite slow in terms of cpu time, but illustrates that the method converges, although quite a few iterations can be needed.

```
> # Gauss Seidel
> mlrsc1 <- function(x,b,j,resp,comp,miss,cov.pred,mi) {
+ # compute jth estimating equation
+   b[j] <- x
+   n <- length(resp)
+   pihat <- table(mi)[1]/n #prob not missing
+   z1 <- cbind(rep(1,n)[!mi],comp[!mi,],miss[!mi,])
+   pr1 <- exp(z1 %*% b)
+   pr1 <- pr1/(1+pr1)
```

```
+   u <- sum((resp[!mi]-pr1)*z1[,j])/pihat
+   z1 <- cbind(rep(1,n),comp,cov.pred)
+   pr1 <- exp(z1 %*% b)
+   pr1 <- pr1/(1+pr1)
+   u+sum((pihat-1+mi)*c(resp-pr1)*z1[,j])/pihat
+ }
> b <- rep(0,(p+1))
> for (k in 1:20) {
+   for (j in 1:(p+1))
+     b[j] <- uniroot(mlrsc1,c(-5,5),b=b,j=j,resp=resp,comp=comp,
+       miss=miss,cov.pred=cov.pred,mi=mi)[[1]]
+   print(b)
+ }
[1] -0.7537806  0.5504519 -0.6482014  0.6544366 -0.9458477
[1] -0.9461827  0.9801123 -1.0161491  1.0989794 -1.4090325
[1] -1.138037  1.186189 -1.275633  1.405184 -1.702155
[1] -1.291558  1.332846 -1.462370  1.617214 -1.923521
[1] -1.409850  1.451551 -1.599529  1.769575 -2.100687
[1] -1.502706  1.548543 -1.702731  1.884346 -2.245064
[1] -1.577033  1.627066 -1.782199  1.974400 -2.363419
[1] -1.637391  1.690202 -1.844622  2.047130 -2.460700
[1] -1.686884  1.740855 -1.894431  2.107039 -2.540878
[1] -1.727840  1.781618 -1.934743  2.156889 -2.607170
[1] -1.761795  1.814547 -1.967636  2.198616 -2.662079
[1] -1.790063  1.841277 -1.994673  2.233657 -2.707679
[1] -1.813657  1.863097 -2.017020  2.263131 -2.745640
[1] -1.833387  1.881004 -2.035569  2.287945 -2.777312
[1] -1.849914  1.895768 -2.051016  2.308844 -2.803789
[1] -1.863774  1.907992 -2.063911  2.326452 -2.825959
[1] -1.875412  1.918148 -2.074698  2.341292 -2.844550
[1] -1.885192  1.926609 -2.083737  2.353801 -2.860158
[1] -1.893418  1.933674 -2.091319  2.364347 -2.873275
[1] -1.900340  1.939585 -2.097688  2.373241 -2.884307
> b <- rep(0,(p+1))
> for (k in 1:10) {
+   for (j in 1:(p+1))
+     b[j] <- uniroot(mlrsc1,c(-5,5),b=b,j=j,resp=resp,comp=comp,
+       miss=miss,cov.pred=cov.pred,mi=mi)[[1]]
+   for (j in p:1) #reverse direction
+     b[j] <- uniroot(mlrsc1,c(-5,5),b=b,j=j,resp=resp,comp=comp,
+       miss=miss,cov.pred=cov.pred,mi=mi)[[1]]
+   print(b)
+ }
[1] -1.0275617  0.9429030 -0.9373084  1.0871376 -0.9458477
[1] -1.253645  1.170589 -1.294156  1.528826 -1.439667
[1] -1.409932  1.331468 -1.499171  1.780198 -1.775423
```

```
[1] -1.522369  1.453633 -1.636888  1.940715 -2.019265
[1] -1.606288  1.549350 -1.737205  2.051020 -2.203048
[1] -1.670577  1.625678 -1.813534  2.130561 -2.345016
[1] -1.720873  1.687204 -1.873127  2.189888 -2.456601
[1] -1.760719  1.737159 -1.920441  2.235307 -2.545491
[1] -1.792638  1.777917 -1.958409  2.270685 -2.616936
[1] -1.818421  1.811412 -1.989159  2.298622 -2.674795
```

In the second implementation, alternating between forward and backward sweeps did not result in much improvement.

3.8 Exercises

Exercise 3.1 Consider the function $f(x, y) = x^2 + 10y^2$. Starting from the point $(1, 1)'$, show that computing the minimum in the direction of the gradient at each step leads to long sequence of short steps converging only slowly to the global minimum of $(0, 0)'$.

Exercise 3.2 Consider minimizing the function

$$f(\theta_1, \theta_2, \theta_3, \theta_4) = (\theta_1 + 10\theta_2)^2 + 5(\theta_3 - \theta_4)^2 + (\theta_2 - 2\theta_3)^4 + (\theta_1 - \theta_4)^4,$$

starting from the values $(\theta_1, \theta_2, \theta_3, \theta_4) = (3, -1, 0, 1)$ (this is Powell's quartic, a standard test problem). Compare the performance of the Nelder-Mead simplex method, the BFGS algorithm, and a modified Newton algorithm in solving this problem (use exact derivative formulas in all algorithms that require them). Compare performance with respect to accuracy of solutions, number of iterations, number of function evaluations, and number of gradient evaluations (where relevant).

Exercise 3.3 Consider the quadratic function $f(\theta) = \theta' A\theta/2 + b'\theta + c$, where A is positive definite. Show that if for all θ_0, $f(\theta_0 - \lambda\nabla f(\theta_0))$ is minimized (over λ) at the global minimum of $f(\theta)$, then A is a constant times the identity matrix, and that the converse also holds. (Hint: consider that for this function, from any point θ_0, the minimum always lies in the direction $-[\nabla^2 f(\theta_0)]^{-1}\nabla f(\theta_0)$.)

Exercise 3.4 File `t1.dat` contains data values (y_i, x_i, z_i, r_i), $i = 1, \ldots, n = 200$, with y_i in the first column in the file, x_i in the second, z_i in the third, and r_i in the fourth. In this data, y_i is a binary response and x_i and z_i are binary covariates. The value of z_i is not always observed, and $r_i = 1$ indicates that z_i is observed and $r_i = 0$ indicates z_i is missing (z_i is coded -1 when it is missing). Assume the sampled vectors (Y_i, X_i, Z_i, R_i) are iid, and that R_i and Z_i are independent given (Y_i, X_i) (this is sometimes referred to as missing at random, although it is not identical to Rubin's definition of that term). The interest here is in estimating the parameters of the logistic regression of y_i on the covariates x_i and z_i; that is

$$\log\left(\frac{P(Y_i = 1|X_i = x_i, Z_i = z_i)}{P(Y_i = 0|X_i = x_i, Z_i = z_i)}\right) = \beta_0 + \beta_1 x_i + \beta_2 z_i.$$

Let $\theta_x = P(Z_i = 1|X_i = x)$, $x = 0, 1$, and assume $P(R_i|X_i, Y_i)$ does not depend on $(\beta_0, \beta_1, \beta_2, \theta_0, \theta_1)$.

In this model, the log likelihood for estimating $(\beta_0, \beta_1, \beta_2, \theta_0, \theta_1)$ can be written

$$\sum_i r_i\{y_i(\beta_0 + \beta_1 x_i + \beta_2 z_i) - \log[1 + \exp(\beta_0 + \beta_1 x_i + \beta_2 z_i)] + \log[\theta_{x_i}^{z_i}(1 - \theta_{x_i})^{1-z_i}]\}$$
$$+ \sum_i (1 - r_i) \log\left(\sum_{u=0}^{1} \frac{\exp[y_i(\beta_0 + \beta_1 x_i + \beta_2 u)]\theta_{x_i}^{u}(1 - \theta_{x_i})^{1-u}}{1 + \exp(\beta_0 + \beta_1 x_i + \beta_2 u)}\right).$$

1. Derive formulas for the gradient of the log likelihood. (The parameters θ_0 and θ_1 should probably be transformed to eliminate constraints, before proceeding with this step.)

2. Write functions in Splus to evaluate the negative log likelihood and its gradient. (I might be helpful to check the gradient formulas by comparing the output of the gradient function to a finite difference approximation.) Also, write a function to evaluate the second derivatives using finite differences of the gradient (the function `fdjac()` may be used here).

3. Find the maximum likelihood estimates (i) using the Nelder-Mead simplex method, (ii) using a BFGS method with gradient calculated from analytical formulas, (iii) a modified Newton-Raphson algorithm. In the Newton-Raphson algorithm, use the finite difference approximation to the Hessian (which may need to be symmetrized for the `chol()` function), but use the analytic gradient formulas. In Splus, for (ii) and (iii) either `nlminb()` or `bfgs()` and `nr()` can be used.

4. Compare the algorithms in (c) with respect to the number of iterations, function evaluations, gradient evaluations, and Hessian evaluations.

5. Compare the inverse Hessian at the MLE (computed using finite differences of the analytic gradient) to the approximate inverse Hessian from the final iteration of the BFGS algorithm.

3.9 References

Cheng B and Titterington DM (1994). Neural networks: a review from a statistical perspective (with discussion). *Statistical Science* 9:2–54.

Dennis JE and Schnabel RB (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. SIAM.

Hastie T, Tibshirani R and Buja A (1994). Flexible discriminant analysis by optimal scoring. *Journal of the American Statistical Association*, 89:1255–1270.

Lange K (1999). *Numerical Analysis for Statisticians*. Springer.

Powers R (1995). *Galatea 2.2*. HarperPerennial.

Press WH, Teukolsky SA, Vetterling WT, and Flannery BP (1992). *Numerical Recipes in C: The Art of Scientific Computing. Second Edition*. Cambridge University Press.

Ripley BD (1996). *Pattern Recognition and Neural Networks*. Cambridge University Press.

Robins JM, Rotnitzky A and Zhao LP (1994). Estimation of regression coefficients when some regressors are not always observed. *Journal of the American Statistical Association*, 89:846–866.

Thisted RA (1988). *Elements of Statistical Computing*. Chapman & Hall.

Venables WN and Ripley BD (1997). *Modern Applied Statistics with S-Plus, 2nd Edition*. Springer. (the third edition is now available)

Warner B and Misra M (1996). Understanding neural networks as statistical tools. *The American Statistician*, 50:284–293.

Chapter 4

The EM Algorithm

4.1 Introduction

The EM Algorithm is not an algorithm in the usual mathematical or computational sense, but is a general approach to finding maximum likelihood estimators in incomplete data problems. Considerable work can still be required to implement the general approach in particular problems. Also, there are many variations on how the details can be implemented. Two examples of incomplete data problems follow. Then the main ideas of the EM algorithm will be described.

Example 4.1 Censored Data Linear Regression. Suppose the true model for responses T_i is

$$T_i = \beta' z_i + \sigma\epsilon_i, \quad i = 1, \ldots, n,$$

where $z_i = (1, z_{i1}, \ldots, z_{ip})'$ is a vector of fixed covariates, $\beta = (\beta_0, \beta_1, \ldots, \beta_p)'$ and $\sigma > 0$ are unknown parameters, and the ϵ_i are iid N(0,1). Suppose also that there are independent variates $C_1, \ldots, C_n$, and that the observed data consist of (Y_i, δ_i, z_i), $i = 1, \ldots, n$, where $Y_i = \min\{T_i, C_i\}$ and $\delta_i = I(T_i \leq C_i)$, so Y_i is the observed right-censored response. When $\delta_i = 0$, it is only known that $T_i > Y_i$.

If the T_i were all observed, then the log likelihood would be

$$l_{comp}(\theta) = -n\log(\sigma) - \sum_{i=1}^{n}(T_i - \beta' z_i)^2/(2\sigma^2), \tag{4.1}$$

where $\theta = (\beta', \sigma^2)'$. $l_{comp}(\theta)$ is maximized by $\hat{\beta}_c = (Z'Z)^{-1}Z'(T_1, \ldots, T_n)'$ and $\hat{\sigma}_c^2 = \sum_i (T_i - \hat{\beta}_c' z_i)^2/n$, where

$$Z = \begin{pmatrix} z_1' \\ \vdots \\ z_n' \end{pmatrix}.$$

In the terminology of the EM algorithm, $l_{comp}(\theta)$ is the complete data log likelihood.

With censored data, the likelihood contribution of the ith subject is the same as above if the case is not censored, and if the case is censored the likelihood contribution is

$$\int_{Y_i}^{\infty} \sigma^{-1}\phi([u - \beta' z_i]/\sigma)\, du = 1 - \Phi([Y_i - \beta' z_i]/\sigma),$$

where $\phi(\cdot)$ and $\Phi(\cdot)$ are the standard normal density and CDF. Thus the log likelihood for the observed data is

$$l_{obs}(\theta) = \sum_i \{\delta_i[-\log(\sigma) - (Y_i - \beta' z_i)^2/(2\sigma^2)] + (1-\delta_i)\log(1 - \Phi([Y_i - \beta' z_i]/\sigma))\}.$$

In the terminology of the EM algorithm, $l_{obs}(\theta)$ is the observed data log likelihood. With censored data the likelihood no longer has a closed form solution for the maximum, and an iterative search is required to find the MLE. Although direct maximization of $l_{obs}(\theta)$ is not particularly difficult in this problem, the EM algorithm will provide a simple iterative method for calculating the MLE based on repeatedly calculating modified versions of the complete data estimators. □

Example 4.2 A Mixed Effects Linear Model. Suppose the true model for responses Y_{ij} is

$$Y_{ij} = \beta' x_{ij} + b_j + \sigma\epsilon_{ij}, \quad i = 1, \ldots, n_j, \quad j = 1, \ldots, N, \tag{4.2}$$

where the ϵ_{ij} are iid $N(0,1)$, the b_j are iid $N(0, \sigma_b^2)$ (and the b_j and ϵ_{ij} are independent), $x_{ij} = (x_{ij1}, \ldots, x_{ijp})'$ is a vector of fixed covariates, which would usually include a constant term (eg $x_{ij1} = 1$), and $\beta = (\beta_1, \ldots, \beta_p)'$, $\sigma_b > 0$ and $\sigma > 0$ are unknown parameters. This model is called a mixed effects model because it contains both fixed effects (the x_{ij}) and random effects (the b_j). The data might have arisen from a clinical trial with N participating centers, with n_j patients entered from the jth center. The model allows for correlation among the patients from the same center. The random effects structure in this model is particularly simple, but the methods are easily extended to more complicated models, such as crossed and nested structures.

The observed data consist of (Y_{ij}, x_{ij}), $i = 1, \ldots, n_j$, $j = 1, \ldots, N$. Let

$$Y_j = \begin{pmatrix} Y_{1j} \\ \vdots \\ Y_{n_j j} \end{pmatrix}, \quad X_j = \begin{pmatrix} x'_{1j} \\ \vdots \\ x'_{n_j j} \end{pmatrix}, \quad \text{and} \quad V_j = \sigma^2 I_{n_j \times n_j} + \sigma_b^2 \mathbf{1}_{n_j} \mathbf{1}'_{n_j},$$

where $I_{k\times k}$ is the $k \times k$ identity matrix and $\mathbf{1}_k$ is a k-vector of ones. It is easily verified that Y_j has a multivariate normal distribution with mean $X_j\beta$ and covariance matrix V_j. The log likelihood for the observed data is therefore

$$l_{obs}(\theta) = \sum_{j=1}^{N} -\log(|V_j|)/2 - (Y_j - X_j\beta)' V_j^{-1} (Y_j - X_j\beta)/2,$$

where in general $|A|$ denotes the determinant of a matrix A, and $\theta = (\beta', \sigma^2, \sigma_b^2)'$. The simple form of V_j allows explicit formulas to be given for $|V_j|$ and V_j^{-1}. In particular, $|V_j| = (\sigma^2)^{n_j - 1}(\sigma^2 + n_j\sigma_b^2)$ and

$$V_j^{-1} = \sigma^{-2}(I_{n_j \times n_j} - [\sigma_b^2/(\sigma^2 + n_j\sigma_b^2)]\mathbf{1}_{n_j}\mathbf{1}'_{n_j}) \tag{4.3}$$

(V_j^{-1} is an example of the Sherman-Morrison-Woodbury formula, see eg Section 2.7 of Press *et. al.*, 1992, or Thisted, 1988, p. 117.) (Explicit formulas may not exist for more general random

effects structures.) Thus $l_{obs}(\theta)$ can be written

$$l_{obs}(\theta) = \frac{1}{2}\sum_j \left(-(n_j - 1)\log(\sigma^2) - \log(\sigma^2 + n_j\sigma_b^2) - \frac{1}{\sigma^2}\sum_i (Y_{ij} - x'_{ij}\beta)^2 + \frac{\sigma_b^2}{\sigma^2(\sigma^2 + n_j\sigma_b^2)}[\sum_i (Y_{ij} - x'_{ij}\beta)]^2 \right). \quad (4.4)$$

As in Example 1, it is not difficult to directly maximize $l_{obs}(\theta)$, although again this requires an iterative search algorithm.

The random variables b_j are a device used to induce a correlation structure for the responses, and would generally not be observable quantities. However, if their values were known, the analysis would be much simpler. Conditional on the b_j, the Y_{ij} are independent normal random variables with means $x'_{ij}\beta$ and common variance σ^2. Letting $n = \sum_j n_j$, the log likelihood for this augmented data is

$$l_{comp}(\theta) = -n\log(\sigma^2)/2 - \sum_{i,j}(Y_{ij} - x'_{ij}\beta - b_j)^2/(2\sigma^2) - N\log(\sigma_b^2)/2 - \sum_j b_j^2/(2\sigma_b^2), \quad (4.5)$$

which is maximized by

$$\hat{\beta}_c = (\sum_j X'_j X_j)^{-1} \sum_j X'_j (Y_j - b_j \mathbf{1}_{n_j}),$$

$\hat{\sigma}_c^2 = \sum_{i,j}(Y_{ij} - x'_{ij}\hat{\beta} - b_j)^2/n$ and $\hat{\sigma}_{bc}^2 = \sum_j b_j^2/N$. Here augmenting the observed data with additional data leads to simple closed form estimators. For purposes of the terminology of the EM algorithm, the 'complete data' consist of the observed data augmented by $(b_1, \ldots, b_N)$. As in Example 1, the EM algorithm will give a simple method for calculating the MLEs for the observed data by repeatedly calculating modified versions of the complete data estimators. □

In general, suppose the observed data Y_{obs} can be augmented with additional information Y_{mis} (the missing data) to give 'complete' data $Y_{comp} = (Y_{obs}, Y_{mis})$. Let $f_{obs}(Y_{obs}; \theta)$ be the observed data likelihood, and $f_{comp}(Y_{obs}, Y_{mis}; \theta)$ be the complete data likelihood. Roughly,

$$f_{obs}(y_{obs}; \theta) = \int f_{comp}(y_{obs}, y_{mis}; \theta)\, dy_{mis},$$

although it is not always easy to give a precise definition to this integral in particular settings. In Example 1, Y_{mis} consists of the values of the T_i for the censored observations, and in Example 2 $Y_{mis} = (b_1, \ldots, b_N)$.

The EM algorithm consists of repeatedly applying two steps. In the first step, called the E-step, the expectation of the complete data log likelihood conditional on the observed data is computed at the parameter values from the previous iteration (or the initial starting values for the first iteration). That is, the E-step computes

$$Q(\theta|\theta_0) = \mathrm{E}_{\theta_0}[\log\{f_{comp}(Y_{obs}, Y_{mis}; \theta)\}|Y_{obs}],$$

where θ_0 denotes the parameter values from the previous iteration. Note that θ_0 is only used in computing the expectation; it is not substituted for θ in the complete data log likelihood, which should be viewed as a function of θ.

The second step of the EM algorithm, called the M-step, is to maximize $Q(\theta|\theta_0)$ as a function of θ. The maximizing value of θ then replaces the old θ_0, and the steps are repeated, until convergence.

Example 4.1 (continued). From (4.1),

$$\begin{aligned} Q(\theta|\theta_0) &= \mathrm{E}(l_{comp}(\theta)|Y_{obs}) \\ &= -n\log(\sigma) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}\mathrm{E}[(T_i-\beta' z_i)^2|(Y_i,\delta_i,z_i)] \\ &= -n\log(\sigma) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(w_i^* - 2\beta' z_i y_i^* + [\beta' z_i]^2), \end{aligned} \tag{4.6}$$

where

$$y_i^* = \delta_i Y_i + (1-\delta_i)\mathrm{E}_{\theta_0}[T_i|T_i > Y_i, z_i]$$

and

$$w_i^* = \delta_i Y_i^2 + (1-\delta_i)\mathrm{E}_{\theta_0}[T_i^2|T_i > Y_i, z_i].$$

From $\phi(z) = (2\pi)^{-1/2}\exp(-z^2/2)$, it is easily verified that $\phi'(z) = -z\phi(z)$, and hence that $\int_a^\infty z\phi(z)\,dz = \phi(a)$ and $\int_a^\infty z^2\phi(z)\,dz = a\phi(a) + 1 - \Phi(a)$ (using integration by parts for the last expression). Given θ_0, $T_i \sim \mathrm{N}(\beta_0' z_i, \sigma_0^2)$, so setting $e_{i0} = (Y_i - \beta_0' z_i)/\sigma_0$,

$$\begin{aligned} \mathrm{E}_{\theta_0}[T_i|T_i > Y_i, z_i] &= \int_{Y_i}^\infty \frac{u}{\sigma_0}\phi\left(\frac{u-\beta_0' z_i}{\sigma_0}\right) du/[1-\Phi(e_{i0})] \\ &= \int_{e_{i0}}^\infty (\sigma_0 u + \beta_0' z_i)\phi(u)\,du/[1-\Phi(e_{i0})] \\ &= \beta_0' z_i + \sigma_0\phi(e_{i0})/[1-\Phi(e_{i0})]. \end{aligned}$$

Similarly,

$$\mathrm{E}_{\theta_0}[T_i^2|T_i > Y_i, z_i] = \sigma_0^2 + (\beta_0' z_i)^2 + \sigma_0(Y_i + \beta_0' z_i)\phi(e_{i0})/[1-\Phi(e_{i0})].$$

The E-step consists of computing these conditional expectations, using them to obtain y_i^* and w_i^*, and substituting these expressions into (4.6).

The M-step consists of finding the parameter values that maximize $Q(\theta|\theta_0)$. It is easily seen that β is maximized by the ordinary least squares estimate with responses y_i^*; that is

$$\hat\beta = (Z'Z)^{-1}Z'(y_1^*, \ldots, y_n^*)'.$$

Then

$$\hat\sigma^2 = \sum_{i=1}^{n}(w_i^* - 2\hat\beta' z_i y_i^* + [\hat\beta' z_i]^2)/n.$$

Note that $\hat\sigma^2$ differs from $\sum_i (y_i^* - \hat\beta' z_i)^2/n$ by $\sum_i (w_i^* - y_i^{*2})/n$. Schmee and Hahn (1979) omitted this correction, which was pointed out by Aitkin (1981).

For this example, it is straightforward to implement the EM algorithm in Splus. The updates for $\hat\beta$ could be calculated using a linear regression function such as `lm()` with responses y_i^*, but that would require a lot of unnecessary repeated calculations. With a little extra work a more efficient version can be given, as follows (although the version given here only works for a single covariate).

```
> ## generate data
> .Random.seed
 [1] 41 58 29 42 54  1 52 44 55 36 18  0
> z <- runif(100)
> y <- rnorm(100)+z-2
> ct <- runif(100)*3-2.5
> failind <- ifelse(ct<y,0,1)
> y <- pmin(y,ct)
> table(failind)
  0  1
 43 57
>
> zbar <- mean(z)
> zc <- z-zbar
> z2 <- sum(zc^2)
> ## EM
> b <- c(0,0,1); bold <- c(-9,-9,-9)
> xb <- b[1]+b[2]*z
> while(sum((bold-b)^2)/sum(b^2) > .00001^2) {
+ # E step
+   e0 <- (y-xb)/b[3]
+   t1 <- b[3]*dnorm(e0)/(1-pnorm(e0))
+   ystar <- failind*y+(1-failind)*(xb+t1)
+   wstar <- failind*y*y+(1-failind)*(b[3]^2+xb^2+t1*(y+xb))
+ # M step
+   bold <- b
+   b[2] <- sum(ystar*zc)/z2
+   b[1] <- mean(ystar)-b[2]*zbar
+   xb <- b[1]+b[2]*z
+   b[3] <- sqrt(sum(wstar-2*ystar*xb+xb^2)/length(y))
+   print(b)
+ }
[1] -2.005234  1.681141  1.297397
[1] -2.289998  1.831539  1.128622
[1] -2.340482  1.749140  1.014409
[1] -2.3564801  1.6703821  0.9476251
[1] -2.3634256  1.6177400  0.9094215
[1] -2.3668774  1.5857778  0.8878729
[1] -2.3686839  1.5671733  0.8758312
[1] -2.369650  1.556585  0.869139
[1] -2.370173  1.550636  0.865431
[1] -2.3704575  1.5473184  0.8633798
[1] -2.3706136  1.5454764  0.8622462
[1] -2.370699  1.544456  0.861620
[1] -2.3707467  1.5438919  0.8612742
[1] -2.3707727  1.5435801  0.8610832
```

```
[1] -2.3707871  1.5434078  0.8609777
[1] -2.3707950  1.5433127  0.8609195
[1] -2.3707994  1.5432601  0.8608874
[1] -2.3708018  1.5432311  0.8608696
[1] -2.3708031  1.5432151  0.8608598
>
```

Note that the rate of convergence as the algorithm approaches the solution is quite slow. It turns out that the rate of convergence is slower when there is more missing data, faster when there is less.

Of course, in this example it is also easy to directly maximize the likelihood in Splus (see also the `survreg()` function).

```
> # function to evaluate -log(likelihood) directly
> assign('i',0,0) #to count # calls to ff
NULL
> ff <- function(b) {
+   assign('i',i+1,0)
+   e0 <- (y-b[1]-b[2]*z)/abs(b[3])
+   sum(ifelse(failind == 1, log(abs(b[3]))+e0^2/2,-log(1-pnorm(e0))))
+ #  sum(ifelse(failind == 1, log(b[3]^2)/2+e0^2/2,-log(1-pnorm(e0))))
+ }
> out <- nlmin(ff,c(0,0,1))
> out
$x:
[1] -2.3708047  1.5431954  0.8608478

$converged:
[1] T

$conv.type:
[1] "relative function convergence"

> i
[1] 71
> options(digits=12)
> ff(b)
[1] 43.6168197992
> ff(out$x)
[1] 43.6168197851
```

`nlmin()` uses a BFGS-type algorithm, with the gradient approximated by numerical differences, similar to `nlminb()` when a gradient function is not included in the call. However, `nlmin()` does more of the work inside a C routine, and can execute more quickly than `nlminb()` with this option. `nlmin()` only uses a finite difference gradient, though. In the example above, the finite

difference gradient approximations and line searches together required a total of 71 function evaluations.

In this problem either approach is easy to program and reasonably fast to execute, and both gave identical results to the accuracy used. □

Example 4.2 (continued). Let

$$b_j^* = \mathrm{E}_{\theta_0}(b_j|Y_j) \quad \text{and} \quad c_j^* = \mathrm{Var}_{\theta_0}(b_j|Y_j).$$

From (4.5),

$$\begin{aligned} Q(\theta|\theta_0) &= \mathrm{E}_{\theta_0}(l_{comp}(\theta)|Y_{obs}) \\ &= \mathrm{E}_{\theta_0}\{-n\log(\sigma^2)/2 - \sum_{i,j}[(Y_{ij} - x'_{ij}\beta)^2 - 2b_j(Y_{ij} - x'_{ij}\beta) + b_j^2]/(2\sigma^2) - N\log(\sigma_b^2)/2 \\ &\quad - \sum_j b_j^2/(2\sigma_b^2) \,|\, Y_{obs}\} \\ &= -n\log(\sigma^2)/2 - \sum_{i,j}[(Y_{ij} - x'_{ij}\beta)^2 - 2b_j^*(Y_{ij} - x'_{ij}\beta) + b_j^{*2} + c_j^*]/(2\sigma^2) - N\log(\sigma_b^2)/2 \\ &\quad - \sum_j (b_j^{*2} + c_j^*)/(2\sigma_b^2) \\ &= -\frac{n}{2}\log(\sigma^2) - \sum_{i,j}\frac{(Y_{ij} - x'_{ij}\beta - b_j^*)^2 + c_j^*}{2\sigma^2} - \frac{N}{2}\log(\sigma_b^2) - \sum_j \frac{b_j^{*2} + c_j^*}{2\sigma_b^2}, \end{aligned} \tag{4.7}$$

so the E-step consists of computing b_j^* and c_j^* (note: the expectations in b_j^* and c_j^* are computed at the current θ_0, while the other parameters in (4.7) are left as the free parameters θ).

The M-step consists of maximizing (4.7). The only difference in the structure of (4.7) and the complete data likelihood is the addition of the c_j^* terms, which only affect the variance estimates. Thus the maximizers are

$$\hat\beta = (\sum_j X_j'X_j)^{-1}\sum_j X_j'(Y_j - b_j^*\mathbf{1}_{n_j}),$$

$$\hat\sigma^2 = \sum_j [n_j c_j^* + \sum_i (Y_{ij} - x'_{ij}\hat\beta - b_j^*)^2]/n \quad \text{and} \quad \hat\sigma_b^2 = \sum_j (b_j^{*2} + c_j^*)/N. \tag{4.8}$$

The EM algorithm consists of calculating b_j^* and c_j^* at the current parameter values, and then updating the estimates using the formulas above, repeating until convergence.

To do this, formulas are needed for b_j^* and c_j^*. From (4.2), $(Y_j', b_j)'$ has a multivariate normal distribution with mean

$$\begin{pmatrix} X_j\beta \\ 0 \end{pmatrix}, \quad \text{and covariance matrix} \quad \begin{pmatrix} V_j & \sigma_b^2\mathbf{1}_{n_j} \\ \sigma_b^2\mathbf{1}'_{n_j} & \sigma_b^2 \end{pmatrix}.$$

Thus from standard formulas and (4.3), the conditional distribution of $b_j|Y_j$ is normal with mean

$$b_j^* = 0 + \sigma_b^2\mathbf{1}'_{n_j}V_j^{-1}(Y_j - X_j\beta) = \frac{\sigma_b^2}{\sigma^2 + n_j\sigma_b^2}\mathbf{1}'_{n_j}(Y_j - X_j\beta) \tag{4.9}$$

and variance

$$c_j^* = \sigma_b^2 - \sigma_b^4 \mathbf{1}'_{n_j} V_j^{-1} \mathbf{1}_{n_j} = \frac{\sigma_b^2 \sigma^2}{\sigma^2 + n_j \sigma_b^2}. \tag{4.10}$$

As in the previous example, the EM algorithm is straightforward to implement. Instead of using `lm()` to calculate $\hat{\beta}$, the QR decomposition of the matrix $X = (X_1' \cdots X_N')'$ is computed once, and the least squares estimates of β obtained from the QR decomposition in each iteration using the updated response vector. The data used are in the file `exmix.dat`. The commands used to generate the data are given below.

```
> # generate data
> options(digits=12)
> n <- 100
> ng <- 10
> x <- rnorm(n)
> g <- sample(ng,n,replace=T)
> y <- 10+x+rnorm(n)+rnorm(ng)[g]
>
> nj <- table(g)
> X <- cbind(1,x)
> Xqr <- qr(X)
>
> #fit model
> emup <- function(beta0,sig0,sigb0){
+ # E step
+   bstar <- tapply(y-X%*%beta0,g,sum)/(nj+sig0/sigb0)
+   cj <- sigb0*sig0/(sig0+nj*sigb0)
+ # M step
+   ystar <- y-bstar[g]
+   beta <- qr.coef(Xqr,ystar)
+   sig <- (sum((ystar-X %*% beta)^2)+sum(nj*cj))/n
+   sigb <- sum(bstar^2+cj)/length(nj)
+   list(beta=beta,sig=sig,sigb=sigb)
+ }
> # initial values
> beta <- c(mean(y),rep(0,ncol(X)-1))
> sig <- 1
> sigb <- 1
> err <- 10
> i <- 0
> while(err>1.e-5){
+   i <- i+1
+   u <- emup(beta,sig,sigb)
+   err <- sqrt(sum(c(u$beta-beta,u$sig-sig,u$sigb-sigb)^2))
+   beta <- u$beta
+   sig <- u$sig
```

```
+   sigb <- u$sigb
+   print(c(i,err,beta))
+ }
                                                  x
 1 0.949843894712 10.0919147446 0.90402815552
                                                   x
 2 0.353776667546 10.0674145089 1.05837358894
                                                    x
 3 0.0654376874088 10.0555500833 1.08498905088
                                                    x
 4 0.0165017986074 10.0466468929 1.08956839248
                                                    x
 5 0.00842823437663 10.0390983495 1.0904090053
                                                     x
 6 0.00668542018088 10.0325493911 1.09060278779
                                                   x
 7 0.00574170211 10.0268462393 1.09067690898
                                                     x
 8 0.00498575993387 10.0218774471 1.09072401586
                                                    x
 9 0.00433774622566 10.017548805 1.09076149343
                                                    x
 10 0.003775656728 10.0137783351 1.09079322052
                                                      x
 11 0.00328688660694 10.0104944152 1.09082049456
                                                      x
 12 0.00286159731258 10.0076344836 1.09084403948
                                                      x
 13 0.00249145705788 10.0051439343 1.09086439754
                                                     x
 14 0.00216927362678 10.0029751377 1.0908820165
                                                      x
 15 0.00188880890312 10.0010865707 1.09089727595
                                                      x
 16 0.00164464438034 9.99944204767 1.09091050022
                                                      x
 17 0.00143207040279 9.99801004306 1.09092196718
                                                      x
 18 0.00124699177004 9.99676309675 1.09093191536
                                                      x
 19 0.00108584650998 9.99567729296 1.09094054981
                                                       x
 20 0.000945535732669 9.99473180481 1.09094804702
                                                      x
 21 0.00082336293512 9.99390849645 1.09095455909
                                                       x
```

```
22 0.000716981412176 9.99319157601 1.09096021726
                                                 x
23 0.00062434864376 9.99256729296 1.09096513487
                                                  x
24 0.000543686701138 9.99202367437 1.09096940989
                                                  x
25 0.000473447855931 9.99155029516 1.09097312709
                                                  x
26 0.000412284693267 9.99113807797 1.09097635987
                                                  x
27 0.000359024128894 9.99077911893 1.09097917184
                                                  x
28 0.000312644813103 9.99046653587 1.09098161812
                                               x
29 0.0002722574752 9.9901943362 1.09098374654
                                                  x
30 0.000237087822272 9.98995730189 1.09098559862
                                                  x
31 0.000206461658094 9.98975088934 1.09098721039
                                                  x
32 0.000179791932159 9.98957114211 1.09098861315
                                                 x
33 0.00015656746734 9.98941461508 1.09098983409
                                                 x
34 0.00013634314783 9.98927830832 1.09099089686
                                                  x
35 0.000118731377489 9.98915960954 1.09099182199
                                                  x
36 0.000103394643786 9.98905624397 1.09099262735
                                                  x
37 9.00390438826e-05 9.98896623078 1.09099332848
                                                  x
38 7.84086480999e-05 9.98888784506 1.09099393889
                                                  x
39 6.82805924762e-05 9.98881958475 1.09099447034
                                                 x
40 5.9460805667e-05 9.98876014184 1.09099493305
                                                  x
41 5.17802885016e-05 9.98870837732 1.09099533592
                                                  x
42 4.50918743298e-05 9.98866329932 1.09099568671
                                                 x
43 3.92674081483e-05 9.9886240441 1.09099599214
                                                x
44 3.41952902929e-05 9.9885898595 1.0909962581
                                                  x
```

```
 45 2.97783375547e-05 9.98856009054 1.09099648968
                                                 x
 46 2.59319207207e-05 9.98853416683 1.09099669132
                                                 x
 47 2.25823426953e-05 9.98851159167 1.09099686691
                                                 x
 48 1.96654262847e-05 9.98849193253 1.09099701981
                                                 x
 49 1.71252843249e-05 9.98847481274 1.09099715295
                                                 x
 50 1.49132488106e-05 9.98845990429 1.09099726889
                                                 x
 51 1.29869383204e-05 9.98844692155 1.09099736985
                                                 x
 52 1.13094459137e-05 9.98843561576 1.09099745777
                                                 x
 53 9.84863197933e-06 9.98842577032 1.09099753432
> c(beta,sig,sigb)
                            x
 9.98842577032 1.09099753432 1.11164062315 0.742579099161
```

Again the rate of convergence is quite slow after the first few iterations. It is also straightforward to directly maximize the observed data likelihood (4.4) here. Note that to avoid range restrictions, the parameters $\log(\sigma^2)$ and $\log(\sigma_b^2)$ are used in the argument to the function `fclik()` below.

```
> #direct maximization of observed data likelihood
> fclik <- function(b) {
+   assign('i',i+1,0)
+   beta <- b[1:2]
+   sig <- exp(b[3])
+   sigb <- exp(b[4])
+   r <- y-X %*% beta
+   rg2 <- tapply(r,g,sum)^2
+   (sum(nj-1)*b[3]+sum(log(sig+nj*sigb))+sum(r*r)/sig-sum(rg2*sigb/
+     (sig+nj*sigb))/sig)/2
+ }
> assign('i',0,0)
NULL
> z <- nlmin(fclik,c(mean(y),rep(0,3)))
> z
$x:
[1]  9.988359604686  1.090998071265  0.105836732789 -0.297623535749

$converged:
[1] T
```

```
$conv.type:
[1] "relative function convergence"

> exp((z$x)[3:4])
[1] 1.111640367267 0.742580842274
> i
[1] 71
```

As in the previous example, this required exactly 71 evaluations of the likelihood, and gave nearly identical results to the EM algorithm. □

4.2 Some General Properties

An important relationship in the EM algorithm is that the score function of the observed data likelihood is equal to the expectation of the score function of the complete data likelihood, conditional on the observed data. A heuristic argument for this result is as follows.

$$\begin{aligned}
\frac{\partial}{\partial\theta} l_{obs}(\theta) &= \frac{\partial}{\partial\theta} \log\left(\int f_{comp}(y_{obs}, y_{mis};\theta)\, dy_{mis}\right) \\
&= \int \partial f_{comp}(y_{obs}, y_{mis};\theta)/\partial\theta\, dy_{mis} / f_{obs}(y_{obs};\theta) \\
&= \int \frac{\partial f_{comp}(y_{obs}, y_{mis};\theta)/\partial\theta}{f_{comp}(y_{obs}, y_{mis};\theta)} \frac{f_{comp}(y_{obs}, y_{mis};\theta)}{f_{obs}(y_{obs};\theta)}\, dy_{mis} \\
&= \int (\partial l_{comp}(\theta)/\partial\theta) f(y_{obs}, y_{mis}|y_{obs};\theta)\, dy_{mis} \\
&= \mathrm{E}_\theta(\partial l_{comp}(\theta)/\partial\theta|Y_{obs}). \qquad (4.11)
\end{aligned}$$

Thus in many applications it is no harder to compute the score vector for the likelihood of the observed data than to compute $Q(\theta|\theta_0)$, since they will involve similar expectations.

Note also that

$$\mathrm{E}_\theta(\partial l_{comp}(\theta)/\partial\theta|Y_{obs}) = \frac{\partial}{\partial\theta} Q(\theta|\theta_0)\Big|_{\theta_0=\theta}. \qquad (4.12)$$

Using these relationships, it can be shown that if the EM algorithm converges, it converges to a solution to the score equations for the likelihood of the observed data. To see this, first note that if the EM algorithm converges to a value $\hat\theta$, then $Q(\theta|\hat\theta)$ is maximized by $\hat\theta$ (that is, convergence means that applying additional iterations continues to return the same value). Thus

$$\frac{\partial}{\partial\theta} Q(\theta|\hat\theta)\Big|_{\theta=\hat\theta} = 0.$$

But from (4.12) and (4.11) this implies

$$\frac{\partial}{\partial\theta} l_{obs}(\hat\theta) = 0,$$

so a stationary point of the EM algorithm is a solution to the observed data score equations.

It is also possible to show that at each iteration, if θ_1 is such that $Q(\theta_1|\theta_0) > Q(\theta_0|\theta_0)$, then $l_{obs}(\theta_1) > l_{obs}(\theta_0)$ as well. Since the EM updates are chosen to maximize $Q(\theta|\theta_0)$, this means that

each iteration of the EM algorithm gives a value of θ which increases the value of the likelihood of the observed data from the previous iteration. (Note that it is not necessary to maximize $Q(\theta|\theta_0)$ at each iteration to achieve this property; it is sufficient to just find a better point.) A heuristic argument for this property follows. Note that

$$l_{obs}(\theta) = Q(\theta|\theta_0) - H(\theta|\theta_0), \tag{4.13}$$

where

$$H(\theta|\theta_0) = \mathrm{E}_{\theta_0}\{\log[f_{comp}(Y_{obs}, Y_{mis}; \theta)/f_{obs}(Y_{obs}; \theta)] \,|\, Y_{obs}\}.$$

By Jensen's inequality, since $-\log(\cdot)$ is a convex function,

$$\begin{aligned}
-H(\theta|\theta_0) + H(\theta_0|\theta_0) &= \mathrm{E}_{\theta_0}\left(-\log\left[\frac{f_{comp}(Y_{obs}, Y_{mis}; \theta) f_{obs}(Y_{obs}; \theta_0)}{f_{obs}(Y_{obs}; \theta) f_{comp}(Y_{obs}, Y_{mis}; \theta_0)} \right] \middle| Y_{obs} \right) \\
&\geq -\log\left(\mathrm{E}_{\theta_0}\left[\frac{f_{comp}(Y_{obs}, Y_{mis}; \theta) f_{obs}(Y_{obs}; \theta_0)}{f_{obs}(Y_{obs}; \theta) f_{comp}(Y_{obs}, Y_{mis}; \theta_0)} \middle| Y_{obs} \right] \right) \\
&= -\log \int \left[\frac{f_{comp}(y_{obs}, y_{mis}; \theta) f_{obs}(y_{obs}; \theta_0)}{f_{obs}(y_{obs}; \theta) f_{comp}(y_{obs}, y_{mis}; \theta_0)} \right] \frac{f_{comp}(y_{obs}, y_{mis}; \theta_0)}{f_{obs}(y_{obs}; \theta_0)} \, dy_{mis} \\
&= -\log\left(\int f_{comp}(y_{obs}, y_{mis}; \theta) \, dy_{mis} / f_{obs}(y_{obs}; \theta) \right) \\
&= -\log\left(f_{obs}(y_{obs}; \theta) / f_{obs}(y_{obs}; \theta) \right) \\
&= 0,
\end{aligned}$$

so

$$H(\theta|\theta_0) \leq H(\theta_0|\theta_0).$$

Thus if θ_1 is such that $Q(\theta_1|\theta_0) \geq Q(\theta_0|\theta_0)$, then from (4.13),

$$l_{obs}(\theta_1) - l_{obs}(\theta_0) = [Q(\theta_1|\theta_0) - Q(\theta_0|\theta_0)] + [H(\theta_0|\theta_0) - H(\theta_1|\theta_0)] \geq 0,$$

since both terms in [ ] are ≥ 0, so

$$l_{obs}(\theta_1) \geq l_{obs}(\theta_0).$$

Since the EM algorithm moves to a better point at each iteration, and if it converges it converges to a solution of the observed data score equations, it is certainly believable that the EM algorithm generally will converge to such a solution. However, this line of reasoning is not sufficient to prove this result. Dempster, Laird and Rubin (1977) give more details, but it turns out their argument is also not sufficient. Wu (1983) gives a correct proof for convergence of the EM algorithm to a solution to the observed data score equations. (Such a solution need not be the global MLE, though.) Lange (1999, Chapter 13) also derives some convergence results for the EM algorithm.

Equation (4.13) has another interesting consequence. Differentiating twice gives

$$\begin{aligned}
-\frac{\partial^2 l_{obs}(\theta)}{\partial\theta\partial\theta'} &= -\frac{\partial^2 Q(\theta|\theta_0)}{\partial\theta\partial\theta'} + \frac{\partial^2 H(\theta|\theta_0)}{\partial\theta\partial\theta'} \\
&= \mathrm{E}_{\theta_0}\left(-\frac{\partial^2 l_{comp}(\theta)}{\partial\theta\partial\theta'} \middle| Y_{obs} \right) - \mathrm{E}_{\theta_0}\left(-\frac{\partial^2}{\partial\theta\partial\theta'} \log\left[\frac{f_{comp}(Y_{obs}, Y_{mis}; \theta)}{f_{obs}(Y_{obs}; \theta)} \right] \middle| Y_{obs} \right).
\end{aligned} \tag{4.14}$$

Setting both θ and θ_0 to the true value of the parameters, the left hand side is the observed information matrix in the sample and the first term on the right is the conditional expected value of the complete data information given the observed data. In the second term on the right, f_{comp}/f_{obs} is essentially the conditional likelihood of the missing data given the observed data, so this term can be thought of as the information in the missing data, conditional on the observed data. Thus (4.14) gives the interesting relationship that the information in the observed data equals the information in the complete data minus the information in the missing data. This was called the missing information principle by Orchard and Woodbury (1972).

4.3 An Accelerated EM Algorithm

There have been a variety of proposals for speeding up the convergence of the EM algorithm. One simple approach due to Jamshidian and Jennrich (1997) will be discussed below. That paper discusses several other approaches, and gives a number of additional references on this topic. Some of the discussion in Meng and van Dyk (1997) is also relevant.

Jamshidian and Jennrich (1997) propose two accelerated versions of the EM algorithm based on quasi-Newton methods for solving equations and minimizing functions. Their second proposal is based on the BFGS minimization algorithm, and can be very fast computationally, but requires evaluating the observed data likelihood and score. Since part of the point of using EM is to avoid those computations, and because if the observed data likelihood and score can be easily computed then the BFGS algorithm (and several other minimization algorithms) can be applied directly, only Jamshidian and Jennrich's first proposal will be discussed here. This approach is based on Broyden's method for solving systems of nonlinear equations, which was briefly mentioned in Section 3.6, although no details were given.

Details of Broyden's method for solving nonlinear equations are discussed in Section 9.7 of Press *et. al.* (1992). This method builds up an approximation to the Jacobian of the system of equations, much the same way the BFGS algorithm builds up an approximation to the Hessian in minimization problems. For solving the system of equations $G(x) = 0$, Broyden's method uses search directions of the form $-A_i G(x_i)$. In Newton's method A_i is the inverse of the Jacobian of $G(x)$, while in Broyden's method an approximation is used. As with the BFGS minimization algorithm, the updates to A_i are chosen to satisfy a secant condition, and there is both a version for updating the approximation to the Jacobian and for updating its inverse. The inverse Jacobian updating formula is

$$A_{i+1} = A_i + (s' A_i h)^{-1}(s - A_i h)(s' A_i) \tag{4.15}$$

where $s = x_{i+1} - x_i$ and $h = G(x_{i+1}) - G(x_i)$.

Let $M(\theta_0)$ be the value of θ given by one EM update from the current value θ_0; that is, the value maximizing $Q(\theta|\theta_0)$. Then $M(\theta)$ is the mapping defined by the EM algorithm updates. The objective of the EM algorithm can be expressed as finding a solution to $g(\theta) = M(\theta) - \theta = 0$, so here $g(\theta)$ takes the role of $G(x)$ in the above discussion. In Jamshidian and Jennrich's approach, the Broyden update step is applied to this function, as follows. First initialize by setting $\theta = \theta_0$, $g_0 = g(\theta_0)$, and $A = -I$ (the negative of an identity matrix). Then

1. Compute $s = -Ag_0$ and $h = g(\theta + s) - g_0$.

2. Update A using (4.15) and s and h from step 1.

3. Replace θ by $\theta + s$ and g_0 by $g_0 + h$, and return to step 1, repeating until convergence.

Note that each cycle through this algorithm computes one standard EM algorithm update ($M(\theta)$), and all other calculations are straightforward. Also note that at the initial step, with $A = -I$, the update is just a standard EM update. Thus this algorithm is easy to implement. There are two potential problems. One is that this modified version is not guaranteed to always move to a better point, unless some form of testing and backtracking is incorporated. This would be a substantial complication that would make this algorithm less appealing. In practice, running a few steps of the standard EM algorithm first often moves close enough to a solution that the above algorithm will converge without modification. The second problem is that the approximation A to the inverse Jacobian may deteriorate, and in some applications it could be necessary to occasionally reset $A = -I$.

The calculations for this accelerated EM algorithm are illustrated in the following example.

Example 4.2 (continued).
The algorithm above is easily implemented for the mixed effects linear model. Note that in the following `emup` is the same as before, and `emqn1` implements one cycle of the accelerated algorithm. Also, note that only the call to `emup` and the initialization of `theta` is specific to this particular application.

```
> emqn1 <- function(theta,A,gg) {
+   # gg=M(theta)-theta on input
+   # components of theta are in the order (beta, sig, sigb)
+   deltheta <- -A %*% gg #=s above
+   thet2 <- theta+deltheta
+   thetaem <- emup(thet2[1:np],thet2[np1],thet2[np2])
+   delgg <- unlist(thetaem)-thet2-gg #=h above
+   adgg <- A %*% delgg
+   A <- A+outer(c(deltheta-adgg)/sum(deltheta*adgg),c(t(A)%*%deltheta))
+   list(thet2,A,gg+delgg)
+ }
> #initialize
> theta <- c(mean(y),rep(0,ncol(X)-1),1,1)
> np <- ncol(X)
> np1 <- np+1
> np2 <- np+2
> A <- -diag(length(theta))
> gg <- unlist(emup(theta[1:np],theta[np1],theta[np2]))-theta
> err <- 10
> i <- 0
> # accelerated iteration loop
> while(err>1.e-5) {
+   i <- i+1
+   u <- emqn1(theta,A,gg)
```

```
+   err <- sqrt(sum((u[[1]]-theta)^2))
+   theta <- u[[1]]
+   A <- u[[2]]
+   gg <- u[[3]]
+   print(c(i,err,theta))
+ }
[1]  1.0000000  0.9498439 10.0919147  0.9040282  1.2491022  1.1119228
[1]  2.0000000  0.3875182 10.0650778  1.0730943  1.1139159  0.7916238
[1]  3.00000000  0.04831467 10.05280778  1.09117128  1.10951595  0.74875634
[1]  4.000000000  0.009988417 10.043655540  1.090790445  1.111614303
[6]  0.745371546
[1]  5.00000000  0.03158347 10.01251726  1.09072925  1.11222393  0.74012297
[1] 6.00000000 0.01262663 9.99996740 1.09093529 1.11163882 0.74136717
[1] 7.00000000 0.01280706 9.98720088 1.09101940 1.11158907 0.74238061
[1] 8.000000000 0.002238957 9.989433650 1.090994868 1.111627516 0.742540497
[1] 9.0000000000 0.0008156216 9.9886185174 1.0909977266 1.1116356038
[6] 0.7425673947
[1] 1.000000e+01 2.471474e-04 9.988372e+00 1.090998e+00 1.111640e+00
[6] 7.425798e-01
[1] 1.100000e+01 1.161950e-05 9.988360e+00 1.090998e+00 1.111640e+00
[6] 7.425807e-01
[1] 1.200000e+01 7.854155e-07 9.988359e+00 1.090998e+00 1.111640e+00
[6] 7.425807e-01
```

Note that only 12 iterations of this accelerated algorithm were required, versus 53 for the standard EM algorithm above. □

4.4 Calculating the Information

While the EM algorithm gives a simple method of calculating MLEs in many incomplete data problems, it does not automatically provide any of the other quantities needed to draw inferences on the parameters, such as standard errors and test statistics. The inverse information matrix is the usual estimator of the covariance matrix of the MLEs, so this matrix is also often of interest.

From (4.11), the observed data score vector can be obtained as the conditional expectation of the complete data score, a calculation that is often no harder than computing $Q(\theta|\theta_0)$. Differentiating the observed data score then gives the observed data information. Since differentiation can be done in symbolic mathematics programs, this would often be a feasible way to obtain the information. However, differentiating a complicated expression does introduce the potential for errors. Formulas for the first and second derivatives of the complete data likelihood are often already available. Louis (1981) gave a formula for the observed data information just in terms of conditional expectations of functions of the complete data first and second derivatives. This formula will be described next.

4.4.1 Louis' Formula

Louis' formula is easily derived by differentiating

$$l_{obs}(\theta) = \log\left(\int f_{comp}(Y_{obs}, y_{mis}; \theta)\, dy_{mis},\right)$$

twice with respect to θ, differentiating $l_{comp}(\theta) = \log[f_{comp}(Y_{obs}, Y_{mis}; \theta)]$ twice, comparing terms, and using the definition of the conditional expectation. The result is that

$$-\frac{\partial^2 l_{obs}(\theta)}{\partial\theta\partial\theta'} = \mathrm{E}_\theta\left(-\frac{\partial^2 l_{comp}(\theta)}{\partial\theta\partial\theta'}\middle| Y_{obs}\right) - \mathrm{E}_\theta\left(\frac{\partial l_{comp}(\theta)}{\partial\theta}^{\otimes 2}\middle| Y_{obs}\right) + \mathrm{E}_\theta\left(\frac{\partial l_{comp}(\theta)}{\partial\theta}\middle| Y_{obs}\right)^{\otimes 2}, \quad (4.16)$$

where in general for a column vector a, $a^{\otimes 2} = aa'$. Thus the observed data information can be computed in terms of expectations of derivatives of the complete data likelihood. If the complete data have a distribution in a regular exponential family, then the log likelihood is linear in the sufficient statistics of the exponential family, and it turns out the conditional expectations needed for the first and third terms on right hand side of (4.16) involve exactly the same expectations required for computing $Q(\theta|\theta_0)$. (Also note that the third term is 0 when evaluated at the MLE, from (4.11)). However, the middle term involves higher order moments of the sufficient statistics.

Comparing (4.14) and (4.16), it follows that the missing data information is

$$-\frac{\partial^2 H(\theta|\theta_0)}{\partial\theta\partial\theta'} = \mathrm{E}_\theta\left(\frac{\partial l_{comp}(\theta)}{\partial\theta}^{\otimes 2}\middle| Y_{obs}\right) - \mathrm{E}_\theta\left(\frac{\partial l_{comp}(\theta)}{\partial\theta}\middle| Y_{obs}\right)^{\otimes 2}. \quad (4.17)$$

4.4.2 The SEM Algorithm

The methods described above for obtaining the observed data information all involve extra analytical calculations beyond those needed to compute the MLEs. Meng and Rubin (1993) proposed a supplemented EM algorithm (SEM) that provides a numerical approximation to the information as a bi-product of the EM calculations themselves. This approach is based on the derivative of the mapping defined by the iterations of the EM algorithm. Again define $M(\theta_0)$ to be the value of θ that maximizes $Q(\theta|\theta_0)$. Then one iteration of the EM algorithm from θ_0 gives the value $\theta_1 = M(\theta_0)$. The derivative of this mapping is the Jacobian matrix of $M(\theta)$. Dempster, Laird and Rubin (1977) observed that this Jacobian satisfies

$$\frac{\partial}{\partial\theta}M(\hat\theta)' = -\frac{\partial^2 H(\theta|\hat\theta)}{\partial\theta\partial\theta'}\bigg|_{\theta=\hat\theta}\left(-\frac{\partial^2 Q(\theta|\hat\theta)}{\partial\theta\partial\theta'}\bigg|_{\theta=\hat\theta}\right)^{-1}, \quad (4.18)$$

the missing data information times the inverse of the complete data information. Defining $I_{obs} = -\partial^2 l_{obs}(\theta)/\partial\theta\partial\theta'$, $I_{comp} = \mathrm{E}_\theta\left(-\partial^2 l_{comp}(\theta)/\partial\theta\partial\theta'\middle| Y_{obs}\right)$ and $I_{mis} = -\partial^2 H(\theta|\theta_0)/\partial\theta\partial\theta'$, the observed, complete and missing data information (in the sense used previously), and writing I for the identity matrix of order equal to the number of components in θ, it follows from (4.14) and (4.18) that

$$\begin{aligned} I_{obs} &= I_{comp} - I_{mis} \\ &= (I - I_{mis}I_{comp}^{-1})I_{comp} \\ &= [I - \partial M(\theta)/\partial\theta]' I_{comp}, \end{aligned}$$

when θ is evaluated at $\hat{\theta}$. Thus if $\partial M(\hat{\theta})'/\partial\theta$ can be estimated and I_{comp} computed, then the observed information can be estimated without further computation.

To see why (4.18) should be true, use $\partial/\partial\theta_1$ to denote derivatives with respect to the first vector argument in $Q(\cdot|\cdot)$, and $\partial/\partial\theta_2$ to denote derivatives with respect to the second vector argument. Since $M(\theta_0)$ maximizes $Q(\theta|\theta_0)$, generally $(\partial/\partial\theta_1)Q(M(\theta)|\theta) = 0$. Taking a Taylor series in both arguments of $Q(\theta_1|\theta_2)$ about $(\theta_1, \theta_2) = (\hat{\theta}, \hat{\theta})$ (where $\hat{\theta}$ is the MLE), gives that

$$0 = \frac{\partial Q(M(\theta)|\theta)}{\partial\theta_1} = \frac{\partial Q(\hat{\theta}|\hat{\theta})}{\partial\theta_1} + \frac{\partial^2 Q(\hat{\theta}|\hat{\theta})}{\partial\theta_1\theta_1'}[M(\theta) - \hat{\theta}] + \frac{\partial^2 Q(\hat{\theta}|\hat{\theta})}{\partial\theta_1\partial\theta_2'}(\theta - \hat{\theta}) + \cdots.$$

Using the fact that $M(\hat{\theta}) = \hat{\theta}$, and recalling that $\partial Q(\hat{\theta}|\hat{\theta})/\partial\theta_1 = 0$, it follows that

$$0 = \frac{\partial^2 Q(\hat{\theta}|\hat{\theta})}{\partial\theta_1\theta_1'}[M(\theta) - M(\hat{\theta})] + \frac{\partial^2 Q(\hat{\theta}|\hat{\theta})}{\partial\theta_1\partial\theta_2'}(\theta - \hat{\theta}) + \cdots.$$

Separately setting $\theta = \hat{\theta} + \epsilon e^{(j)}$ for each j, where $e^{(j)}$ is the unit vector in the jth coordinate direction, dividing by ϵ, and taking the limit as $\epsilon \to 0$, it follows that

$$0 = \frac{\partial^2 Q(\hat{\theta}|\hat{\theta})}{\partial\theta_1\theta_1'}\left(\frac{\partial M(\hat{\theta})}{\partial\theta}\right) + \frac{\partial^2 Q(\hat{\theta}|\hat{\theta})}{\partial\theta_1\partial\theta_2'}$$

Formula (4.18) then follows by showing

$$\frac{\partial^2 Q(\hat{\theta}|\hat{\theta})}{\partial\theta_1\partial\theta_2'} = -\frac{\partial^2 H(\hat{\theta}|\hat{\theta})}{\partial\theta_1\theta_1'}.$$

This follows from differentiating

$$Q(\theta_1|\theta_2) = \int l_{comp}(\theta_1)\frac{f_{comp}(Y_{obs}, y_{mis}; \theta_2)}{f_{obs}(Y_{obs}; \theta_2)}\, dy_{mis},$$

and using (4.17).

In the SEM algorithm, the derivatives in the Jacobian $\partial M(\hat{\theta})'/\partial\theta$ are approximated with numerical differences. Given the EM algorithm maximizer $\hat{\theta} = (\hat{\theta}_1, \ldots, \hat{\theta}_p)$, and some $\theta = (\theta_1, \ldots, \theta_p)'$, with $\theta_j \neq \hat{\theta}_j$ for all j, the row vector $\partial M(\hat{\theta})'/\partial\theta_j$ is approximated as follows. Set $\theta(j) = (\hat{\theta}_1, \ldots, \hat{\theta}_{j-1}, \theta_j, \hat{\theta}_{j+1}, \ldots, \hat{\theta}_p)'$. Use the EM algorithm to compute $M(\theta(j))$. Approximate $\partial M(\hat{\theta})'/\partial\theta_j$ with

$$[M(\theta(j))' - M(\hat{\theta})']/(\theta_j - \hat{\theta}_j). \tag{4.19}$$

Repeating these steps for $j = 1, \ldots, p$, then gives the full Jacobian matrix. Since at the exact solution, $M(\hat{\theta}) = \hat{\theta}$, this substitution could also be made in the previous expression. However, usually the EM algorithm is terminated a little short of the exact maximizer, so $M(\hat{\theta})$ will not be exactly equal to $\hat{\theta}$, and a small difference between $M(\hat{\theta})$ and $\hat{\theta}$ can affect the accuracy of the results (Meng and Rubin, 1993, seem to have overlooked this point).

The remaining question is how to choose the step $\theta - \hat{\theta}$ for the differences. Numerical differences as approximations to derivatives can be quite sensitive to the size of the difference used, and the appropriate step size can be different for different components of θ. Meng and Rubin (1993)

proposed starting the EM algorithm from some arbitrary point, running it for a moderate number of iterations, then using the current value for θ in (4.19). The advantage of this is that the components of $\theta - \hat{\theta}$ tend to be scaled in a way that takes into account differences in the slopes of $M(\theta)$ in the different directions. They also suggest continuing to run the EM algorithm for a number of additional iterations, recomputing the Jacobian approximation at each iteration, until the Jacobian approximation stabilizes. Of course other sequences of perturbations of $\hat{\theta}$ could be used, too.

The EM algorithm has a linear rate of convergence in the neighborhood of the maximum, unlike say Newton's method and quasi-Newton methods that have quadratic (or at least super-linear) convergence. The magnitude of the linear rate of convergence for the EM algorithm is related to the Jacobian of the algorithmic map, $\partial M(\theta)'/\partial\theta$. Assuming the eigenvalues of this matrix are all < 1, which should usually be the case, the largest eigenvalue gives the rate of convergence. Since from (4.18), $\partial M(\theta)'/\partial\theta = I_{mis}I_{comp}^{-1}$, it follows that the rate of convergence of the EM algorithm is related to the fraction of the complete data information which is missing in the observed data.

Example 4.2 (continued). Direct calculation of the observed information matrix and using the SEM algorithm will now be illustrated. First, the formulas for the observed data scores and information and the conditional expectation of the complete data information are given. From (4.5),

$$\frac{\partial l_{comp}(\theta)}{\partial\beta} = \frac{1}{\sigma^2}\sum_{i,j}(Y_{ij} - x'_{ij}\beta - b_j)x_{ij},$$

$$\frac{\partial l_{comp}(\theta)}{\partial(\sigma^2)} = -\frac{n}{2\sigma^2} + \frac{1}{2\sigma^4}\sum_{i,j}(Y_{ij} - x'_{ij}\beta - b_j)^2$$

(note this is one derivative with respect to the parameter σ^2, not two derivatives with respect to σ), and

$$\frac{\partial l_{comp}(\theta)}{\partial(\sigma_b^2)} = -\frac{N}{2\sigma_b^2} + \frac{1}{2\sigma_b^4}\sum_j b_j^2.$$

From (4.11), the observed data scores can be obtained from these by taking their expectations conditional on the observed data. Using (4.9) and (4.10), the observed data scores are thus

$$\frac{\partial l_{obs}(\theta)}{\partial\beta} = \frac{1}{\sigma^2}\sum_{i,j}(Y_{ij} - x'_{ij}\beta)x_{ij} - \frac{1}{\sigma^2}\sum_j \frac{\sigma_b^2}{\sigma^2 + n_j\sigma_b^2}\left(\sum_i(Y_{ij} - x'_{ij}\beta)\right)\left(\sum_i x_{ij}\right),$$

$$\frac{\partial l_{obs}(\theta)}{\partial(\sigma^2)} = -\frac{n-N}{2\sigma^2} - \frac{1}{2}\sum_j\left(\frac{1}{\sigma^2 + n_j\sigma_b^2} - \frac{1}{\sigma^4}\sum_i(Y_{ij} - x'_{ij}\beta)^2 + \frac{\sigma_b^2(2\sigma^2 + n_j\sigma_b^2)}{\sigma^4(\sigma^2 + n_j\sigma_b^2)^2}[\sum_i(Y_{ij} - x'_{ij}\beta)]^2\right),$$

and

$$\frac{\partial l_{obs}(\theta)}{\partial(\sigma_b^2)} = \frac{1}{2}\sum_j\left(-\frac{n_j}{\sigma^2 + n_j\sigma_b^2} + \frac{[\sum_i(Y_{ij} - x'_{ij}\beta)]^2}{(\sigma^2 + n_j\sigma_b^2)^2}\right).$$

To obtain analytical expressions for the observed data information, either the derivatives of the observed data scores can be calculated, or the complete data derivatives and expectations in

Louis' formula (4.16) can be calculated. Note that here Louis' formula would require calculating $\mathrm{E}[b_j^4|Y_j]$ (which is not that difficult, since b_j and Y_j have a joint normal distribution). Directly differentiating the observed data scores yields the following formulas:

$$\begin{aligned}
-\frac{\partial^2 l_{obs}(\theta)}{\partial\beta\partial\beta'} &= \frac{1}{\sigma^2}\sum_j\left(X_j'X_j - \frac{\sigma_b^2}{\sigma^2+n_j\sigma_b^2}(\sum_i x_{ij})(\sum_i x_{ij}')\right),\\
-\frac{\partial^2 l_{obs}(\theta)}{\partial\beta\partial(\sigma^2)} &= \frac{1}{\sigma^2}\frac{\partial l_{obs}(\theta)}{\partial\beta} - \sum_j \frac{\sigma_b^2}{\sigma^2(\sigma^2+n_j\sigma_b^2)^2}[\sum_i(Y_{ij}-x_{ij}'\beta)](\sum_i x_{ij}),\\
-\frac{\partial^2 l_{obs}(\theta)}{\partial\beta\partial(\sigma_b^2)} &= \sum_j[\sum_i(Y_{ij}-x_{ij}'\beta)](\sum_i x_{ij})/(\sigma^2+n_j\sigma_b^2)^2,\\
-\frac{\partial^2 l_{obs}(\theta)}{\partial(\sigma^2)^2} &= -\frac{n-N}{2\sigma^4} + \sum_j\left(-\frac{1}{2(\sigma^2+n_j\sigma_b^2)^2} + \frac{1}{\sigma^6}\sum_i(Y_{ij}-x_{ij}'\beta)^2 + \left[\frac{\sigma_b^2}{\sigma^4(\sigma^2+n_j\sigma_b^2)^2} - \right.\right.\\
&\qquad \left.\left.\frac{\sigma_b^2(2\sigma^2+n_j\sigma_b^2)^2}{\sigma^6(\sigma^2+n_j\sigma_b^2)^3}\right][\sum_i(Y_{ij}-x_{ij}'\beta)]^2\right),\\
-\frac{\partial^2 l_{obs}(\theta)}{\partial(\sigma_b^2)^2} &= \sum_j\left(-\frac{n_j^2}{2(\sigma^2+n_j\sigma_b^2)^2} + [\sum_i(Y_{ij}-x_{ij}'\beta)]^2\frac{n_j}{(\sigma^2+n_j\sigma_b^2)^3}\right),\\
-\frac{\partial^2 l_{obs}(\theta)}{\partial(\sigma_b^2)\partial(\sigma^2)} &= \sum_j\left(-\frac{n_j}{2(\sigma^2+n_j\sigma_b^2)^2} + [\sum_i(Y_{ij}-x_{ij}'\beta)]^2\frac{1}{(\sigma^2+n_j\sigma_b^2)^3}\right).
\end{aligned}$$

The conditional expectations of the complete data second derivatives given the observed data are

$$\begin{aligned}
-\mathrm{E}\left(\frac{\partial^2 l_{comp}(\theta)}{\partial\beta\partial\beta'}\middle| Y_{obs}\right) &= \frac{1}{\sigma^2}\sum_j X_j'X_j,\\
-\mathrm{E}\left(\frac{\partial^2 l_{comp}(\theta)}{\partial\beta\partial(\sigma^2)}\middle| Y_{obs}\right) &= \frac{1}{\sigma^4}\sum_{i,j}(Y_{ij}-x_{ij}'\beta)x_{ij} - \frac{1}{\sigma^4}\sum_j b_j^*(\sum_i x_{ij}),\\
-\mathrm{E}\left(\frac{\partial^2 l_{comp}(\theta)}{\partial\beta\partial(\sigma_b^2)}\middle| Y_{obs}\right) &= -\mathrm{E}\left(\frac{\partial^2 l_{comp}(\theta)}{\partial(\sigma_b^2)\partial(\sigma^2)}\middle| Y_{obs}\right) = 0,\\
-\mathrm{E}\left(\frac{\partial^2 l_{comp}(\theta)}{\partial(\sigma^2)^2}\middle| Y_{obs}\right) &= -\frac{n}{2\sigma^4} + \sum_{i,j}\frac{(Y_{ij}-x_{ij}'\beta)^2}{\sigma^6} + \sum_j\frac{-2b_j^*\sum_i(Y_{ij}-x_{ij}'\beta)+n_j(b_j^{*2}+c_j^*)}{\sigma^6},\\
-\mathrm{E}\left(\frac{\partial^2 l_{comp}(\theta)}{\partial(\sigma_b^2)^2}\middle| Y_{obs}\right) &= -\frac{N}{2\sigma_b^4} + \frac{1}{\sigma_b^6}\sum_j(b_j^{*2}+c_j^*),
\end{aligned}$$

where the parameters in b_j^* and c_j^* are evaluated at the same θ used elsewhere in the formulas. (These formulas are obtained by differentiating the complete data scores and taking the conditional expectations of b_j and b_j^2.)

Below is an S function to compute the observed data information, based on the formulas above. Note that the data and related quantities are assumed to be in the working directory. Also, the variance terms need to be input, not their logarithms.

```
> z
$x:
[1]  9.9883596  1.0909981  0.1058367 -0.2976235

$converged:
[1] T

$conv.type:
[1] "relative function convergence"

> # function to compute likelihood score and observed information for
> # the observed data
> fclik2 <- function(b) {
+   beta <- b[1:2]
+   sig <- b[3]
+   sigb <- b[4]
+   v2 <-sig+nj*sigb
+   r <- y-X %*% beta
+   rg2 <- tapply(r,g,sum)
+   x2 <- matrix(0,ncol=ncol(X),nrow=length(rg2))
+   for (i in 1:length(rg2))  x2[i,] <- rep(1,nj[i]) %*% X[g==i,]
+   l <- -(sum(nj-1)*log(b[3])+sum(log(sig+nj*sigb))+sum(r*r)/sig-
+   sum(rg2^2*sigb/v2)/sig)/2
+ # scores: sb is beta, s2 is sigma^2, s3 is sigma_b^2
+   sb <- (apply(c(r)*X,2,sum)-apply(c((sigb/v2)*rg2)*x2,2,sum))/sig
+   s2 <- (-sum((nj-1)/sig+1/v2)+sum(r*r)/sig^2-sum(rg2^2*sigb*
+   (sig+v2)/(sig*v2)^2))/2
+   s3 <- sum(-nj/v2+(rg2/v2)^2)/2
+ # information ib for beta, ibs for betaxsigma^2 ibs2 for betaxsigma_b^2
+ # is for sigma^2, is2 for sigma_b^2, is3 for sigma^2xsigma_b^2
+   ib <- (t(X)%*%X-t(x2) %*% diag(sigb/v2) %*% x2)/sig
+   ibs <- (sb-apply(c((sigb/v2^2)*rg2)*x2,2,sum))/sig
+   ibs2 <- apply(c((1/v2^2)*rg2)*x2,2,sum)
+   is <- -sum((nj-1)/sig^2+1/v2^2)/2+sum(r*r)/sig^3+sum(rg2^2*sigb*
+          (sig*v2-(sig+v2)^2)/(sig*v2)^3)
+   is2 <- sum((-nj/(2*v2)+(rg2/v2)^2)*nj/v2)
+   is3 <- sum((-nj/(2*v2)+(rg2/v2)^2)/v2)
+   list(lik=l,score=c(sb,s2,s3),inf=cbind(rbind(ib,ibs,ibs2),
+        c(ibs,is,is3),c(ibs2,is3,is2)))
+ }
> bb <- c((z[[1]])[1:2],exp((z[[1]])[3:4]))
> z2 <- fclik2(bb)
> z2
$lik:
[1] -65.36781
```

```
$score:
                            x
 -2.516083e-06 -2.174082e-06 1.085196e-06 -1.032213e-06

$inf:
                          x
   11.62765249   1.8197904  -0.09724661 0.1455744
x   1.81979042  82.0244773  -0.32338890 0.4841089
   -0.09724661  -0.3233889  36.64068459 0.4897123
    0.14557437   0.4841089   0.48971226 7.0961166
```

If only the observed data scores were available, the forward difference approximation (or even better a central difference approximation) could be used. The forward difference approximation from the function `fdjac()` gives the following. It gives quite close agreement with the analytical results above.

```
> fcs <- function(b) fclik2(b)[[2]]
> -fdjac(bb,fcs)
             [,1]        [,2]         [,3]      [,4]
[1,]  11.62765250   1.8197907  -0.09724664 0.1455744
[2,]   1.81979042  82.0244775  -0.32338888 0.4841089
[3,]  -0.09724631  -0.3233867  36.64068360 0.4897116
[4,]   0.14557437   0.4841088   0.48971230 7.0961166
```

The conditional expectation of the complete data information is given next. The Jacobian of the algorithmic map will then be estimated from the SEM algorithm, to obtain an alternate estimate of the observed information.

```
> # conditional expectation of complete data information
> r <- y-X%*%beta
> bstar <- tapply(r,g,sum)/(nj+sig/sigb)
> cj <- sigb*sig/(sig+nj*sigb)
> ib <- t(X)%*%X/sig
> ibs <- c(t(X)%*%r - apply(X*(bstar[g]),2,sum))/sig^2
> is <- -n/(2*sig^2)+(sum(r*r)+sum(bstar*bstar*(nj-2*sig/cj)+nj*cj))/sig^3
> is2 <- -length(cj)/(2*sigb^2)+sum(bstar*bstar+cj)/sigb^3
> ic <- cbind(rbind(ib,ibs,rep(0,ncol(X))),c(ibs,is,0),c(rep(0,ncol(X)),0,is2))
> ic
                            x
   8.995713e+01  9.818038e+00 -6.932212e-04 0.000000
x  9.818038e+00  9.846385e+01 -6.981834e-05 0.000000
  -6.932212e-04 -6.981834e-05  4.046142e+01 0.000000
   0.000000e+00  0.000000e+00  0.000000e+00 9.067442
```

Note that this suggests that the complete data would have much more information about the constant term, but only a little more about the other terms.

Recall that the EM algorithm required 53 iterations to reach convergence, but after the first few iterations the parameters were close to the final values. Below, after every 10 iterations of the EM algorithm, the approximation to the Jacobian from the SEM algorithm is computed.

```
> # function to compute the Jacobian of the EM algorithm mapping
> fdm <- function(b,s,sb,beta,sig,sigb,nrep=10) {
+ # b,s,sb mles from em
+ # beta,sig,sigb starting values for sem
+ # unlike Meng and Rubin, using M(mle) instead of assuming this is
+ # =mle.  This seems to work better for small increments
+ # nrep is the # reps of EM at which to approximate the Jacobian
+   us <- emup(b,s,sb)
+   for (j in 1:nrep) {
+     u <- emup(beta,sig,sigb)
+     beta <- u$beta
+     sig <- u$sig
+     sigb <- u$sigb
+     dm <- matrix(0,length(beta)+2,length(beta)+2)
+     for (i in 1:length(beta)) {
+       bb <- b
+       bb[i] <- beta[i]
+       u <- emup(bb,s,sb) # only perturb one component of mle at a time
+       dm[i,] <- c(u$beta-us$beta,u$sig-us$sig,u$sigb-us$sigb)/(bb[i]-b[i])
+     }
+     u <- emup(b,sig,sb)
+     dm[length(beta)+1,] <- c(u$beta-us$beta,u$sig-us$sig,u$sigb-us$sigb)/
+       (sig-s)
+     u <- emup(b,s,sigb)
+     dm[length(beta)+2,] <- c(u$beta-us$beta,u$sig-us$sig,u$sigb-us$sigb)/
+       (sigb-sb)
+     print(dm)
+   }
+   list(dm=dm,beta=beta,sig=sig,sigb=sigb)
+ }
> # run a few iterations of EM from the initial starting value
> bb <- list(beta=c(mean(y),rep(0,ncol(X)-1)),sig=1,sigb=1)
> for (j in 1:10) bb <- emup(bb$beta,bb$sig,bb$sigb)
> w <- fdm(beta,sig,sigb,bb$beta,bb$sig,bb$sigb,1)
             [,1]         [,2]         [,3]          [,4]
[1,]  0.871359206 -0.005654803  0.002420528  0.0005174388
[2,]  0.071467412  0.159832398  0.007971319 -0.0534126775
[3,]  0.000723939  0.003211234  0.094429755 -0.0540090387
[4,] -0.001083199 -0.004806531 -0.012099036  0.2173713377
> for (j in 1:10) bb <- emup(bb$beta,bb$sig,bb$sigb)
> w <- fdm(beta,sig,sigb,bb$beta,bb$sig,bb$sigb,1)
              [,1]         [,2]         [,3]        [,4]
```

```
[1,]  0.8713592057 -0.005654803  0.002407766 -0.01186342
[2,]  0.0714674123  0.159832398  0.007986871 -0.05338475
[3,]  0.0007239027  0.003211260  0.094429905 -0.05400955
[4,] -0.0010839546 -0.004807739 -0.012106584  0.21742746
> for (j in 1:10) bb <- emup(bb$beta,bb$sig,bb$sigb)
> w <- fdm(beta,sig,sigb,bb$beta,bb$sig,bb$sigb,1)
             [,1]         [,2]         [,3]        [,4]
[1,]  0.871359206 -0.005654803  0.002404567 -0.01496695
[2,]  0.071467412  0.159832398  0.007990658 -0.05337795
[3,]  0.000723893  0.003211268  0.094429950 -0.05400970
[4,] -0.001083757 -0.004807423 -0.012104608  0.21741277
> for (j in 1:10) bb <- emup(bb$beta,bb$sig,bb$sigb)
> w <- fdm(beta,sig,sigb,bb$beta,bb$sig,bb$sigb,1)
              [,1]         [,2]         [,3]        [,4]
[1,]  0.8713592057 -0.005654803  0.002403765 -0.01574515
[2,]  0.0714674116  0.159832398  0.007991601 -0.05337626
[3,]  0.0007238922  0.003211270  0.094429962 -0.05400974
[4,] -0.0010836821 -0.004807305 -0.012103870  0.21740729
> for (j in 1:10) bb <- emup(bb$beta,bb$sig,bb$sigb)
> w <- fdm(beta,sig,sigb,bb$beta,bb$sig,bb$sigb,1)
              [,1]         [,2]         [,3]        [,4]
[1,]  0.8713592054 -0.005654803  0.002403564 -0.01594032
[2,]  0.0714674524  0.159832393  0.007991835 -0.05337584
[3,]  0.0007238967  0.003211277  0.094429966 -0.05400974
[4,] -0.0010836605 -0.004807273 -0.012103671  0.21740580
```

Except for the upper right corner, the estimated Jacobian matrix is very stable over the entire range. Using the last update, the estimated information is as follows.

```
> # compute the estimated information
> dm2 <- -w$dm
> diag(dm2) <- diag(dm2)+1
> dm2 %*% ic
                           x
[1,]  11.62767741   1.8197940  -0.09734119 0.1445379
[2,]   1.81979577  82.0244664  -0.32337014 0.4839823
[3,]  -0.09727587  -0.3233652  36.64065385 0.4897302
[4,]   0.14467259   0.4839812   0.48973067 7.0961278
```

This agrees closely with the values obtained above.

In this example explicit formulas can be given for the mapping $M(\theta)$, and hence also for its derivatives. For example, from (4.8), the σ_b^2 component of $M(\theta_0)$ is

$$\sum_j (b_j^{*2} + c_j^*)/N = \frac{1}{N}\sum_j \left(\left[\frac{\sigma_{b0}^2 \sum_i (Y_{ij} - x_{ij}'\beta_0)}{\sigma_0^2 + n_j\sigma_{b0}^2} \right]^2 + \frac{\sigma_{b0}^2\sigma_0^2}{\sigma_0^2 + n_j\sigma_{b0}^2} \right).$$

Differentiating this expression gives the elements in the last column of the Jacobian. In particular, the derivative with respect to the constant term, which is the slowly converging element in the upper right corner above, is

$$-\frac{2}{N}\sum_j \left[\frac{\sigma_{b0}^2}{\sigma_0^2 + n_j\sigma_{b0}^2}\right]^2 n_j \sum_i (Y_{ij} - x_{ij}'\beta_0).$$

Evaluating this expression at the MLE gives a value of -0.015956, in close agreement with the numerical value above at the 50th iteration. □

In general it is not clear that the SEM algorithm has any advantage over calculating the observed data scores and using numerical differences to approximate the information. That is, both make use of numerical approximations to derivatives, and one requires computing $\mathrm{E}_\theta[\partial l_{comp}(\theta)/\partial\theta|Y_{obs}]$ and the other requires computing $\mathrm{E}_\theta[\partial^2 l_{comp}(\theta)/\partial\theta\partial\theta'|Y_{obs}]$. In any case using analytical formulas, either by differentiating the observed data scores or by using Louis' formula, will be more reliable.

4.5 The ECM Algorithm

In many applications (unlike the examples above), the M-step would require iterative search methods for finding the maximum. Recall though that generally it is not necessary to find the exact maximum in the M-step for the EM algorithm to converge. In applications where iterative search methods are needed, sometimes subsets of the parameters can be maximized much more easily than the full parameter vector. Suppose that $\theta = (\alpha', \beta')'$, and that $Q(\alpha, \beta|\alpha_0, \beta_0)$ is easy to maximize over either α or β with the other held fixed, but that jointly maximizing over both is more difficult. In these settings Meng and Rubin (1993) proposed using a generalization of the EM algorithm called the Expectation/Conditional Maximization (ECM) algorithm. Given the parameter values from the previous iteration (or the initial values), the algorithm proceeds by computing $Q(\alpha, \beta|\alpha_0, \beta_0)$ in the E-step as before. Next $Q(\alpha, \beta_0|\alpha_0, \beta_0)$ is maximized over α to obtain α_1. Then $Q(\alpha_1, \beta|\alpha_0, \beta_0)$ is maximized over β to obtain β_1. Then the algorithm returns to the E-step, computing the expectation at the new parameter values (α_1, β_1), with the steps repeated until convergence.

The two maximizations within the M-step could be iterated, to get improved M-step estimates before returning to the next E-step (often such iteration would converge to the joint maximizers of $Q(\alpha, \beta|\alpha_0, \beta_0)$, in which case this algorithm would be an EM algorithm with a special type of computations in the M-step). However, since the EM algorithm often requires many iterations through both the E- and M-steps, there is often little advantage to further iteration within each M-step.

The phrase 'Conditional maximization' in ECM is used to denote the process of maximizing over a subset of the parameters with the other parameters held fixed. This is an unfortunate choice of terminology. It would be better in statistical contexts to limit the word 'conditional' to conditional probabilities and conditional distributions. 'Conditional maximization' sounds like it means maximizing the likelihood for a conditional distribution, but all that is meant is fixing the values of some of the parameters.

The algorithm as described above has an obvious generalization to settings where there are three or more different subsets of parameters, each of which is easy to maximize when the others are

held fixed.

Example 4.3 Multivariate normal regression with incomplete response data. Suppose the complete data response vectors Y_i are independent with

$$Y_i \sim \mathrm{N}(X_i\beta, V), \quad i = 1, \ldots, n,$$

where $Y_i = (Y_{i1}, \ldots, Y_{ik})'$, X_i is a $k \times p$ matrix of covariates (usually including a constant term), β is a $p \times 1$ vector of unknown parameters, and the covariance matrix V is only required to be positive definite (which means it has $k(k+1)/2$ unknown parameters).

Suppose that Y_i is incomplete for some cases. For example, the components of Y_i could be longitudinal measurements, and some subjects may drop out before completing all measurements (the missing data is assumed to be missing at random, though).

Let θ consist of the components of β and the free parameters in V. The complete data likelihood is

$$\begin{aligned} l_{comp}(\theta) &= -\frac{n}{2}\log|V| - \frac{1}{2}\sum_i (Y_i - X_i\beta)'V^{-1}(Y_i - X_i\beta) \\ &= -\frac{n}{2}\log|V| - \frac{1}{2}\mathrm{trace}[V^{-1}\sum_i (Y_i - X_i\beta)(Y_i - X_i\beta)'] \\ &= -\frac{n}{2}\log|V| - \frac{1}{2}\mathrm{trace}[V^{-1}\sum_i (Y_iY_i' - Y_i(X_i\beta)' - X_i\beta Y_i' + X_i\beta\beta' X_i')]. \end{aligned} \quad (4.20)$$

Let S_i be the matrix of zeros and ones which 'selects' the elements of Y_i which are actually observed; that is, the product S_iY_i gives the observed components of Y_i. For example, if $k = 4$ and only Y_{i1} and Y_{i3} are observed, then

$$S_i = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix},$$

giving $S_iY_i = (Y_{i1}, Y_{i3})'$. If all components of Y_i are observed, then S_i is the $k \times k$ identity matrix.

Since the trace is the sum of the diagonal elements, the expectation operator can be taken inside the trace, so from (4.20), the E-step of the EM algorithm will consist of computing

$$Y_i^* = \mathrm{E}_{\theta_0}(Y_i | S_iY_i)$$

and

$$W_i^* = \mathrm{E}_{\theta_0}(Y_iY_i' | S_iY_i),$$

and replacing unobserved components in (4.20) by the corresponding conditional expectations. These conditional expectations are functions of the means and variances of the conditional normal distribution of the unobserved components, given the observed components. The calculations are somewhat similar to those in Example 2, and are left as an exercise.

The M-step then consists of maximizing

$$\begin{aligned} &-\frac{n}{2}\log|V| - \frac{1}{2}\mathrm{trace}[V^{-1}\sum_i (W_i^* - Y_i^*(X_i\beta)' - X_i\beta Y_i^{*'} + X_i\beta\beta' X_i')] \\ &= -\frac{n}{2}\log|V| - \frac{1}{2}\mathrm{trace}[V^{-1}\sum_i (W_i^* - Y_i^*Y_i^{*'})] - \frac{1}{2}\sum_i (Y_i^* - X_i\beta)'V^{-1}(Y_i^* - X_i\beta). \end{aligned}$$

In general iterative methods are required to calculate the joint maximizing parameter values. However, for fixed V it is easily seen that the maximizer of β is

$$(\sum_i X_i' V^{-1} X_i)^{-1} \sum_i X_i' V^{-1} Y_i^*, \tag{4.21}$$

and not quite so easily it can be shown that for β fixed the maximizer of V is

$$\frac{1}{n} \sum_i (W_i^* - Y_i^* (X_i\beta)' - X_i\beta Y_i^{*'} + X_i\beta\beta' X_i'). \tag{4.22}$$

Starting from initial values β_0 and V_0, a series of ECM updates can be given by first computing Y_i^* and W_i^*, then using these to compute a new value of β from (4.21), and using Y_i^*, W_i^* and the new value of β to calculate a new value of V from (4.22). These new parameter values are then used to compute new values of Y_i^* and W_i^*, with the process repeated until convergence. Although joint maximizers are not computed at each iteration, generally this algorithm will still converge to the maximizers of the observed data likelihood.

For comparison, the observed data log likelihood is

$$l_{obs}(\theta) = -\frac{1}{2} \sum_i \log |S_i V S_i'| - \frac{1}{2} \sum_i (S_i Y_i - S_i X_i \beta)' (S_i V S_i')^{-1} (S_i Y_i - S_i X_i \beta).$$

Maximizing this would require an iterative search with possibly lots of determinant and matrix inversion calculations within each step. □

4.6 Comments

The EM algorithm provides a convenient framework to approach estimation in incomplete data problems. It is widely used in the statistical literature, and there are many other extensions and variations than those discussed above. For example, Liu and Rubin (1994) extend the ECM algorithm to an ECME algorithm, where in some of the substeps the observed data likelihood is directly maximized over a subset of the parameters, and Wei and Tanner (1990) proposed an MCEM algorithm, where the E step in the EM algorithm is carried out using Monte Carlo integration. Other recent extensions are discussed in Meng and van Dyk (1997).

The EM algorithm is most useful when (1) computing the conditional expectation of the complete data log likelihood is easier than directly computing the observed data likelihood, and (2) maximizing $Q(\theta|\theta_0)$, or at least finding sufficiently improved values of θ through the ECM algorithm or other means, is sufficiently fast. Since the EM algorithm usually converges at a fairly slow rate, individual iterations have to be fast or it will not be competitive with directly maximizing the observed data likelihood.

Another important reason for using the EM algorithm (possibly the main reason) is to compute MLEs from incomplete data using available software for fitting complete data. As has been seen in the examples above, in many problems the M-step involves complete data estimates computed from a modified data set, or simple modifications of such estimates. Provided the E-step calculations can be programmed, it is then often straightforward to compute the steps in the EM algorithm without much additional work. Of course, there is still the problem computing the inverse information or other variance estimates.

In some (probably many) incomplete data problems, the EM algorithm gives no advantage at all. Suppose the complete data $T_1, \ldots, T_n$ are a random sample from a Weibull distribution with density

$$f(t; \alpha, \gamma) = \alpha^{-1}\gamma(t/\alpha)^{\gamma-1}\exp[-(t/\alpha)^{\gamma}].$$

If the observed data are a right censored sample (Y_i, δ_i), $i = 1, \ldots, n$, where as in Example 1, $\delta_i = 1$ indicates that Y_i is the observed failure time T_i, and $\delta_i = 0$ indicates only that $T_i > Y_i$, then the complete and observed data likelihoods are

$$l_{comp} = \sum_i [\log(\gamma) + \gamma\log(T_i/\alpha) - (T_i/\alpha)^{\gamma}],$$

and

$$l_{obs} = \sum_i [\delta_i\{\log(\gamma) + \gamma\log(Y_i/\alpha)\} - (Y_i/\alpha)^{\gamma}]$$

The complete data likelihood is no easier to maximize than the observed data likelihood, and the E-step would require computing $\mathrm{E}(T_i^{\gamma}|T_i > Y_i)$ and $\mathrm{E}(\log[T_i]|T_i > Y_i)$. The second of these is not a closed form integral. Thus the EM algorithm appears to be substantially harder than direct maximization of l_{obs} in this example.

4.7 Exercises

Exercise 4.1 In the usual case where the data consist of independent observations, Louis' formula (4.16) simplifies. Let $l_{c,i}(\theta)$ be the complete data log likelihood contribution from the ith subject. When the subjects are independent, $l_{comp}(\theta) = \sum_{i=1}^n l_{c,i}(\theta)$. Show that in this case,

$$\mathrm{E}_\theta\left(\frac{\partial l_{comp}(\theta)}{\partial\theta}^{\otimes 2}\middle| Y_{obs}\right) - \mathrm{E}_\theta\left(\frac{\partial l_{comp}(\theta)}{\partial\theta}\middle| Y_{obs}\right)^{\otimes 2}$$

$$= \sum_{i=1}^n \left[\mathrm{E}_\theta\left(\frac{\partial l_{c,i}(\theta)}{\partial\theta}^{\otimes 2}\middle| Y_{obs}\right) - \mathrm{E}_\theta\left(\frac{\partial l_{c,i}(\theta)}{\partial\theta}\middle| Y_{obs}\right)^{\otimes 2}\right],$$

so the last two terms in Louis' formula can be replaced by the right hand side of this expression, which is usually more straightforward to evaluate than the left hand side.

Exercise 4.2 Suppose $y_1, \ldots, y_n$ are a random sample from a finite mixture model with density

$$\sum_{l=1}^r \pi_l f(y; \mu_l, \sigma_l),$$

where $f(y; \mu_l, \sigma_l)$ is the normal density with mean μ_l and variance σ_l^2. The π_l, μ_l, σ_l, $l = 1, \ldots, r$, are unknown parameters, but assume r is known (also note $\sum_l \pi_l = 1$). One way to think of this is that each y_i is drawn from one of the normal populations with density $f(y; \mu_l, \sigma_l)$, but we don't know which one (and different y's may be drawn from different populations). The π_l are the marginal probabilities that y_i is actually from population l. Finite mixture models also provide a convenient flexible family that gives a good approximation to many non-normal distributions.

A natural augmented data model in this problem is to let $z_{ik} = 1$ if y_i is drawn from population k and $z_{ik} = 0$ otherwise, $k = 1, \ldots, r$. The z_{ik} are not observed, but if they were the analysis would be considerably simplified.

1. Give the joint distribution of $(y_i, z_{i1}, \ldots, z_{ir})$.
2. (E step) Give a formula for $Q(\theta|\theta_0)$, the expectation (at θ_0) of the complete data log likelihood conditional on the observed data. (θ represents all the unknown parameters.)
3. (M step) Give formulas for the values θ_1 of θ which maximize $Q(\theta|\theta_0)$.
4. Using the formulas derived for the previous parts, write a program to implement the EM algorithm for arbitrary r and $(y_1, \ldots, y_n)$.
5. The file `mix.data` contains data drawn from a finite mixture model. Assume $r = 2$, and use your EM algorithm program to find the MLE for this data. Repeat assuming $r = 3$.
6. Apply the accelerated EM algorithm from Section 4.3 to fitting the model with $r = 2$.
7. Calculate the observed data information matrix at the MLE when $r = 2$, using the following methods:
   (a) Use formula (4.11) to give expressions for the observed data scores, and analytically differentiate these formulas to obtain formulas for the information;
   (b) Louis' method (4.16) (this is not too hard using the result from Exercise 4.1, and taking advantage of the fact that the z_{ik} are binary and $\sum_k z_{ik} = 1$);
   (c) the SEM algorithm.

Exercise 4.3 Consider the missing covariate data problem from Exercise 3.4.

1. Using the same model, formulate the steps in the EM algorithm for maximizing the likelihood.
2. For the same data (file `t1.dat`) as before, fit the model using the EM algorithm from part 1.
3. Apply the accelerated EM algorithm from Section 4.3 to this problem. How does the computational burden compare with the ordinary EM algorithm?

(Note: the M step updates do not have closed form solutions here, so the basic EM algorithm might not be an ideal approach.)

4.8 References

Aitkin M (1981). A note on the regression analysis of censored data. *Technometrics*, **23**:161-163.

Dempster AP, Laird NM and Rubin DB (1977). Maximum likelihood estimation from incomplete data via the EM algorithm (with discussion). *Journal of the Royal Statistical Society, Series B*, **39**:1–38.

Jamshidian M and Jennrich RI (1997). Acceleration of the EM algorithm by using quasi-Newton methods. *Journal of the Royal Statistical Society*, **59**: 569–587.

Lange K (1999). *Numerical Analysis for Statisticians*. Springer.

Liu C and Rubin DB (1994). The ECME algorithm: a simple extension of EM and ECM with faster monotone convergence. *Biometrika*, **81**:633–648.

Louis TA (1982). Finding the observed information matrix when using the EM algorithm. *Journal of the Royal Statistical Society, Series B*, **44**:226–233.

Meng X-L and Rubin DB (1991). Using EM to obtain asymptotic variance-covariance matrices: the SEM algorithm. *Journal of the American Statistical Association*, **86**:899–909.

Meng X-L and Rubin DB (1993). Maximum likelihood estimation via the ECM algorithm: a general framework. *Biometrika*, **80**:267–278.

Meng X-L and van Dyk D (1997). An old folk-song sung to a fast new tune (with discussion). *Journal of the Royal Statistical Society*, **59**:511–567.

Orchard T and Woodbury MA (1972). A missing information principle: theory and application. *Proceedings of the 6th Berkeley Symposium on Mathematical Statistics and Probability*, **1**:697–715.

Press WH, Teukolsky SA, Vetterling WT, and Flannery BP (1992). *Numerical Recipes in C: The Art of Scientific Computing. Second Edition*. Cambridge University Press.

Schmee J and Hahn GJ (1979). A simple method for regression analysis with censored data. *Technometrics*, **21**:417-432.

Wei GCG and Tanner MA (1990). A Monte Carlo implementation of the EM algorithm and the poor man's data augmentation algorithms. *Journal of the American Statistical Association*, **85**:699–704.

Wu CFJ (1983). On the convergence properties of the EM algorithm. *Annals of Statistics*, **11**:95–103.

Chapter 5

Laplace Approximations

5.1 Introduction

The Laplace approximation is a method for approximating integrals using local information about the integrand at its maximum. As such it is most useful when the integrand is highly concentrated about its maxizing value. A common application is to computing posterior means and variances of parametric functions in Bayesian analyses. In these settings, the posterior becomes increasingly concentrated about the posterior mode as the sample size increases, and the Laplace approximation converges to the correct value as the sample size approaches infinity. The Laplace approximation is related to the large sample normal approximation to the posterior, but is generally more accurate. The Laplace approximation is also useful for approximating the likelihood in various nonlinear random effects models, when the integrals in the likelihood do not have closed form solutions, and in other models with similar structures.

Example 5.1 Stanford heart transplant data. Turnbull, Brown and Hu (1974) give data on survival times for a series of patients registered in a heart transplant program. (This data set is used as an example by Tierney and Kadane (1986) and Tierney, Kass and Kadane (1989).) Let Y_i be the time from entry until transplant and X_i the time from entry until death. The primary interest is in modeling mortality as a function of transplant. The time to transplant Y_i has a hypothetical value even for patients who died before transplant, or who are still awaiting transplant at the time of analysis, and the analysis is conducted conditional on these times to transplant. The mortality hazard for patient i at time t is assumed to be

$$\phi_i \tau^{I(Y_i \le t)},$$

where ϕ_i is the mortality hazard rate for subject i in the absence of transplant. When transplant occurs the hazard is shifted by a factor τ, which then represents the effect of transplant. To allow heterogeneity in the baseline mortality rates ϕ_i, suppose the ϕ_i are a random sample from a gamma distribution with shape p and rate λ. Set

$$a_i(t) = \int_0^t \tau^{I(Y_i \le u)}\, du = \min(t, Y_i) + I(t > Y_i)(t - Y_i)\tau.$$

Then the integrated hazard conditional on ϕ_i is $\phi_i a_i(t)$, and

$$\begin{aligned} P(X_i > t) &= \int_0^\infty \exp[-\phi a_i(t)]\lambda^p \phi^{p-1} \exp(-\lambda\phi)\, d\phi/\Gamma(p) \\ &= \left(\frac{\lambda}{\lambda + a_i(t)}\right)^p \int_0^\infty [\lambda + a_i(t)]^p \phi^{p-1} \exp(-\phi[\lambda + a_i(t)])\, d\phi/\Gamma(p) \\ &= \left(\frac{\lambda}{\lambda + a_i(t)}\right)^p. \end{aligned}$$

Note that

$$a_i(X_i) = \begin{cases} X_i, & X_i \le Y_i, \\ Y_i + \tau Z_i, & X_i > Y_i, \end{cases} \quad \text{and} \quad a_i'(X_i) = \begin{cases} 1, & X_i \le Y_i, \\ \tau, & X_i > Y_i, \end{cases}$$

where $Z_i = X_i - Y_i$. Suppose there are N patients who do not get transplants, and M who do. The observed data consist of censored survival times (x_i, δ_i), $i = 1, \ldots, N$, for patients who do not receive transplant ($\delta_i = 0$ if censored), and (x_i, δ_i, y_i), $i = N+1, \ldots, N+M$ for patients who do, where y_i is the observed time to transplant and (x_i, δ_i) are again the censored survival times (measured from entry in the program). Set $z_i = x_i - y_i$, $i = N+1, \ldots, N+M$. (In the actual data, $N = 30$ and $M = 52$.) From the above formulas, the log-likelihood is easily seen to be

$$l(\theta) = (N+M)p\log(\lambda) + \sum_{i=1}^{N}[\delta_i \log(p) - (p+\delta_i)\log(\lambda + x_i)] + \sum_{i=N+1}^{N+M}[\delta_i \log(p\tau) - (p+\delta_i)\log(\lambda + y_i + \tau z_i)], \tag{5.1}$$

where for convenience define $\theta = (p, \lambda, \tau)$. As a first step towards Bayesian inference in this setting, it might be desired to compute the means and variances of the posterior distribution. If flat priors are used, this requires evaluating expressions of the form

$$\frac{\int g(\theta) \exp[l(\theta)]\, d\theta}{\int \exp[l(\theta)]\, d\theta}, \tag{5.2}$$

where $g(\theta) = \theta_j^k$ for $j = 1, 2, 3$ and $k = 1, 2$. If a prior distribution with density $p(\theta)$ was used, then the integrand in the numerator and denominator would be multiplied by $p(\theta)$. Calculation of (5.2) for the first and second moments of all 3 parameters requires evaluating several 3 dimensional integrals. Naylor and Smith (1982) discuss the use of Gauss-Hermite quadrature in this example. Below their results will be compared with the results from the Laplace approximation. □

Example 5.2 Matched pair Poisson data. The NCI conducted a study of cancer deaths in regions near nuclear power plants. For each of N counties containing a nuclear power plant, they obtained the number of cancer deaths. For each of the N counties, they also selected a control county with similar demographics, but without a nuclear power plant, and obtained the number of cancer deaths for each of the control counties. Let y_{i1}, $i = 1, \ldots, N$, be the number of cancer deaths in the counties with nuclear power plants, and y_{i2} be the number of cancer deaths in the corresponding control counties. A reasonable model for such data might be that $y_{i1} \sim$ Poisson(mean $= E_{i1}\rho_i \exp(\theta_i)$) and $y_{i2} \sim$ Poisson(mean $= E_{i2}\rho_i$), where E_{ij} is the expected number of cancer deaths in each county, based on U.S. national rates, the ρ_i allow for variation from national rates within each pair, and $\exp(\theta_i)$ represents the cancer relative risk for the ith pair.

The unknown parameters ρ_i can be eliminated by conditioning. In particular, the conditional distribution of y_{i1} given $y_{i1} + y_{i2}$ is

$$\text{Binomial}(n_i = y_{i1} + y_{i2}, p_i = E_{i1}\exp(\theta_i)/[E_{i1}\exp(\theta_i) + E_{i2}]).$$

It is probably not reasonable to expect that the θ_i will all be equal, but there would be particular interest in the question of whether on average they tend to be greater than 0. One way to proceed is to use a random effects model. Suppose

$$\theta_i = \theta + \epsilon_i,$$

where the ϵ_i are iid $\text{N}(0, \sigma^2)$. Dropping factors that do not involve unknown parameters, the conditional likelihood contribution for the ith pair can be written

$$Q_i = \int \exp(y_{i1}[\theta + \epsilon])(E_{i1}\exp(\theta + \epsilon) + E_{i2})^{-y_{i1}-y_{i2}}\sigma^{-1}\phi(\epsilon/\sigma)\, d\epsilon, \tag{5.3}$$

where $\phi(\cdot)$ is the standard normal density function. The conditional log-likelihood is then $l(\theta, \sigma) = \sum_i \log(Q_i)$. The integral in Q_i does not have a closed form expression. It could be approximated using Gauss-Hermite quadrature methods. A simple alternative is provided by the Laplace approximation, as discussed by Liu and Pierce (1993). Here the integral is more peaked for small σ and less peaked for large σ, and it will turn out that the Laplace approximation is more accurate for small σ. □

5.2 Definition and Theory

For the most part the discussion of the Laplace approximation which follows will be confined to the one-dimensional case. Formulas for Laplace approximations for positive functions of several dimensions are given at (5.8) and (5.9) below.

For one-dimensional integrals of positive functions, the Laplace approximation in its simplest form is

$$\int \exp[h(\theta)]\, d\theta = \sqrt{2\pi}\exp[h(\hat{\theta})]/[-h''(\hat{\theta})]^{1/2}, \tag{5.4}$$

where $\hat{\theta}$ maximizes $h(\theta)$. Note that this form of the approximation only applies to positive integrands. This is sometimes called the "fully exponential" approximation, since the integrand can be written in the form $\exp[h(\theta)]$. The idea for this approximation comes from a second order Taylor series expansion of $h(\theta)$ about $\theta = \hat{\theta}$. That is, for θ near $\hat{\theta}$,

$$h(\theta) \doteq h(\hat{\theta}) + h'(\hat{\theta})(\theta - \hat{\theta}) + h''(\hat{\theta})(\theta - \hat{\theta})^2/2,$$

and $h'(\hat{\theta}) = 0$ since $\hat{\theta}$ maximizes h. Thus

$$\begin{aligned}
\int \exp[h(\theta)]\, d\theta &\doteq \int \exp[h(\hat{\theta}) + h''(\hat{\theta})(\theta - \hat{\theta})^2/2]\, d\theta \\
&= \exp[h(\hat{\theta})]\sqrt{\frac{2\pi}{-h''(\hat{\theta})}} \int \sqrt{\frac{-h''(\hat{\theta})}{2\pi}} \exp[-(-h''(\hat{\theta})/2)(\theta - \hat{\theta})^2]\, d\theta \\
&= \exp[h(\hat{\theta})]\sqrt{\frac{2\pi}{-h''(\hat{\theta})}},
\end{aligned}$$

since the integrand in the middle line is the density function of a normal distribution with mean $\hat{\theta}$ and variance $-1/h''(\hat{\theta})$.

Suppose that the function h is part of a sequence of functions indexed by a parameter n, such that $q_n(\theta) = h_n(\theta)/n$ converges to a well behaved function $q(\theta)$ as $n \to \infty$. For computing posterior moments as in Example 5.1, usually $h_n(\theta) = \log[g(\theta)] + l_n(\theta) + \log[p(\theta)]$ for a fixed function g, where l_n and p are the log-likelihood and prior density, and $q_n(\theta) = h_n(\theta)/n$ converges to the expected log likelihood per subject as $n \to \infty$ ($\log[g(\theta)]/n$ and $\log[p(\theta)]/n$ both $\to 0$ as $n \to \infty$). In Example 5.2, the parameter $1/\sigma^2$ plays the role of n in the expansions.

For sequences h_n with the properties given above, a more careful analysis gives the result

$$\int \exp[h_n(\theta)]\, d\theta = \int \exp[nq_n(\theta)]\, d\theta = \sqrt{\frac{2\pi}{-nq_n''(\hat{\theta})}}\, \exp[nq_n(\hat{\theta})](1 + a_n/n + O(1/n^2)), \tag{5.5}$$

where

$$a_n = (1/8)q_n^{(4)}(\hat{\theta})/q_n''(\hat{\theta})^2 + (5/24)q_n^{(3)}(\hat{\theta})^2/[-q_n''(\hat{\theta})^3].$$

Since the derivatives q_n'', $q_n^{(3)}$ and $q_n^{(4)}$ are generally $O(1)$, the relative error in the Laplace approximation is $O(1/n)$. It is also generally true that the higher order derivatives in the formula for a_n can be used to obtain an approximation with a relative error of $O(1/n^2)$; see Tierney, Kass and Kadane (1989) and Section 3.3 of Barndorff-Nielsen and Cox (1989).

Computing posterior moments requires evaluating the ratio of two related integrals. It turns out that the errors in the two integrals tend to be similar, so the Laplace approximation to the ratio is more accurate than for either integral by itself. Again letting $l_n(\theta)$ be the log likelihood and $p(\theta)$ the prior density, writing

$$d_n(\theta) = \{l_n(\theta) + \log[p(\theta)]\}/n \tag{5.6}$$

and $q_n(\theta) = \log[g(\theta)]/n + d_n(\theta)$, and letting $\hat{\theta}_q$ be the maximizer of q_n and $\hat{\theta}_d$ be the maximizer of d_n, and applying (5.4), the Laplace approximation to the posterior mean of a positive function $g(\theta)$ is given by

$$\frac{\int g(\theta)\exp[nd_n(\theta)]\, d\theta}{\int \exp[nd_n(\theta)]\, d\theta} = \frac{\int \exp[nq_n(\theta)]\, d\theta}{\int \exp[nd_n(\theta)]\, d\theta} \doteq \left(\frac{-d_n''(\hat{\theta}_d)}{-q_n''(\hat{\theta}_q)}\right)^{1/2} \exp[nq_n(\hat{\theta}_q) - nd_n(\hat{\theta}_d)]. \tag{5.7}$$

More precisely, setting

$$a_n = (1/8)q_n^{(4)}(\hat{\theta}_q)/q_n''(\hat{\theta}_q)^2 + (5/24)q_n^{(3)}(\hat{\theta}_q)^2/[-q_n''(\hat{\theta}_q)^3]$$

and

$$b_n = (1/8)d_n^{(4)}(\hat{\theta}_d)/d_n''(\hat{\theta}_d)^2 + (5/24)d_n^{(3)}(\hat{\theta}_d)^2/[-d_n''(\hat{\theta}_d)^3],$$

then from (5.5),

$$\begin{aligned}
\frac{\int \exp[nq_n(\theta)]\, d\theta}{\int \exp[nd_n(\theta)]\, d\theta} &= \left(\frac{-d_n''(\hat{\theta}_d)}{-q_n''(\hat{\theta}_q)}\right)^{1/2} \exp[nq_n(\hat{\theta}_q) - nd_n(\hat{\theta}_d)]\frac{1 + a_n/n + O(1/n^2)}{1 + b_n/n + O(1/n^2)} \\
&= \left(\frac{-d_n''(\hat{\theta}_d)}{-q_n''(\hat{\theta}_q)}\right)^{1/2} \exp[nq_n(\hat{\theta}_q) - nd_n(\hat{\theta}_d)][1 + (a_n - b_n)/n + O(1/n^2)],
\end{aligned}$$

and since q_n and d_n differ only by the term $\log[g(\theta)]/n$, typically $a_n - b_n = O_p(1/n)$, so the Laplace approximation to the ratio is accurate to $O(1/n^2)$.

In Example 5.2, Q_i in (5.3) can be rewritten

$$\frac{1}{\sqrt{2\pi}\sigma}\int \exp[\log(g(\epsilon)) - \epsilon^2/(2\sigma^2)]\,d\epsilon,$$

where

$$g(\epsilon) = \exp[y_{i1}(\theta+\epsilon)]/[E_{i1}\exp(\theta+\epsilon) + E_{i2}]^{y_{i1}+y_{i2}}.$$

Then $-\epsilon^2/(2\sigma^2)$ is similar to the term nd_n above, with $1/\sigma^2$ as 'n'. Thus applying (5.5), the Laplace approximation to $\int \exp[\log(g(\epsilon)) - \epsilon^2/(2\sigma^2)]\,d\epsilon$ has relative accuracy of $O(\sigma^2)$. However, for a ratio of the likelihood evaluated at different θ, σ values, generally the accuracy improves to $O(\sigma^4)$. That is, the likelihood ratio has ratios of such integrals, and as before the leading terms in the errors of the approximations to the integrals in the ratios roughly cancel. Since the likelihood is only determined to a constant of proportionality, which does not depend on the parameters (θ and σ in Example 5.2), the likelihood itself can be thought of as including the ratio of each $Q_i(\theta,\sigma)$ to $Q_i(\theta_0,\sigma_0)$ at a fixed (θ_0,σ_0). This suggests that for any practical purpose (such as computing the MLEs), the Laplace approximation to the likelihood is accurate to $O(\sigma^4)$.

Similar results are available for p-dimensional integrals of positive functions. The Laplace approximation is

$$\int \exp[h(\theta)]\,d\theta = (2\pi)^{p/2}\det\left(-\frac{\partial^2 h(\hat\theta)}{\partial\theta\partial\theta'}\right)^{-1/2}\exp[h(\hat\theta)], \tag{5.8}$$

where $\hat\theta$ maximizes $h(\cdot)$, and $\det(A)$ denotes the determinant of a matrix A. The only differences from the univariate case are that the second derivative of h has been replaced by the determinant of the matrix of second derivatives, and the power of 2π. For approximating the posterior mean of a parametric function $g(\theta)$, using notation analogous to that used in (5.7),

$$\frac{\int \exp[nq_n(\theta)]\,d\theta}{\int \exp[nd_n(\theta)]\,d\theta} \doteq \left(\frac{\det[-\partial^2 d_n(\hat\theta_d)/\partial\theta\partial\theta']}{\det[-\partial^2 q_n(\hat\theta_q)/\partial\theta\partial\theta']}\right)^{1/2}\exp[nq_n(\hat\theta_q) - nd_n(\hat\theta_d)]. \tag{5.9}$$

As in the univariate case, this approximation is usually accurate to $O(1/n^2)$.

Tierney and Kadane (1986) also give an approximation for marginal posterior densities. If $\theta = (\theta_1, \theta_2')'$, then the marginal posterior density of θ_1 is

$$\frac{\int \exp\{l_n(\theta_1,\theta_2) + \log[p(\theta_1,\theta_2)]\}\,d\theta_2}{\int \exp\{l_n(\theta) + \log[p(\theta)]\}\,d\theta},$$

where the integrands are the same in the numerator and denominator, but the integral in the numerator is only over the components in θ_2 with θ_1 fixed. Applying (5.8) to the integrals in the numerator and denominator then gives an approximation to the posterior density at θ_1. To get an approximation to the full posterior density, this approximation needs to be computed at a number of different θ_1 values. The resulting approximation generally only has a pointwise accuracy of $O(1/n)$, because of the different dimensions of the integrals in the numerator and denominator. However, renormalizing the approximate density to integrate to 1 can substantially improve the resulting approximation.

All of the previous discussion applies to positive functions. Tierney, Kass and Kadane (1989) consider general Laplace approximations for computing posterior moments of the form

$$\frac{\int g(\theta)\exp(nd_n(\theta))\,d\theta}{\int \exp(nd_n(\theta))\,d\theta},$$

where $g(\theta)$ can take on negative values, and where typically nd_n is the sum of the log-likelihood and the log-prior density, as in (5.6). Applying a quadratic expansion to d_n in both the numerator and denominator does not lead to the same accuracy as when g is positive. By explicitly correcting for higher order terms they obtain an approximation which is accurate to $O(1/n^2)$ ((5.13) below is a special case). They also consider using (5.7) or (5.8) to approximate the moment generating function of $g(\theta)$, which has a strictly positive integrand. Evaluating the derivative of the approximation to the moment generating function at 0 then gives an approximation to the posterior mean of $g(\theta)$. They showed that the moment generating function approach is equivalent to the explicit correction, so it is also accurate to $O(1/n^2)$. They seem to prefer the moment generating function approach in practice, and it will now be described in more detail.

The moment generating function of the posterior distribution of $g(\theta)$ is given by

$$M(s) = \frac{\int \exp[sg(\theta) + nd_n(\theta)]\,d\theta}{\int \exp[nd_n(\theta)]\,d\theta},$$

with d_n as in (5.6). Applying (5.4) to the numerator and denominator, as in (5.7), then gives an approximation

$$\tilde{M}(s) = \left(\frac{nd_n''(\hat{\theta}_d)}{sg''(\hat{\theta}_s) + nd_n''(\hat{\theta}_s)}\right)^{1/2} \exp[sg(\hat{\theta}_s) + nd_n(\hat{\theta}_s) - nd_n(\hat{\theta}_d)], \tag{5.10}$$

where $\hat{\theta}_s$ maximizes $sg(\theta) + nd_n(\theta)$ and $\hat{\theta}_d$ maximizes $d_n(\theta)$. Then

$$\mathrm{E}[g(\theta)] \doteq \tilde{M}'(0).$$

Analytic differentiation of (5.10) requires an expression for $d\hat{\theta}_s/ds$. Since $\hat{\theta}_s$ satisfies $sg'(\hat{\theta}_s) + nd_n'(\hat{\theta}_s) = 0$, and differentiating this expression with respect to s gives

$$g'(\hat{\theta}_s) + [sg''(\hat{\theta}_s) + nd_n''(\hat{\theta}_s)]d\hat{\theta}_s/ds = 0,$$

it follows that

$$d\hat{\theta}_s/ds = -\frac{g'(\hat{\theta}_s)}{sg''(\hat{\theta}_s) + nd_n''(\hat{\theta}_s)}.$$

Using this result it is possible to give an explicit formula for $\tilde{M}'(0)$. In the special case where $g(\theta) = \theta$ some simplification results, since then $g'(\theta) = 1$, $g''(\theta) = 0$ and

$$d\hat{\theta}_s/ds = -[nd_n''(\hat{\theta}_s)]^{-1}. \tag{5.11}$$

In this case, differentiating (5.10) gives

$$\tilde{M}'(s) \quad = \quad [-nd_n''(\hat{\theta}_d)]^{1/2} \exp[s\hat{\theta}_s + nd_n(\hat{\theta}_s) - nd_n(\hat{\theta}_d)]$$

$$
\begin{aligned}
&\times \left(-\frac{1}{2} \frac{-nd_n'''(\hat{\theta}_s)}{[-nd_n''(\hat{\theta}_s)]^{3/2}} \frac{d\hat{\theta}_s}{ds} + \frac{1}{[-nd_n''(\hat{\theta}_s)]^{1/2}} \left[\hat{\theta}_s + s\frac{d\hat{\theta}_s}{ds} + nd_n'(\hat{\theta}_s)\frac{d\hat{\theta}_s}{ds} \right] \right) \\
= \;& \tilde{M}(s) \left(\hat{\theta}_s + \left[s + nd_n'(\hat{\theta}_s) - \frac{1}{2}\frac{nd_n'''(\hat{\theta}_s)}{nd_n''(\hat{\theta}_s)} \right] \frac{d\hat{\theta}_s}{ds} \right) \\
= \;& \tilde{M}(s) \left(\hat{\theta}_s - \frac{1}{2}\frac{nd_n'''(\hat{\theta}_s)}{nd_n''(\hat{\theta}_s)} \frac{d\hat{\theta}_s}{ds} \right),
\end{aligned}
$$

since $\hat{\theta}_s$ is determined by solving $s + nd_n'(\theta) = 0$. Substituting for $d\hat{\theta}_s/ds$ then gives

$$
\tilde{M}'(s) = \tilde{M}(s) \left(\hat{\theta}_s + \frac{nd_n'''(\hat{\theta}_s)}{2[nd_n''(\hat{\theta}_s)]^2} \right). \tag{5.12}
$$

Note that $\tilde{M}(0) = 1$. Also, since $\hat{\theta}_s$ and $\hat{\theta}_d$ are determined by $s + nd_n'(\hat{\theta}_s) = 0$ and $nd_n'(\hat{\theta}_d) = 0$, generally $\lim_{s\to 0} \hat{\theta}_s = \hat{\theta}_d$. Thus

$$
\tilde{M}'(0) = \hat{\theta}_d + \frac{nd_n'''(\hat{\theta}_d)}{2[nd_n''(\hat{\theta}_d)]^2} \tag{5.13}
$$

(for $g(\theta) = \theta$). Formula (5.13) approximates the posterior mean with the posterior mode plus a correction term. The correction term is related to the asymmetry in the distribution in the neighborhood of the mode (if the posterior is symmetric, then $d_n'''(\hat{\theta}_d) = 0$).

Evaluating $d_n'''(\theta)$ is sometimes substantial work, so in some settings it is more convenient to approximate $\tilde{M}'(0)$ with the numerical difference

$$
[\tilde{M}(\delta) - \tilde{M}(-\delta)]/(2\delta), \tag{5.14}
$$

for some small δ.

For vector parameters, the second derivatives in (5.10) are again replaced by determinants of matrices of second derivatives, and similar formulas can be obtained.

To calculate the posterior variance of $g(\theta)$, the second moment is also needed. Since $g(\theta)^2$ is always nonnegative, the second moment can often be approximated directly from (5.7). Tierney, Kass and Kadane (1989) also proposed computing the second moment directly from the approximation to the moment generating function. Recall that if $M(s)$ is the moment generating function of $g(\theta)$, then $\mathrm{E}[g(\theta)^2] = M''(0)$ and $\mathrm{Var}[g(\theta)] = d^2 \log[M(0)]/ds^2$. Thus the variance can be approximated by either $\tilde{M}''(0) - \tilde{M}'(0)^2$ or $d^2 \log[\tilde{M}(0)]/ds^2$.

For the special case $g(\theta) = \theta$, it is again straightforward to give an explicit formula for $\tilde{M}''(0)$. Differentiating (5.12) gives

$$
\begin{aligned}
\tilde{M}''(s) &= \tilde{M}'(s) \left(\hat{\theta}_s + \frac{nd_n'''(\hat{\theta}_s)}{2[nd_n''(\hat{\theta}_s)]^2} \right) + \tilde{M}(s)\frac{d\hat{\theta}_s}{ds} \left(1 + \frac{nd_n^{(4)}(\hat{\theta}_s)}{2[nd_n''(\hat{\theta}_s)]^2} - \frac{[nd_n'''(\hat{\theta}_s)]^2}{[nd_n''(\hat{\theta}_s)]^3} \right) \\
&= \tilde{M}(s) \left[\left(\hat{\theta}_s + \frac{nd_n'''(\hat{\theta}_s)}{2[nd_n''(\hat{\theta}_s)]^2} \right)^2 - \frac{1}{nd_n''(\hat{\theta}_s)} - \frac{nd_n^{(4)}(\hat{\theta}_s)}{2[nd_n''(\hat{\theta}_s)]^3} + \frac{[nd_n'''(\hat{\theta}_s)]^2}{[nd_n''(\hat{\theta}_s)]^4} \right],
\end{aligned}
$$

using (5.11). Thus

$$\tilde{M}''(0) = \left(\hat{\theta}_d + \frac{nd_n'''(\hat{\theta}_d)}{2[nd_n''(\hat{\theta}_d)]^2}\right)^2 - \frac{1}{nd_n''(\hat{\theta}_d)} - \frac{nd_n^{(4)}(\hat{\theta}_d)}{2[nd_n''(\hat{\theta}_d)]^3} + \frac{[nd_n'''(\hat{\theta}_d)]^2}{[nd_n''(\hat{\theta}_d)]^4}. \tag{5.15}$$

The first term in this expression is the square of the approximation to the mean, so the remaining terms give the approximation to the variance. As with the mean, the Laplace approximation to the moment generating function gives a variance approximation consisting of the large sample approximation to the variance $(-[nd_n''(\hat{\theta}_d)]^{-1})$ plus a correction term. If the posterior was exactly normal, then d_n would be a quadratic, and $d_n'''(\hat{\theta}_d)$ and $d_n^{(4)}(\hat{\theta}_d)$ would both be 0.

Second difference approximations can also be used for the variance, although their accuracy is questionable. Since $\tilde{M}(0) = 1$, the second derivative $\tilde{M}''(0)$ could be approximated by

$$[\tilde{M}(\delta) - 2\tilde{M}(0) + \tilde{M}(-\delta)]/\delta^2 = [\tilde{M}(\delta) + \tilde{M}(-\delta) - 2]/\delta^2,$$

and $d^2 \log[\tilde{M}(0)]/d\theta^2$ could be approximated by

$$(\log[\tilde{M}(\delta)] - 2\log[\tilde{M}(0)] + \log[\tilde{M}(-\delta)])/\delta^2 = \log[\tilde{M}(\delta)\tilde{M}(-\delta)]/\delta^2. \tag{5.16}$$

In the setting of the continuation of Example 5.1, below, these approximations seemed sensitive to the choice of δ and to the accuracy of the maximizations used, and in some cases led to negative values for the estimated variances.

It is important to remember that the Laplace approximation only uses information about the integrand and its derivatives at a single point. If the integrand is not highly concentrated about that point, then the Laplace approximation is unlikely to be very accurate. This is especially true for multimodal functions.

5.3 Examples

It is instructive to consider how the Laplace approximations perform in an example where the true values are known.

Example 5.3 Moments of a beta distribution. Suppose X has a beta distribution with density

$$f(x) \propto x^{\alpha-1}(1-x)^{\beta-1},$$

with $\alpha > 1$ and $\beta > 1$. Then

$$\begin{aligned} \mathrm{E}(X^k) &= \frac{\int_0^1 x^{k+\alpha-1}(1-x)^{\beta-1}\,dx}{\int_0^1 x^{\alpha-1}(1-x)^{\beta-1}\,dx} \\ &= \frac{\int_0^1 \exp[(k+\alpha-1)\log(x) + (\beta-1)\log(1-x)]\,dx}{\int_0^1 \exp[(\alpha-1)\log(x) + (\beta-1)\log(1-x)]\,dx} \\ &= \frac{(k+\alpha-1)\cdots\alpha}{(k+\alpha+\beta-1)\cdots(\alpha+\beta)}, \end{aligned} \tag{5.17}$$

from the definition of the beta function. (The notation is somewhat confusing relative to the earlier discussion, but α and β are fixed values and x plays the role of θ above.)

Since $g(x) = x^k$ is positive on the domain of X, the approximation (5.7) can be applied. The role of n here is played by $\alpha + \beta$, but it is not necessary to explicitly factor out this term to use the approximation. The quantity $nq_n(x) = (k + \alpha - 1)\log(x) + (\beta - 1)\log(1 - x)$ is maximized by $\hat{x}_q = (k + \alpha - 1)/(k + \alpha + \beta - 2)$ (which can be verified by solving $nq_n'(x) = 0$). Also, as a special case, $nd_n(x) = (\alpha - 1)\log(x) + (\beta - 1)\log(1 - x)$ is maximized by $\hat{x}_d = (\alpha - 1)/(\alpha + \beta - 2)$. Further,

$$-nq_n''(x) = (k + \alpha - 1)/x^2 + (\beta - 1)/(1 - x)^2,$$

so

$$-nq_n''(\hat{x}_q) = \frac{(k + \alpha + \beta - 2)^3}{(k + \alpha - 1)(\beta - 1)}$$

and

$$-nd_n''(\hat{x}_d) = \frac{(\alpha + \beta - 2)^3}{(\alpha - 1)(\beta - 1)}. \tag{5.18}$$

Thus the Laplace approximation to $\mathrm{E}(X^k)$ is

$$\begin{aligned}
&\left(\frac{(\alpha + \beta - 2)^3(k + \alpha - 1)}{(\alpha - 1)(k + \alpha + \beta - 2)^3}\right)^{1/2} \left(\frac{k + \alpha - 1}{k + \alpha + \beta - 2}\right)^{k+\alpha-1} \left(\frac{\beta - 1}{k + \alpha + \beta - 2}\right)^{\beta-1} \\
&\qquad \times \left(\frac{\alpha + \beta - 2}{\alpha - 1}\right)^{\alpha-1} \left(\frac{\alpha + \beta - 2}{\beta - 1}\right)^{\beta-1} \\
&= \frac{(\alpha + \beta - 2)^{\alpha+\beta-1/2}(k + \alpha - 1)^{k+\alpha-1/2}}{(k + \alpha + \beta - 2)^{k+\alpha+\beta-1/2}(\alpha - 1)^{\alpha-1/2}}.
\end{aligned} \tag{5.19}$$

To analyze this rather complicated expression, it is somewhat easier to think in terms of the parameters $\gamma = \alpha + \beta$ and $p = \alpha/(\alpha + \beta)$. Then (5.19) can be expressed

$$\begin{aligned}
&\left(\frac{\gamma - 2}{\gamma + k - 2}\right)^{\gamma-1/2} \left(\frac{p\gamma + k - 1}{p\gamma - 1}\right)^{p\gamma-1/2} \left(\frac{p\gamma + k - 1}{\gamma + k - 2}\right)^{k} \\
&= \left(\frac{1 - 2/\gamma}{1 + (k - 2)/\gamma}\right)^{\gamma-1/2} \left(\frac{1 + (k - 1)/(p\gamma)}{1 - 1/(p\gamma)}\right)^{p\gamma-1/2} \left(\frac{p + (k - 1)/\gamma}{1 + (k - 2)/\gamma}\right)^{k}.
\end{aligned}$$

In the limit as $\gamma \to \infty$ with p held fixed, this expression converges to

$$p^k. \tag{5.20}$$

To see this note that using the first order approximation $\log(1 + u) \doteq u$, which is valid in the limit as $u \to 0$, or alternately using L'Hospital's rule, it follows that

$$(\gamma - 1/2)[\log(1 - 2/\gamma) - \log(1 + (k - 2)/\gamma)] \doteq (\gamma - 1/2)[-2/\gamma - (k - 2)/\gamma] \to -k$$

as $\gamma \to \infty$, so

$$\left(\frac{1 - 2/\gamma}{1 + (k - 2)/\gamma}\right)^{\gamma-1/2} \to e^{-k}.$$

Similarly

$$\left(\frac{1 + (k - 1)/(p\gamma)}{1 - 1/(p\gamma)}\right)^{p\gamma-1/2} \to e^{k}.$$

Then the result follows from

$$\left(\frac{p+(k-1)/\gamma}{1+(k-2)/\gamma}\right)^k \to p^k.$$

At first glance (5.20) may look a bit different from (5.17). However, in the limit as $\gamma = \alpha + \beta \to \infty$, (5.17) $\to p^k$ as well. Thus the Laplace approximation gives the correct result in the limit. The following calculations give some idea of the accuracy of the approximation for finite γ when $k = 1$, so the true value is $\mathrm{E}(X) = p$.

```
> k <- 1
> p <- .1
> g <- trunc(1/p+1):40 # g=gamma; recall alpha=p*g must be > 1
> ((g-2)/(k+g-2))^(g-.5)*((k+p*g-1)/(p*g-1))^(p*g-.5)*((p*g+k-1)/(k+g-2))^k
 [1] 0.1534 0.1278 0.1180 0.1129 0.1098 0.1077 0.1062 0.1052 0.1044 0.1037
[11] 0.1032 0.1028 0.1025 0.1022 0.1020 0.1018 0.1016 0.1015 0.1014 0.1012
[21] 0.1011 0.1011 0.1010 0.1009 0.1009 0.1008 0.1007 0.1007 0.1007 0.1006
> p <- .5
> g <- trunc(1/p+1):40 # recall alpha=p*g must be > 1
> ((g-2)/(k+g-2))^(g-.5)*((k+p*g-1)/(p*g-1))^(p*g-.5)*((p*g+k-1)/(k+g-2))^k
 [1] 0.3977 0.4562 0.4757 0.4846 0.4894 0.4922 0.4940 0.4953 0.4962 0.4969
[11] 0.4974 0.4978 0.4981 0.4983 0.4985 0.4987 0.4988 0.4990 0.4991 0.4991
[21] 0.4992 0.4993 0.4993 0.4994 0.4994 0.4995 0.4995 0.4996 0.4996 0.4996
[31] 0.4996 0.4997 0.4997 0.4997 0.4997 0.4997 0.4997 0.4998
```

For small values of γ, the integrand is not very concentrated about the mode, and the approximation is not very good. As γ increases, the accuracy of the approximation quickly improves.

Even though $g(x)$ is positive in this example, the moment generating function method can also be applied. First consider the case $k = 1$, so $g(x) = x$. For the notation in (5.10), $nd_n = (\alpha-1)\log(x) + (\beta-1)\log(1-x)$, and $\hat{x}_d = (\alpha-1)/(\alpha+\beta-2)$, as above. Also, $g''(x) = 0$. $\hat{x}_s$ is the solution to $s + nd_n'(x) = 0$. It will be convenient to modify the notation. Let $v = \alpha - 1$ and $w = \alpha + \beta - 2$, so $\beta - 1 = w - v$. Then

$$s + nd_n'(x) = s + v/x - (w-v)/(1-x) = 0$$

yields the quadratic equation

$$sx^2 + (w-s)x - v = 0.$$

The roots are $\left[(s-w) \pm \sqrt{(s-w)^2 + 4sv}\right]/(2s)$. Since $w > v > 0$ and s is arbitrarily close to 0, the roots are real. The root

$$\hat{x}_s = \left[(s-w) + \sqrt{(s-w)^2 + 4sv}\right]/(2s) \in (0,1), \tag{5.21}$$

and thus is the relevant solution. Substituting these expressions into (5.10) then gives the Laplace approximation to the moment generating function. Since this will be an explicit function of s, the moments can be determined by differentiation. However, the expression will be complicated, and

this direct approach is not for the algebraically challenged (evaluating the limit as $s \to 0$ is also not completely trivial). It is substantially simpler to use (5.13) and (5.15) here.

First note that applying L'Hospital's rule,

$$\lim_{s\to 0} \hat{x}_s = \lim_{s\to 0}(1 + [(s-w)^2 + 4sv]^{-1/2}[2(s-w) + 4v]/2)/2 = v/w = \hat{x}_d,$$

as is needed to use (5.13). Also, from (5.18), $-nd_n''(\hat{x}_d) = w^3/[v(w-v)]$. Further

$$nd_n'''(x) = 2vx^{-3} - 2(w-v)(1-x)^{-3},$$

so

$$nd_n'''(\hat{x}_d) = 2w^3\frac{(w-v)^2 - v^2}{v^2(w-v)^2} = 2w^4\frac{w-2v}{v^2(w-v)^2}.$$

Substituting in (5.13) then gives that the approximation for $\mathrm{E}(X)$ is

$$v/w + \frac{w^4(w-2v)}{v^2(w-v)^2}\frac{v^2(w-v)^2}{w^6} = \frac{vw + w - 2v}{w^2} = \frac{\alpha^2 + \alpha\beta - 4\alpha + 2}{(\alpha+\beta-2)^2} = \frac{p\gamma(\gamma-4)+2}{(\gamma-2)^2},$$

where again $\gamma = \alpha + \beta$ and $p = \alpha/\gamma$. When $p = 1/2$, this formula is exactly $1/2$ (the correct value). For $p = .1$, it gives the following values for different values of γ.

```
> p <- .1
> g <- trunc(1/p+1):40
> (p*g*(g-4)+2)/(g-2)^2
 [1] 0.1198 0.1160 0.1132 0.1111 0.1095 0.1082 0.1071 0.1062 0.1055 0.1049
[11] 0.1044 0.1040 0.1036 0.1033 0.1030 0.1028 0.1026 0.1024 0.1022 0.1020
[21] 0.1019 0.1018 0.1017 0.1016 0.1015 0.1014 0.1013 0.1012 0.1012 0.1011
```

In this case the Laplace approximation to the moment generating function is more accurate than the fully exponential form given earlier for small γ, but less accurate as γ increases.

To use (5.15) to approximate $\mathrm{Var}(X)$, it is also easily verified that

$$nd_n^{(4)}(x) = -6[v/x^4 + (w-v)/(1-x)^4],$$

so

$$nd_n^{(4)}(\hat{x}_d) = -6w^4[1/v^3 + 1/(w-v)^3].$$

Using these expressions, and some algebra, (5.15) gives

$$\mathrm{Var}(X) \doteq \frac{\alpha^2\beta + \alpha\beta^2 - 9\alpha\beta + 6\alpha + 6\beta - 5}{(\alpha+\beta-2)^4}.$$

□

As a more realistic example of using the Laplace approximation, it will be applied to computing posterior moments in Example 5.1.

Example 5.1 (continued). First formula (5.9) will be used to give approximations for the posterior moments (5.2). In the notation of (5.9),

$$nq_n(\theta) = k\log(\theta_j) + l(\theta)$$

for $k = 1, 2$, and

$$nd_n(\theta) = l(\theta),$$

where the log-likelihood $l(\theta)$ is given by (5.1) (recall that flat priors are being used in this example).

The derivatives of (5.1) are

$$\begin{aligned}
\frac{\partial l(\theta)}{\partial p} &= (N+M)\log(\lambda) + \sum_{i=1}^{N}[\delta_i/p - \log(\lambda + x_i)] + \sum_{i=N+1}^{N+M}[\delta_i/p - \log(\lambda + y_i + \tau z_i)],\\
\frac{\partial l(\theta)}{\partial \lambda} &= (N+M)p/\lambda - \sum_{i=1}^{N}(p+\delta_i)/(\lambda + x_i) - \sum_{i=N+1}^{N+M}(p+\delta_i)/(\lambda + y_i + \tau z_i),\\
\frac{\partial l(\theta)}{\partial \tau} &= \sum_{i=N+1}^{N+M}[\delta_i/\tau - (p+\delta_i)z_i/(\lambda + y_i + \tau z_i)],\\
\frac{\partial^2 l(\theta)}{\partial p^2} &= -\sum_{i=1}^{N+M}\delta_i/p^2,\\
\frac{\partial^2 l(\theta)}{\partial p \partial \lambda} &= (N+M)/\lambda - \sum_{i=1}^{N}1/(\lambda + x_i) - \sum_{i=N+1}^{N+M}1/(\lambda + y_i + \tau z_i),\\
\frac{\partial^2 l(\theta)}{\partial p \partial \tau} &= -\sum_{i=N+1}^{N+M}z_i/(\lambda + y_i + \tau z_i)],\\
\frac{\partial^2 l(\theta)}{\partial \lambda^2} &= -(N+M)p/\lambda^2 + \sum_{i=1}^{N}(p+\delta_i)/(\lambda + x_i)^2 + \sum_{i=N+1}^{N+M}(p+\delta_i)/(\lambda + y_i + \tau z_i)^2,\\
\frac{\partial^2 l(\theta)}{\partial \lambda \partial \tau} &= \sum_{i=N+1}^{N+M}(p+\delta_i)z_i/(\lambda + y_i + \tau z_i)^2,\\
\frac{\partial^2 l(\theta)}{\partial \tau^2} &= \sum_{i=N+1}^{N+M}[-\delta_i/\tau^2 + (p+\delta_i)z_i^2/(\lambda + y_i + \tau z_i)^2].
\end{aligned}$$

The second derivatives of $nq_n(\theta)$ are given by

$$\partial^2 nq_n(\theta)/\partial\theta_j^2 = -k/\theta_j^2 + \partial^2 l(\theta)/\partial\theta_j^2$$

and

$$\partial^2 nq_n(\theta)/\partial\theta_j\partial\theta_r = \partial^2 l(\theta)/\partial\theta_j\partial\theta_r$$

for $r \neq j$.

The standard large sample approximation to the posterior distribution is that θ is normally distributed with

$$\mathrm{E}(\theta) = \hat{\theta} \quad \text{and} \quad \mathrm{Var}(\theta) = -H^{-1}, \tag{5.22}$$

where $\hat{\theta}$ is the posterior mode and H is the matrix of second derivatives of the log-posterior evaluated at $\theta = \hat{\theta}$. Here, with flat priors, $H = \partial^2 l(\hat{\theta})/\partial\theta\partial\theta'$ and $\hat{\theta}$ is just the MLE.

In the Splus code which follows, `x` and `sx` are the vectors of survival times and status variables (δ_i) for nontransplant patients, and `y`, `z` and `sz` are the times to transplant, survival beyond transplant, and survival status for transplant patients. First the standard large sample approximation to the posterior mean and variance will be obtained, then the approximations based on (5.9).

```
> # x=survival for nontransplant patients; sx= status
> # y=days to transplant
> # z=survival from transplant; sz=status
> x <- c(49,5,17,2,39,84,7,0,35,36,1400,5,34,15,11,2,1,39,8,101,2,148,1,68,31,
+        1,20,118,91,427)
> sx <- c(1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0)
> y <- c(0,35,50,11,25,16,36,27,19,17,7,11,2,82,24,70,15,16,50,22,45,18,4,1,
+        40,57,0,1,20,35,82,31,40,9,66,20,77,2,26,32,13,56,2,9,4,30,3,26,4,
+        45,25,5)
> z <- c(15,3,624,46,127,61,1350,312,24,10,1024,39,730,136,1379,1,836,60,1140,
+        1153,54,47,0,43,971,868,44,780,51,710,663,253,147,51,479,322,442,65,
+        419,362,64,228,65,264,25,193,196,63,12,103,60,43)
> sz <- c(1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,0,0,1,1,1,1,0,0,1,0,1,0,0,1,
+         1,1,0,1,0,1,0,0,1,1,1,0,1,0,0,1,1,0,0,0)
>
> ff <- function(b) { # calculate -log likelihood - input values are log(params)
+   p <- exp(b[1]);  l <- exp(b[2]);  tau <- exp(b[3]);
+   -sum(p*log(l/(l+x))+sx*log(p/(l+x)))-sum(
+     p*log(l/(l+y+tau*z))+sz*log(tau*p/(l+y+tau*z)))
+ }
> ff2 <- function(b) {# - second derivatives of log likelihood
+   p <- exp(b[1]);  l <- exp(b[2]);  tau <- exp(b[3]);
+   v <- matrix(0,3,3)
+   v[1,1] <- sum(c(sx,sz))/p^2
+   w <- y+tau*z
+   N <- length(sx)+length(sz)
+   v[1,2] <- v[2,1] <- -N/l+sum(1/(l+x))+sum(1/(l+w))
+   v[1,3] <- v[3,1] <- sum(z/(l+w))
+   v[2,2] <- N*p/l^2-sum((p+sx)/(l+x)^2)-sum((p+sz)/(l+w)^2)
+   v[2,3] <- v[3,2] <- -sum((p+sz)*z/(l+w)^2)
+   v[3,3] <- sum(sz)/tau^2-sum((p+sz)*(z/(l+w))^2)
+   v
+ }
> w <- nlmin(ff,c(0,0,0),rfc.tol=1e-14)
> exp(w$x) # first order estimate of mean
[1]  0.4342928 21.8721078  0.8135526
> wd <- ff2(w$x)
> sqrt(diag(solve(wd))) # first order estimate of standard deviations
```

```
[1]  0.1101879 10.2539312  0.3322589
>
> wl <- -ff(w$x)  #nd_n
> wsdet <- prod(diag(chol(wd))) # sqrt of det of 2nd derivs of nd_n
>
> # Laplace approx to E(theta_j) and Var(theta_j)
> gg <- function(b) ff(b)-b[j] #-l-log(theta_j)
> gg2 <- function(b) {
+   v <- ff2(b)
+   v[j,j] <- v[j,j]+exp(-2*b[j]) # +1/theta_j^2
+   v
+ }
> gg3 <- function(b) ff(b)-2*b[j] #-l-2log(theta_j)
> gg4 <- function(b) {
+   v <- ff2(b)
+   v[j,j] <- v[j,j]+2*exp(-2*b[j]) # +2/theta_j^2
+   v
+ }
>
> for (j in 1:3) {
+   u <- nlmin(gg,w$x,rfc.tol=1e-14)
+   ul <- -gg(u$x) #nq_n for theta_j
+   ud <- gg2(u$x)
+   usdet <- prod(diag(chol(ud))) # sqrt of det of nq_n
+   pm <- (wsdet/usdet)*exp(ul-wl) #mean
+   u <- nlmin(gg3,w$x,rfc.tol=1e-14)
+   ul <- -gg3(u$x) #nq_n for theta_j^2
+   ud <- gg4(u$x)
+   usdet <- prod(diag(chol(ud))) # sqrt of det of nq_n
+   pvar <- (wsdet/usdet)*exp(ul-wl)-pm^2 #var
+   print(c(pm,sqrt(pvar)))
+ }
[1] 0.4926045 0.1380890
[1] 32.10656 16.08997
[1] 1.0439007 0.4943295
```

Note that there are substantial differences between the Laplace approximations and the first order results (these differences would be smaller, though, if the logs of the parameters were used instead of the parameters themselves). Naylor and Smith (1982), using Gauss-Hermite quadrature methods, determined the posterior means to be .50, 32.5 and 1.04 and the posterior standard deviations to be .14, 16.2 and .47, for p, λ, and τ, so the Laplace approximation is very close here.

The moment generating function method can also be used in this example. It would be possible to give formulas analogous to (5.13) and (5.15) for the multiparameter case. Instead though, the Laplace approximation will be used to approximate the moment generating function, as in (5.10), and numerical differences will be used to approximate the derivatives, as in (5.14) and (5.16).

In the following, `w`, `wd`, `wl`, `ff`, `ff2`, etc., are as defined earlier. The values of s used are of the form ϵ/σ_{jj}, where σ_{jj} is the estimated posterior standard deviation for the jth parameter obtained from the standard large sample first order approximation (by inverting the second derivative matrix). The reason for scaling the differences by $1/\sigma_{jj}$ is that the curvature in the moment generating function is quite different for the different parameters, making different step sizes appropriate.

```
> hh <- function(b) ff(b)-s*exp(b[j]) #-l-s*theta_j
> wdi <- 1/sqrt(diag(solve(wd)))  #1/\sigma_{jj}
> wdi
[1] 9.07540899 0.09752357 3.00970128
> for (j in 1:3) {
+   for (D in c(.1,.01,.001,.0001)) { # actually D=epsilon
+     s <- D*wdi[j]
+     u1 <- nlmin(hh,w$x,rfc.tol=1e-14)
+     if (u1$conve) {
+       u1l <- -hh(u1$x)
+       u1d <- ff2(u1$x)
+       u1sdet <- prod(diag(chol(u1d)))
+       m1 <- (wsdet/u1sdet)*exp(u1l-wl)
+       s <- -s
+       u2 <- nlmin(hh,w$x,rfc.tol=1e-14)
+       if (u2$conve) {
+       u2l <- -hh(u2$x)
+       u2d <- ff2(u2$x)
+       u2sdet <- prod(diag(chol(u2d)))
+       m2 <- (wsdet/u2sdet)*exp(u2l-wl)
+       pm <- (m2-m1)/(2*s)
+       pvar <- log(m1*m2)/s^2
+       print(c(j,D,pm,pvar,sqrt(pvar)))
+       }
+     }
+   }
+ }
[1] 1.00000000 0.10000000 0.50709367 0.01729363 0.13150523
[1] 1.00000000 0.01000000 0.48733590 0.01725193 0.13134660
[1] 1.00000000 0.00100000 0.48714124 0.01672953 0.12934269
[1]  1.0000000  0.0001000  0.4865873 -1.2092876         NA
[1]   2.00000   0.10000  30.99081 196.13113  14.00468
[1]   2.00000   0.01000  30.21429 195.24472  13.97300
[1]   2.00000   0.00100  30.20651 192.81457  13.88577
[1]   2.00000   0.00010  30.20565 516.53107  22.72732
[1] 3.0000000 0.1000000 1.0338185 0.1916737 0.4378055
[1] 3.0000000 0.0100000 1.0077132 0.1906006 0.4365783
[1] 3.0000000 0.0010000 1.0074374 0.1997018 0.4468801
[1]  3.000000  0.000100  1.007411 -0.386718        NA
```

From these calculations $.001 < \epsilon < .01$ seems to work well in this example. The values in this range are stable, and not too different from the results obtained earlier. A more careful analysis appears to indicate the values of the difference approximations are close to the correct derivatives of the moment generating function approximation. In this example the moment generating function approach is a little less accurate than the direct Laplace approximation given previously.

In discussing finite differences to approximate derivatives in connection with optimization methods, it was suggested that $\epsilon_m^{1/2}$, where ϵ_m is the relative machine precision, would often be a reasonable choice for the step size in the finite difference. That recommendation assumed that the error in evaluating the function was $O(\epsilon_m)$. Here the errors may be larger, and $\epsilon_m^{1/2} \doteq 10^{-8}$ appears to be too small. The value $\epsilon_m^{1/2}$ was also only recommended for the finite difference approximation to the first derivative. A similar analysis for the finite difference approximation to the second derivative suggests a value of $\epsilon_m^{1/4} \doteq 10^{-4}$ for that case. Again here $\epsilon_m^{1/4}$ is too small to give good results for approximating the second derivative.

This example also illustrates the general difficulty with using numerical differences to approximate derivatives. The values obtained can be quite sensitive to the step size used, and the appropriate step size tends to be heavily dependent on the particular problem. If the step size is too large, then the slope over the step may be a poor approximation to the derivative at the point of interest, while if the step is too small, the difference in the function values can be the same order as the accuracy of the function evaluations, leading to meaningless results (like the negative variances with $\epsilon = .0001$ above). Part of the error in the function evaluations here is the accuracy with which the maximizing values of the integrands are determined.

Another way to estimate the derivatives of the approximation to the moment generating function is to calculate it at a number of values of s, fit a parametric curve to the values, and use the derivatives of the fitted curve to estimate the derivatives. In particular, if we fit a qth degree polynomial $r(s) = \beta_0 + \beta_1 s + \cdots + \beta_q s^q$, then $r'(0) = \beta_1$ and $r''(0) = 2\beta_2$. The degree of the polynomial needs to be chosen large enough to model the function well over the range considered, but small enough to smooth out inaccuracies in the evaluation of the function. Below this idea is applied to estimating the mean and variance of the posterior distribution of p from the moment generating function. The method in this example does not seem very sensitive to the degree of the polynomial or the width of the interval used. (There are better ways to fit the polynomials, but the method used below was adequate to illustrate the idea.)

```
> j <- 1
> D <- .1
> ss <- seq(-D,D,length=11)*wdi[j]
> mm <- ss
> for (i in 1:length(ss)) {
+   s <- ss[i]
+   u1 <- nlmin(hh,w$x,rfc.tol=1e-14)
+   u1l <- -hh(u1$x)
+   u1d <- ff2(u1$x)
+   u1sdet <- prod(diag(chol(u1d)))
+   mm[i] <- (wsdet/u1sdet)*exp(u1l-wl)
+ }
```

```
> a <- coef(lm(mm~ss+ss^2+ss^3+ss^4+ss^5+ss^6+ss^7+ss^8+ss^9+ss^10))
> c(a[2],sqrt(2*a[3]-a[2]^2))
        ss   I(ss^2)
 0.4871398 0.1313269
> a <- coef(lm(mm~ss+ss^2+ss^3+ss^4+ss^5+ss^6+ss^7+ss^8))
> c(a[2],sqrt(2*a[3]-a[2]^2))
        ss   I(ss^2)
 0.4871385 0.1313297
> a <- coef(lm(mm~ss+ss^2+ss^3+ss^4+ss^5+ss^6))
> c(a[2],sqrt(2*a[3]-a[2]^2))
        ss   I(ss^2)
 0.4871403 0.1313767
> a <- coef(lm(mm~ss+ss^2+ss^3+ss^4))
> c(a[2],sqrt(2*a[3]-a[2]^2))
       ss   I(ss^2)
 0.487041 0.1315763
> a <- coef(lm(mm~ss+ss^2))
> c(a[2],sqrt(2*a[3]-a[2]^2))
        ss    I(ss^2)
 0.5013096 0.09625772
> j <- 1
> D <- .01
> ss <- seq(-D,D,length=11)*wdi[j]
> mm <- ss
> for (i in 1:length(ss)) {
+   s <- ss[i]
+   u1 <- nlmin(hh,w$x,rfc.tol=1e-14)
+   u1l <- -hh(u1$x)
+   u1d <- ff2(u1$x)
+   u1sdet <- prod(diag(chol(u1d)))
+   mm[i] <- (wsdet/u1sdet)*exp(u1l-wl)
+ }
> a <- coef(lm(mm~ss+ss^2+ss^3+ss^4+ss^5+ss^6+ss^7+ss^8+ss^9+ss^10))
> c(a[2],sqrt(2*a[3]-a[2]^2))
        ss   I(ss^2)
 0.4871405 0.1268419
> a <- coef(lm(mm~ss+ss^2+ss^3+ss^4+ss^5+ss^6+ss^7+ss^8))
> c(a[2],sqrt(2*a[3]-a[2]^2))
       ss   I(ss^2)
 0.487139 0.1305032
> a <- coef(lm(mm~ss+ss^2+ss^3+ss^4+ss^5+ss^6))
> c(a[2],sqrt(2*a[3]-a[2]^2))
        ss   I(ss^2)
 0.4871373 0.1315923
> a <- coef(lm(mm~ss+ss^2+ss^3+ss^4))
> c(a[2],sqrt(2*a[3]-a[2]^2))
```

```
      ss   I(ss^2)
 0.48714 0.1310105
> a <- coef(lm(mm~ss+ss^2))
> c(a[2],sqrt(2*a[3]-a[2]^2))
        ss   I(ss^2)
 0.4872798 0.1309879
```

In the second case, with a smaller interval, the 10th degree polynomial overfits noise in the function evaluations, while in the first case, with a larger interval, the quadratic fit is not flexible enough to give a good fit over the range of the points, and the estimated curvature at 0 is distorted by the influence of distant points.

5.4 Exercises

Exercise 5.1

1. The gamma function is defined by

$$\Gamma(a) = \int_0^\infty u^{a-1} \exp(-u)\, du$$

(recall that for integers $\Gamma(n+1) = n!$). Apply the Laplace approximation to this formula and show that the result is Stirling's formula:

$$\Gamma(a+1) \doteq (2\pi)^{1/2} a^{a+1/2} \exp(-a).$$

2. Suppose that X has a gamma distribution with density proportional to $x^{a-1}\exp(-\lambda x)$. Recall that $\mathrm{E}(X^k) = \Gamma(k+a)/[\Gamma(a)\lambda^k]$. Apply (5.7) to the ratio

$$\int_0^\infty x^{k+a-1} \exp(-\lambda x)\, dx \Big/ \int_0^\infty x^{a-1} \exp(-\lambda x)\, dx$$

to find the Laplace approximation to $\mathrm{E}(X^k)$. Show that the only difference between the Laplace approximation and the exact formula is that in the Laplace approximation the gamma functions have been replaced by Stirling's approximation.

3. Use (5.10) to find the Laplace approximation to the moment generating function of the distribution in (b) (again pretend you do not know the normalizing constant for the density). Show that in this case the approximation is exact.

Exercise 5.2 The standard large sample approximation usually performs better if the parameters are transformed to a reasonably symmetric scale first. In Example 5.1, instead of (p, λ, τ), suppose the parameters $(\log(p), \log(\lambda), \log(\tau))$ are used.

1. Find the large sample approximations for the posterior means and standard deviations of $(\log(p), \log(\lambda), \log(\tau))$. The data can be found in the file `stanford.s`.

2. Since the distributions of the new parameters have mass on negative values, approximation (5.7) is not appropriate. Apply the Laplace approximation to the moment generating function, as in (5.10), to approximate the posterior means and standard deviations.

Exercise 5.3 In Example 5.3, plot the integrand $x^{k+\alpha-1}(1-x)^{\beta-1}$ for $k = 0, 1, 2$, for (a) $p = \alpha/(\alpha+\beta) = .5$ and $\gamma = \alpha+\beta = 3, 5, 10, 20, 40$, and (b) $p = .1$ and $\gamma = 11, 20, 40$.

Exercise 5.4 The following questions refer to Example 5.2. The integrand of Q_i in (5.3) can be written as the product of

$$A_i(\epsilon) = \exp[y_{i1}(\theta+\epsilon)][E_{i1}\exp(\theta+\epsilon) + E_{i2}]^{-y_{i1}-y_{i2}}$$

and

$$B(\epsilon) = \sigma^{-1}\phi(\epsilon/\sigma).$$

There is not an explicit solution for the maximizing value $\hat{\epsilon}_i$ of the integrand $A_i(\epsilon)B(\epsilon)$.

1. Give a formula for the Laplace approximation to Q_i in terms of $\hat{\epsilon}_i$. (Give explicit formulas except for the maximizing value $\hat{\epsilon}_i$.)
2. Using the result of part 1, give a formula for the approximate log-likelihood $l(\theta, \sigma)$.
3. Give formulas for the scores based on the approximate likelihood in part 2, keeping in mind that $\hat{\epsilon}_i$ is a function of θ and σ. (Express the scores in terms of the $\hat{\epsilon}_i$ and their derivatives).
4. Using reasoning similar to that leading to (5.11), give formulas for $\partial\hat{\epsilon}_i/\partial\theta$ and $\partial\hat{\epsilon}_i/\partial\sigma$.
5. Exact calculation of the approximate likelihood and scores (and information, if that is also calculated) requires iterative searches to compute $\hat{\epsilon}_i$ for each term in the likelihood at each set of parameter values where these quantities are evaluated. Since this is a lot of iterative searches, Liu and Pierce (1993) proposed an approximation to $\hat{\epsilon}_i$, which is easier to compute. There approximation was based noting that each of the factors $A_i(\epsilon)$ and $B(\epsilon)$ has an explicit maximizing value of ϵ, say $\hat{\epsilon}_{ia}$ and $\hat{\epsilon}_b$, and approximating $\hat{\epsilon}_i$ with a weighted average of $\hat{\epsilon}_{ia}$ and $\hat{\epsilon}_b$. (Liu and Pierce also suggest using a one-step Newton update from this weighted average.)
   (a) Give formulas for $\hat{\epsilon}_{ia}$ and $\hat{\epsilon}_b$.
   (b) There are various possibilities for the weights in the weighted average. Roughly, A_i could be thought of as a likelihood for ϵ and B_i as a prior, in which case it would roughly be appropriate to weight the two components proportionately to the information in each component (where information refers to the usual statistical sense of the negative of the second derivative of the log of the density or likelihood—this is more or less weighting inversely proportional to the variances of the two components). This leads to the formula
   $$\hat{\epsilon}_i \doteq (i_a\hat{\epsilon}_{ia} + i_b\hat{\epsilon}_b)/(i_a + i_b),$$
   where $i_a = -\partial^2 \log[A_i(\hat{\epsilon}_{ia})]/\partial\epsilon^2$ and $i_b = -\partial^2 \log[B_i(\hat{\epsilon}_{ib})]/\partial\epsilon^2$. Give a formula for this approximation to $\hat{\epsilon}_i$.
6. Suppose $y_{i1} = 100$, $y_{i2} = 205$, $E_{i1} = 85$, $E_{i2} = 200$. Plot the integrand $A_i(\epsilon)B(\epsilon)$ for combinations of $\theta = 0, \log(1.25), \log(1.5)$ and $\sigma^2 = .1, 1, 10$. Which combinations are more concentrated?

5.5 References

Barndorff-Nielsen OE and Cox DR (1989). *Asymptotic Techniques for Use in Statistics*. London: Chapman and Hall.

Liu Q and Pierce DA (1993). Heterogeneity in Mantel-Haenszel-type models. *Biometrika*, **80**:543–556.

Naylor JC and Smith AFM (1982). Applications of a method for the efficient computation of posterior distributions. *Applied Statistics*, **31**:214–225.

Tierney L and Kadane JB (1986). Accurate approximations for posterior moments and marginal densities. *Journal of the American Statistical Association*, **81**:82–86.

Tierney L, Kass RE, and Kadane JB (1989). Fully exponential Laplace approximations to expectations and variances of nonpositive functions. *Journal of the American Statistical Association*, **84**:710–716.

Turnbull BW, Brown BW and Hu M (1974). Survivorship analysis of heart transplant data. *Journal of the American Statistical Association*, **69**:74–80.

Chapter 6

Quadrature Methods

This section of the course is concerned with the numerical evaluation of integrals that do not have closed form antiderivatives. The main topic is numerical quadrature rules. Intractable integrals occur in many different contexts in statistics, including in Bayesian inference problems, nonlinear mixed effects models, measurement error models, and missing data problems.

Initially standard methods for one-dimensional integrals, such as the trapezoidal rule and Simpson's rule, will be considered, followed by a discussion of Gaussian quadrature rules in the one-dimensional case, and then by extensions to multi-dimensional integrals.

In addition to the methods discussed below, there have been special algorithms developed for a variety of problems that are encountered frequently in statistics, such as computing the CDFs for standard probability distributions. Many of these approximations are based on series expansions, continued fraction expansions, or rational function approximations. Some examples are given in Chapters 2 and 3 of Lange (1999) and Chapter 6 of Press *et. al.* (1992). The Statistical Algorithms section of the journal *Applied Statistics* gives many useful algorithms of this type, and source code for many of these is available in STATLIB (`http://lib.stat.cmu.edu`).

6.1 Newton-Cotes Rules

Consider the problem of evaluating

$$\int_{x_0}^{x_1} f(x)\,dx.$$

If the integrand $f(x)$ is expanded in a Taylor series about $m = (x_0 + x_1)/2$, then

$$f(x) = f(m) + \sum_{j=1}^{k-1} f^{(j)}(m)(x-m)^j/j! + R_k(x), \tag{6.1}$$

where $R_k(x) = f^{(k)}(m^*(x))(x-m)^k/k!$ for some $m^*(x)$ between x and m, and $f^{(j)}(x) = d^j f(x)/dx^j$. Assume k is even. Since $x_1 - m = m - x_0 = (x_1 - x_0)/2$, integrating the Taylor series gives the midpoint formula

$$\int_{x_0}^{x_1} f(x)\,dx \;=\; f(m)(x_1-x_0) + \sum_{j=1}^{k/2-1} \frac{f^{(2j)}(m)}{(2j+1)!}[(x_1-m)^{2j+1} - (x_0-m)^{(2j+1)}] + \int_{x_0}^{x_1} R_k(x)\,dx$$

$$= f(m)(x_1 - x_0) + \sum_{j=1}^{k/2-1} f^{(2j)}(m)2^{-2j}(x_1 - x_0)^{2j+1}/(2j+1)! + \int_{x_0}^{x_1} R_k(x)\,dx. \quad (6.2)$$

Also, direct substitution in (6.1) gives

$$[f(x_0) + f(x_1)]/2 = f(m) + \sum_{j=1}^{k/2-1} f^{(2j)}(m)2^{-2j}(x_1 - x_0)^{2j}/(2j)! + [R_k(x_1) + R_k(x_0)]/2.$$

Multiplying the last expression by $(x_1 - x_0)$ and subtracting from (6.2) gives

$$\int_{x_0}^{x_1} f(x)\,dx - \frac{f(x_0) + f(x_1)}{2}(x_1 - x_0) = -\sum_{j=1}^{k/2-1} \frac{f^{(2j)}(m)(x_1 - x_0)^{2j+1} j}{2^{2j-1}(2j+1)!} + \tilde{R}_k, \quad (6.3)$$

where $\tilde{R}_k$ is the difference in the remainder terms from the two expressions, which satisfies

$$|\tilde{R}_k| \le \sup_{x\in[x_0,x_1]} |f^{(k)}(x)| \frac{(x_1 - x_0)^{k+1}(2k+3)}{2^k(k+1)!}.$$

With $k = 2$, (6.3) gives

$$\int_{x_0}^{x_1} f(x)\,dx = \frac{f(x_0) + f(x_1)}{2}(x_1 - x_0) + O((x_1 - x_0)^3 \sup|f''(x)|). \quad (6.4)$$

Dividing $[a, b]$ into n intervals $[x_{j-1}, x_j]$ of equal size, so that $x_j = a + j(b - a)/n$, and applying (6.4) to each interval, gives the approximation

$$\int_a^b f(x)\,dx = \left([f(a) + f(b)]/2 + \sum_{j=1}^{n-1} f(x_j) \right) (b - a)/n + O((b - a)^3 n^{-2} \sup|f''(x)|), \quad (6.5)$$

which is known as the trapezoidal rule.

A substantially better approximation can be obtained with only slightly more work. Adding 2 times (6.2) to (6.3), with $k = 4$ in both cases, gives

$$3\int_{x_0}^{x_1} f(x)\,dx - \frac{f(x_0) + f(x_1)}{2}(x_1 - x_0) = 2f(m)(x_1 - x_0) + O((x_1 - x_0)^5 \sup|f^{(4)}(x)|),$$

since the $f''(m)$ terms cancel, so

$$\int_{x_0}^{x_1} f(x)\,dx = \frac{1}{6}[f(x_0) + 4f(m) + f(x_1)](x_1 - x_0) + O\{(x_1 - x_0)^5 \sup|f^{(4)}(x)|\}. \quad (6.6)$$

This formula can also be obtained by fitting a quadratic approximation to f through the points $(x_0, f(x_0))$, $(m, f(m))$, and $(x_1, f(x_1))$. It is interesting that this formula is exact when f is a cubic function (since the error depends only on the 4th derivative), even though it is obtained from a quadratic approximation. Also note that if T_1 is the two point trapezoidal approximation and T_2 is the 3 point trapezoidal approximation, then the approximation in (6.6) can be obtained from $(4T_2 - T_1)/3$.

Again dividing $[a, b]$ into n intervals (where n is even), with $x_j = a + j(b-a)/n$, and applying (6.6) to each of the $n/2$ intervals $[x_{2j}, x_{2j+2}]$, so $m_j = x_{2j+1}$, gives

$$\begin{aligned}\int_a^b f(x)\,dx &= [f(a) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(b)]\frac{(b-a)}{3n} \\ &\quad + O((b-a)^5 n^{-4} \sup|f^{(4)}(x)|). \end{aligned} \tag{6.7}$$

This formula is known as Simpson's rule. The alternating multipliers of 4 and 2 are a curiosity of this approach. In Section 4.1 of Press *et. al.* (1992), it is noted that applying cubic interpolants to groups of 4 points leads to the formula

$$\begin{aligned}\int_a^b f(x)\,dx &= [3\{f(a) + f(b)\}/8 + 7\{f(x_1) + f(x_{n-1})\}/6 + 23\{f(x_2) + f(x_{n-2})\}/24 + \\ &\quad f(x_3) + \cdots + f(x_{n-3})](b-a)/n + O(n^{-4}).\end{aligned}$$

This has the same order of error as Simpson's rule, but the form is similar to the trapezoidal rule, except for the weights given to the points near the end of the interval.

The trapezoidal rule and Simpson's rule are part of a general family of quadrature rules based on approximating the function with interpolating polynomials over an equally spaced grid of points. These quadrature rules are generally known as Newton-Cotes formulas.

The trapezoidal rule and Simpson's rule above are known as closed formulas, since they require evaluation of the function at the endpoints of the interval. There are analogous open formulas based on (6.2), that do not require evaluating the function at the endpoints, which could be used if the function value is not defined (except in a limiting sense) at an endpoint of the interval; see Sections 4.1 and 4.4 of Press *et. al.* (1992), and (6.9), below.

Since analytic bounds on the higher order derivatives of the integrand can be difficult to obtain, the formal error bounds above are not often very useful. In practice these rules are often applied by evaluating the rules for an increasing sequence of values of n. Since the rules use equally spaced points, if the number of intervals is doubled at each step, then the function evaluations from the previous steps can be reused in the current step; only the function values at the midpoints of the old intervals need to be evaluated. The sequence is usually terminated when the change from one step to the next is small enough. This does not guarantee a good approximation to the integral, since it is possible to construct functions where doubling the number of intervals at some step will give the same value, so the algorithm will terminate, but where the integral is poorly approximated (eventually adding more intervals would lead to a better approximation, if the function is smooth, but the algorithm could terminate before then).

As noted above, Simpson's rule can be obtained from two successive evaluations of the trapezoidal rule. This leads to the following stepwise algorithm for calculating Simpson's rule for evaluating $\int_a^b f(x)\,dx$:

Algorithm 6.1
Iterative Evaluation of Simpson's Rule

1. Set $T_1 = (b-a)(f(b) + f(a))/2$, $h = (b-a)/2$, $T_2 = T_1/2 + f((a+b)/2)h$, and $n = 2$.

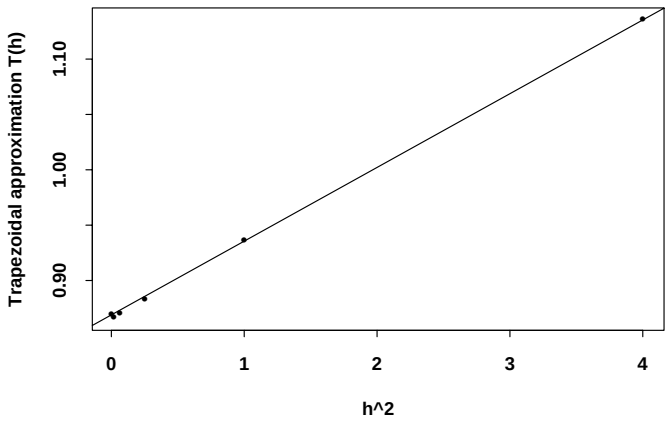

Figure 6.1: The trapezoidal rule (dots) for evaluating $\int_0^2 \exp(-x)\,dx$, for $n = 1, 2, 4, 8, 16$ intervals ($h^2 = 4, 1, 1/4, 1/16, 1/64$). The extrapolation to 0 of the line through the first two trapezoidal approximations ($h^2 = 4, 1$) gives Simpson's rule with 3 points (dot at $h^2 = 0$).

2. Set $S_1 = (4T_2 - T_1)/3$ and $T_1 = T_2$.
3. Set $h = h/2$ and then evaluate $T_2 = T_1/2 + (\sum_{i=1}^{n} f(a + (2i-1)h))\,h$.
4. Set $S_2 = (4T_2 - T_1)/3$.
5. If $|S_2 - S_1| < \epsilon(|S_1| + \delta)$, where $\delta > 0$, then return S_2 as the approximate value of the integral; otherwise set $n = 2n$, $T_1 = T_2$ and $S_1 = S_2$ and go to step 3.

At steps 3 and 4, T_1 and T_2 are the previous and current evaluations of the trapezoidal rule, and S_1 and S_2 are the previous and current evaluations of Simpson's rule.

6.1.1 Romberg Integration

There is another interpretation of the Simpson's rule formula $S = (4T_2 - T_1)/3$ used in Algorithm 6.1. If the integrand $f(x)$ is a sufficiently smooth function, and if $T(h)$ is the trapezoidal approximation with interval length h, then the exact value of the integral is given by $T(0) = \lim_{h \downarrow 0} T(h)$. Think of T_1 and T_2 as being the points $(h^2, T(h))$ and $(h^2/4, T(h/2))$, and use linear extrapolation to $(0, T(0))$ to estimate $T(0)$. The result is $T(0) \doteq (4T(h/2) - T(h))/3$, giving the formula for Simpson's rule. As was seen above, this difference cancels the leading error term in the expansion (6.3). The reason for using the square of interval width instead of the interval width itself as the ordinate in the extrapolation is because the terms in the error expansion (6.3) only depend on powers of h^2.

This is illustrated if Figure 6.1. For $\int_0^2 \exp(-x)\,dx$ (used in Example 6.1, below), the first 5 steps in sequential evaluation of the trapezoidal rule (doubling the number of intervals at each step) are plotted against h^2, as is the linear extrapolation to 0 from the first 2 steps. This extrapolant gives

Simpson's rule based on the same set of points, and substantially improves on the trapezoidal approximation. The code used for generating Figure 6.1 is as follows.

```
> f <- function(x) exp(-x)
> a <- 0
> b <- 2
> h <- b-a
> n <- 1
> TR <- mean(f(c(a,b))) * h
> for (i in 1:4) {
+   h <- c(h[1]/2,h)
+   TR <- c(TR[1]/2+sum(f(seq(a+h[1],b-h[1],length=n)))*h[1],TR)
+   n <- n*2
+ }
> h <- rev(h)^2
> TR <- rev(TR)
> plot(h,TR,xlab='h^2',ylab='Trapezoidal approximation T(h)')
> b <- (TR[1]-TR[2])/(h[1]-h[2])
> abline(a=TR[1]-b*h[1],b=b)
> points(0,TR[1]-b*h[1])
> TR[1]-b*h[1] # extrapolation to 0
[1] 0.868951
> (4*TR[2]-TR[1])/3 # 3 point Simpson's rule
[1] 0.868951
```

Since a linear extrapolation from two points substantially decreases the error in the trapezoidal rule, it might be hoped that a quadratic extrapolation from three successive evaluations would lead to further improvement. This turns out to be the case. If the intervals are again split in two, the point $(h^2/16, T(h/4))$ results (the point at $h^2 = 1/4$ in Figure 6.1. Fitting a quadratic to the two earlier points and this new point, and extrapolating to $(0, T(0))$, will cancel the h^4 error term and give an approximation accurate to h^6. Adding more points and and using higher order extrapolations continues to cancel higher order terms in the error. This suggests evaluating $\int_a^b f(x)\,dx$ by sequentially evaluating trapezoidal rules as in Algorithm 6.1, and after each evaluation performing a polynomial extrapolation as described above. This is known as Romberg integration, and is implemented in the Splus function `rint()` below. The polynomial extrapolation is done in the function `polint()`, which is adapted from the routine of the same name in Press *et. al.* (1992).

```
rint <- function(f,a,b,eps=1.e-5,mk=20,dm=5,prt=F) { # Romberg integration
# f= univariate function to be integrated (given a vector of values
# x, f(x) must return a vector giving the values of f at x), a to b is
# interval of integration (finite).  stops when relative change < eps
# mk=max number Romberg iterations, dm=max degree in the interpolation
#   alternative interpolation:  {u <- poly(c(0,h2),min(k,dm));
#   R1 <- sum(lm(Q~u[-1,])$coef*c(1,u[1,]))} (slower and less accurate)
  h <- b-a
```

```
  h2 <- h^2
  Q <- mean(f(c(a,b)))*h
  R0 <- Q
  j <- 1
  for (k in 1:mk){
    h <- h/2
    h2 <- c(h2,h2[j]/4)
    i <- sum(f(seq(a+h,b-h,length=2^(k-1))))
    Q <- c(Q,Q[j]/2+h*i)
    if (k>dm) {h2 <- h2[-1]; Q <- Q[-1]}
    R1 <- polint(h2,Q,0)
    j <- length(Q)
    if(prt) print(c(k,trapezoid=Q[j],Simpson=(4*Q[j]-Q[j-1])/3,Romberg=R1))
    if (abs(R1-R0)<eps*(abs(R1)+1e-8)) break
    R0 <- R1
  }
  R1
}
polint <- function(xa,ya,x){ #polynomial interpolation
 # given a vector of points xa and ya=f(xa) for some function f,
 # computes the length(xa)-1 degree polynomial interpolant at x
 # based on numerical recipes polint
  n <- length(xa)
  ns <- 1
  cc <- d <- ya
  u <- abs(x-xa)
  dif <- min(u)
  ns <- match(dif,u)
  y <- ya[ns]
  ns <- ns-1
  for (m in 1:(n-1)) {
    ii <- 1:(n-m)
    ho <- xa[ii]-x
    hp <- xa[ii+m]-x
    w <- cc[ii+1]-d[ii]
    den <- w/(ho-hp)
    d[ii] <- hp*den
    cc[ii] <- ho*den
    if (2*ns<n-m) {
      dy <- cc[ns+1]
    } else {
      dy <- d[ns]
      ns <- ns-1
    }
    y <- y+dy
  }
```

```
  y
}
```

The following is a simple example using these functions.

Example 6.1 Again consider numerically evaluate $\int_0^2 \exp(-x)\,dx$. The true value is $1 - \exp(-2)$. Note that the function `f()` has to be able to evaluate the integrand at a vector of values in a single call. As part of the calculations for Romberg integration, it is trivial to evaluate the stepwise trapezoidal and Simpson's approximations as well.

```
> # simple example
> i <- 0
> f <- function(x) { # integrand
+   assign('i',i+length(x),where=1)
+   exp(-x)
+ }
> u <- rint(f,0,2,eps=1.e-7,prt=T)
   trapezoid  Simpson  Romberg
 1 0.9355471 0.868951 0.868951
   trapezoid   Simpson   Romberg
 2  0.882604 0.8649562 0.8646899
   trapezoid   Simpson   Romberg
 3 0.8691635 0.8646833 0.8646648
   trapezoid   Simpson   Romberg
 4 0.8657903 0.8646659 0.8646647
> u
[1] 0.8646647
> i
[1] 17
> truval <- 1-exp(-2)
> u-truval
[1] 1.548361e-11
```

□

6.1.2 Singularities

The methods above require the range of integration to be a finite interval. Many integrals encountered in statistics have infinite limits. These can often be dealt with through an appropriate transformation. For example, to evaluate

$$\int_1^\infty \exp(-u)u^{-1/2}\,du \tag{6.8}$$

the transformations $x = \exp(-u)$ or $x = 1/u$ could be considered. The first leads to

$$\int_0^{1/e} \{-\log(x)\}^{-1/2}\,dx,$$

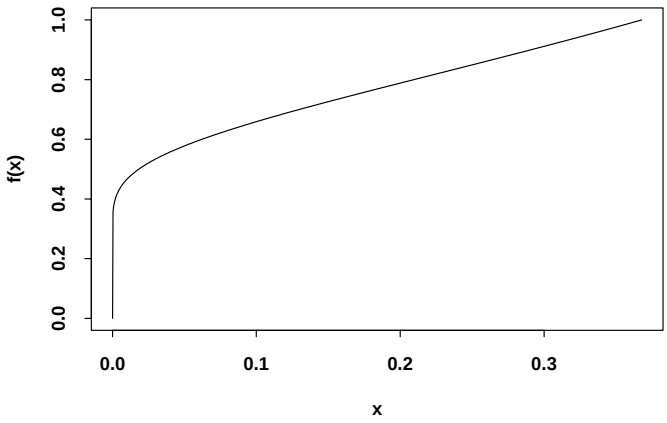

Figure 6.2: $f(x) = \{-\log(x)\}^{-1/2}$, the integrand following the first transformation

and the second to

$$\int_0^1 \exp(-1/x)x^{-3/2}\,dx.$$

Note that both transformations have given an integrand that cannot be evaluated at 0, although in both cases the limit of the integrand as $x \to 0$ is 0. Thus the singularity at 0 can be removed by defining the function to be 0 at 0, or an open formula (see Sections 4.1 and 4.4 of Press *et. al.* (1992), and (6.9) below) or Gaussian quadrature methods (see Section 6.2, below), which do not require evaluating the integrand at the endpoints of the interval, could be used. The integrand after the first transformation has a very steep slope near 0 (see Figure 6.2), which could require a large number of intervals to get a good approximation.

Applying the Romberg algorithm to both transformations gives the following.

```
> # int_1^Inf u^(-.5)exp(-u)
> # transformation to a finite interval
> # x=exp(-u)
> f <- function(x) {
+   assign('i',i+length(x),where=1)
+   ifelse(x==0,0,1/sqrt(-log(x)))
+ }
> i <- 0
> u <- rint(f,0,exp(-1),prt=T)
   trapezoid   Simpson   Romberg
 1 0.2333304 0.2497939 0.2497939
   trapezoid   Simpson   Romberg
 2 0.2572495 0.2652226 0.2662511
   trapezoid   Simpson   Romberg
 3 0.2686219 0.2724127 0.2729975
   trapezoid Simpson   Romberg
```

```
 4 0.2739905 0.27578 0.2760659
   trapezoid   Simpson   Romberg
 5 0.2765225 0.2773666 0.2775027
   trapezoid   Simpson   Romberg
 6 0.2777193 0.2781182 0.2781827
   trapezoid   Simpson   Romberg
 7 0.2782869 0.2784761 0.2785068
   trapezoid   Simpson   Romberg
 8 0.2785571 0.2786471 0.2786618
   trapezoid   Simpson   Romberg
 9 0.2786861 0.2787292 0.2787362
    trapezoid   Simpson  Romberg
 10  0.278748 0.2787686 0.278772
    trapezoid   Simpson   Romberg
 11 0.2787778 0.2787877 0.2787893
    trapezoid   Simpson   Romberg
 12 0.2787921 0.2787969 0.2787977
    trapezoid   Simpson   Romberg
 13 0.2787991 0.2788014 0.2788017
    trapezoid   Simpson   Romberg
 14 0.2788024 0.2788035 0.2788037
> truval <- (1-pgamma(1,shape=.5))*gamma(.5)
> c(i,u,u-truval)
[1]  1.638500e+04  2.788037e-01 -1.868625e-06
>
> # can split into two parts
> i <- 0
> u1 <- rint(f,.01,exp(-1))
> i
[1] 65
> u2 <- rint(f,0,.01)
> c(i,u1+u2,u1,u2)
[1] 3.283400e+04 2.788055e-01 2.745401e-01 4.265423e-03
> u1+u2-truval
[1] -8.471474e-08
>
> # 2nd transformation to a finite interval
> # x=1/u
> f <- function(x) {
+   assign('i',i+length(x),where=1)
+   ifelse(x==0,0,exp(-1/x)*x^(-1.5))
+ }
> i <- 0
> u <- rint(f,0,1,prt=T)
   trapezoid   Simpson   Romberg
 1 0.2833629 0.3165039 0.3165039
```

```
   trapezoid   Simpson   Romberg
 2 0.2797713 0.2785741 0.2760454
   trapezoid   Simpson   Romberg
 3 0.2784363 0.2779914 0.2779828
   trapezoid   Simpson   Romberg
 4 0.2787456 0.2788487 0.2789246
   trapezoid   Simpson   Romberg
 5 0.2787906 0.2788056 0.2788005
   trapezoid   Simpson   Romberg
 6 0.2788018 0.2788056 0.2788056
   trapezoid   Simpson   Romberg
 7 0.2788046 0.2788056 0.2788056
> c(i,u,u-truval)
[1]  1.290000e+02  2.788056e-01 -1.092346e-10
```

As expected, the second transformation worked much better.

A semi-automated method for dealing with endpoint singularities and semi-infinite intervals can be given by using 'open' Newton-Cotes formulas. An open formula does not require evaluating the function at the endpoints of the interval. An open trapezoidal rule, similar to (6.5), can be obtained by applying the midpoint formula (6.2) to successive intervals, giving

$$\int_a^b f(x)\,dx = \left(\sum_{i=1}^{n} f(x_i)\right)(b-a)/n + O\left(\frac{(b-a)^3}{n^2}\right), \tag{6.9}$$

where $x_i = a + (i - 1/2)(b-a)/n$. Also, higher order error terms again involve only even powers of n^{-1}. Thus a Romberg polynomial extrapolation algorithm can be applied, that will reduce the error by an order of n^{-2} at each step. This is implemented in the S function `rinto()` below. This function triples the number of points at each step, because tripling allows both reusing the points already computed and maintaining the relative spacing in (6.9), while doubling would not.

The function `rinto()` also automates the transformation $x = 1/u$ for semi-infinite intervals, by redefining the function appropriately. Note that `rinto()` only allows intervals of the form $(-\infty, -b)$ $(b > 0)$, (a, ∞) $(a > 0)$, or (a, b) $(-\infty < a < b < \infty)$. Other intervals of integration can be handled by making multiple calls over separate intervals. For example, $\int_{-\infty}^{\infty} f = \int_{-\infty}^{-a} f + \int_{-a}^{b} f + \int_{b}^{\infty} f$.

```
rinto <- function(f,aa,bb,eps=1e-5,mk=13,dm=5,prt=F) {
# Romberg integration using the open trapezoidal formula
# triples the number of points at each step
# f= univariate function to be integrated (given a vector of values
# x, f(x) must return a vector giving the values of f at x), (aa,bb) is
# interval of integration; only aa=-Inf and bb<0, aa>0 and bb=Inf, or
# aa<bb both finite, are allowed. Stops when relative change < eps
# mk=max number Romberg iterations, dm=max degree in the interpolation
  if (aa >= bb) stop('aa must be < bb')
  if (is.inf(aa)) {
```

```
    if (bb >= 0) stop('bb must be <0 when aa=-Inf')
    ff <- function(x,f) f(1/x)/x^2
    b <- 0
    a <- 1/bb
  } else if (is.inf(bb)) {
    if (aa <= 0) stop('aa must be >0 when bb=Inf')
    ff <- function(x,f) f(1/x)/x^2
    b <- 1/aa
    a <- 0
  } else {
    ff <- function(x,f) f(x)
    a <- aa
    b <- bb
  }
  h <- b-a
  h2 <- h^2
  R0 <- Q <- h*ff((a+b)/2,f)
  j <- 1
  for (k in 1:mk){
    h2 <- c(h2,h2[j]/9)
    x <- c(seq(a+h/6,b,by=h),seq(a+5*h/6,b,by=h))
    i <- sum(ff(x,f))*h
    Q <- c(Q,(Q[j]+i)/3)
    h <- h/3
    if (k>dm) {h2 <- h2[-1]; Q <- Q[-1]}
    R1 <- polint(h2,Q,0)
    j <- length(Q)
    if(prt) print(c(k,trapezoid=Q[j],Romberg=R1))
    if (abs(R1-R0)<eps*(abs(R1)+1e-8)) break
    R0 <- R1
  }
  R1
}
```

Applying this function to (6.8) gives the following.

```
> f <- function(x) {assign('i',i+length(x),where=1); exp(-x)/sqrt(x)}
> i <- 0
> print(u <- rinto(f,1,Inf))
[1] 0.2788056
> c(i,u-truval)
[1]  2.430000e+02 -1.319426e-10
```

This used about twice as many function evaluations (243 versus 129) as the closed rule Romberg integration required on this problem.

Another type of problem is when the integrand is infinite at one or both endpoints. In this case use of an open formula or Gaussian quadrature methods (below) can be essential.

6.2 Gaussian Quadrature

The basic form of a Gaussian quadrature rule is

$$\int_a^b W(x)f(x)\,dx \doteq \sum_{j=1}^{n} \omega_j f(x_j), \tag{6.10}$$

where both the weights ω_j and the abscissas x_j are determined to give an accurate approximation. In the Newton-Cotes rules with a grid of n equally spaced points, it is possible to give a formula that is exact when the integrand is a polynomial of degree $n-1$. In Gaussian quadrature, by varying both the weights and abscissas, formulas are obtained which are exact when $f(x)$ is a polynomial of degree $2n-1$. Another advantage of Gaussian quadrature methods is that the weight function $W(x)$ in the integrand can be chosen to deal with various types of singularities.

The key to determining the weights and abscissas of Gaussian quadrature rules are systems of orthogonal polynomials. Consider the space of functions $\{u(x) : \int_a^b W(x)u(x)^2\,dx < \infty\}$. A polynomial basis $\{p_j(x), j = 0, 1, 2, \ldots\}$ can be given for this space of functions such that $p_j(x)$ is a polynomial of degree j, and the $p_j(x)$ are orthogonal in the sense that $\int_a^b W(x)p_j(x)p_k(x)\,dx = 0$ for $j \neq k$. There are in fact simple recurrence formulas for the $p_j(x)$, starting from $p_0(x) \propto 1$. It can be shown that each $p_j(x)$ will have j distinct real roots.

Given any set of possible abscissas $x_1, \ldots, x_n$ for (6.10), the weights ω_j, $j = 1, \ldots, n$, can be chosen to make (6.10) exact for the functions $p_0(x), \ldots, p_{n-1}(x)$. In principle, this can be done by solving the system of equations

$$\int_a^b W(x)p_i(x)\,dx = \sum_{j=1}^{n} \omega_j p_i(x_j), \qquad i = 0, \ldots, n-1,$$

for the ω_j, although in practice there are better methods than direct solution of this system (which tends to be poorly conditioned). (In this system, $\int W(x)p_i(x)\,dx \propto \int W(x)p_i(x)p_0(x)\,dx = 0$ for $i > 0$.) The abscissa values can then be varied to make (6.10) exact for $p_j(x)$, $j = n, \ldots, 2n-1$ as well. It turns out that the solution is to take the abscissas to be the roots of $p_n(x)$. See Section 16.6 of Lange (1999) and Section 4.5 of Press *et. al.* (1992) for additional details. The roots of the polynomials are always in the interior of the interval, so the integrand need not be evaluated at the limits of integration.

The quantity actually evaluated in (6.10) is $\int W(x)q(x)\,dx$, where $q(x) = \sum_{i=0}^{n-1} \alpha_i p_i(x)$ is an interpolating polynomial through the points $(x_j, f(x_j))$, $j = 1, \ldots, n$, since then the α_i satisfy $\sum_{i=0}^{n-1} \alpha_i p_i(x_j) = f(x_j)$, and

$$\begin{aligned} \sum_{j=1}^{n} \omega_j f(x_j) &= \sum_{j=1}^{n} \omega_j \sum_{i=0}^{n-1} \alpha_i p_i(x_j) \\ &= \sum_{i=0}^{n-1} \alpha_i \sum_{j=1}^{n} \omega_j p_i(x_j) \end{aligned}$$

Table 6.1: Weight functions in Gaussian quadrature rules

Name	$W(x)$	(a, b)	S function
Legendre	1	(a, b)	`gauleg()`
Laguerre	$x^\alpha \exp(-x)$	$(0, \infty)$	`gaulag()`
Hermite	$\exp(-x^2)$	$(-\infty, \infty)$	`gauher()`
Jacobi	$(1-x)^\alpha(1+x)^\beta$	$(-1, 1)$	—

$$\begin{aligned} &= \sum_{i=0}^{n-1} \alpha_i \int W(x) p_i(x)\, dx \\ &= \int W(x) q(x)\, dx \end{aligned} \tag{6.11}$$

(recall that the ω_j were chosen to make (6.10) exact for the p_i, $i = 0, \ldots, n-1$). Thus the success of a Gaussian quadrature rule will depend on how closely $f(x)$ is approximated by the interpolating polynomial $q(x)$. (Actually, the requirement is somewhat weaker, since it is a remarkable fact that any polynomial $q^*(x)$ of degree $2n-1$ or less, which interpolates this same set of n points, will have the same value of $\int W(x)q^*(x)\, dx$, since the Gaussian quadrature rule is exact for all such polynomials, so all that is needed is for $W(x)q^*(x)$ to be close enough to $W(x)f(x)$ for some such q^*.)

S functions to calculate the abscissas and weights for several standard weight functions are given in the Appendix, below. These functions are based on the corresponding routines in Press *et. al.* (1992). They use nested loops, and will tend to be quite slow if n is very large. The algorithms execute much faster in FORTRAN or C programs.

Some standard weight functions are given in Table 6.1. The weights and abscissas returned by the S functions can be used as indicated in (6.10). By using a few simple transformations, it is possible to fit a variety of integrals into this framework. For example $\int_0^1 f(v)[v(1-v)]^{-1/2}\, dv$ can be put in the form of a Gauss-Jacobi integral with the transformation $v = x^2$.

Since Gaussian quadrature rules give a higher order approximation than in Newton-Cotes rules with the same number of points, they tend to give better accuracy with a smaller number of points than is achieved with Romberg integration.

One disadvantage of Gaussian quadrature rules is that the abscissas from an n-point formula generally cannot be used in higher order formulas, so for example if the rule with n and $2n$ points are to be computed, the calculations from the first cannot be used in the second. However, given n points from a Gaussian quadrature rule, it is possible to choose $n+1$ additional points and choose weights for all $2n+1$ points in a way that will make the resulting formula exact for polynomials of degree $3n+1$. The pair of rules consisting of the original n point Gaussian quadrature rule and the optimal $n+1$ point extension are known as Gauss-Kronrod pairs. Evaluating the extension in these pairs gives a check on the accuracy of the original Gaussian quadrature rule. The function `gaukron()` below implements the $(7, 15)$ point Gauss-Kronrod pair (7 initial points plus 8 in the extension) for an arbitrary finite interval. Because the weights are symmetric, some multiplications could be saved by adding terms with the same weights before

multiplying, but with greater complexity in the Splus code.

```
gaukron <- function(f,a,b) {
# compute 7-15 Gauss-Kronrod rule for the function f on the
#   finite interval (a,b)
# x=abscissas of 7 point Gauss-Legendre rule on (-1,1)
  x <- c(-0.949107912342758524526189684047851,
         -0.741531185599394439863864773280788,
         -0.405845151377397166906606412076961,
         0,
         0.405845151377397166906606412076961,
         0.741531185599394439863864773280788,
         0.949107912342758524526189684047851)
# w=weights of 7 point G-L rule on (-1,1)
  w <- c(0.129484966168869693270611432679082,
         0.279705391489276667901467771423780,
         0.381830050505118944950369775488975,
         0.417959183673469387755102040816327,
         0.381830050505118944950369775488975,
         0.279705391489276667901467771423780,
         0.129484966168869693270611432679082)
# xk=additional abscissas for 15 point Kronrod extension
  xk <- c(-0.991455371120812639206854697526329,
          -0.864864423359769072789712788640926,
          -0.586087235467691130294144838258730,
          -0.207784955007898467600689403773245,
          0.207784955007898467600689403773245,
          0.586087235467691130294144838258730,
          0.864864423359769072789712788640926,
          0.991455371120812639206854697526329)
# wk=weights of 15 point Kronrod rule, corresponding to c(x,xk)
  wk <- c(0.063092092629978553290700663189204,
          0.140653259715525918745189590510238,
          0.190350578064785409913256402421014,
          0.209482141084727828012999174891714,
          0.190350578064785409913256402421014,
          0.140653259715525918745189590510238,
          0.063092092629978553290700663189204,
          0.022935322010529224963732008058970,
          0.104790010322250183839876322541518,
          0.169004726639267902826583426598550,
          0.204432940075298892414161999234649,
          0.204432940075298892414161999234649,
          0.169004726639267902826583426598550,
          0.104790010322250183839876322541518,
          0.022935322010529224963732008058970)
```

```
# transform to interval (a,b)
  p1 <- (a+b)/2
  p2 <- (b-a)/2
  x <- p2*x+p1
  xk <- p2*xk+p1
  y <- f(x)
  yk <- f(xk)
  p2*c(sum(w*y),sum(wk*c(y,yk)))
}
```

Here is a simple example.

```
> gaukron(function(x) x^4*exp(-x),0,10)
[1] 23.29599 23.29794
> integrate(function(x) x^4*exp(-x),0,10)[1:4]
$integral:
[1] 23.29794

$abs.error:
[1] 2.055201e-06

$subdivisions:
[1] 2

$message:
[1] "normal termination"
```

Gauss-Kronrod pairs can be used in adaptive quadrature schemes. The `integrate()` function, which comes with Splus, implements such an algorithm. It is based on algorithms given in the QUADPACK FORTRAN library, available from Netlib (`http://www.netlib.org/`). For evaluating $\int_a^b f(x)\,dx$, `integrate()` first evaluates the $(7, 15)$ Gauss-Kronrod pair, and stops if the difference is small enough. If not, the interval is subdivided and the $(7, 15)$ pair is applied within each subinterval. The process continues until the difference is small enough within each subinterval. Semi-infinite intervals are handled via an automated transformation, such as $u = 1/x$. In the example above, the initial call to `gaukron()` indicates larger error than the default error bound in `integrate()`, so the interval needed to be divided to guarantee sufficient accuracy in the call to `integrate()`. It is also easy to implement an adaptive Gauss-Kronrod integration algorithm directly in Splus. The function `agk()` below does this for integration over finite intervals. The algorithm could be combined with automated transformations for infinite intervals to give a general routine for evaluating one-dimensional integrals. In `agk()`, the interval with the largest difference between the 7 and 15 point rules is split in half at each step, with the iteration continuing until the sum of the absolute differences is less than `eps` times the absolute value of the integral.

```
agk <- function(f,a,b,eps=1e-6,maxint=50) {
```

```
# adaptive Gauss-Kronrod integration of the function f over
#   the finite interval (a,b)
# the subinterval with the largest abs difference between the 7 and 15
#   point rules is divided in half, and the 7-15 rule applied to each half,
#   continuing until the sum of the absolute differences over all
#   subintervals is < eps*|Integral|, to a maximum of maxint subintervals.
  intervals <- matrix(c(a,b),ncol=1)
  tmp <- gaukron(f,a,b)
  error <- abs(tmp[1]-tmp[2])
  integral <- tmp[2]
  while (sum(error)>eps*(abs(sum(integral))+1e-8) & length(error)<=maxint) {
    split <- (1:length(error))[error == max(error)]
    aa <- intervals[1,split]
    bb <- intervals[2,split]
    tmp <- (aa+bb)/2
    i1 <- gaukron(f,aa,tmp)
    i2 <- gaukron(f,tmp,bb)
    error <- c(error[-split],abs(i1[1]-i1[2]),abs(i2[1]-i2[2]))
    intervals <- cbind(intervals[,-split],c(aa,tmp),c(tmp,bb))
    integral <- c(integral[-split],i1[2],i2[2])
  }
  c(value=sum(integral),error=sum(error),nsubint=ncol(intervals))
}
```

Here are two simple examples.

```
> agk(function(x) x^4*exp(-x),0,10)
    value        error nsubint
 23.29794 1.642814e-06       2
> agk(function(x) sin(x)*exp(-x),0,20)
 value       error nsubint
   0.5 1.42681e-08       3
> integrate(function(x) sin(x)*exp(-x),0,20)[1:4]
$integral:
[1] 0.5

$abs.error:
[1] 7.66039e-09

$subdivisions:
[1] 3

$message:
[1] "normal termination"
```

The QUADPACK routines, and hence probably `integrate()`, use a more sophisticated error

estimate, and the termination criterion and possibly the splitting algorithm are a little different than in `agk()`, so the two functions do not produce identical results.

Experience suggests that in Gaussian quadrature it is generally better to use only a moderate number of points in the quadrature rule, and to subdivide the interval and apply the rule within subintervals if more accuracy is needed, as in the adaptive algorithms discussed above, than to keep using ever larger values of n. This is because it is generally easier to get good local approximations using moderate degree polynomials on short intervals than using a high degree polynomial over a long interval.

To further illustrate, again consider the simple integral from Example 6.1, $\int_0^2 \exp(-x)\,dx$. Here 5-point and 10-point Gauss-Legendre quadrature will be considered, as will the $(7, 15)$ Gauss-Kronrod algorithm from the `gaukron()` and `integrate()` functions.

```
> f <- function(x) {
+   assign('i',i+length(x),where=1)
+   exp(-x)
+ }
> truval <- 1-exp(-2)
> u <- gauleg(5,0,2)
> u2 <- sum(u$w*f(u$x))
> c(u2,u2-truval)
[1]  8.646647e-01 -3.034261e-10
> u <- gauleg(10,0,2)
> u3 <- sum(u$w*f(u$x))
> c(u3,u3-truval,u2-u3)
[1]  8.646647e-01 -8.437695e-15 -3.034176e-10
> i <- 0
> u <- integrate(f,0,2)[1:4]
> u
$integral:
[1] 0.8646647

$abs.error:
[1] 9.599707e-15

$subdivisions:
[1] 1

$message:
[1] "normal termination"

> i
[1] 15
> u[[1]]-truval
[1] -1.110223e-16
> i <- 0
```

```
> u <- gaukron(f,0,2)
> u
[1] 0.8646647 0.8646647
> i
[1] 15
> u-truval
[1] -7.771561e-16  1.110223e-16
```

Here 5-point Gaussian quadrature is almost as accurate as Romberg integration after evaluating the integrand at 17 points, and the 10-point Gauss-Legendre rule is accurate to 14 significant digits. In `integrate()`, the interval is not subdivided, and the result from the $(7, 15)$ Gauss-Kronrod pair was essentially accurate to machine precision.

Next Gaussian quadrature methods are applied to evaluating $\int_1^\infty u^{-1/2} \exp(-u)\, du$.

```
> # int_1^Inf u^(-.5)exp(-u)
> # transformation to a finite interval
> # x=exp(-u)
> f <- function(x) {
+   assign('i',i+length(x),where=1)
+   ifelse(x==0,0,1/sqrt(-log(x)))
+ }
> truval <- (1-pgamma(1,shape=.5))*gamma(.5)
> # Gauss-Legendre of transformed integral
> for (n in c(10,20,40,80)) {
+   u <- gauleg(n,0,exp(-1))
+   u <- sum(u$w*f(u$x))
+   print(c(u,u-truval))
+ }
[1] 2.788540e-01 4.837192e-05
[1] 2.788157e-01 1.006516e-05
[1] 2.788077e-01 2.100210e-06
[1] 2.788060e-01 4.426965e-07
>
> # adaptive integration of transformed integral
> i <- 0
> u <- integrate(f,0,exp(-1))[1:4]
> u
$integral:
[1] 0.2788056

$abs.error:
[1] 5.469256e-05

$subdivisions:
[1] 8
```

```
$message:
[1] "normal termination"

> i
[1] 225
> u[[1]]-truval
[1] 4.720109e-08
>
> # Gauss-Laguerre, x=u-1 in original integral (so interval is 0,infty)
> f <- function(x) 1/sqrt(x+1)
> for (n in c(5,10,20)) {
+   u <- gaulag(n,0)
+   u <- sum(u$w*f(u$x))/exp(1)
+   print(c(u,u-truval))
+ }
[1]  0.2786623754 -0.0001432099
[1]  2.788022e-01 -3.350869e-06
[1]  2.788056e-01 -1.631527e-08
>
> # adaptive integration of original integral
> i <- 0
> u <- integrate(function(u) {assign('i',i+length(u),where=1);
+        exp(-u)/sqrt(u)},1,Inf)[1:4]
> u
$integral:
[1] 0.2788056

$abs.error:
[1] 1.189216e-06

$subdivisions:
[1] 3

$message:
[1] "normal termination"

> i
[1] 75
> u[[1]]-truval
[1] 1.327549e-11
>
> # transformation to a finite interval
> # x=1/u
> f <- function(x) {
+   assign('i',i+length(x),where=1)
```

```
+   ifelse(x==0,0,exp(-1/x)*x^(-1.5))
+ }
> # Gauss-Legendre of transformed integral
> for (n in c(5,10,20)) {
+   u <- gauleg(n,0,1)
+   u <- sum(u$w*f(u$x))
+   print(c(u,u-truval))
+ }
[1]  0.2784703115 -0.0003352738
[1]  2.787817e-01 -2.387048e-05
[1] 2.788056e-01 2.738277e-08
> # adaptive integration of transformed integral
> i <- 0
> u <- agk(f,0,1)
> u
     value        error nsubint
 0.2788056 1.177239e-07       3
> i
[1] 75
> u[1]-truval
        value
 1.327549e-11
```

Note that `integrate()` applied to the original semi-infinite interval worked quite well, as did Gauss-Legendre and `agk()` applied to the second transformation. The values for `integrate()` on the original interval and `agk()` on the second transformation were identical, suggesting that `integrate()` used this same transformation. Generally the Gaussian quadrature methods give better accuracy for a smaller number of points than seen previously for Romberg integration.

6.2.1 Gauss-Hermite Quadrature

A critical aspect of applying quadrature rules is that points need to be located in the main mass of the integrand. Adaptive rules try to do this automatically, but in general this can require careful thought. This is especially true for Gauss-Hermite quadrature. Gauss-Hermite quadrature is a natural method for many statistical applications, especially for integrals arising in Bayesian calculations, since the Gauss-Hermite weight function is proportional to a normal density. In Bayesian inference, the posterior distribution is often approximately normal, and this approximation improves as the sample size increases.

It is always possible to put a doubly infinite integral into the form of the Gauss-Hermite integral, through

$$\int_{-\infty}^{\infty} g(x)\,dx = \int_{-\infty}^{\infty} \exp(-x^2) f(x)\,dx, \tag{6.12}$$

where $f(x) = g(x)\exp(x^2)$. However, if $g(x)$ is concentrated about a point far from 0, or if the spread in $g(x)$ is quite different than for the weight function $\exp(-x^2)$, then applying

Gauss-Hermite quadrature directly to the right hand side of (6.12) can give a very poor approximation, because the abscissas in the quadrature rule will not be located where most of the mass of g is located. Recalling from (6.11) that Gaussian quadrature really evaluates $\int W(x)q^*(x)\,dx$ for an interpolating polynomial q^* through the points $(x_j, f(x_j))$, another way to think of this is that such a q^* may provide a poor approximation to the integrand over an important region, if the x_j are in region where f is say nearly constant.

To illustrate, suppose $g(x) = \exp[-(x-\mu)^2/(2\sigma^2)]$, which is proportional to the normal density with mean μ and variance σ^2. If the change of variables $u = (x-\mu)/(\sigma\sqrt{2})$ is made, then

$$\int_{-\infty}^{\infty} g(x)\,dx = \sqrt{2}\sigma \int_{-\infty}^{\infty} \exp(-u^2)\,du$$

so the function $f(x)$ is constant in (6.12), and a one-point formula should give an exact result. If the change of variables is not made, $f(x) = \exp[x^2 - (x-\mu)^2/(2\sigma^2)]$ in (6.12), which might be difficult to approximate with the polynomial interpolating the points in a Gauss-Hermite quadrature rule if μ is far from 0 or $2\sigma^2$ is far from 1.

More generally, the transformation $u = (x-\hat{x})/(\hat{\sigma}\sqrt{2})$ could be considered, where $\hat{x}$ is the mode of $g(x)$ and $\hat{\sigma}^2 = -1/[d^2 \log\{g(\hat{x})\}/dx^2]$. This is based on approximating g with a normal density in the neighborhood of the mode of g. Note the similarity to the Laplace approximation. In a sense, Gauss-Hermite quadrature is a generalization of the Laplace approximation. Applying Gauss-Hermite quadrature with this transformation gives

$$\int_{-\infty}^{\infty} g(x)\,dx = \sqrt{2}\hat{\sigma} \int_{-\infty}^{\infty} \exp(-u^2) g(\sqrt{2}\hat{\sigma}u + \hat{x}) \exp(u^2)\,du \doteq \sqrt{2}\hat{\sigma} \sum_{i=1}^{n} \omega_i g(\sqrt{2}\hat{\sigma}x_i + \hat{x}) \exp(x_i^2).$$

The success of this formula will depend on how close g is to a normal density times a low order polynomial.

In the context of Bayesian inference, using a parameter value θ for the argument of the integrand, $g(\theta)$ would often be of the form $g(\theta) = q(\theta)\exp[l(\theta) + \log\{\pi(\theta)\}]$, where $l(\theta)$ is the log-likelihood and $\pi(\theta)$ is the prior density. Often several such integrals will be evaluated. For example, to compute the posterior mean and variance of θ, integrals of this form with $q(\theta) = 1$, θ and $(\theta - \tilde{\theta})^2$ would need to be evaluated, where $\tilde{\theta}$ is the posterior mean, given by the ratio of the integrals with the first two values of $q(\theta)$. In this case it might sometimes be reasonable to determine $\hat{\theta}$ and $\hat{\sigma}$ just from the $\exp[l(\theta) + \log\{\pi(\theta)\}]$ portion of the integrand. Then $\hat{\theta}$ is just the posterior mode, and $\hat{\sigma}^2$ is the large sample approximation to the posterior variance. If this is done the same abscissas and weights could be applied to calculating moments of various parametric functions. This approach is used in the following example.

Example 6.2 Suppose $X \sim \text{Binomial}(20, p)$ is observed to have the value $X = 18$. Let $\theta = \log(p/(1-p))$. The log-likelihood is

$$l(\theta) = X\theta - n\log\{1 + \exp(\theta)\}.$$

Suppose the prior is $N(0, 100)$, so

$$\log\{\pi(\theta)\} \propto -\theta^2/200.$$

Quantities that might be of interest are the posterior mean and variance of θ, and the posterior mean and variance of p. Let

$$I(q) = \int_{-\infty}^{\infty} q(\theta) \exp\{l(\theta)\}\pi(\theta)\, d\theta.$$

Then the posterior mean of θ is $I(\theta)/I(1)$, and the posterior mean of p is $I[\exp(\theta)/\{1+\exp(\theta)\}]/I(1)$, with similar formulas for the variances.

In the following calculations, first the normalizing constant $I(1)$ is computed using both the naive approach (6.12) and using Gauss-Hermite quadrature following transformation, which is more accurate, even though $\hat{\theta}$ is not terribly far from from 0 and $2\hat{\sigma}^2$ is close to 1. Also, direct application of `integrate()` seems to work well here, too.

```
> # one sample binomial
> X <- 18
> n <- 20
> mu <- 0
> sig2 <- 100
> # f1 is proportional to -log posterior
> f1 <- function(theta) -X*theta+n*log(1+exp(theta))+(theta-mu)^2/(2*sig2)
> # normalizing constant--naive approach
> for (k in c(5,10,20,40)) {
+   u <- gauher(k)
+   post <- exp(-f1(u$x)+u$x^2)
+   u1 <- sum(u$w*post)
+   print(u1)
+ }
[1] 0.001906739
[1] 0.002729944
[1] 0.002826549
[1] 0.002829062
> # more accurate approach
> theta.hat <- nlmin(f1,0)[[1]]
> theta.hat
[1] 2.185144
> sig.hat <- exp(theta.hat)
> sig.hat <- 1/sqrt(n*sig.hat/(1+sig.hat)^2+1/sig2)
> sig.hat
[1] 0.7397356
> #normalizing constant
> for (k in c(5,10,20,40)) {
+   u <- gauher(k)
+   theta <- sqrt(2)*sig.hat*u$x+theta.hat
+   post <- exp(-f1(theta)+u$x^2)
+   u1 <- sum(u$w*post)
+   print(u1*sqrt(2)*sig.hat)
+ }
```

```
[1] 0.002812167
[1] 0.002828169
[1] 0.002829058
[1] 0.002829073
> # different method
> i <- 0
> integrate(function(x) {assign('i',i+length(x),where=1); exp(-f1(x))},
+   -Inf,Inf)[1:4]
$integral:
[1] 0.002829073

$abs.error:
[1] 4.827169e-08

$subdivisions:
[1] 3

$message:
[1] "normal termination"

> i
[1] 150
> # posterior mean and variance of theta; 40 pt G-H Q
> theta.mean <- sum(u$w*post*theta)/u1
> c(theta.mean,theta.hat)
[1] 2.420751 2.185144
> theta.var <- sum(u$w*post*(theta-theta.mean)^2)/u1
> c(theta.var,sig.hat^2)
[1] 0.6854480 0.5472088
> # check with different # points
> u2 <- gauher(30)
> theta2 <- sqrt(2)*sig.hat*u2$x+theta.hat
> post2 <- exp(-f1(theta2)+u2$x^2)
> theta.mean2 <- sum(u2$w*post2*theta2)/u1
> c(theta.mean2,theta.hat)
[1] 2.420749 2.185144
> theta.var2 <- sum(u2$w*post2*(theta2-theta.mean2)^2)/u1
> c(theta.var2,sig.hat^2)
[1] 0.6854337 0.5472088
> # posterior mean and variance of p
> p <- exp(theta)
> p <- p/(1+p)
> p.mean <- sum(u$w*post*p)/u1
> c(p.mean,sum(u$w*post*(p-p.mean)^2)/u1)
[1] 0.898789623 0.004308108
> # different method
```

```
> i <- 0
> integrate(function(x) {assign('i',i+length(x),where=1); x2 <- exp(x);
+   ifelse(abs(x)<700,x2*exp(-f1(x))/(1+x2),0)},
+   -Inf,Inf)[[1]]/(u1*sqrt(2)*sig.hat)
[1] 0.8987896
> i
[1] 150
> exp(theta.hat)/(1+exp(theta.hat))
[1] 0.8989075
```

6.3 Multi-dimensional Integrals

The principal method for numerical evaluation of higher dimensional integrals is nested application of the one-dimensional methods above. Consider, for example, evaluating the three dimensional integral

$$\int_A f(x_1, x_2, x_3)\, dx_1 dx_2 dx_3,$$

where $A \subset R^3$. Assume that A is of the form

$$\{(x_1, x_2, x_3) : u(x_1, x_2) \le x_3 \le v(x_1, x_2), q(x_1) \le x_2 \le r(x_1), a \le x_1 \le b\},$$

for some functions u, v, q, and r, and values a and b. If A is not of this form but can be divided into subregions that are of this form, then evaluate each such subregion separately. Then

$$\int_A f(x_1, x_2, x_3)\, dx_1 dx_2 dx_3 = \int_a^b \int_{q(x_1)}^{r(x_1)} \int_{u(x_1,x_2)}^{v(x_1,x_2)} f(x_1, x_2, x_3)\, dx_3 dx_2 dx_1 = \int_a^b g(x_1)\, dx_1, \quad (6.13)$$

where

$$g(x_1) = \int_{q(x_1)}^{r(x_1)} \int_{u(x_1,x_2)}^{v(x_1,x_2)} f(x_1, x_2, x_3)\, dx_3 dx_2.$$

The final integral in (6.13) can be evaluated using a one-dimensional quadrature rule. However, doing so will require evaluating $g(x_1)$ at a number of points, and $g(x_1)$ is defined in terms of integrals. At each x_1 where the value of $g(x_1)$ is needed, it can be evaluated by applying a one-dimensional quadrature rule to

$$g(x_1) = \int_{q(x_1)}^{r(x_1)} h(x_1, x_2)\, dx_2,$$

where

$$h(x_1, x_2) = \int_{u(x_1,x_2)}^{v(x_1,x_2)} f(x_1, x_2, x_3)\, dx_3. \quad (6.14)$$

This in turn requires evaluating $h(x_1, x_2)$ at many values of x_2, which can be done by applying a one-dimensional quadrature rule to the integral in (6.14), which only requires evaluating $f(x_1, x_2, x_3)$ at many values of x_3 (for each value of x_1 and x_2).

In principle, this procedure could be performed for integrals of any dimension, but in practice there are serious limitation. The number of points where the original integrand needs to be evaluated grows exponentially with the dimension of the integral, so the computational burden

becomes excessive. Also, since the quadrature rules do not necessarily give highly accurate approximations, the overall error in the approximation can grow with the number of nested applications.

In general, defining the functions bounding the region of integration can also be difficult. For rectangular regions, this is simple, though. In this case q, r, u, and v are constants. If the same fixed abscissas and weights are used for each evaluation in each dimension, say $(x_i^{(j)}, \omega_i^{(j)})$ for the integrals over x_i, then the nested integration rule above reduces to evaluating f over the Cartesian product of the abscissas for each variable. This can be written

$$\int_A f(x_1, x_2, x_3)\, dx_3 dx_2 dx_1 \doteq \sum_j \sum_k \sum_l f(x_1^{(j)}, x_2^{(k)}, x_3^{(l)}) \omega_1^{(j)} \omega_2^{(k)} \omega_3^{(l)}$$

Transformations of the variables of integration can greatly affect the performance of nested integration formulas. This will be discussed in more detail for Gauss-Hermite Quadrature, below.

6.3.1 Gauss-Hermite Quadrature

For multi-dimensional integrals, as in the univariate case, Gauss-Hermite quadrature is most useful when the integrand is fairly concentrated about a single mode. It is again important to make a change of variables to center the quadrature rule near the mode, to make the scaling of the variables similar, and to reduce association among the variables of integration. Near the mode this can be done by first locating the mode of the integrand, and then calculating (or approximating) the second derivative matrix of the log of the integrand at the mode. For Bayesian inference applications, it may again be appropriate to just use the mode and second derivative of the posterior density, rather than the entire integrand. The transformation is then based on the mode and the inverse of the second derivative matrix.

In general, suppose the integral is of the form

$$\int q(\theta) \exp[h(\theta)]\, d\theta, \tag{6.15}$$

where θ is p-dimensional and the region of integration is essentially all of R^p. Let $\hat{\theta}$ be the mode of h, let $H = -\partial^2 h(\hat{\theta})/\partial\theta\partial\theta'$, and let $B'B = H$ be the Choleski factorization of H. If θ is thought of as a random quantity with log-density given by $h(\theta)$, then the first order normal approximation to this distribution is the $N(\hat{\theta}, H^{-1})$ distribution. To use a Gauss-Hermite product rule, a function proportional to a $N(0, 2^{-1}I)$ density would be preferred, since the Gauss-Hermite weight function is proportional to this density. Setting $\alpha = 2^{-1/2}B(\theta - \hat{\theta})$ will transform θ to a variable that does have an approximate $N(0, 2^{-1}I)$ distribution. With this transformation (6.15) becomes

$$2^{p/2}|B|^{-1} \int \exp(-\alpha'\alpha) f(\alpha)\, d\alpha, \tag{6.16}$$

where

$$f(\alpha) = q(2^{1/2}B^{-1}\alpha + \hat{\theta}) \exp\{h(2^{1/2}B^{-1}\alpha + \hat{\theta}) + \alpha'\alpha\}.$$

The advantage is that to first order, $\exp\{h(2^{1/2}B^{-1}\alpha + \hat{\theta}) + \alpha'\alpha\}$ should be constant. (6.16) can be approximated using the n-point Gauss-Hermite product rule with abscissas $x_1, \ldots, x_n$ and

weights $\omega_1, \ldots, \omega_n$, giving

$$2^{p/2}|B|^{-1} \sum_{i_1=1}^{n} \cdots \sum_{i_p=1}^{n} \omega_{i_1} \cdots \omega_{i_p} f(x_{i_1}, \ldots, x_{i_p}). \tag{6.17}$$

This approach is described for Bayesian inference problems in Smith et al (1987). They also suggest starting from an initial guess at the posterior mean and variance, using these in place of $\hat{\theta}$ and H^{-1} above, using Gauss-Hermite quadrature to calculate new approximations to the posterior mean and variance, and iterating this process until convergence. The iteration seems unnecessary, though, if a reasonably good approximation has been used initially. They also suggest that this approach can be used effectively for integrals of up to 5 or 6 variables (with increases in computing power since then it might be possible to go a little higher, now). Note that if $n = 10$ and $p = 6$, then f has to be evaluate at 10^6 points, and with $n = 20$ and $p = 6$ there are 6.4×10^7 points.

To illustrate, Gauss-Hermite quadrature will be applied to the Stanford heart transplant data discussed in Example 5.1. Recall that the data consist of survival times X_i for non-transplant patients, time to transplant Y_i, and survival beyond transplant Z_i, and that the model specifies

$$P(X_i > t) = [\lambda/\{\lambda + a_i(t)\}]^p, \quad \text{where} \quad a_i(t) = \begin{cases} t & t \le Y_i \\ Y_i + \tau(t - Y_i) & t > Y_i \end{cases}$$

with parameters $\theta = (p, \lambda, \tau)'$. Flat priors were also assumed, so the posterior is proportional to $\exp\{l(\theta)\}$, where $l(\theta)$ is the log likelihood; see (5.1). Since all of the parameters are constrained to be positive, it is convenient here to first transform to $\gamma = (\log(p), \log(\lambda), \log(\tau))$. For these parameters the posterior is then proportional to $\exp[l(\gamma) + \gamma_1 + \gamma_2 + \gamma_3]$. Below the posterior means and variances for the original parameters (θ) are computed using (6.17), with $n = 10$ and $n = 20$. Comparing the two suggests the results are reasonably accurate.

```
> # x=survival for nontransplant patients; sx= status
> # y=days to transplant
> # z=survival from transplant; sz=status
> x <- c(49,5,17,2,39,84,7,0,35,36,1400,5,34,15,11,2,1,39,8,101,2,148,1,68,31,
+        1,20,118,91,427)
> sx <- c(1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0)
> y <- c(0,35,50,11,25,16,36,27,19,17,7,11,2,82,24,70,15,16,50,22,45,18,4,1,
+        40,57,0,1,20,35,82,31,40,9,66,20,77,2,26,32,13,56,2,9,4,30,3,26,4,
+        45,25,5)
> z <- c(15,3,624,46,127,61,1350,312,24,10,1024,39,730,136,1379,1,836,60,1140,
+        1153,54,47,0,43,971,868,44,780,51,710,663,253,147,51,479,322,442,65,
+        419,362,64,228,65,264,25,193,196,63,12,103,60,43)
> sz <- c(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,0,0,1,1,1,1,0,0,1,0,1,0,0,1,
+         1,1,0,1,0,1,0,0,1,1,1,0,1,0,0,1,1,0,0,0)
>
> ff <- function(b) { # calculate -log likelihood -log prior
+ # - input values are log(params);  final term because flat prior on
+ # original scale is rescaled by the log transformation
```

```
+   p <- exp(b[1]);  l <- exp(b[2]);  tau <- exp(b[3])
+   -sum(p*log(l/(l+x))+sx*log(p/(l+x)))-sum(
+     p*log(l/(l+y+tau*z))+sz*log(tau*p/(l+y+tau*z)))-sum(b)
+ }
> ff2 <- function(b) {# - second derivatives of log likelihood wrt log
+ # params (only valid at posterior mode)
+   p <- exp(b[1]);  l <- exp(b[2]);  tau <- exp(b[3]);
+   v <- matrix(0,3,3)
+   v[1,1] <- sum(c(sx,sz))
+   w <- y+tau*z
+   N <- length(sx)+length(sz)
+   v[1,2] <- v[2,1] <- (-N/l+sum(1/(l+x))+sum(1/(l+w)))*p*l
+   v[1,3] <- v[3,1] <- sum(z/(l+w))*p*tau
+   v[2,2] <- N*p-(sum((p+sx)/(l+x)^2)+sum((p+sz)/(l+w)^2))*l^2
+   v[2,3] <- v[3,2] <- -sum((p+sz)*z/(l+w)^2)*l*tau
+   v[3,3] <- sum(sz)-sum((p+sz)*(z/(l+w))^2)*tau^2
+   v
+ }
>
> gamma.hat <- nlmin(ff,c(0,0,0),rfc.tol=1e-14)$x
> exp(gamma.hat) # first order estimate of mean
[1]  0.4853519 29.5188871  0.9117213
> h <- ff2(gamma.hat)
> hc <- solve(chol(h))*sqrt(2)
> solve(h) # first order estimate of posterior variances of log params
            [,1]        [,2]        [,3]
[1,]  0.09062597  0.12995558 -0.06730149
[2,]  0.12995558  0.28901209 -0.02226088
[3,] -0.06730149 -0.02226088  0.21314900
>
> # 10 point G-H quadrature
> u <- gauher(10)
> alpha <- as.matrix(expand.grid(u$x,u$x,u$x))
> dim(alpha)
[1] 1000    3
> gamma <- hc %*% t(alpha) + gamma.hat
> post <- apply(gamma,2,ff)
> post <- exp(-post+apply(alpha^2,1,sum))
> weights <- expand.grid(u$w,u$w,u$w)
> weights <- weights[,1]*weights[,2]*weights[,3]
> u1 <- sum(weights*post)
> # means
> theta <- exp(gamma)
> p.m <- sum(theta[1,]*weights*post)/u1
> l.m <- sum(theta[2,]*weights*post)/u1
> tau.m <- sum(theta[3,]*weights*post)/u1
```

```
> c(p.m,l.m,tau.m)
[1]  0.4968976 32.5958054  1.0469074
> #variances
> p.var <- sum((theta[1,]-p.m)^2*weights*post)/u1
> l.var <- sum((theta[2,]-l.m)^2*weights*post)/u1
> tau.var <- sum((theta[3,]-tau.m)^2*weights*post)/u1
> c(p.var,l.var,tau.var)
[1]   0.02069934 279.62275496   0.25345330
> sqrt(c(p.var,l.var,tau.var))
[1]  0.1438727 16.7219244  0.5034415
>
> # 20 point G-H quadrature
> u <- gauher(20)
... (other commands the same as before)
> c(p.m,l.m,tau.m)
[1]  0.4968993 32.5960503  1.0469256
> #variances
...
> c(p.var,l.var,tau.var)
[1]   0.02071147 279.88172827   0.25381589
> sqrt(c(p.var,l.var,tau.var))
[1]  0.1439148 16.7296661  0.5038014
>
```

Naylor and Smith, 1982, used 8, 9 and 10 point Gauss-Hermite quadrature on the original scale [instead of using the log parameters], and obtained slightly different results. The values here should be more accurate.

The calculations were quite fast for $n = 10$, but noticeably slower for $n = 20$. For higher dimensional integrals, coding the calculations in FORTRAN or C to speed execution would be very important.

6.4 Exercises

Exercise 6.1 Verify that if T_1 is the trapezoidal approximation for $\int_a^b f(x)\,dx$ with n intervals, and T_2 is the trapezoidal approximation with $2n$ intervals, then $(4T_2 - T_1)/3$ gives Simpson's approximation (6.7) (see Algorithm 6.1).

Exercise 6.2 Evaluate

$$\int_0^2 \{\log(x)\}^2 x^3 \exp(-x)\,dx,$$

and

$$\int_2^\infty \{\log(x)\}^2 x^3 \exp(-x)\,dx,$$

using Romberg integration. (It may be easiest to use an 'open' formula, although that is not the only possibility.) Compare the number of points where the integrand is evaluated to that for the Splus `integrate()` function.

Exercise 6.3 Evaluate

$$\int_1^2 2 + 7x + 3x^2 - 15x^3 - 5x^4 - 2x^5 + 14x^7 + 6x^8 + x^9 \, dx$$

using 5 point Gauss-Legendre quadrature. Verify that this result is exact by explicitly evaluating the antiderivative. (The formula for the antiderivative need not be written out. This calculation can be done by storing the coefficients in a vector, and evaluating the integral from the term-by-term antiderivative using a `for()` loop.)

Exercise 6.4 In Example 6.2, suppose $n = 100$ and $X = 98$. Evaluate the normalizing constant of the posterior using both the naive approach (6.12) to Gauss-Hermite quadrature, and also by using the transformation $\alpha = (\theta - \hat{\theta})/(\sqrt{2}\hat{\sigma})$ before applying Gauss-Hermite quadrature.

Exercise 6.5 Suppose $y_1, \ldots, y_n$ are a random sample from a Poisson distribution with mean $\exp(\alpha)$. Suppose the prior on α is normal with mean 0 and variance 100, and that the observed data are 11, 19, 27, 12, 14, 11, 10, 13, 15, 10.

1. Use Gauss-Hermite quadrature to evaluate the mean and variance of the posterior distribution of α. Remember to make a change of variables in the integral, if appropriate, before applying the quadrature formulas. Give some discussion of the accuracy of these calculations.
2. Use the `integrate()` command in Splus to calculate the mean and variance of the posterior distribution of α.
3. Use the Laplace approximation to approximate the mean and variance of the posterior distribution of α.

Exercise 6.6 Suppose

$$f(x, y) = (4x^4 + x^4y^3 + 23x^2y^4 + 12x + 2y + 1)^{-2/3}.$$

Evaluate

$$\int_0^5 \int_0^8 f(x, y) \, dy \, dx$$

using nested application of a univariate quadrature rule, as described in class. (Choose an appropriate univariate quadrature method.)

Exercise 6.7 Consider again the logistic regression model with incomplete covariate data in Exercise 3.4, only here suppose that there are p covariates, and that the covariate vectors z_i are sampled iid from a p-dimensional normal distribution with mean μ and covariance matrix V. Suppose again that the covariate values are missing completely at random. In this case, the likelihood function involves integrals of the form

$$\int \frac{\exp([\beta_0 + \beta' z_i] y_i)}{1 + \exp(\beta_0 + \beta' z_i)} f(z_i; \mu, V) \tag{6.18}$$

where y_i is the binary response, $f(z; \mu, V)$ is the normal density with mean μ and covariance matrix V, and the integral is over any components of z_i that are missing.

Suppose $p = 3$, $\mu = 0$, $v_{11} = v_{22} = 1$, $v_{33} = 2$, $v_{ij} = .5$ for $i \neq j$. Also suppose the second and third components of z_i are not observed, that $z_{i1} = .5$, and that $(\beta_0, \beta_1, \beta_2, \beta_3) = (1, 2, -1, -2)$. Use Gauss-Hermite quadrature to evaluate (6.18) in this case both for $y_i = 1$ and $y_i = 0$. Try both $n = 10$ and $n = 20$ for each dimension.

Also, evaluate these integrals using the Laplace approximation (5.8).

Note that in performing an iterative search for the MLEs of the logistic regression model, it would be necessary to evaluate such integrals for every observation missing some covariates, at every value of the parameters where the likelihood was evaluated during the search.

6.5 Appendix: Gaussian Quadrature Functions

```
# nested loops in the following routines are slow-should not be used
# with large n
gauleg <- function(n,a=-1,b=1) {# Gauss-Legendre: returns x,w so that
#\int_a^b f(x) dx \doteq \sum w_i f(x_i)
  EPS <- 3.e-14
  m <- trunc((n+1)/2)
  xm <- 0.5*(b+a)
  xl <- 0.5*(b-a)
  x <- w <- rep(-1,n)
  for (i in 1:m) {
    z <- cos(pi*(i-.25)/(n+.5))
    z1 <- z+1
    while (abs(z-z1) > EPS) {
      p1 <- 1
      p2 <- 0
      for (j in 1:n) {# recursively evaluates pn(x)
        p3 <- p2
        p2 <- p1
        p1 <- ((2*j-1)*z*p2-(j-1)*p3)/j
      }
      pp <- n*(z*p1-p2)/(z*z-1)
      z1 <- z
      z <- z1-p1/pp  #Newton iteration
    }
    x[i] <- xm-xl*z
    x[n+1-i] <- xm+xl*z
    w[i] <- 2*xl/((1-z*z)*pp*pp)
    w[n+1-i] <- w[i]
  }
  list(x=x,w=w)
}
```

```
gaulag <- function(n,alf) {# Gauss-Laguerre:  returns x,w so that
#\int_0^\infty x^alf*exp(-x) f(x) dx \doteq \sum w_i f(x_i)
  EPS <- 3.e-14
  MAXIT <- 10
  x <- w <- rep(-1,n)
  for (i in 1:n) {
    if(i==1){
      z <- (1+alf)*(3+.92*alf)/(1+2.4*n+1.8*alf)
    } else if(i==2){
      z <- z+(15+6.25*alf)/(1+.9*alf+2.5*n)
    }else {
      ai <- i-2
      z <- z+((1+2.55*ai)/(1.9*ai)+1.26*ai*alf/(1+3.5*ai))*(z-x[i-2])/(1+.3*alf)
    }
    for (its in 1:MAXIT) {
      p1 <- 1
      p2 <- 0
      for (j in 1:n) {
        p3 <- p2
        p2 <- p1
        p1 <- ((2*j-1+alf-z)*p2-(j-1+alf)*p3)/j
      }
      pp <- (n*p1-(n+alf)*p2)/z
      z1 <- z
      z <- z1-p1/pp
      if(abs(z-z1) <= EPS) break
    }
    x[i] <- z
    w[i] <- -exp(lgamma(alf+n)-lgamma(n))/(pp*n*p2)
  }
  list(x=x,w=w)
}

gauher <- function(n) {# Gauss-Hermite:  returns x,w so that
#\int_-\infty^\infty exp(-x^2) f(x) dx \doteq \sum w_i f(x_i)
  EPS <- 3.e-14
  PIM4 <- .7511255444649425D0
  MAXIT <- 10
  m <- trunc((n+1)/2)
  x <- w <- rep(-1,n)
  for (i in 1:m) {
    if (i==1) {
      z <- sqrt(2*n+1)-1.85575*(2*n+1)^(-.16667)
    } else if(i==2) {
      z <- z-1.14*n^.426/z
```

```
    } else if (i==3) {
      z <- 1.86*z-.86*x[1]
    } else if (i==4) {
      z <- 1.91*z-.91*x[2]
    } else {
      z <- 2.*z-x[i-2]
    }
    for (its in 1:MAXIT) {
      p1 <- PIM4
      p2 <- 0.d0
      for (j in 1:n) {
        p3 <- p2
        p2 <- p1
        p1 <- z*sqrt(2.d0/j)*p2-sqrt((j-1)/j)*p3
      }
      pp <- sqrt(2.d0*n)*p2
      z1 <- z
      z <- z1-p1/pp
      if(abs(z-z1) <= EPS) break
    }
    x[i] <- z
    x[n+1-i] <- -z
    w[i] <- 2/(pp*pp)
    w[n+1-i] <- w[i]
  }
  list(x=x,w=w)
}
```

6.6 References

Lange K (1999). *Numerical Analysis for Statisticians*. Springer.

Naylor JC and Smith AFM (1982). Applications of a method for the efficient computation of posterior distributions. *Applied Statistics*, 31:214–225.

Press WH, Teukolsky SA, Vetterling WT, and Flannery BP (1992). *Numerical Recipes in C: The Art of Scientific Computing. Second Edition*. Cambridge University Press.

Smith AFM, Skeene AM, Shaw JEH, and Naylor JC (1987). Progress with numerical and graphical methods for practical Bayesian statistics. *The Statistician*, 36:75–82.

Chapter 7

Basic Simulation Methodology

7.1 Generating Pseudo-Random Numbers

Sequences of 'random' numbers generated on a computer are generally not random at all, but are generated by deterministic algorithms, which are designed to produce sequences that look random. The term 'pseudo-random' is often used to distinguish the generated numbers from truly random processes (if there are such things). The basic problem is generating a sequence of values u_i, $0 < u_i < 1$, that appear to be an iid sample for the $U(0,1)$ distribution. Once such a sequence is available, transformations and other techniques can be used to obtain samples from other distributions. Generation of uniform deviates is discussed in the following section, and methods of using these to obtain samples from other distributions are discussed in the 2 sections following that. A much more extensive discussion of these topics is given in Kennedy and Gentle (1980, Chapter 6).

7.1.1 Uniform Deviates

Probably the most commonly used random number generators in statistical applications are multiplicative congruential generators, which generate a sequence of integers $\{k_i\}$ by calculating

$$k_{i+1} = ak_i \bmod m, \tag{7.1}$$

for suitably chosen positive integers a and m, where $b \bmod m$ is the remainder from dividing b by m. That is, if $c = b \bmod m$, then c is an integer, $0 \le c < m$, and $b = lm + c$ for some integer l. The k_i are converted to uniform deviates by setting

$$u_i = k_i/m.$$

If

$$a^{m-1} = 1 \bmod m \quad \text{and} \quad a^l \neq 1 \bmod m \quad \text{for } 0 < l < m-1, \tag{7.2}$$

and if k_0 is a positive integer not equal to a multiple of m, then it can be shown that $k_1, \ldots, k_{m-1}$ will be a permutation of $\{1, 2, \ldots, m-1\}$. In this case the period of the random number generator is $m - 1$. In general the period is the number of values until the generator starts to repeat.

There is also an extension called linear congruential generators, which take the form

$$k_{i+1} = (ak_i + b) \bmod m, \tag{7.3}$$

for suitable constants a, b and m.

Good generators should have long periods, should have low (near 0) autocorrelations, and should give samples which appear to be drawn from a uniform distribution. There are obvious limitations. For example, a sample can only appear to be random if the size of the sample is much less than the period of the generator.

There are two common choices for m in (7.1). One is $2^{31} - 1$, which is the largest prime integer that can be stored on most computers. For many years the most widely used multiplier a used with this m was 7^5. This combination satisfies (7.2), and so gives a period of $2^{31} - 2$. This particular generator is still used in the IMSL random number generator, and was also used in early versions of MATLAB, for example. Since 7^5 is fairly small relative to $2^{31} - 1$, there will be a small autocorrelation in this generator, although it is still adequate for many purposes. There are many other values of a that will give a full period when $m = 2^{31} - 1$. Fishman and Moore (1986) identified all such multipliers a (there are apparently over 500,000,000), and evaluated them using a variety of criteria. They identified 5 multipliers as being the best. One of these, with $a = 950,706,376$, is available as an option in IMSL. (Another IMSL option uses $a = 397,204,094$, which scored better in Fishman and Moore's evaluation than 7^5, but not as well as the best multipliers in that evaluation.)

The other commonly used value of m in (7.1) is 2^{32}. Since this is an even number, it is not possible to get a period of length $m - 1$, but if $a = 5 + 8l$ for some l, then the period will be 2^{30} for appropriately chosen starting values (the starting value must at least be an odd number). A commonly used value of a for this m is $a = 69069$. For certain choices of a and b in (7.3), it is possible to get a generator of period 2^{32}; for example, $a = 69069$ and $b = 23606797$.

Random number generators require starting values to initialize the sequence. These starting values are usually referred to as *seeds*. As noted above, the performance of some generators can depend on the initial values.

When using a generator of type (7.1), depending on the magnitude of a and m, it is often true that the product ak_i cannot be represented exactly using standard arithmetic. If the multiplication is done in standard integer arithmetic, it will often overflow. Thus special procedures are needed to compute k_{i+1} without explicitly forming the product ak_i. One such algorithm (due to Schrage, 1979) is given in Section 7.1 of Press *et. al.* (1992). In the special case of $m = 2^{32}$, on a 32 bit machine using the C language unsigned integer data type, the result of the overflow of the multiplication ak_i will be precisely $ak_i \bmod 2^{32}$, so special procedures are not needed in this case. This is probably a major reason for the use of this value of m.

Multiplicative congruential generators can have some autocorrelation in the sequence. A closely related concept is n-dimensional uniformity, which is the degree to which vectors of n consecutive generated values distribute uniformly in the n-dimensional unit cube. That is, given a sequence of uniform pseudo-random numbers, use the first n values to form an n-dimensional vector, the second n values to form a second n-dimensional vector, and so on. If the generated values are iid $U(0, 1)$, then these n-dimensional vectors should be uniformly distributed in the unit cube.

However, it is known that multiplicative congruential generators tend to give n-vectors that concentrate near hyperplanes in n-dimensional space, for some n, and hence tend to exhibit deviations from n-dimensional uniformity. Another difficulty with multiplicative congruential generators is that the leading bits in the representation tend to be more random than the low order bits, so one should not treat subsets of the bits in the representation as separate random numbers.

Because of these problems, multiplicative congruential generators are often modified in some way. A simple modification is to introduce shuffling in the sequence. In this approach, some fixed number (say somewhere between 30 and 150) of values are generated and stored in a table (vector). Then at each step, one element of the table is selected to be the next number reported, and a new value generated to replace that number in the table. The replacement value cannot be the value used to select the element of the table, since then the values in the table would be in a nonrandom order. A standard approach is to use the last value returned to also choose which element to select next. This is implemented in the following algorithm.

Shuffling Algorithm
Given a means of generating a (nearly) iid sequence $u_1, u_2, \ldots$, with $u_i \sim U(0,1)$

- Initialize: `s(i)=`u_i, $i = 1, \ldots, N$, and set `y=s(N)`.
- Generate a new value u, and set `j=int(y*N)+1`, where `int(x)` is the largest integer $\leq$`x`.
- Set `y=s(j)`, `s(j)=`u, and return `y` as the uniform deviate.

The standard congruential generators also have periods that are a little too short for some statistical applications. This problem can be addressed by combining two generators with long but unequal periods to get a new generator with a much longer period. This can help reduce autocorrelation in the sequence, too.

The `ran2()` function in Press *et. al.* (1992), based on L'Ecuyer (1988), uses two parallel streams generated using multiplicative congruential methods, with one stream generated using $a = 40014$ and $m = 2147483563$, and the other using $a = 40692$, $m = 2147483399$. The first stream is shuffled (as above, using the last combined output value to select the next value from the shuffle table), but the second is not (shuffling one is sufficient). The two streams (say v_i and w_i) are combined by setting $u_i = (v_i - w_i)I(v_i \geq w_i) + (1 - v_i + w_i)I(v_i < w_i)$, which has a $U(0,1)$ distribution if both v_i and w_i do. The period of this generator is the product of the periods of the two streams, divided by any common divisors, or about 2.3×10^{18}. This generator should be reliable for virtually any current statistical applications.

Wichmann and Hill (1982) combined 3 streams of uniforms, each with different periods of about 30,000, to get a generator with period $> 10^{12}$. The reason for using such short periods in the streams may have been that then each value of m can be stored as an ordinary integer on a 16-bit computer. If $u_i^{(1)}$, $u_i^{(2)}$ and $u_i^{(3)}$ are the ith value from the 3 streams, then the combined value is simply $u_i^{(1)} + u_i^{(2)} + u_i^{(3)} \bmod 1.0$ (for floating point values, $x \bmod 1.0$ is just the fractional part of x). That this value has a $U(0,1)$ distribution follows from the symmetry of the distribution of $u_i^{(1)} + u_i^{(2)} + u_i^{(3)}$ about the point 1.5. It has also been noted by Zeisel (1986) that this generator is

equivalent to the multiplicative congruential generator with $m = 27,817,185,604,309$ and $a = 16,555,425,264,690$. This generator is simple, and has proven to be excellent for many purposes, but its period is a little shorter than ideal for some applications.

Another way to extend the multiplicative congruential generator concept is to combine several previous generated values to generate the next value, for example, by setting

$$k_{i+1} = a_1 k_i + \cdots + a_l k_{i+1-l} \bmod m.$$

If the a_j are coefficients of a primitive polynomial mod m, then the period of such a generator is $m^l - 1$. The simplest form of these generators is

$$k_{i+1} = a k_{i-l} - k_i \bmod m.$$

Deng and Lin (2000) give a number of different values of a that can be used for $l = 2, 3, 4$ and $m = 2^{31} - 1$. The following FORTRAN subroutine gives one of these combinations for $l = 4$. The resulting generator has a period of roughly 2×10^{37} (it is not a terribly efficient implementation, though).

```
* generate uniforms using a fast multiplicative recursive generator
* from Deng and Lin, American Statistician, 2000
* n (input) = # uniforms to generate
* u (output) = vector of generated uniforms
* iseed (input, output), 4 integers giving the seeds for the generator
* initial seeds must be >0 and < 2^31-1
      subroutine fmrg(n,u,iseed)
      double precision u(n),v,a,m
      integer n,iseed(4),i,j
      data m,a/2147483647.d0,39532.d0/
      do 10 i=1,n
         v=a*iseed(1)-iseed(4)
         if (v.lt.0) then
            v=v+m
         else
            v=v-int(v/m)*m
         endif
         do 11 j=1,3
            iseed(j)=iseed(j+1)
 11      continue
         iseed(4)=v
         u(i)=v/m
 10   continue
      return
      end
```

The following Splus function provides an interface to the FORTRAN routine. If an initial set of starting values is not specified, one is generated from the Splus `runif()` function. The value of the sequence at the end of the call is include as an attribute of the output vector, so it can be used as the input seed for a subsequent call.

```
ufmrg <- function(n,seed=NULL) {
# seed should be a vector of 4 integers giving the initial values for fmrg
  if (length(seed) != 4) seed <- round(runif(4)*2147483646+.5)
  else seed <- round(seed)
  u <- .C('fmrg_',as.integer(n),double(n),as.integer(seed))
  seed <- u[[3]]
  u <- u[[2]]
  attr(u,'seed') <- seed
  u
}
```

There are a variety of other methods that have been proposed for generating pseudo-random numbers. Fibonacci generators start with a sequence of random uniform numbers generated by some other method, and then generate new values as differences of lagged values in the sequence. Two lags that have been proposed are $u_i = u_{i-17} - u_{i-5}$, and $u_i = u_{i-97} - u_{i-33}$ (the first requires an initial sequence of 17 values, the second a sequence of 97). When the lagged difference is negative, 1 is added to the result, so that $0 \leq u_i < 1$ at all steps.

Another type of generator uses bitwise operations on the bits of the current value to generate the next value (the 'bits' are the individual binary digits in the computer representation of a number). In this method the bits of the current value are combined with the bits of a shifted version of the current value using a bitwise exclusive or operation. The exclusive or operator ('^' in C) works the like the ordinary or, except that `1 ^ 1 = 0` (instead of 1 for the ordinary or operator). Thus the bitwise exclusive or is equivalent to adding the bits modulo 2. A typical version is given in the following C function, which also uses the left shift `<<` and right shift `>>` operators.

```
double ush (j)
     unsigned long *j;
{
  double v = 4294967296; /* v= 2^32 */
  *j = *j ^ (*j << 17);
  *j = *j ^ (*j >> 15);
  return (*j/v);
}
```

To understand this, think of `*j` as being represented as 32 binary digits $(b_{32}, b_{31}, \ldots, b_1)$, `*j` $= \sum_{l=1}^{32} b_l 2^{l-1}$. The command `*j = *j ^ (*j << 17)` produces as output $(b_{32} + b_{15}, b_{31} + b_{14}, \ldots, b_{18} + b_1, b_{17}, \ldots, b_1)$ mod 2, where the addition in each component is performed mod 2. That is, the left shift operator shifts the bits to the left, creating a value with the original bits 1 to 15 in positions 18 to 32, and with 0's in positions 1 to 17. This is then combined with the original value using a bitwise exclusive or. The command `*j = *j ^ (*j >> 15)` then converts this to $(b_{32} + b_{15}, b_{31} + b_{14}, \ldots, b_{18} + b_1, b_{17} + (b_{32} + b_{15}), \ldots, b_3 + (b_{18} + b_1), b_2 + b_{17}, b_1 + b_{16})$ mod 2, where again each individual addition is performed mod 2. According to Robert and Casella (1999, p. 42), for most starting values, this generator has a period of $2^{32} - 2^{21} - 2^{11} + 1$, but would have shorter periods for other starting values. These generators are often referred to as

shift generators or Tausworthe generators. See Ripley (1987, Chapter 2) or Kennedy and Gentle (1980, Section 6.2.2) for more details.

The uniform generator in Splus uses a Tausworthe generator, similar to that above, except with the order of the left and right shift operations switched, in combination with the multiplicative congruential generator with $a = 69069$ and $m = 2^{32}$. At each update the values from these 2 sequences are combined using a bitwise exclusive or. Since the 2 generators have different periods, the period of the combined generator is quite long (about 4.6×10^{18}). (There is an additional modification to skip the value 0, when it occurs.) Details are given on Page 167 of Venables and Ripley (1997). This algorithm is adequate for most purposes, but has failed some tests of randomness.

A somewhat similar, but improved, algorithm was given by Marsaglia and and Zaman (1993). Called the Keep It Simple Stupid algorithm, it is also described in Robert and Casella (1999, p. 42). This algorithm combines 3 generators: the linear congruential generator (7.3) with $a = 69069$, $b = 23606797$ and $m = 2^{32}$; the shift generator given above; and a second shift generator, which uses a left shift of 18 followed by a right shift of 13, with all values modulo 2^{31}. A C function for generating a vector of uniforms based on the C code given by Robert and Casella follows.

```
/* KISS generator, Marsaglia and Zaman, 1993; adapted from Robert
   and Casella, 1999, p 42
   n (input) = # uniforms to generate
   u (output) = vector of length n containing U(0,1) deviates
   i, j, k (input) = seeds; j and k cannot be 0
   All arguments must be references to addresses
   Skips 0, so uniforms will always be 0<u<1
*/
void uks (n, u, i, j, k)
     unsigned long *i, *j, *k ;
     long int *n;
     double *u;
{
  double v = 4294967296; /* v= 2^32 */
  long int l;
  unsigned long m;
  l=0;
  while (l< *n) {
    *j = *j ^ (*j << 17);
    *k = (*k ^ (*k << 18)) & 0X7FFFFFFF; /* & 2^31-1 <=> mod 2^31 */
    m = ((*i = 69069 * (*i) + 23606797) +
           (*j ^= (*j >> 15)) + (*k ^= (*k >> 13)));
    if (m>0) { /* skip 0 when it occurs */
      u[l] = m/v;
      l++;
    }
  }
}
```

Here `0X7FFFFFFF` is a base 16 constant equal to $2^{31} - 1$, so the quantity x `&` `0X7FFFFFFF` $= x \bmod 2^{31}$ ('`&`' is the bitwise 'and' operator). It is straightforward to verify by direct computation that the `*i` sequence above has period 2^{32} for any starting seed, the `*k` sequence has period $2^{31} - 1$ for any nonzero starting value, and the `*j` sequence has period $2^{32} - 2^{21} - 2^{11} + 1$ for most seeds. (If the seed of the `*j` sequence is chosen at random, the chance of getting less than the maximal period is about 2^{-11}. One initial value that does give the maximal period is `*j = 2`.) Thus for the seeds which give a maximal period for `*j`, the period of this generator is approximately 4×10^{28}. Even in the unlikely event of choosing a poor seed for the `*j` sequence, the period is still $> 10^{19}$.

The following Splus function gives an interface to the `uks` C function. As with `ufmrg()`, an initial seed can be generated from `runif()`, and the final seed is returned as an attribute of the output vector.

```
uks <- function(n,seed=NULL) {
# seed should be a vector of 3 integers giving the initial values
# for the 3 generators
  if (length(seed) != 3) seed <- round(runif(3)*2147483646+.5)
  else seed <- round(seed)
  u <- .C('uks',as.integer(n),double(n),as.integer(seed[1]),
          as.integer(seed[2]),as.integer(seed[3]))
  seed <- c(u[[3]],u[[4]],u[[5]])
  u <- u[[2]]
  attr(u,'seed') <- seed
  u
}
```

The constants used in the multiplicative congruential generators and the lags used in the Fibonacci and Tausworthe generators are not arbitrary. Poor choices can lead to very nonrandom behavior in the sequences.

In statistical simulations, generally the total time spent generating sequences of uniform random numbers is a tiny fraction of the total computational burden, so the speed of the uniform random number generator is generally not important. If one generator takes twice as long as another, but generated sequences have slightly better statistical properties, then it is better to use the slower generator.

Some methods given above will include 0 in the sequence of random uniform numbers, while others do not. A 0 can create problems in some transformation method algorithms, so it might generally be better to use generators which exclude 0 as a possible value, or are modified to skip 0 when it occurs.

Most generators have some facility for automatically setting an initial seed, but also give the user the option of specifying a value. Automatically chosen seeds might always use the same fixed value, or might use some value based on the clock time and/or process id at the time the function is called. As noted above, some generators have restrictions on the initial values, and others can use arbitrary values.

On repeated calls to a random number generator, it would usually be preferred to generate new sets of random numbers, instead of just repeating the same set over and over (the latter would not be very random). Many random number generators are designed so it is possible to save the information on the current state of the generator, and then to use that information to initialize the generator on the next call, so that it resumes at the same point where it left off in the previous call, as was done with the `ufmrg()` and `uks()` functions above. If the generator has a long period, and it always resumes from the state at the last call, then it will not repeat any section of the sequence of generated values for a very long time. If arbitrarily chosen seeds are used at different calls, there is no guarantee that the generator will not restart in the middle of a recent run (although that is not likely). Thus to avoid reusing the same numbers over and over, it is best to always save the updated seed from the last call, and use the saved value as the starting seed in the next call. Random number generators that use shuffling may also require saving the current shuffle table to resume from the same point on restarting, although it may be sufficient in this case to reinitialize the table from the new seed, instead of continuing with the same sequence. Fibonacci generators require saving the sequence of values back to the longest lag in the generator in order to resume with the same sequence at a new call.

It is also a good idea in simulation experiments to save the value of the initial seed (and other initialization information, if used), so that the experiment could later be repeated on exactly the same generated numbers.

The seed in standard Splus generator is stored as an object `.Random.seed`. `.Random.seed` is a vector of 12 values, each of which is an integer between 0 and 63. According to Venables and Ripley (1997, p. 167), the seed for the multiplicative congruential part of the Splus generator is

$$\sum_{i=1}^{6} \texttt{.Random.seed[i]} 2^{6(i-1)},$$

and the seed for the Tausworthe generator is

$$\sum_{i=1}^{6} \texttt{.Random.seed[i+6]} 2^{6(i-1)},$$

in which case the first component of `.Random.seed` should be odd, and the 6th and 12th components should be < 4 (so the seeds are $< 2^{32}$). (There may be other restrictions as well, so it is best not to use arbitrarily selected seeds for this generator.) There is a default value of `.Random.seed` in a system directory, which never changes. When any of the functions generating random numbers in Splus are called, the current value of `.Random.seed` is read from its first occurrence in the current search path. When the call is completed, the updated value of `.Random.seed` is written to the current working directory. If that value is available the next time a function using random numbers is called, the random number generator will read that value and resume where it had left off at the last call. If the updated value is not available, however, it will reinitialize using the system default seed. Splus also comes with a function `set.seed()` that can be used to pick one of 1000 different preset seeds. These preset seeds are chosen to be far apart in the overall sequence, so overlap between among sequences generated from these different starting values would require very long simulation runs.

Random numbers play important roles in data encryption, and conversely, encryption algorithms provide a means for generating random numbers (see the `ran4` routine in Press *et. al.*, 1992, for

an example). Good data encryption requires random numbers that arise from an algorithm that is not easily reproducible. Researchers at Silicon Graphics have found that numbers obtained from digitized images of Lava-Lite lamps are useful source of random numbers that are not easily reproducible. They use values obtained from these images as initial seeds in a pseudo random number generator, which is then run for a period of time also determined by the digitized image, to generate random values for use in encryption. See `http://lavarand.sgi.com` for more information.

7.1.2 Transformation Methods

In principle, once a sequence of independent uniform random numbers is available, samples from virtually any distribution can be obtained by applying transformations to the sequence.

Inverse CDF.

For a cumulative distribution function (CDF) $F(x)$, an inverse CDF can be defined by $F^{-1}(u) = \inf\{x : F(x) \geq u\}$. If X is a continuous random variable with CDF $P(X \leq x) = F(x)$, then

$$P[F(X) \leq u] = P[X \leq F^{-1}(u)] = F[F^{-1}(u)] = u,$$

by the continuity of F, so $F(X) \sim U(0,1)$. Conversely, if $U \sim U(0,1)$, then $P[F^{-1}(U) \leq x] = F(x)$. Thus if $F^{-1}(u)$ can easily be calculated, then given an iid $U(0,1)$ sample $\{u_1, \ldots, u_n\}$, an iid sample $\{x_1, \ldots, x_n\}$ from F can be obtained by setting $x_i = F^{-1}(u_i)$. For example, the CDF of the logistic distribution is $F(x; \mu, \sigma) = \{1 + \exp[-(x-\mu)/\sigma]\}^{-1}$. Thus $F^{-1}(u; \mu, \sigma) = -\log(1/u - 1)\sigma + \mu$, and this formula can be used to obtain samples from the logistic distribution from iid uniform samples.

For continuous distributions, the same results hold if the survivor function $S(x) = 1 - F(x)$ is used in place of $F(x)$, and this is sometimes more convenient. The Weibull distribution with shape γ and scale α has survivor function $S(x; \alpha, \gamma) = \exp[-(x/\alpha)^\gamma]$, which has inverse $S^{-1}(u; \alpha, \gamma) = [-\log(u)]^{(1/\gamma)}\alpha$. Again if $u_i \sim U(0,1)$, then setting $x_i = S^{-1}(u_i; \alpha, \gamma)$ gives an x_i from this Weibull distribution. If the CDF were used instead of the survival function, the only difference would be replacing u by $1 - u$. This would give the same distribution, since u and $1 - u$ have the same distribution.

For distributions such as the normal and gamma, there are not simple closed form expressions for the inverse CDF, so the inverse CDF method is not so straightforward. For the normal distribution, there are good analytical approximations to the inverse CDF, and using them would often be adequate. However, the methods discussed below, based on polar coordinates transformations, are simpler and faster.

The obvious transformations are not always the fastest. For example, there are algorithms for generating exponential random variables that do not require taking logs, which are faster than the standard method (the obvious transformation is $x_i = -\log(u_i)$). When a simple transformation like this is available, its simplicity still might be preferred to a faster algorithm, unless the simple method turns out to be too slow.

Polar Coordinates and the Normal Distribution.

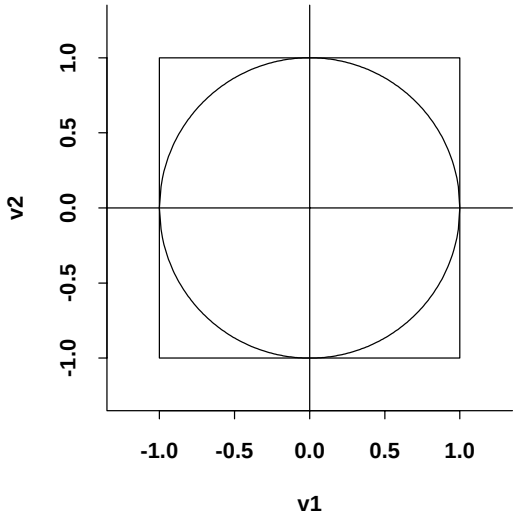

Figure 7.1: The unit disk $v_1^2 + v_2^2 < 1$ and the square $\{(v_1, v_2) : -1 < v_i < 1\}$. If (v_1, v_2) has the uniform distribution over the square, then conditional on lying in the unit disk, its distribution is uniform over the unit disk.

Suppose X, Y are independent $N(0, 1)$ random variables. Consider the polar coordinates transformation defined by

$$X = R\cos(\theta), \quad Y = R\sin(\theta), \quad R \geq 0, \quad 0 \leq \theta < 2\pi.$$

It is easily verified that θ is uniformly distributed on $[0, 2\pi)$, and $R^2 = X^2 + Y^2$ has a χ^2_2 distribution (chi-square with 2 degrees of freedom), so $P(R > r) = P(R^2 > r^2) = \exp(-r^2/2)$. Thus given two independent uniform deviates u_1 and u_2, two independent normal deviates x_1 and x_2 can be generated by setting $R = \{-2\log(u_1)\}^{1/2}$, $\theta = 2\pi u_2$, $x_1 = R\cos(\theta)$ and $x_2 = R\sin(\theta)$.

The use of the sine and cosine functions can also be avoided. Suppose (v_1, v_2) is drawn from the distribution that is uniform on the unit disk in two-dimensional Euclidean space; that is, the density of (v_1, v_2) is $I(v_1^2 + v_2^2 \leq 1)/\pi$. Let θ be the counterclockwise angle from the positive v_1 axis to the point (v_1, v_2). Then $\cos(\theta) = v_1/(v_1^2 + v_2^2)^{1/2}$ and $\sin(\theta) = v_2/(v_1^2 + v_2^2)^{1/2}$. Also, $P(v_1^2 + v_2^2 \leq u) = (\pi u)/\pi = u$ (that is, the ratio of the area of the disk with radius $u^{1/2}$ to the area of the disk with with radius 1), so $v_1^2 + v_2^2 \sim U(0, 1)$. Furthermore, $v_1^2 + v_2^2$ is independent of the angle θ. Thus an appropriate R for the polar coordinates transformation can be obtained from $\{-2\log(v_1^2 + v_2^2)\}^{1/2}$. Thus given values (v_1, v_2), two normal deviates x_1 and x_2 can be obtained by computing $u = v_1^2 + v_2^2$, $w = \{-2\log(u)/u\}^{1/2}$, $x_1 = v_1 w$ and $x_2 = v_2 w$.

There remains the problem of obtaining (v_1, v_2). This is most easily accomplished by generating two uniforms u_1 and u_2, and setting $v_i = 2u_i - 1$. Then (v_1, v_2) is uniformly distributed over the square $\{(v_1, v_2) : -1 < v_i < 1\}$. The subset of such points lying in the unit disk (which is contained in this square), will be uniformly distributed over the unit disk; see Figure 7.1. Thus an algorithm to generate (v_1, v_2) uniformly over the unit disk is as follows.

1. Generate (v_1, v_2) uniformly from the set $\{(v_1, v_2) : -1 < v_i < 1\}$.

2. If $v_1^2 + v_2^2 \geq 1$, then reject (v_1, v_2) and try again. If $v_1^2 + v_2^2 < 1$, then keep (v_1, v_2).

This algorithm is an example of rejection method sampling, which is described in more detail in the next section. Since the ratio of the area of the disk to that of the square is $\pi/4 \doteq .785$, only about 21.5% of the candidate pairs will be rejected.

The idea of multivariate change of variables underlying the polar coordinates transformation above can be used to obtain a variety of other distributions. For example, if $Z \sim N(0,1)$ and $V \sim \chi^2_\nu$, then $T = Z/(V/\nu)^{1/2}$ has a t_ν distribution.

To obtain a vector $x = (x_1, \ldots, x_p)'$ from the p-variate normal distribution with mean vector μ and covariance matrix V, first compute the Choleski factorization of $U'U$ of V. Then generate $z_1, \ldots, z_p$ iid $N(0,1)$, and set $x = U'(z_1, \ldots, z_p)' + \mu$. Then $\mathrm{Var}(x) = U'U = V$, and $\mathrm{E}(x) = \mu$, as required.

Discrete Distributions.

All of the preceding applies to continuous distributions. For a random variable X with a discrete distribution with a small number of support points, $s_1, \ldots, s_k$, set $p_j = \sum_{i=1}^{j} P(X = s_j)$. Then independent observations x_i from this distribution can be generated by generating uniform deviates u_i, and setting $x_i = s_j$ if $p_{j-1} < u_i \leq p_j$ (where $p_0 = 0$). Essentially this is inverting the CDF by direct search. If k is small, then only a few comparisons are needed to determine j, while if k is large then searching for the value of j may take a significant amount of time, and other techniques, such as rejection methods (discussed below) may be needed.

7.1.3 Rejection Sampling

There are two basic principles underlying rejection method sampling. The first (used above in the algorithm for generating normal deviates) is that if $B \subset A \subset R^k$, and if X has a uniform distribution on A, then the conditional distribution of $X|X \in B$ is a uniform distribution on B. Thus if there is a means available to sample from the uniform distribution on A, then a sample from the uniform distribution on B can be obtained by sampling from A, and keeping just those points that are also in B. The main applications are to situations where B is an irregularly shaped set that is difficult to sample from directly, and A is a more regular set (such as a rectangle) that is easier to sample from.

The second principle concerns regions bounded by densities. Suppose $f(x)$ is a density on $D \subset R^k$, and let $B = \{(x', y)' : x \in D, y \in R^1, 0 < y < f(x)\}$. Suppose $(X', Y)'$ is a random vector with a uniform distribution on B. Then the distribution of X is given by the density $f(x)$ on D. That is, since $(X', Y)'$ has a uniform distribution on B, $P(X \leq c)$ is the ratio of the volume of the set $C = B \cap \{(x', y)' : x \leq c\}$ to the volume of B. The volume of C is

$$\int_{D \cap \{x \leq c\}} \int_0^{f(x)} 1 \, dy \, dx = \int_{D \cap \{x \leq c\}} f(x) \, dx,$$

and the volume of B is $\int_D f(x) \, dx = 1$, since f is a density on D, so the ratio of volumes is $\int_{D \cap \{x \leq c\}} f(x) \, dx$, and X has the density f, as claimed.

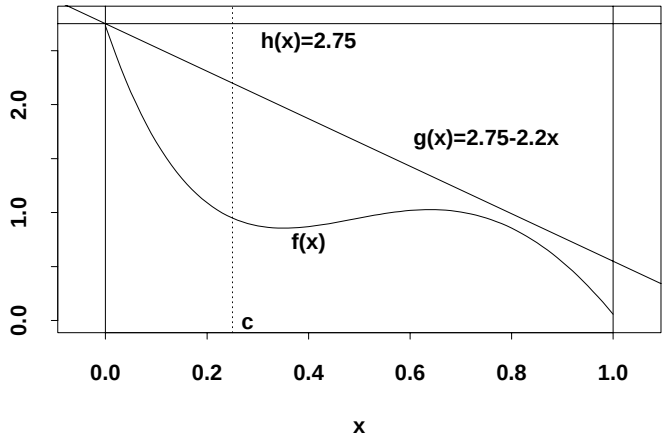

Figure 7.2: Density $f(x)$ and possible dominating functions $h(x)$ and $g(x)$.

To visualize this, suppose $k = 1$, $D = [0, 1]$, and the density is given by the curve $f(x)$ in Figure 7.2. The set B in the previous paragraph is the region bounded by the x-axis, the vertical lines at 0 and 1, and the curve $f(x)$. Since $f(x)$ is a density, the area of B is $\int_0^1 f(x)\,dx = 1$. If $c = .25$, as indicated by the dotted vertical line in the figure, then C is the region bounded by the x-axis, the vertical lines at 0 and 0.25, and the curve $f(x)$. The area of this region is $\int_0^{.25} f(x)\,dx = P(X \le c)$, where (X, Y) has a uniform distribution on B.

Rejection sampling combines the two principles given above. Given a density $f(x)$ defined on a set D, again let $B = \{(x', y)' : x \in D, y \in R^1, 0 < y < f(x)\}$. As just discussed, a sample from $f(x)$ can be generated by sampling points uniformly from B. But $f(x)$ may have an irregular shape, and it may not be easy to directly sample points uniformly from B. Let $g(x)$ be a function satisfying $g(x) \ge f(x)$ for all $x \in D$. Define $A = \{(x', y)' : x \in D, y \in R^1, 0 < y < g(x)\}$. Then $B \subset A$, and from the first principle, points can be sampled uniformly from B by first sampling uniformly from A, and rejecting those points that are not also in B. The function $g(x)$ satisfying $g(x) \ge f(x)$ is called a dominating function. Thus if $(X', Y)'$ is sampled uniformly from A, and points retained only if they are also in B, then the retained X values will have density $f(x)$.

In Figure 7.2, the function $h(x) = 2.75$ is a dominating function. A variate can be generated from the density $f(x)$ by first generating $X \sim U(0, 1)$ and $Y \sim U(0, 2.75)$. Then (X, Y) has a uniform distribution on the region $\{(x, y)' : 0 < x < 1, 0 < y < 2.75\}$. If $Y \le f(X)$, then X is accepted. If $Y > f(X)$ then X is rejected, and another attempt made. Clearly the efficiency of this algorithm is affected by the size of the region A, in that the proportion of points accepted is the ratio of the area of B to the area of A. In Figure 7.2, the function $g(x)$ is also a dominating function, and since it leads to a smaller bounding region A, use of this function will lead to rejecting fewer candidate points. Generating points uniformly from the region $A = \{(x, y)' : 0 < x < 1, 0 < y < 2.75 - 2.2x\}$ is not quite as straightforward as when $h(x)$ is used as the dominating function, but is still fairly easy. $g(x)$ can be normalized to a density $g^*(x) = (5 - 4x)/3$, which has corresponding CDF $G^*(x) = (5x - 2x^2)/3$. The inverse CDF

method to sample from G^* then requires solving a quadratic equation. Once X is generated from g^*, sample $Y \sim U(0, g(X))$, and the pair (X, Y) will then be uniformly distributed on A. Then if $Y \leq f(X)$, the point X is accepted, and if not, another attempt is made.

A better way to sample from g^* in the previous paragraph is to note that $g^*(x) = I(0 < x < 1)[1/3 + (2/3)(2 - 2x)]$, so g^* is the mixture of a $U(0, 1)$ and the triangular distribution with density $I(0 < x < 1)(2 - 2x)$. This latter density is the same as that of $|u_1 - u_2|$, where u_1 and u_2 are iid $U(0, 1)$. Thus an x from g^* can be obtained from 3 independent uniform deviates u_1, u_2, u_3 through

$$x = I(u_1 < 1/3)u_2 + I(u_1 \geq 1/3)|u_2 - u_3|.$$

Mixtures of linear combinations of uniforms can be used to obtain dominating functions for many densities over a finite interval. Combined with other approximations in the tails, this method can be used to sample from many univariate densities.

Another important feature of rejection sampling is that the density need not be normalized for the method to work. This often arises in Bayesian inference problems, where determining the normalizing constant for the posterior density requires evaluation of a large dimensional integral. If the density is only known to within a normalizing constant, the sets A and B can still be defined as before. The dominating function is still any function that is everywhere at least as large as the unnormalized density. The accepted points will still be a random sample from the normalized density. This follows because normalizing the density just requires rescaling the y coordinated in the definitions of A and B, by exactly the same factor in both sets, and the distribution of the X coordinate is not affected by the rescaling.

Example 7.1 Consider generating samples from the $\Gamma(a, 1)$ distribution, which has density proportional to $f^*(x) = x^{a-1} \exp(-x) I(x > 0)$, where $a > 1$ (the case $a = 1$ is trivial, and for $0 < a < 1$, values can be generated using the fact that if $Y \sim \Gamma(a + 1, 1)$ and independently $U \sim U(0, 1)$, then $YU^{1/a} \sim \Gamma(a, 1)$). $f^*(x)$ has a maximum value of $(a - 1)^{a-1} \exp(-a + 1)$ at $x = a - 1$. It turns out that the function

$$g(x) = \frac{(a - 1)^{a-1} \exp(-a + 1)}{1 + (x - a + 1)^2/(2a - 1)}$$

dominates $f^*(x)$. This is believable since $g(a - 1) = f^*(a - 1)$, and g has heavier tails than f^*. Also,

$$-\partial^2 \log[g(x)]/\partial x^2 \Big|_{x=a-1} = (a - 1/2)^{-1} < (a - 1)^{-1} = -\partial^2 \log[f^*(x)]/\partial x^2 \Big|_{x=a-1},$$

so $g(x)$ has a slightly broader peak than $f^*(x)$. The functions f^* and g are shown in Figure 7.3 for $a = 2, 6$.

Since $g(x)$ is proportional to a Cauchy density, it is easy to generate data from the region A in this case (again, A is the region between $g(x)$ and the x-axis). Inverting the Cauchy CDF, it follows that an observation from the Cauchy distribution with location $a - 1$ and scale $(2a - 1)^{1/2}$ can be generated as

$$x_i = (2a - 1)^{1/2} \tan(\pi u_i - \pi/2) + (a - 1),$$

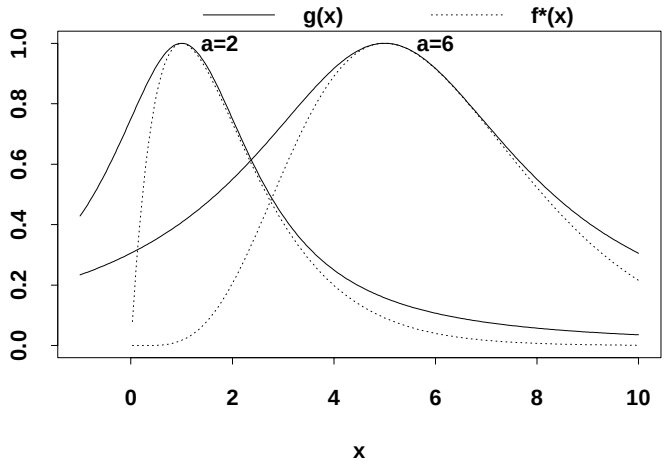

Figure 7.3: Non-normalized gamma density $f^*(x)$ and dominating function $g(x)$, for $a = 2, 6$. Both f^* and g are divided by $(a-1)^{a-1}\exp(-a+1)$.

where u_i is a $U(0,1)$ deviate (because of the periodic nature of the tangent function, the constant $\pi/2$ can also be dropped without changing the overall distribution). As in the polar coordinate transformation for the normal distribution, it is faster to generate the value of $\tan(\pi u_i - \pi/2)$ directly from pairs uniformly distributed over the unit disk. If (v_1, v_2) is such a pair, then the required tangent is given by v_2/v_1. The complete algorithm then is as follows:

1. Generate u_1 and u_2 from the $U(0,1)$ distribution, and set $v_i = 2u_i - 1$, $i = 1, 2$.
2. If $v_1^2 + v_2^2 \geq 1$ reject (v_1, v_2) and return to step 1.
3. If $v_1^2 + v_2^2 < 1$, set $x = (2a-1)^{1/2} v_2/v_1 + a - 1$.
4. If $x \leq 0$, reject x and return to step 1.
5. If $x > 0$, generate $u_3 \sim U(0,1)$. If $y = u_3 g(x) \leq f^*(x)$ then retain x; otherwise reject x and return to step 1.

There are many other methods for generating observations from gamma distributions. See Section 6.5.2 of Kennedy and Gentle (1980) for an extended discussion. □

Rejection sampling is a powerful tool that could be applied to many problems. The main difficulty is establishing that the dominating function $g(x)$ really does dominate. If $g(x) < f(x)$ at some values, then the sample will have the distribution with density proportional to $\min\{f(x), g(x)\}$, which probably is not what is wanted.

Hazard based rejection sampling.

Consider generating data from a continuous distribution with hazard function

$\lambda(t) = f(t)/[1 - F(t)]$. If the cumulative hazard $\Lambda(t) = \int_0^t \lambda(u)\,du$ has a closed form inverse, then the inverse CDF transformation can be applied, since $F(t) = 1 - \exp[-\Lambda(t)]$. If $\Lambda(t)$ does not have a closed form inverse, or is too complicated, then straightforward transformations may not be possible. In this case a rejection sampling algorithm based directly on the hazard function can be used.

Suppose $\lambda_d(t)$ is another hazard rate function with $\lambda(t) \le \lambda_d(t)$ for all t, and that it is easy to generate data from the distribution with hazard rate $\lambda_d(t)$. The algorithm to generate X from the distribution with hazard $\lambda(t)$ is quite simple. First generate T_1 from the distribution with hazard $\lambda_d(t)$ and $U_1 \sim U(0,1)$. If

$$\lambda_d(T_1)U_1 \le \lambda(T_1),$$

then set $X = T_1$. If not, generate T_2 from the distribution of $Y|Y > T_1$, where Y has the distribution with hazard rate $\lambda_d(t)$, independent of T_1. The hazard function of this conditional distribution is $I(t > T_1)\lambda_d(t)$. Also generate $U_2 \sim U(0,1)$. If

$$\lambda_d(T_2)U_2 \le \lambda(T_2),$$

set $X = T_2$. Otherwise, continue in the same fashion. At stage j, generate T_j from the conditional distribution with hazard rate $I(t > T_{j-1})\lambda_d(t)$, and $U_j \sim U(0,1)$. If

$$\lambda_d(T_j)U_j \le \lambda(T_j), \tag{7.4}$$

then set $X = T_j$; otherwise, continue.

Once the first X is generated, if the process is continued, retaining only the T_j that satisfy (7.4), then the retained values will be the event times of a nonhomogeneous Poisson process with mean function $\Lambda(t)$. To see this, let $N_d(t)$ be the number of $T_i \le t$, and let $N(t)$ be the number of $T_i \le t$ that also satisfy (7.4) (that is, the number of retained $T_i \le t$). Since $N_d(t)$ is a nonhomogeneous Poisson process with mean function $\Lambda_d(t) = \int_0^t \lambda_d(u)\,du$,

$$P[N_d(t+h) - N_d(t) = 1] = \lambda_d(t)h + o(h).$$

Then

$$\begin{aligned} P[N(t+h) - N(t) = 1] &= P[N_d(t+h) - N_d(t) = 1, U < \lambda(t)/\lambda_d(t)] + o(h) \\ &= P[N_d(t+h) - N_d(t) = 1]P[U < \lambda(t)/\lambda_d(t)] + o(h) \\ &= [\lambda_d(t)h + o(h)]\lambda(t)/\lambda_d(t) + o(h) \\ &= \lambda(t)h + o(h), \end{aligned}$$

where $U \sim U(0,1)$ independently of the N_d process. Thus $N(t)$ has the distribution of a nonhomogeneous Poisson process with mean function $\Lambda(t)$. From this it also follows that the first retained value (the time of the first jump in $N(t)$, which is value of X defined above) has the distribution of a survival time with hazard rate $\lambda(t)$.

Example 7.2 To illustrate, consider the hazard function

$$\lambda(t) = .1 + .2\exp(-t),$$

which has cumulative hazard

$$\Lambda(t) = .1t + .2[1 - \exp(-t)].$$

Although $\Lambda(t)$ is simple, it does not have a closed form inverse. Consider $\lambda_d(t) = .3$. This is the hazard of an exponential distribution, and due to the memoryless property of the exponential, the conditional distribution of $Y|Y > t$ is just an exponential with origin t, so generating observations from the required conditional distributions is trivial (that is, the quantities $T_j - T_{j-1}$ are iid, where $T_0 = 0$). Below is some inefficient Splus code to illustrate the method.

```
> k1 <- .1
> k2 <- .2
> ld <- function(t) k1+k2
> l <- function(t) k1+k2*exp(-t)
> n <- 5000
> re <- rexp(5*n)/(k1+k2)
> ru <- runif(5*n)
> y <- rep(0,n)
> k <- 1
> for (i in 1:n) {
+   u <- re[k]
+   while (ru[k] > l(u)/ld(u)) {
+     k <- k+1
+     u <- u+re[k]
+   }
+   y[i] <- u
+   k <- k+1
+ }
> k
[1] 12664
> # survivor function transform
> uu <- exp(-k1*y-k2*(1-exp(-y)))
> hist(uu,20)
> # chi-square goodness-of-fit test
> ff <- table(cut(uu,0:20/20))
> X2 <- sum((ff-n/20)^2/(n/20))
> 1-pchisq(X2,19)
[1] 0.933394
```

The output from the `hist()` command is given in Figure 7.4. The random number functions in Splus have a certain amount of overhead, including reading and writing the value of `.Random.seed`, so it is usually more efficient to generate random numbers in large blocks, as above. However, nearly twice as many values were generated (25,000) as were needed (12,664). Also $12644/5000 \doteq 2.53$ exponentials and uniforms were needed for each observation generated from the hazard $\lambda(t)$.

If y_i denotes the ith generated value, then $u_i = S(y_i) = \exp[-\Lambda(y_i)]$ should have a $U(0, 1)$ distribution. The histogram suggests close agreement with this distribution. The χ^2

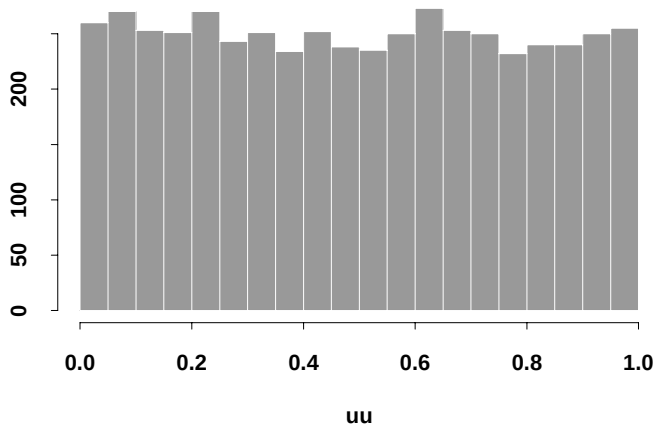

Figure 7.4: Histogram of the survival function of a sample of size $n = 5000$ generated by the hazard rejection sampling method.

goodness-of-fit test in the Splus code uses 20 cells, each of which has $p = .05$. The general formula for the χ^2 test statistic is just

$$\sum_{j=1}^{k}(f_j - np_j)^2/(np_j),$$

where f_j is the observed count in the jth cell, p_j is the probability of an observation falling in the jth cell, n is the sample size, and k is the number of cells. This statistic is asymptotically chi-square with $k - 1$ degrees-of-freedom if the sample is from the intended distribution. The p-value again suggests close agreement with the hypothesized distribution. □

7.1.4 Testing Random Number Generators

Random number generators can be informally evaluated by plotting histograms of large generated samples, to see if they reasonably approximate the intended distribution. Computing autocorrelations at various lags is also recommended, to check that they are close to 0. Both of these methods can be formalized into statistical tests. The fit of the distribution can be evaluated using chi-square (as above), Kolmogorov-Smirnov and other types of goodness-of-fit tests. Formal tests of whether the autocorrelation equals 0 can also be given.

Almost any aspect of the sequence of generated numbers can be examined to see if it exhibits nonrandom behavior. A number of tests have been proposed to examine patterns of digits. Digits in any base can be obtained from the sequence using a variety of methods, and the frequency of individual digits and patterns of digits examined. The runs test examines the distribution of the length of increasing and decreasing sequences of digits. Lattice tests examine how uniformly consecutive n-tuples of generated values appear in an n-dimensional hypercube. These and other procedures are discussed in Section 6.3.2 of Kennedy and Gentle (1980). See also Fishman and Moore (1982). George Marsaglia has developed a standard battery of tests called DIEHARD.

Source code for these tests is available on his web page at Florida State University, Department of Statistics.

7.1.5 Splus

Splus has many functions for generating random values from different distributions. See the help files for the functions `rbeta()`, `rcauchy()`, `rchisq()`, `rexp()`, `rf()`, `rgamma()`, `rlogis()`, `rlnorm()`, `rnorm()`, `rstab()`, `rt()`, `runif()`, `rbinom()`, `rgeom()`, `rhyper()`, `rpois()`, `rwilcox()`, and `sample()`, for details.

As discussed above, Splus stores the seed in an object called `.Random.seed`. If this object is not in the current user directories in the search path, the system default value will be used.

As already mentioned, Splus random number functions have a certain amount of overhead associated with each call (such as reading and writing `.Random.seed`), so there is some advantage to generating random numbers in large blocks.

7.2 Statistical Simulations

When new methods of analyzing data are developed, it is usually appropriate to perform some simulations to evaluate the methodology. To take a concrete example, consider the partial likelihood analysis of the proportional hazards model. For independent censored failure time data (t_i, δ_i, z_i), $i = 1, \ldots, n$, where t_i is the failure or censoring time, $\delta_i = 1$ for observed failures and $= 0$ for censored observations, and z_i is a vector of covariates, the proportional hazards model specifies that the failure hazard for the ith case is of the form

$$\lambda(t|z_i) = \lambda_0(t) \exp(\beta' z_i),$$

where $\lambda_0(t)$ is an unspecified underlying hazard function. The partial likelihood analysis estimates the regression parameters β by maximizing the log partial likelihood

$$l(\beta) = \sum_i \delta_i [\beta' z_i - \log\{\sum_j I(t_j \geq t_i) \exp(\beta' z_j)\}].$$

Let $\hat{\beta}$ be the maximizing value. Th estimator $\hat{\beta}$ should be consistent and asymptotically normal, and $V = (v_{ij}) = [-\partial^2 l(\hat{\beta})/\partial\beta\partial\beta']^{-1}$ should give a consistent estimate of $\mathrm{Var}(\hat{\beta})$. Here are some questions that might be of interest:

1. Are the sizes of Wald, score and likelihood ratio tests of hypotheses on the parameters, using asymptotic approximations to critical regions, close to the nominal levels (and related, what are the powers of such tests)?

2. What is the finite sample bias of the estimators?

3. How efficient are the estimators compared with estimators from fully specified parametric models.

4. How accurate are the variance estimators in finite samples.

5. Do confidence intervals based on the asymptotic distributions give correct coverage in small samples?

All these issues are straightforward to examine through simulations, although a thorough investigation would require examining different censoring patterns, different numbers of covariates, different covariate distributions, different true values of the parameters, and various sample sizes. (Since the estimators are invariant to monotone transformations of the time axis, the results do not depend on the underlying hazard used, except through the censoring pattern.)

Simulations to examine all of the above questions have the same general structure. First, data sets of a specified sample size are generated from a specified distribution. In regression problems, either one fixed covariate pattern can be used throughout the simulation run, or new sets of covariates can be generated from a specified covariate distribution for each sample. In the first case the results will be conditional on the fixed covariate distribution, while in the second case the results will reflect an average over the distribution of the covariates. For each simulated data set, the analysis is then carried out exactly as it would be for real data. The results from the individual data sets are then combined to estimate quantities addressing the questions of interest.

To be more specific, consider a simulation to address questions 1, 2 and 4 above. The case of 2 fixed binary covariates z_1 and z_2, with each of the 4 covariate combinations occurring for 1/4 of the observations, will be considered, although as already mentioned, a thorough investigation would require looking at many different covariate configurations, and varying the number of covariates. For simplicity, exponential distributions will be used for both the failure and censoring distributions, although again in a thorough investigation the effect of different censoring patterns should also be investigated. The failure hazard is $\lambda(t|z_1, z_2) = \exp(\beta_1 z_1 + \beta_2 z_2)$, corresponding to a baseline hazard of 1, and the censoring distribution will also have hazard rate equal to 1. In the particular case examined below, the sample size is $n = 100$.

For a simulated sample s, $s = 1, \ldots, S$, the estimated coefficients $\hat{\beta}_1^{(s)}$ and $\hat{\beta}_2^{(s)}$, and their estimated variances $v_{11}^{(s)}$ and $v_{22}^{(s)}$ (the diagonal elements of the inverse information) are computed. The test statistics $T_j^{(s)} = \hat{\beta}_j/(v_{jj}^{(s)})^{1/2}$ for the hypotheses $H_0 : \beta_j = 0$ are also computed. The test statistics should be asymptotically $N(0, 1)$ under the null hypothesis.

A simulation run with $\beta_1 = \beta_2 = 0$ allows examination of accuracy of the asymptotic distribution for determining the critical values of the test. Let z_α be the αth quantile of the standard normal distribution. The standard approximation uses the critical region $|T_j| \geq z_{1-\alpha/2}$ for two-sided tests of nominal size α. The quantities

$$\sum_{s=1}^{S} I(|T_j^{(s)}| \geq z_{1-\alpha/2})/S, \quad j = 1, 2,$$

which are just the proportions of simulated samples where each test rejects H_0, can be used to estimate the actual sizes of the tests using these approximate critical regions.

Similarly, if β_1 and β_2 are the true parameter values, then

$$\sum_{s=1}^{S} \hat{\beta}_j^{(s)}/S - \beta_j$$

estimates the bias in $\hat{\beta}_j$. Also the true variance $\mathrm{Var}(\hat{\beta}_j)$ can be estimated by

$$\hat{\mathrm{Var}}(\hat{\beta}_j) = \sum_{s=1}^{S} [\hat{\beta}_j^{(s)} - \sum_{r=1}^{S} \hat{\beta}_j^{(r)}/S]^2/(S-1),$$

the usual sample variance over the simulations, and

$$\sum_{s=1}^{S} v_{jj}^{(s)}/S - \hat{\mathrm{Var}}(\hat{\beta}_j)$$

estimates the bias in v_{jj} as an estimate of $\mathrm{Var}(\hat{\beta}_j)$ (often the two terms are reported separately, instead of just giving the estimate of bias in the variance estimator).

For any finite n, there is a positive probability that the partial likelihood will be maximized in the limit as $\hat{\beta}_j \to \pm\infty$. Thus strictly speaking, $\mathrm{E}(\hat{\beta}_j)$ does not exist, and $\mathrm{Var}(\hat{\beta}_j)$ is infinite. To avoid this problem, interpret means and variances as conditional on those outcomes that give finite parameter estimates. In practice, tests for divergence to $\pm\infty$ can be built into the model fitting algorithm, and samples which appear to be giving unbounded estimates excluded from the results, if any are encountered. For many parameter configurations, the probability of unbounded parameter estimates is negligible, though, and can be ignored.

It is often convenient to put the commands to run a simulation in a batch file. Here this takes the form of a shell script, given below. A shell script is a file containing commands to be executed at the shell level. The name of the file below is `coxsim1.s`. In this file, the initial `#` identifies the script as a C-shell script, so it will be executed under the C-shell. The second line executes the Splus command. The `<<` means that the following lines are to be used as input to the Splus command. The characters after that are delimiters. The second occurrence of this string identifies where the Splus commands end and control returns to the shell. The quotes in this string tell the shell not to change anything in the Splus commands. Without these the shell would attempt to perform expansion of shell variables, and give special characters their shell interpretation. Use of `%%` as the delimiting characters is entirely arbitrary.

```
#
Splus << '%%'
NT <- 845 # number simulated samples
N <- 100   # sample size--must be a multiple of 4
beta <- c(0,0) # true coefficients
lamc <- 1  # hazard rate for censoring distribution
fp <- 0
out <- matrix(-1,nrow=4,ncol=NT) # store the results
z1 <- rep(c(0,1,0,1),N/4) #covariates stay fixed throughout the run
z2 <- rep(c(0,0,1,1),N/4)
thr <- exp(c(0,beta,beta[1]+beta[2])) # true exponential hazards
                        # for different covariate combinations
gendata <- function(n,thr,lamc) { #function to generate censored outcomes
  y <- rexp(n)/thr
  cen <- rexp(n,rate=lamc)
```

```
  Surv(pmin(y,cen),ifelse(y <= cen,1,0))
}

print(.Random.seed) # always print the seed for future reference
for ( i in 1:NT) { # simulation loop
  u <- gendata(N,thr,lamc)
  fp <- fp+mean(u[,2])
  u <- coxph(u~z1+z2)
  out[,i] <- c(u$coef,diag(u$var))
}
print(memory.size()) # can become excessive
print(fp/NT) # proportion of observed failures
cat('true beta=',format(beta),'\n')
u1 <- apply(out,1,mean)
u2 <- apply(out,1,var)
cat('estimated bias\n')
print(u1[1:2]-beta)
cat('standard error of estimated bias\n')
print(sqrt(u2[1:2]/NT))
cat('average information\n')
print(u1[3:4])
cat('standard error of ave inform\n')
print(sqrt(u2[3:4]/NT))
cat('Estimated variance\n')
print(u2[1:2])
cat('approximate standard error of estimated variance\n')
print(sqrt(c(var((out[1,]-u1[1])^2),var((out[2,]-u1[2])^2))/NT))
#test statistics
t1 <- abs(out[1,]/sqrt(out[3,]))
t2 <- abs(out[2,]/sqrt(out[4,]))
zq <- qnorm(.975)
rp <- c(sum(t1>=zq)/NT,sum(t2>=zq)/NT)
cat('estimated rejection probabilities of tests\n')
print(rp)
cat('standard errors of est rejection probs of tests\n')
print(sqrt(rp*(1-rp)/NT))
q()
'%%'
```

Since a function to fit the proportional hazards model is built into Splus, and a separate function was written to generate the failure and censoring times, the simulation loop is quite short. In some versions of Splus, there is a substantial efficiency gain from encapsulating all the commands within the simulation `for()` loop in a single function call. That was not critical in Splus 3.4, which was used here. The commands at the end of the program are to tabulate the simulation results and estimate the quantities of interest.

Estimated quantities computed from a finite number of replications of a stochastic simulation will have some sampling error, so it is always good practice to give standard errors for all quantities being estimated. The estimated bias and the average information are just averages over independent samples, so the variance of the estimated bias of $\hat{\beta}_j$ is just $\mathrm{Var}(\hat{\beta}_j)/\texttt{NT}$, which can be estimated by substituting the simulation estimate $\hat{\mathrm{Var}}(\hat{\beta}_j)$. A similar variance estimator can be constructed for the average information. The estimated rejection probability of one of the tests is just the average of NT iid Bernoulli random variables, and hence its variance can be estimated by $\hat{p}(1-\hat{p})/\texttt{NT}$, where $\hat{p}$ is the estimated rejection probability. The exact variance of the estimate of $\mathrm{Var}(\hat{\beta}_j)$ is more complicated. The approximation used above is based on the fact that if $X_1, X_2, \ldots$ are iid with mean μ, variance σ^2 and finite fourth moments, and $\hat{\sigma}^2_S = \sum_{s=1}^S (X_s - \overline{X}_S)^2/(S-1)$ denotes the usual sample variance, then

$$S^{1/2}(\hat{\sigma}^2_S - \sigma^2) \quad \text{and} \quad S^{1/2}\left[\sum_{s=1}^S (X_s - \mu)^2/S - \sigma^2\right]$$

have the same asymptotic normal distribution, and hence the same asymptotic variances. For known μ, an unbiased estimator of

$$\mathrm{Var}\left[\sum_{s=1}^S (X_s - \mu)^2/S\right]$$

is given by

$$\sum_{s=1}^S \left[(X_s - \mu)^2 - \sum_{l=1}^S (X_l - \mu)^2/S\right]^2 /[S(S-1)]. \tag{7.5}$$

Thus a consistent estimate of the asymptotic variance of $\hat{\sigma}^2_S$ is given by substituting $\overline{X}_S$ for μ in (7.5). The estimate used for the variance of $\hat{\mathrm{Var}}(\hat{\beta}_j)$ in the program above is of this form.

Here is the output from a run with `beta <- c(0,0)`, as above.

```
% coxsim1.s
S-PLUS : Copyright (c) 1988, 1996 MathSoft, Inc.
S : Copyright AT&T.
Version 3.4 Release 1 for Sun SPARC, SunOS 5.3 : 1996
Working data will be in /usr/stats/bobg/.Data
 [1] 37 14  5 19 20  0 58 38  4 48 59  2
[1] 7831160
[1] 0.4967219
true beta= 0 0
estimated bias
[1]  0.006329357 -0.020323796
standard error of estimated bias
[1] 0.01066831 0.01052263
average information
[1] 0.08650828 0.08649603
standard error of ave inform
[1] 0.0003438038 0.0003395060
```

```
Estimated variance
[1] 0.09617193 0.09356332
approximate standard error of estimated variance
[1] 0.004680766 0.004530143
estimated rejection probabilities of tests
[1] 0.05917160 0.05207101
standard errors of est rejection probs of tests
[1] 0.008116775 0.007642889
111.81u 0.68s 2:09.93 86.5%
111.97u 0.97s 2:10.40 86.6%
```

With $\beta_1 = \beta_2 = 0$ and the hazard for the censoring distribution equal to 1, the probability of observing the failure time for each case should be exactly 1/2, and the observed proportion of failures (.4967) is very close to this value. The estimated bias in $\hat{\beta}_2$ is just under 2 standard errors from 0. However, from the symmetry of the configuration, the bias in $\hat{\beta}_1$ and $\hat{\beta}_2$ should be the same, and the combined estimate of bias would be well within sampling variation of 0. On average, it appears the inverse information slightly underestimates the true variance, although the standard errors of the estimated variances are large enough that the difference might be within sampling variation. Note that the average information is estimated much more precisely than the true $\mathrm{Var}(\hat{\beta}_j)$. This is often the case, so when evaluating the accuracy of the inverse information or other types of variance estimates, it is not sufficient to simply compute the standard error of the average of these estimates over the samples—the variation in the empirical estimate of the true variance also must be taken into account. The estimated sizes of the tests are within sampling variation of the nominal levels. The number of samples was chosen so that if the exact size of the tests was .05, then the true standard errors of the estimated sizes, $\{(.05)(.95)/\texttt{NT}\}^{1/2}$, would be .0075. Since the estimated sizes were slightly larger than the nominal level, the estimated standard errors were slightly larger than the target value.

When running simulations in Splus, it is often the case that memory usage will grow with the number of replicates of the simulation. When memory usage becomes too large, the job can become a serious inconvenience to other users, and ultimately can cause the job to terminate with an 'unable to obtain requested dynamic memory' error. In the example above, at the termination of the simulation loop, the process had grown to about 8 MB. This is not large enough to be much of a problem on most modern computers. If substantially more memory were required, it might be necessary to break the job down into several separate runs, say of 50 or 100 replicates each. The output of `memory.size()` can be checked within the simulation loop, with a command to terminate if it grows too large; for example, `if (memory.size()>20000000) break`. It is important that the termination be done in a way that is graceful enough to save the results to that point, so they may be combined with results from other runs.

Also note that only about 2 minutes of cpu time were required for the entire run (on a SPARC Ultra 1). Thus a simulation like this is now quite feasible even within an inefficient computing environment like Splus. Of course, it should be noted that most of the work in `coxph()` takes place inside C routines. A much faster program could be written by coding everything in C or FORTRAN. If efficient algorithms are used, the entire run above should take no more than a few seconds in such a program. Of course, coding and debugging the program would probably take at least several days of work.

Here is the output from another run with `beta <- c(1,1)`.

```
% coxsim1.s
S-PLUS : Copyright (c) 1988, 1996 MathSoft, Inc.
S : Copyright AT&T.
Version 3.4 Release 1 for Sun SPARC, SunOS 5.3 : 1996
Working data will be in /usr/stats/bobg/.Data
 [1]  5 35 63 26 46  0 61 23 21 57  7  0
[1] 7937232
[1] 0.7103905
true beta= 1 1
estimated bias
[1] 0.02192803 0.02329385
standard error of estimated bias
[1] 0.008778498 0.009213349
average information
[1] 0.06809622 0.06824705
standard error of ave inform
[1] 0.0002545156 0.0002668420
Estimated variance
[1] 0.06511741 0.07172851
approximate standard error of estimated variance
[1] 0.003234047 0.003701250
estimated rejection probabilities of tests
[1] 0.9869822 0.9846154
standard errors of est rejection probs of tests
[1] 0.003899369 0.004233975
111.72u 0.64s 2:08.78 87.2%
111.90u 0.89s 2:09.25 87.2%
```

In this case there is clear evidence of bias in the estimators, in that their means are slightly larger than the true value of 1. The average information appears closer to the true variance here than in the previous simulation, but the number of samples was again not large enough to give highly precise estimates of the true variance. The estimated powers of the tests are $> 98\%$ for this alternative. The estimated failure probability here is higher than in the previous run because here the failure hazard rates are higher for 75% of the cases (those that have either covariate equal to 1), while the censoring distribution has not changed.

In general, simulation experiments like those above should be approached using the same principles and techniques as any other type of experiment. Principles of sound experimental design should be used. The number of replicates in the simulation should be chosen to give reasonable precision for the quantities being estimated. Since variances are often unknown prior to collecting some data, it is often convenient to focus on binary endpoints. For example, in the simulation above, it is known that the variance of the rejection probabilities of the tests will be $p(1-p)/\texttt{NT}$, where p is the true rejection probability, and that p should be close to the nominal level .05. This makes it possible to determine a value of `NT` to give reasonable precision in the

estimates of the rejection probabilities, as was done above. More generally, when variances of quantities to be estimated from the simulations are unknown, it is usually feasible to make some preliminary simulation runs to estimate the variances. These estimated variances can then be used in planning the final simulation study. Conducting preliminary experiments to estimate unknown design parameters is generally much more feasible in computer experiments than in general scientific research.

Variance reduction techniques, such as blocking and regressing on covariates, should also be employed in simulation experiments, where feasible. Some particular techniques are discussed in the following section.

7.3 Improving Efficiency

7.3.1 Control Variates

Consider estimating $\mu_X = \mathrm{E}(X_s)$ based on an iid sample $X_1, \ldots, X_S$. Let $\sigma_X^2 = \mathrm{Var}(X_s)$. The standard estimator of μ_X,

$$\overline{X} = \sum_{s=1}^{S} X_s/S,$$

has variance σ_X^2/S. Suppose also that there is another variable Y_s measured on the same experimental units, and that $\mu_Y = \mathrm{E}(Y_s)$ is known. Let $\sigma_Y^2 = \mathrm{Var}(Y_s)$, and $\rho = \mathrm{Cov}(X_s, Y_s)/(\sigma_X \sigma_Y)$. Consider estimating μ_X with

$$\tilde{\mu}_X(c) = \overline{X} - c(\overline{Y} - \mu_Y).$$

Since μ_Y is known, this is a well-defined estimator, and clearly it is unbiased for μ_X. Now

$$\mathrm{Var}[\tilde{\mu}_X(c)] = S^{-1}[\sigma_X^2 + c^2\sigma_Y^2 - 2c\rho\sigma_X\sigma_Y]$$

is minimized by

$$c^* = \rho\sigma_X/\sigma_Y,$$

and

$$\mathrm{Var}[\tilde{\mu}_X(c^*)] = S^{-1}\sigma_X^2[1 - \rho^2],$$

which is less than σ_X^2/S if $\rho \neq 0$.

In practice, c^* will often need to be estimated. If the usual moment estimators are substituted for ρ, σ_X and σ_Y, the resulting estimator is

$$\hat{c}^* = \sum_{s=1}^{S}(X_s - \overline{X})(Y_s - \overline{Y}) / \sum_{s=1}^{S}(Y_s - \overline{Y})^2,$$

which is just the usual least squares estimator of the regression coefficient for the regression of X_s on Y_s. Thus $\tilde{\mu}_X(\hat{c}^*)$ is a regression adjustment for estimating μ_X, and more simply, is just the least squares estimate of μ_X in the model

$$X_s = \mu_X + \gamma(Y_s - \mu_Y) + \epsilon_s,$$

where the ϵ_s are iid with mean 0. Since this model is not necessarily true, $\tilde{\mu}_X(\hat{c}^*)$ is not necessarily unbiased for μ_X. However, since

$$S^{1/2}[\tilde{\mu}_X(\hat{c}^*) - \tilde{\mu}_X(c^*)] = -S^{1/2}(\hat{c}^* - c^*)(\overline{Y} - \mu_Y) \xrightarrow{P} 0,$$

$\tilde{\mu}_X(\hat{c}^*)$ and $\tilde{\mu}_X(c^*)$ are asymptotically equivalent, as the number of simulated samples $S \to \infty$.

While it is not often used in simulation studies appearing in the statistical literature, the technique described above can be a powerful tool for variance reduction in simulations. In this setting, Y_s is referred to as a *control variate*. To use this method there needs to be a second quantity that can be computed from the simulated samples, whose mean is known exactly, and that is correlated with the primary quantity of interest. Note that $1 - .5^2 = .75$, $1 - .7^2 = .51$ and $1 - .9^2 = .19$, so a high correlation is needed to get a large reduction in the variance.

In the simulation to study the proportional hazards model in the previous section, consider just the problem of estimating the bias in $\hat{\beta}_1$. In the notation of this section, s indexes the simulated samples, and $X_s = \hat{\beta}_1^{(s)} - \beta_1$. Consider defining Y_s by

$$Y_s = \sum_{i=1}^{n} \delta_i^{(s)} z_{i1} / \sum_{i=1}^{n} z_{i1} - \sum_{i=1}^{n} \delta_i^{(s)} (1 - z_{i1}) / \sum_{i=1}^{n} (1 - z_{i1}), \tag{7.6}$$

the difference in the proportion observed to fail in the groups with $z_{i1} = 1$ and $z_{i1} = 0$. For the exponential distributions used in the previous section,

$$\begin{aligned} \mathrm{E}(\delta_i | z_{i1}, z_{i2}) &= \int \exp(-t\lambda_c) \exp(\beta_1 z_{i1} + \beta_2 z_{i2}) \exp[-t \exp(\beta_1 z_{i1} + \beta_2 z_{i2})]\, dt \\ &= \frac{\exp(\beta_1 z_{i1} + \beta_2 z_{i2})}{\exp(\beta_1 z_{i1} + \beta_2 z_{i2}) + \lambda_c}. \end{aligned}$$

Also, z_{i1} and z_{i2} are binary, and each of the 4 combinations occurs in exactly 1/4 of the cases, so

$$\mathrm{E}(Y_s) = (1/2) \left(\frac{\exp(\beta_1 + \beta_2)}{\exp(\beta_1 + \beta_2) + \lambda_c} + \frac{\exp(\beta_1)}{\exp(\beta_1) + \lambda_c} - \frac{\exp(\beta_2)}{\exp(\beta_2) + \lambda_c} - \frac{1}{1 + \lambda_c} \right).$$

All these quantities are known in the simulation experiment, so Y_i can be used as a control variate for estimating the bias of $\hat{\beta}_1$.

Below is a modified section of the shell script for the proportional hazards simulation (file `coxsim1.s` above), which performs this control variate adjustment. The first part of the file (omited) is unchanged from before. The seed is set to the value from the earlier run to use exactly the same samples, so the results can be compared with those from before.

```
out <- matrix(-1,nrow=5,ncol=NT) # store the results

.Random.seed <- c(37,14,5,19,20,0,58,38,4,48,59,2)
for ( i in 1:NT) { # simulation loop
  u <- gendata(N,thr,lamc)
  fp <- fp+mean(u[,2])
  y <- 2*(mean(z1*u[,2])-mean((1-z1)*u[,2])) # the control variate
```

```
  u <- coxph(u~z1+z2)
  out[,i] <- c(u$coef,diag(u$var),y)
}
print(memory.size()) # can become excessive
print(fp/NT) # proportion of observed failures
cat('true beta=',format(beta),'\n')
u1 <- apply(out,1,mean)
u2 <- apply(out,1,var)
h1 <- exp(c(beta[1]+beta[2],beta,0))
h1 <- h1/(h1+lamc)
h1 <- (h1[1]+h1[2]-h1[3]-h1[4])/2 # true E(Y)
print(u1[5]-h1)
rho <- cor(out[1,],out[5,])
cat('rho=',format(rho),'\n')
cs <- rho*sqrt(u2[1]/u2[5])
cat('ordinary and adjusted estimate of bias (beta 1)\n')
print(c(u1[1]-beta[1],u1[1]-beta[1]-cs*(u1[5]-h1)))
cat('standard errors\n')
print(sqrt(u2[1]/NT*c(1,(1-rho^2))))
```

Below is the output from this portion of the modified program.

```
[1] 8325024
[1] 0.4967219
true beta= 0 0
[1] -7.100592e-05
rho= 0.6886141
ordinary and adjusted estimate of bias (beta 1)
[1] 0.006329357 0.006477450
standard errors
[1] 0.010668314 0.007735894
```

The proportion of observed failures, and the unadjusted estimate of bias and its standard error are identical to the earlier run, because the same seed was used. The correlation of .689 gives a reduction in the standard error of about 27.5%. This same reduction could have been achieved by increasing the number of replications in the simulations by a factor of about 1.9. The estimate of bias changed only slightly, because $\overline{Y}$ was nearly equal to its expectation in this sample.

For estimating the rejection probabilities of the test $H_0 : \beta_1 = 0$, the quantity of interest is $X_s = I(|T_1^{(s)}| \geq z_{1-\alpha/2}) = I(T_1^{(s)} \geq z_{1-\alpha/2}) + I(T_1^{(s)} \leq -z_{1-\alpha/2})$. Here it might be better to use separate control variates for estimating the upper and lower tail rejection probabilities; for example, Y_s in (7.6) for the upper tail and $-Y_s$ for the lower.

If the proportion of censored observations was smaller than in the example above, then (7.6) might be less useful as a control variate (in the extreme case of no censoring (7.6) would be identically 0). In this case there are a variety of similar quantities that could be used. For example δ_i in (7.6) could be replaced by $I(t_i \leq t_0, \delta_i = 1)$, for some t_0.

When working with nonlinear statistics, it will sometimes turn out that a linear approximation to the statistic, often the Taylor series about the true value of the parameter, will be a useful control variate. That is, in some settings it will be possible to compute the expectation of the linearized statistic exactly, and the linear approximation will have high enough correlation with the original statistic to lead to significant variance reduction. In Exercise 7.4, it will also be seen that the components of the score vector evaluated at the true value of the parameters can be a useful control variate.

A recent application of control variates to estimating p-values for exact tests in contingency tables was given by Senchaudhuri, Mehta and Patel (1995). Control variates have also shown promise in bootstrap sampling; see, for example, Chapter 23 of Efron and Tibshirani (1993).

7.3.2 Importance Sampling

Let X denote a sample from a distribution, let $f(\cdot)$ be the density of X, let $q(X)$ be a statistic of interest, and consider estimating

$$\mu_q = \mathrm{E}_f\{q(X)\} = \int q(u)f(u)\,du, \tag{7.7}$$

where E_f denotes expectation with respect to the distribution with density f (of course a similar formula holds for discrete distributions with the integral replaced by a sum). Examples of such statistics from the proportional hazards model simulation include the estimators $\hat{\beta}_j$, whose expectations are needed to determine the bias, and the statistics $I(|T_j| \geq z_{1-\alpha/2})$, whose expectations are the rejection probabilities of the tests.

To estimate (7.7), samples $Y^{(s)}$, $s = 1, \ldots, S$, can be generated from the density f, and

$$\hat{q} = \sum_{s=1}^{S} q(Y^{(s)})/S$$

computed, as before. Alternately, let $g(\cdot)$ be a different density defined on the same sample space, such that $f(X)/g(X) < \infty$ for all X in the sample space, and suppose now $X^{(s)}$, $s = 1, \ldots, S$, are independent samples from $g(\cdot)$. Then

$$\tilde{q}_g = \sum_{s=1}^{S} q(X^{(s)})w_s/S, \tag{7.8}$$

where

$$w_s = f(X^{(s)})/g(X^{(s)}),$$

is also an unbiased estimator for $\mathrm{E}_f\{q(X)\}$, since

$$\mathrm{E}_g(\tilde{q}_g) = \int q(u)\frac{f(u)}{g(u)}g(u)\,du = \int q(u)f(u)\,du = \mathrm{E}_f\{q(X)\}.$$

There are two ways this can be used. In one, a density g is selected to give an estimator $\tilde{q}_g$ that is more efficient than $\hat{\mu}$. In the other, having generated samples from g, (7.8) is used with different

densities f to estimate $\mathrm{E}_f\{q(X)\}$ for different f, without generating and analyzing other simulated data sets. In either case, the general technique of sampling from one distribution to estimate an expectation under a different distribution is referred to as *importance sampling*.

To see how importance sampling can improve efficiency, the variances of $\hat{q}$ and $\tilde{q}_g$ need to be considered. Now

$$\mathrm{Var}(\hat{q}) = \mathrm{Var}_f\{q(X)\}/S = S^{-1}\int\{q(u)-\mu_q\}^2 f(u)\,du,$$

and

$$\mathrm{Var}(\tilde{q}_g) = \mathrm{Var}_g\{q(X)f(X)/g(X)\}/S = S^{-1}\int\{q(u)f(u)/g(u)-\mu_q\}^2 g(u)\,du. \tag{7.9}$$

If $q(u)f(u)/g(u)$ is constant, then (7.9) will be 0. It would almost never be practical to sample from such a g, and this condition may not even define a density (if for example $q(u) < 0$ for some u). As a general rule, though, $g(u)$ should be chosen to be large where $q(u)f(u)$ is large, and small where $q(u)f(u)$ is small. The name 'importance sampling' comes form this observation, since the goal is to choose g to sample more heavily in regions where $q(u)f(u)$ is large (the 'more important' regions). However, if there are regions where $g(u)$ is much smaller than $f(u)$, then (7.9) will be very large, or possibly even infinite. This places limits on how large $g(u)$ can be in regions where $q(u)f(u)$ is large, since $g(u)$ cannot be too small anywhere where $q(u)f(u)$ is nonzero (and g must integrate to 1).

Since $\tilde{q}_g$ is the average of iid terms, $\mathrm{Var}(\tilde{q}_g)$ can be estimated by

$$\frac{1}{S(S-1)}\sum_{s=1}^{S}\{q(X^{(s)})w_s - \tilde{q}_g\}^2.$$

To illustrate, consider the problem of estimating the size of the Wald test for $H_0 : \beta_1 = 0$, in the proportional hazards model simulation. In this case $q(X) = I(|T_1| \geq z_{1-\alpha/2})$, where $T_1 = \hat{\beta}_1/v_{11}^{1/2}$. The density for the observed data in each simulated sample is

$$f(X;\beta_1,\beta_2,\lambda_c) = \prod_{i=1}^{n}\exp[\delta_i(\beta_1 z_{i1}+\beta_2 z_{i2})]\lambda_c^{1-\delta_i}\exp(-t_i[\exp(\beta_1 z_{i1}+\beta_2 z_{i2})+\lambda_c]),$$

where the true value of β_1 is 0 under the null hypothesis. The product $q(X)f(X)$ is 0 for any X where the null hypothesis is accepted. While it would be quite difficult to sample from a g defined to be 0 on this region, it does suggest that to estimate the size of the test, it might be more efficient to use importance sampling with samples generated from alternative distributions than to just generate the data under the null. However, considerable care is needed. If samples are just generated from a fixed alternative, such as $g(X) = f(X;1,\beta_2,\lambda_c)$, then the samples will concentrate in the upper tail of the critical region $T_1 \geq z_{1-\alpha/2}$, but there might be few if any samples in the lower tail $T_1 \leq -z_{1-\alpha/2}$, giving a poor overall estimate of the size. One way to address this would be to estimate the upper tail and lower tail rejection probabilities separately in different simulation runs. Alternately, a mixture distribution could be used for the importance sampling distribution; for example,

$$g(X) = [f(X;\gamma,\beta_2,\lambda_c) + f(X;-\gamma,\beta_2,\lambda_c)]/2.$$

For this choice of g, and f as above, the importance sampling weights can be written

$$\begin{aligned} w &= \frac{2\exp[-\sum_{i=1}^n t_i \exp(\beta_2 z_{i2})]}{\exp[\sum_{i=1}^n \delta_i \gamma z_{i1} - t_i \exp(\gamma z_{i1} + \beta_2 z_{i2})] + \exp[\sum_{i=1}^n (-\delta_i \gamma z_{i1} - t_i \exp(-\gamma z_{i1} + \beta_2 z_{i2}))]} \\ &= 2\left[\sum_{j=1}^2 \exp\left(\sum_{i=1}^n \{(-1)^j \delta_i \gamma z_{i1} - t_i[\exp((-1)^j \gamma z_{i1}) - 1]\exp(\beta_2 z_{i2})\}\right)\right]^{-1} \end{aligned}$$

(note that only terms with $z_{i1} = 1$ contribute to the sums).

A modified version of `coxsim1.s` implementing this importance sampling scheme is given below.

```
#
S << '%%'
# importance sampling for estimating size of tests
NT <- 845 # number simulated samples
N <- 100   # sample size--must be a multiple of 4
beta <- c(.4,0) # true coefficients
lamc <- 1  # hazard rate for censoring distribution
fp <- 0
out <- matrix(-1,nrow=4,ncol=NT) # store the results
z1 <- rep(c(0,1,0,1),N/4) #covariates fixed throughout the run
z2 <- rep(c(0,0,1,1),N/4)
thr <- exp(c(0,beta,beta[1]+beta[2])) # true exponential hazards
                        # for different covariate combinations
mthr <- exp(c(0,-beta[1],beta[2],beta[2]-beta[1]))
gendata <- function(n,thr,lamc) { #function to generate censored outcomes
  y <- rexp(n)/thr
  cen <- rexp(n,rate=lamc)
  Surv(pmin(y,cen),ifelse(y <= cen,1,0))
}

print(.Random.seed)
mm <- runif(NT)<.5 #which part of the mixture to use
w <- rep(0,NT)
sub <- z1 == 1
for ( i in 1:NT) { # simulation loop
  if (mm[i]) u <- gendata(N,thr,lamc)
  else u <- gendata(N,mthr,lamc)
  # importance sampling weights; assumes beta[2]=0
  w[i] <- 2/(exp(sum(u[sub,2]*beta[1]-u[sub,1]*(thr[2]-1))) +
     exp(sum(-u[sub,2]*beta[1]-u[sub,1]*(mthr[2]-1))))
  fp <- fp+mean(u[,2])
  u <- coxph(u~z1+z2)
  out[,i] <- c(u$coef,diag(u$var))
}
print(memory.size()) # can become excessive
```

```
print(fp/NT) # proportion of observed failures
cat('true beta=',format(beta),'\n')
t1 <- abs(out[1,]/sqrt(out[3,]))
zq <- qnorm(.975)
print(summary(w))
print(summary(w[t1>=zq]))
print(summary(w[t1<zq]))
w <- w*(t1>=zq)
rp <- c(sum(t1>=zq)/NT,sum(w)/NT)
cat('estimated rejection probabilities of tests\n')
print(rp)
cat('standard errors of est rejection probs of tests\n')
print(sqrt(c(rp[1]*(1-rp[1]),var(w))/NT))
q()
'%%'
```

Executing this gives the following.

```
[1] 37 12  0 46 25  1 20 41 23 25 14  2
[1] 4997752
[1] 0.5024142
true beta= 0.4 0.0
      Min. 1st Qu. Median  Mean 3rd Qu.  Max.
 0.0002781 0.07801 0.2808 1.042   1.055 9.133
      Min. 1st Qu.  Median   Mean 3rd Qu. Max.
 0.0002781 0.01941 0.05656 0.1436  0.1454 3.26
     Min. 1st Qu. Median  Mean 3rd Qu.  Max.
 0.005581  0.1754 0.4752 1.366   1.672 9.133
estimated rejection probabilities of tests
[1] 0.26508876 0.03807723
standard errors of est rejection probs of tests
[1] 0.015183949 0.005599944
108.95u 0.61s 2:01.99 89.8%
109.12u 0.85s 2:02.36 89.8%
```

In this run with $\gamma = .4$, the empirical rejection probability of .265 was adjusted by the importance weights to .038, which is 2 standard errors below the nominal .05 level, suggesting that the test may be conservative (this is not completely consistent with the earlier results). In essence, the rejection probability under both the null and an alternative has been estimated from a single run (this could be extended by calculating importance weights for other alternatives, to give estimated rejection probabilities under several distributions from the same run). The size of the importance weights varied from .00028 to 9.13, which is larger variation than would be ideal, although the range is a little smaller for samples falling within the critical region, which was part of the objective. In ordinary sampling under the null hypothesis, the binomial standard error with 845 replications would have been .0066, compared with .0056 above. Thus this importance sampling algorithm did result in some improvement in efficiency in this example, but not a large

amount. The difficulty here is that the importance sampling density g did not match qf closely enough to give much improvement. The value of γ used did not move very much mass of the sampling density into the critical region of the test, but if a substantially larger value of γ is used, the probability given to regions where $g << qf$ will be much larger, resulting in a much larger variance. (There are regions where $g << qf$ in the example above, but they occur with relatively low probability.) There are examples where this technique works better, and there may be better sampling densities even in this problem.

Importance sampling can often be used in conjunction with control variates. In many situations the importance sampling density will be chosen to be of a simple form, such as a multivariate normal or t distribution. It will then often be the case that either $\mathrm{E}_g[q(X)]$ or the expectation of a linear approximation to $q(X)$ can be computed exactly, so $q(X) - \mathrm{E}_g[q(X)]$ (or the corresponding quantity from a linear approximation) can be used as a control variate.

Importance sampling is widely used in Monte Carlo integration, and later will be discussed from that perspective. In addition to its efficiency enhancing properties, it can be used to get an answer when the true distribution is difficult to sample from. Importance sampling has been used to enhance efficiency of bootstrap sampling; see Efron and Tibshirani (1993) and Section 10.3.1, below. Mehta, Patel and Senchaudhuri (1988) gave an application of importance sampling to permutational inference.

7.3.3 Antithetic Sampling

The general principle of antithetic sampling is to use the same sequence of underlying random variates to generate a second sample in such a way that the estimate of the quantity of interest from the second sample will be negatively correlated with the estimate from the original sample.

Consider again the problem of estimating the bias in the regression coefficient estimates in the proportional hazards model. The exponential deviates for the failure times are

$$-\log(u_i)/\exp(z_{i1}\beta_1 + z_{i2}\beta_2),$$

where $u_i \sim U(0,1)$. A second sample from the same distribution can be obtained by calculating

$$-\log(1-u_i)/\exp(z_{i1}\beta_1 + z_{i2}\beta_2), \tag{7.10}$$

using the same u_i. This sample is not independent of the first, but marginally it has the same distribution.

Suppose the first sample in this example is such that $\hat{\beta}_1$ is large. That means that on average the failure times for the cases with $z_{i1} = 1$ are smaller than those for cases with $z_{i1} = 0$. In the second sample obtained from (7.10) using the same underlying uniform deviates, the failure times that were large in the first sample will tend to be small, so the second sample will tend to have longer failure times in the group with $z_{i1} = 1$, leading to a small value for $\hat{\beta}_1$ in the second sample. Thus the estimates of bias from the 2 samples should tend to be negatively correlated. Write $\hat{\beta}_1^{(s1)}$ for the estimate of β_1 from the first sample in the sth replication of the simulation, and $\hat{\beta}_1^{(s2)}$ for the corresponding estimate from the second sample. The combined estimate of bias is

$$\sum_{s=1}^{S}\{(\hat{\beta}_1^{(s1)} + \hat{\beta}_1^{(s2)})/2 - \beta_1\}/S.$$

The variance of this estimate is

$$\mathrm{Var}(\hat{\beta}_1)(1+\rho)/(2S), \tag{7.11}$$

where ρ is the correlation between $\hat{\beta}_1^{(s1)}$ and $\hat{\beta}_1^{(s2)}$. If there is substantial negative correlation in the estimates from the two samples, then (7.11) can be much smaller than $\mathrm{Var}(\hat{\beta}_1)/S$, the variance of the ordinary unadjusted bias estimate.

Below is a modification of the `coxsim1.s` program to implement this antithetic sampling algorithm for bias estimation. Note that since two samples are analyzed within each replicate of the simulation, the computational burden per replicate is roughly twice that of the original simulation.

```
#
S << '%%'
# antithetic sampling for estimating bias of \hat{beta}
NT <- 845 # number simulated samples
N <- 100   # sample size--must be a multiple of 4
beta <- c(0,0) # true coefficients
lamc <- 1  # hazard rate for censoring distribution
fp <- 0
z1 <- rep(c(0,1,0,1),N/4) #covariates stay fixed throughout the run
z2 <- rep(c(0,0,1,1),N/4)
thr <- exp(c(0,beta,beta[1]+beta[2])) # true exponential hazards
                        # for different covariate combinations
gendata <- function(n,thr,lamc) { #function to generate censored outcomes
  y <- runif(n)
  y3 <- -log(1-y)/thr # the antithetic variates
  y <- -log(y)/thr  # the original variates
  cen <- rexp(n,rate=lamc)
  list(Surv(pmin(y,cen),ifelse(y <= cen,1,0)),
       Surv(pmin(y3,cen),ifelse(y3 <= cen,1,0)))
}

out <- matrix(-1,nrow=6,ncol=NT) # store the results
.Random.seed <- c(37,14,5,19,20,0,58,38,4,48,59,2) #repeat 1st simulation
for ( i in 1:NT) { # simulation loop
  u <- gendata(N,thr,lamc)
  fp <- fp+mean(u[[1]][,2])
  w <- coxph(u[[2]]~z1+z2)
  u <- coxph(u[[1]]~z1+z2)
  out[,i] <- c(u$coef,diag(u$var),w$coef)
}
print(memory.size()) # can become excessive
print(fp/NT) # proportion of observed failures
cat('true beta=',format(beta),'\n')
u1 <- apply(out,1,mean)
u2 <- apply(out,1,var)
```

```
u <- (out[1,]+out[5,])/2
cat('ordinary and adjusted estimates of bias (beta 1)\n')
print(c(u1[1],u1[5],mean(u))-beta[1])
cat('standard errors\n')
print(sqrt(c(u2[1],u2[5],var(u))/NT))
u <- (out[2,]+out[6,])/2
cat('ordinary and adjusted estimate of bias (beta 2)\n')
print(c(u1[2],u1[6],mean(u))-beta[2])
cat('standard errors\n')
print(sqrt(c(u2[2],u2[6],var(u))/NT))
cat('correlations:\n')
print(c(cor(out[1,],out[5,]),cor(out[2,],out[6,])))
q()
'%%'
```

Here is the output from the repeats of the two simulation runs given earlier.

```
[1] 8139472
[1] 0.4967219
true beta= 0 0
ordinary and adjusted estimates of bias (beta 1)
[1]  0.006329357 -0.001471003  0.002429177
standard errors
[1] 0.010668314 0.010599054 0.003764409
ordinary and adjusted estimate of bias (beta 2)
[1] -0.020323796  0.016505287 -0.001909255
standard errors
[1] 0.010522634 0.010311908 0.003746515
correlations:
[1] -0.7493752 -0.7414896

[1] 8077480
[1] 0.7103905
true beta= 1 1
ordinary and adjusted estimates of bias (beta 1)
[1] 0.02192803 0.01807757 0.02000280
standard errors
[1] 0.008778498 0.008793153 0.003817970
ordinary and adjusted estimate of bias (beta 2)
[1] 0.02329385 0.02377239 0.02353312
standard errors
[1] 0.009213349 0.009183816 0.003959552
correlations:
[1] -0.6223160 -0.6294256
```

In the first case the reduction in the standard error is about 65%, and in the second case about

57%, so in both cases the reduction in variance has more than compensated for the increase in computation time. That is, antithetic sampling has approximately doubled the computation time. If instead the computation time was doubled by doubling the number of independent replications in the simulation, then the standard error would have been reduced by a factor of $1/2^{1/2} \doteq .707$, or about 29%.

An application of a type of antithetic sampling to bootstrap methods, based on defining an antithetic permutation of a sample, is given in Section 10.3.1, below.

7.4 Exercises

Exercise 7.1 Consider the distribution defined by the density

$$f(x) = \frac{x^4 + \exp(-x^2)}{6.6 + \pi^{1/2}[\Phi(2^{3/2}) - \Phi(-2^{1/2})]}, \qquad -1 < x < 2,$$

where $\Phi(\cdot)$ is the standard normal CDF.

1. Give a detailed algorithm for generating random numbers from this distribution.
2. Generate 10,000 independent pseudo-random numbers from this distribution.
3. Using the 10,000 observations from part (b), perform a chi-square goodness-of-fit test, using 50 equal-probability categories, to check whether the sample really appears to be from the distribution with density $f(x)$.

Exercise 7.2 Develop a rejection sampling algorithm to sample from the beta distribution with density $\propto x^{\alpha-1}(1-x)^{\beta-1}I(0 < x < 1)$. The overall algorithm should work for any $\alpha > 0$ and $\beta > 0$, but it may be appropriate to consider separate cases based on the values of α and β.

Exercise 7.3 Consider the binary logistic regression model with

$$P(Y_i = 1|z_{i1}, z_{i2}) = \frac{\exp(\beta_0 + \beta_1 z_{i1} + \beta_2 z_{i2})}{1 + \exp(\beta_0 + \beta_1 z_{i1} + \beta_2 z_{i2})}, \quad i = 1, \ldots, n.$$

Let $\hat{\beta}_j$ be the MLE of β_j, $j = 1, 2, 3$, and v_{jj} be the corresponding diagonal element of the inverse information, which is the usual estimate of the variance of $\hat{\beta}_j$. An approximate 90% confidence interval on β_j is given by

$$\hat{\beta}_j \pm 1.6449 v_{jj}^{1/2}, \qquad j = 1, 2, 3.$$

Conduct a simulation to estimate the true coverage of these approximate confidence intervals with $n = 100$ and with the true values of the parameters given by $\beta_0 = -.5$, $\beta_1 = 1$ and $\beta_2 = 0$. In the simulations, sample (z_{i1}, z_{i2}) from a bivariate normal distribution with means=0, variances=1, and correlation= ρ, generating new covariate values for each simulated sample. Conduct two separate simulation runs, one with $\rho = 0$ and the other with $\rho = .7$. Choose the sample size to give reasonable precision for the estimated coverage probabilities (give a justification). For both simulation runs, give the estimated coverage probabilities and their standard errors. (The `glm( ,family=binomial)` function in Splus may be used to fit the models. If you use this, the inverse information matrix can be obtained from output `glm.out` with the command `summary(glm.out)$cov.unscaled`.)

Exercise 7.4 Consider the Cox model simulation in Sections 7.2 and 7.3.1. Let the true value of β here be $(0, .6)$, but keep other aspects of the true distribution (censoring and failure distributions, covariates, sample size) the same as before.

The partial likelihood score vector evaluated at the true β has exact mean 0, and thus its components are candidates for control variates. Using the variable definitions from the code in the file `coxsim1.s` in Section 7.2, this score vector can be computed using the Splus commands

```
u <- gendata(N,thr,lamc)
uu <- coxph(u~z1+z2,init=beta,iter.max=0)
y <- apply(residuals(uu,'score'),2,sum)
u <- coxph(u~z1+z2)
```

(`y` is then the score vector evaluated at `beta`). The first and last lines above, identical to lines in `coxsim1.s`, are included to show the appropriate location.

Perform a simulation to estimate the bias in $\hat{\beta}_j$, $j = 1, 2$, and the coverage probabilities of upper 95% confidence limits on β_j, $j = 1, 2$. In each case use the corresponding component of the score vector as a control variate. For each of these 4 quantities, give the simulation estimate with and without the control variate correction, standard errors for each, and the correlation between the uncorrected values and the corresponding control variate.

7.5 References

Deng L-Y and Lin DKJ (2000). Random number generation for the new century. *The American Statistician*, 54:145–150.

Efron B and Tibshirani RJ (1993). *An Introduction to the Bootstrap*. Chapman and Hall.

Fishman GS and Moore LR (1982). A statistical evaluation of multiplicative congruential random number generators with modulus $2^{31} - 1$. *Journal of the American Statistical Association*, 77:129–136.

Fishman GS and Moore LR (1986). An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$. *SIAM Journal of Scientific and Statistical Computing*, 7:24–45.

Hall P (1989). Antithetic resampling for the bootstrap. *Biometrika* 76:713–724.

Kennedy WJ and Gentle JE (1980). Statistical Computing. Marcel Dekker.

L'Ecuyer P (1988). *Communications of the ACM*, 31:742–774.

Marsaglia G and Zaman A (1993). The KISS generator. Technical Report, Department of Statistics, Florida State University (?).

Mehta CR, Patel NR and Senchaudhuri P, (1988). Importance sampling for estimating exact probabilities in permutational inference. *Journal of the American Statistical Association*, 83:999–1005.

Press WH, Teukolsky SA, Vetterling WT, and Flannery BP (1992). *Numerical Recipes in C: The Art of Scientific Computing. Second Edition*. Cambridge University Press.

Ripley BD (1987). *Stochastic Simulation*. Wiley.

Robert CP and Casella G (1999). *Monte Carlo Statistical Methods*. Springer.

Schrage L (1979). *ACM Transactions on Mathematical Software*, 5:132–138.

Senchaudhuri P, Mehta CR and Patel NR (1995). Estimating exact p values by the method of control variates or Monte Carlo rescue. *Journal of the American Statistical Association*, 90:640–648.

Venables WN and Ripley BD (1997). *Modern Applied Statistic with Splus, 2nd Ed*. Springer.

Wichmann BA and Hill ID (1982). [Algorithm AS 183] An efficient and portable pseudo-random number generator. *Applied Statistics*, 31:188–190. (Correction, 1984, 33:123.)

Zeisel H (1986). A remark on AS 183. *Applied Statistics*, 35:89.

Chapter 8

More on Importance Sampling

8.1 Introduction

The discussion of importance sampling In Section 7.3.2 was formulated in terms of computing an expectation, where the integrand included a density function. That formulation is completely general, though, since any integral $\int h(x)\,dx$ can be written

$$\int [h(x)/f(x)]f(x)\,dx = \mathrm{E}_f[h(X)/f(X)],$$

where $f(x)$ is a density with support on the region of integration. Then if $X^{(1)}, \ldots, X^{(S)}$ are a sample from f, $\int h(x)\,dx$ can be estimated by

$$S^{-1}\sum_{s=1}^{S} h(X^{(s)})/f(X^{(s)}). \tag{8.1}$$

Thus the method of Section 7.3.2 can be applied to general integration problems. As discussed there, the choice of f is very important, since a poor choice will result in (8.1) having a large variance. In particular, f cannot be too close to 0 except at points where h is also. In general, locating local modes of h and examining the amount of dispersion about the modes can be very useful to help in constructing an appropriate sampling distribution. The following sections illustrate this idea for integrals arising in the context of Bayesian inference. Some extensions of the basic importance sampling methods are also presented.

8.2 Importance Sampling in Bayesian Inference

Given a likelihood $L(\beta)$ for a parameter vector β, based on data y, and a prior $\pi(\beta)$, the posterior is given by

$$\pi(\beta|y) = c^{-1}L(\beta)\pi(\beta),$$

where the normalizing constant $c = \int L(\beta)\pi(\beta)\,d\beta$ is determined by the constraint that the density integrate to 1. This normalizing constant often does not have an analytic expression. General problems of interest in Bayesian analyses are computing means and variances of the

posterior distribution, and also finding quantiles of marginal posterior distributions. In general let $q(\beta)$ be a parametric function for which

$$q^* = \int q(\beta)\pi(\beta|y)\,d\beta \tag{8.2}$$

needs to be evaluated. Typical examples of such functions include β_j, β_j^2 (needed for determining the mean and variance of β_j), $\exp(\beta_j)$, and $I(\beta_j \le x)$ (the posterior expectation of the last is the probability that $\beta_j \le x$, and solving for a quantile of the posterior might require evaluating such integrals for many different values of x).

In many applications, (8.2) cannot be evaluated explicitly, and it is difficult to sample directly from the posterior distribution. (8.2) is of the same form as (7.7), so importance sampling can be applied. As discussed in Section 7.3.2, samples can be drawn from a distribution with density g, and the value of (8.2) estimated using (7.8). However, this estimate still depends on the normalizing constant c, which is often unknown. In this case, if $\beta^{(1)}, \ldots, \beta^{(S)}$ is a sample from g, then (8.2) can be estimated with

$$\tilde{q} = \sum_s w_s q(\beta^{(s)}) / \sum_s w_s, \tag{8.3}$$

where $w_s = L(\beta^{(s)})\pi(\beta^{(s)})/g(\beta^{(s)})$. To use (8.3), the sampling density g need not be normalized, either. In general, let

$$c = \mathrm{E}_g(w_s) = \int \frac{L(\beta)\pi(\beta)}{g(\beta)} \frac{g(\beta)}{\int g(x)\,dx}\,d\beta = \frac{\int L(\beta)\pi(\beta)\,d\beta}{\int g(x)\,dx},$$

which is just the normalizing constant of the posterior, as before, in the case where g is normalized.

To investigate the properties of $\tilde{q}$, note that

$$\tilde{q} - q^* = \frac{\sum_{s=1}^S w_s[q(\beta^{(s)}) - q^*]/S}{\sum_{s=1}^S w_s/S}.$$

By the weak law of large numbers, $\sum_{s=1}^S w_s/S \xrightarrow{P} \mathrm{E}_g(w_s) = c$. Since

$$\begin{aligned}
\mathrm{E}[w_s q(\beta^{(s)})] &= \int q(\beta)\frac{L(\beta)\pi(\beta)}{g(\beta)}\frac{g(\beta)}{\int g(x)\,dx}\,d\beta \\
&= \int q(\beta) c \frac{\pi(\beta|y)}{g(\beta)} g(\beta)\,d\beta \\
&= c\int q(\beta)\pi(\beta|y)\,d\beta \\
&= \mathrm{E}_g(w_s) q^*,
\end{aligned}$$

$\mathrm{E}\{w_s[q(\beta^{(s)}) - q^*]\} = 0$. Thus for large S, $\tilde{q}$ is approximately normal with mean q^* and variance

$$\mathrm{Var}\{w_s[q(\beta^{(s)}) - q^*]\}/(Sc^2). \tag{8.4}$$

Since $\mathrm{E}\{w_s[q(\beta^{(s)}) - q^*]\} = 0$, (8.4) can be estimated by

$$\sum_s \left(\frac{w_s[q(\beta^{(s)}) - \tilde{q}]}{\sum_k w_k}\right)^2, \tag{8.5}$$

which is $S-1$ times the sample variance of the quantities $w_s[q(\beta^{(s)})-\tilde{q}]/(\sum_k w_k)$.

In some Bayesian applications, the sampling density g can be based on the normal approximation to the posterior, as given in (5.22). The normal approximation itself will not always have tails as heavy as the true posterior, so it can be necessary to use some multiple (>1) of the approximate posterior variance, or to use a t distribution instead of the normal, to get a sampling distribution with sufficiently heavy tails. Also, if the posterior has multiple local modes, a mixture distribution with components centered on each mode may improve performance. Thus in any case a preliminary search for modes of the posterior is generally prudent.

In these applications, importance sampling can often be combined with a control variate correction to improve efficiency. Consider the case where the sampling distribution has a mean μ that can be evaluated explicitly (for example, in the normal approximation to the posterior, the mean is the posterior mode, $\hat{\beta}$). Then the components of $\beta^{(s)}-\mu$ can be used as control variates. For example, $\mathrm{E}(\beta_k|y)$ can be estimated with

$$\tilde{\beta}_k = \sum_{s=1}^{S} \frac{\beta_k^{(s)} w_s}{\sum_l w_l} - \frac{r}{S}\sum_{s=1}^{S}(\beta_k^{(s)}-\mu_k), \tag{8.6}$$

where $\beta_k^{(s)}$ is the kth component of $\beta^{(s)}$ and S is the total number of $\beta^{(s)}$ sampled. Applying the argument from Section 7.3.1, the optimal value of r is

$$r = S\frac{\mathrm{Cov}(w_s[\beta_k^{(s)}-\beta_k^*]/\sum_l w_l, \beta_k^{(s)}-\mu_k)}{\mathrm{Var}(\beta_k^{(s)}-\mu_k)},$$

where β_k^* is the true posterior mean, and the variances and covariances are with respect to the sampling distribution g. At this optimal value of r, $\mathrm{Var}(\tilde{\beta}_k)$ is

$$\begin{aligned}\mathrm{Var}(\tilde{\beta}_k) &= S\left(\mathrm{Var}\left[w_s(\beta_k^{(s)}-\beta_k^*)/\sum_l w_l\right] - \frac{\mathrm{Cov}(w_s[\beta_k^{(s)}-\beta_k^*]/\sum_l w_l, \beta_k^{(s)}-\mu_k)^2}{\mathrm{Var}(\beta_k^{(s)}-\mu_k)}\right)\\ &= S\,\mathrm{Var}\left[w_s(\beta_k^{(s)}-\beta_k^*)/\sum_l w_l\right](1-\rho^2),\end{aligned} \tag{8.7}$$

where ρ is the correlation between $w_s[\beta_k^{(s)}-\beta_k^*]/\sum_l w_l$ and $\beta_k^{(s)}-\mu_k$.

In the next section, these methods are applied to Bayesian analysis of a logistic regression model.

8.3 Bayesian Analysis For Logistic Regression

Consider the problem of Bayesian inference in the standard binary logistic regression model with likelihood

$$L(\beta) = \prod_{i=1}^{n}\frac{\exp(y_i x_i'\beta)}{1+\exp(x_i'\beta)},$$

where the y_i are binary responses and the x_i are p-dimensional covariate vectors. Suppose that the prior distribution on the unknown parameters β specifies that the individual components are iid $\mathrm{N}(0,\sigma^2)$, so the joint prior density is

$$\pi(\beta) \propto \exp[-\beta'\beta/(2\sigma^2)].$$

The posterior is then

$$\pi(\beta|y) \propto L(\beta)\pi(\beta).$$

Some quantities that might be of interest in a Bayesian analysis using this model are the posterior mean of β, quantiles of the posterior distribution of individual components of β, which can be thought of as interval estimates of the parameters, means and quantiles of the posterior distributions of the $\exp(\beta_j)$ (the odds ratio parameters), and the means and quantiles of the distributions of

$$P(y = 1|x) = p(x'\beta) = \frac{\exp(x'\beta)}{1 + \exp(x'\beta)}$$

(the response probabilities) for various values of x.

In the following example $n = 200$ with $y_i = 1$ for 132 cases and $y_i = 0$ for 68 cases. There are 5 covariates plus a constant term for $p = 6$ total parameters, and the prior variance $\sigma^2 = 1000$ (an almost noninformative prior). With 6 parameters it might be feasible to get accurate results using Gauss-Hermite quadrature methods, but to calculate all of the quantities of interest would require many different multidimensional numerical integrals, so direct calculation is not an attractive prospect. The posterior is also not easy to sample from directly. In the following, importance sampling is to perform the calculations.

The importance sampling density considered is the normal approximation to the posterior distribution, which is a multivariate normal distribution with mean equal to the mode $\hat{\beta}$ of the posterior distribution and covariance matrix equal to $I^{-1}(\hat{\beta})$, where

$$I(\beta) = -\frac{\partial^2}{\partial\beta\partial\beta'} \log[L(\beta)\pi(\beta)].$$

To sample from this distribution, it is first necessary to find $\hat{\beta}$ and $I(\hat{\beta})$. In the following this is done by specifying a function `flog()` which evaluates minus the log posterior and calling `nlminb()` to minimize this function.

```
> n <- 200
> ncov <- 5
> np <- ncov+1
> y <- scan("data.y")
> X <- matrix(scan("data.X"),ncol=np)
> pen <- .001      # 1/(prior variance)
> flog <- function(b) {
+   lp <- X %*% b
+   - sum(y * lp - log(1 + exp(lp))) + (pen * sum(b^2))/2
+ }
> z <- nlminb(rep(0,np),flog)
> bhat <- z$param
> # bhat is the mode of the posterior
> bhat
[1]  0.8282945 -0.7588482  1.1381240 -0.3916353  0.5320517 -0.0423460
> # I(bhat)
```

```
> d <- exp(X %*% bhat)
> d <- d/(1+d)^2
> # at this point the elements of d are p(xi'bhat)(1-p(xi'bhat)), where
> # the xi are the rows of X.  The following line then gives the usual
> # logistic regression information
> inf <- t(X) %*% (c(d)*X)
> # has to be exactly symmetric to use chol()
> inf <- (inf+t(inf))/2
> # and include the contribution from the prior
> diag(inf) <- diag(inf)+pen
> B <- solve(chol(inf))
```

At this point B is a matrix such that $\mathtt{B}\mathtt{B}' = I^{-1}(\hat{\beta})$. To sample from the normal approximation to the posterior, 6-dimensional vectors $Z^{(s)}$ with iid $\mathrm{N}(0,1)$ components can be generated, and $\beta^{(s)}$ calculated from $\beta^{(s)} = \mathtt{B}Z^{(s)} + \hat{\beta}$. The $\beta^{(s)}$ then have the required multivariate normal distribution. The following 3 lines produce a sample of nt=10,000 such vectors, as columns of the matrix H.

```
> nt <- 10000
> Z <- matrix(rnorm(nt*np),nrow=np)
> H <- B %*% Z + bhat
```

In Splus, generating a large number of samples all at once is much more efficient than generating them one at a time in a loop.

The importance sampling weights are proportional to

$$w_s = \frac{L(\beta^{(s)})\pi(\beta^{(s)})/[L(\hat{\beta})\pi(\hat{\beta})]}{\exp[-(\beta^{(s)} - \hat{\beta})'I(\hat{\beta})(\beta^{(s)} - \hat{\beta})/2]}. \tag{8.8}$$

Since the normalization constant of the posterior is unknown, integrals will be estimated using (8.3). As discussed there, in this form it is also only necessary to know the sampling density g to within a normalizing constant, so for simplicity the weights are left in the above form. The factor $L(\hat{\beta})\pi(\hat{\beta})$ is included to standardize both the numerator and denominator to be 1 at $\beta^{(s)} = \hat{\beta}$, which helps to avoid underflow problems. Substituting $\beta^{(s)} = \mathtt{B}Z^{(s)} + \hat{\beta}$, the denominator simplifies to $\exp(-{Z^{(s)}}'Z^{(s)}/2)$. The following commands calculate the w_s as components of the vector wt. The apply() command is the slowest part of the calculations. It is performing 10,000 evaluations of the log posterior, so it takes several minutes.

```
> w <- rep(1,np) %*% (Z*Z)
> w2 <- apply(H,2,flog)
> wt <- exp(flog(bhat)-w2+w/2)
> wtsum <- sum(wt)
```

Note that Z*Z is an element by element product, which gives squares of the elements of Z, and that flog() gives the negative of the log posterior.

Now that the samples and the weights have been computed, the other quantities of interest can be calculated. The mean of the posterior is estimated by $\sum_s w_s \beta^{(s)} / \sum_s w_s$.

```
> bb  <- t(H) * c(wt)
> bpm <- apply(bb,2,sum)/wtsum
> bpm
[1]  0.86455126 -0.79344280  1.19705198 -0.40929410  0.56454239 -0.04644663
```

It must be kept in mind that these (and other quantities below) are just approximations based on simulations, not exact calculations. They become exact though in the limit as the number of samples in the simulation becomes infinite. Applying (8.5) to estimate the precision of the estimated posterior mean of β, gives the following.

```
> bb2 <- (bb-outer(c(wt),bpm))/wtsum
> sqrt(apply(bb2^2,2,sum))
[1] 0.002805072 0.003156647 0.003536797 0.002749287 0.004001108 0.002282230
```

Thus the standard errors vary from .002 to .004, and so there should be at least 2 decimal places of accuracy for the posterior means of all the parameters. This is quite small relative to the spread of the posterior distribution (see below), and so should be sufficiently accurate for statistical inferences. If the true posterior distribution really was equal to the normal approximation, and if estimates were calculated from 10,000 samples from that posterior, then the variance of the estimates would be as follows.

```
> sqrt(diag(solve(inf))/nt)
[1] 0.001865649 0.001976781 0.002226847 0.001822120 0.002043587 0.001902633
```

That the actual standard errors are all less than twice as large as these indicates that the normal approximation was a reasonably good choice for the sampling distribution.

Information about the range of likely parameter values can be obtained from quantiles of the posterior distribution. To estimate these quantiles, think of the posterior as a discrete distribution with mass $w_s / \sum_k w_k$ on each of the sampled points $\beta^{(s)}$. The marginal distribution of any component of β puts the same mass on the corresponding components of the $\beta^{(s)}$ (the method of generation should give all distinct values). Estimating a quantile of the marginal posterior of a particular component of β is then just a matter of finding the quantile of this discrete approximation. The following function does this.

```
> qntl <- function(p,wt,pct=c(.025,.5,.975)) {
+   o <- order(p)
+   wtp <- cumsum(wt[o]/sum(wt))
+   out <- NULL
+   for (i in pct) {
+     ind <- max((1:length(p))[wtp<i])+0:1
+     out <- rbind(out,c(p[o][ind],wtp[ind]))
```

```
+   }
+   out
+ }
```

To find the quantiles of a scalar quantity $q(\beta)$, the input parameters are `p` = the vector of $q(\beta^{(s)})$, `wt` = the vector of $w^{(s)}$, and `pct` the quantiles (percentiles) to be calculated. In the function, the vector `wtp` is the CDF of the marginal distribution, at the points `p[o]`. For the functions $q(\beta) = \beta_k$, this function gives the following. In the output the first 2 values bracket the estimated quantile, and the next 2 give the estimated CDF of the marginal posterior at those values.

```
> qntl(H[1,],wt)
          [,1]      [,2]       [,3]       [,4]
[1,] 0.4961963 0.4964762 0.02490576 0.02500103
[2,] 0.8590689 0.8590836 0.49999317 0.50009727
[3,] 1.2611304 1.2612538 0.97479210 0.97502347
> qntl(H[2,],wt)
           [,1]       [,2]       [,3]       [,4]
[1,] -1.2050674 -1.2046193 0.02452151 0.02515693
[2,] -0.7930513 -0.7930312 0.49997168 0.50006287
[3,] -0.4048549 -0.4048147 0.97498370 0.97505701
> qntl(H[3,],wt)
          [,1]      [,2]       [,3]      [,4]
[1,] 0.7700427 0.7703188 0.02490795 0.0250034
[2,] 1.1873297 1.1874560 0.49993259 0.5000374
[3,] 1.6874651 1.6875951 0.97461611 0.9750564
> qntl(H[4,],wt)
            [,1]        [,2]       [,3]       [,4]
[1,] -0.78548781 -0.78463854 0.02489136 0.02519407
[2,] -0.40580561 -0.40580381 0.49992123 0.50001046
[3,] -0.05365666 -0.05362017 0.97491367 0.97500327
> qntl(H[5,],wt)
          [,1]      [,2]       [,3]       [,4]
[1,] 0.1597441 0.1598140 0.02495999 0.02506874
[2,] 0.5578549 0.5578882 0.49991835 0.50001017
[3,] 1.0070437 1.0076522 0.97488420 0.97507542
> qntl(H[6,],wt)
            [,1]        [,2]       [,3]       [,4]
[1,] -0.43373646 -0.43321462 0.02499914 0.02511654
[2,] -0.04473495 -0.04467982 0.49988329 0.50013270
[3,]  0.32910177  0.32956713 0.97489659 0.97500700
```

For most of the parameters, the difference between the .975 and .025 quantiles is about .8, so the spread in the posterior is substantially larger than the sampling error in the simulation, as claimed above. The fact that the marginal posteriors concentrate in regions which do not include 0 for all parameters but the last could be interpreted as evidence for nonnegligible effects of these

covariates. For the last covariate 0 is near the center of the posterior distribution, and it is less clear whether it has an important effect.

The real power of sampling based methods for Bayesian inference becomes apparent for analyzing more complicated functions of the parameters. For the odds ratio $q(\beta) = \exp(\beta_2)$ (for example), all that is needed is to calculate the $q(\beta^{(s)})$ and proceed as before.

```
> qq <- exp(H[2,])
> sum(qq*wt)/wtsum
[1] 0.4616637
> qntl(qq,wt)
          [,1]      [,2]       [,3]       [,4]
[1,] 0.2996718 0.2998061 0.02452151 0.02515693
[2,] 0.4524621 0.4524712 0.49997168 0.50006287
[3,] 0.6670736 0.6671004 0.97498370 0.97505701
```

Since $q(\beta)$ is a monotone function of β_2, the quantiles are just the exponential of those given earlier for β_2. Also, since quantiles are being computed from a (weighted) sample from the actual posterior, it is not necessary to transform to approximate normality to calculate the quantiles (as would be advisable if large sample approximations were used).

The success probability $p(x_0)$, with $x_0' = (1, .5, 1, 0, -.5, -1)$ (for example), can be calculated in a similar manner.

```
> x0 <- c(1,.5,1,0,-.5,-1)
> p <- exp(x0 %*% H)
> p <- p/(1+p)
> pm <- sum(wt*p)/wtsum
> pm
[1] 0.799744
> sqrt(sum((wt*(p-pm)/wtsum)^2))
[1] 0.0007656173
> qntl(p,wt)
          [,1]      [,2]       [,3]      [,4]
[1,] 0.6675866 0.6677404 0.02494496 0.0250452
[2,] 0.8042370 0.8042542 0.49991240 0.5000179
[3,] 0.9030322 0.9031865 0.97492025 0.9750304
```

Again note the sampling error in the estimates is much smaller than the spread of the posterior.

Since the sampling density is a normal distribution with mean $\hat{\beta}$, the control variate corrected version of the importance sampling estimator (8.6) can also be used, with $\mu_k = \hat{\beta}_k$. Applying this control variate correction with `bb` an ¡- d `bb2` as before gives the following.

```
> bb  <- t(H) * c(wt)
> bb2 <- (bb-outer(c(wt),bpm))/wtsum
```

```
> se.mean <- sqrt(apply(bb2^2,2,sum))
> H2 <- H-bhat
> for (i in 1:np) {
+   rho <- cor(H2[i,],bb2[,i])
+   cc <- nt*var(H2[i,],bb2[,i])/var(H2[i,])
+   bpmc <- sum(bb[,i]/wtsum-cc*H2[i,]/nt)
+   print(c(rho=rho,correct=cc,mean=bpmc,se=se.mean[i]*sqrt(1-rho^2)))
+ }
       rho  correct      mean          se
 0.7122092 1.079125 0.8670506 0.001969069
       rho  correct       mean         se
 0.6623438 1.057443 -0.7931535 0.00236496
       rho  correct     mean          se
 0.6862681 1.094498 1.195384 0.002572485
       rho  correct       mean          se
 0.6930217 1.058342 -0.4124472 0.001982007
       rho  correct      mean          se
 0.5696292 1.107328 0.5660097 0.003288518
       rho  correct        mean          se
 0.8613971 1.033094 -0.04610898 0.001159218
```

The first column is the correlation, the second the value of r, the 3rd the corrected estimate of the posterior mean, and the last the estimated standard error of the corrected estimate. The posterior means have changed only slightly, but with a correlation of $\rho = 1/\sqrt{2} = .707$, the standard error is reduced by a factor of $1/\sqrt{2}$. To get an equivalent reduction from increasing the sample size would require doubling the number of samples drawn. A correlation of .866 would reduce the standard error by a factor of 2, which would be the equivalent of drawing 4 times as many $\beta^{(s)}$.

8.4 Exercises

Exercise 8.1 Consider again Exercise 6.7. Evaluate the integrals defined in that problem using the following methods:

1. Direct Monte Carlo integration, sampling from the conditional normal distribution of $(z_2, z_3)|z_1$.

2. Importance sampling, sampling from the $N_2(\hat{z}, \hat{H})$ distribution, where $\hat{z} = (\hat{z}_2, \hat{z}_3)'$ maximizes the integrand $g(z_2, z_3)$ of (6.18) and $\hat{H}^{-1} = -(\partial^2 \log\{g(\hat{z}_2, \hat{z}_3)\}/\partial z_i \partial z_j)$ is minus the matrix of second derivatives of $\log\{g(\cdot, \cdot)\}$,

$$\hat{H}^{-1} = \hat{p}(1-\hat{p}) \begin{pmatrix} 1 & -2 \\ -2 & 4 \end{pmatrix} + R,$$

where $\hat{p} = \exp(2 - \hat{z}_2 + 2\hat{z}_3)/\{1 + \exp(2 - \hat{z}_2 + 2\hat{z}_3)\}$ and R is the lower right 2×2 block of V^{-1}.

Exercise 8.2 Suppose

$$Y_i = \beta_0 + \sum_{j=1}^{p} x_{ij}\beta_j + e_i/\nu^{1/2},$$

where the x_{ij} are fixed covariates, $\beta = (\beta_0, \ldots, \beta_p)'$ and ν are unknown parameters, and the e_i are iid from a known symmetric distribution with mean 0, finite variance, and density $f(\cdot)$. Suppose the prior distribution for the parameters specifies that all components are independent, with $\beta_j \sim \mathrm{N}(0, c^{-1})$ (c^{-1} is the variance) and $\nu \sim \Gamma(a, b)$, where a, b and c are known constants, and $\Gamma(a, b)$ is the gamma distribution with density proportional to $x^{a-1}\exp(-bx)$.

1. Show that the log posterior can be written as

$$\sum_i [h(r_i) + \gamma] - c\beta'\beta/2 + 2a\gamma - b\exp(2\gamma),$$

   plus a constant not involving the parameters, where $h(u) = \log(f(u))$, $r_i = (Y_i - \beta_0 - \sum_j x_{ij}\beta_j)\exp(\gamma)$ and $\gamma = \log(\nu)/2$.

2. Develop an importance sampling algorithm that can be used to estimate posterior moments and distributions. Be specific about the details, including what distribution to sample from, how to draw the samples, and give formulas for calculating the weights.

3. Suppose $c^{-1} = 10,000$ and $a = b = .1$, and that $f(u) = \exp(u)/[1 + \exp(u)]^2$, the standard logistic density. The file `impsampexr.dat` contains Y_i in the first column and 2 covariates in columns 2 and 3, for 100 cases. For this data set, use your importance sampling algorithm to calculate the posterior means of the parameters (and give estimates of the precision of the estimated means).

Chapter 9

Markov Chain Monte Carlo

By sampling from a Markov chain whose stationary distribution is the desired sampling distribution, it is possible to generate observations from distributions that would otherwise be very difficult to sample from. The drawbacks of this technique are that it is generally unknown how long the chain must be run to reach a good approximation to the stationary distribution, and once the distribution converges, the values generated are not independent.

First some background material on Markov chains will be reviewed, and then some specific methods for constructing Markov chains with a specified stationary distribution will be considered.

The major applications driving development of Markov chain Monte Carlo methods have been to problems of Bayesian inference, but they are also useful for a variety of other problems where direct generation of independent observations from the joint distribution is difficult, such as in conditional frequentist inference problems for categorical data, where the conditional sampling distributions can have complex forms, and for Monte Carlo evaluation of integrals appearing in the E step of EM algorithms.

9.1 Markov Chains

A Markov chain is a discrete time stochastic process. Only chains with a finite number of possible states will be considered formally here, although in applications it will often be convenient to think in terms of continuous distributions. The discrete state model is always technically true if the calculations are done on a computer, since there are only a finite number of values that can be represented. That is, when generating data from continuous distributions, such as the uniform or normal, the values are actually drawn from a discrete approximation to the true distribution.

Let $X^{(n)}$ be the value the chain takes at time n (formally known as the 'state'). Let $\mathcal{S} = \{x_1, \ldots, x_S\}$ be the state space, which is the set of possible values for the $X^{(n)}$ (the number of possible states S could be enormously large, as long as it is finite). The possible states x_j could be virtually anything, but in statistical applications they can usually be thought of as points in R^p.

The chain starts from an initial state $X^{(0)}$. The distribution of the state of the chain at time

$n+1$, given the state at time n, is given by a set of transition probabilities. The Markov property states that these transition probabilities depend on the past history of the chain only through the current value. That is,

$$P(X^{(n+1)} = y | X^{(j)} = x^{(j)}, j = 0, \ldots, n) = P(X^{(n+1)} = y | X^{(n)} = x^{(n)}).$$

Set

$$p(x, y) = P(X^{(n+1)} = y | X^{(n)} = x).$$

Throughout it will be assumed that the chain is homogeneous in time, meaning that these transition probabilities are the same for all n. Let

$$p^{(m)}(x, y) = P(X^{(n+m)} = y | X^{(n)} = x)$$

be the m-step transition probabilities. A Markov chain is *irreducible* if $p^{(m)}(x_i, x_j) > 0$ for some m for each pair of possible states x_i and x_j (the value of m can be different for different pairs). That is, for an irreducible chain, every state can be reached from every other state.

A state x_i is *periodic* if there exists an integer $d > 1$ such that $p^{(m)}(x_i, x_i) = 0$ whenever m is not divisible by d. A state is *aperiodic* if it is not periodic. A chain is aperiodic if all states are aperiodic. For irreducible chains with a finite state space, it can be shown that either all states are aperiodic or all are periodic.

The transition probabilities $p(x_i, x_j)$ can be organized in an $S \times S$ matrix $P = (p_{ij})$, where $p_{ij} = p(x_i, x_j)$. Also, in general the m-step transition probabilities can be put in a matrix $P^{(m)} = (p^{(m)}(x_i, x_j))$. Since the two-step transition probabilities satisfy

$$p^{(2)}(x_i, x_j) = \sum_{l=1}^{S} p(x_i, x_l) p(x_l, x_j),$$

it follows that $P^{(2)} = P^2 = P \times P$, the ordinary matrix product of P with itself. Continuing in the same fashion, it can be seen that $P^{(m)} = P^m$.

Let $\pi^{(0)}(x_j) = P(X^{(0)} = x_j)$ be the initial distribution of the chain (if the chain always starts in a particular state x_i, then this will be the degenerate distribution with $\pi^{(0)}(x_i) = 1$). Also let $\pi^{(n)}(x_j) = P(X^{(n)} = x_j)$ be the marginal distribution of $X^{(n)}$, and let $\pi^{(n)} = (\pi^{(n)}(x_1), \ldots, \pi^{(n)}(x_S))'$. Since

$$\pi^{(1)}(x_j) = \sum_{l=1}^{S} \pi^{(0)}(x_l) p(x_l, x_j),$$

it follows that $\pi^{(1)} = P\pi^{(0)}$, and continuing in the same fashion, $\pi^{(n)} = P^n \pi^{(0)}$

For irreducible, aperiodic chains, there is a unique probability distribution with mass probabilities $\pi_j = \pi(x_j)$ (for state x_j) satisfying

$$\pi = P\pi,$$

where $\pi = (\pi_1, \ldots, \pi_S)'$. This distribution is known as the stationary distribution. If the initial distribution $\pi^{(0)}$ is the stationary distribution π, then $\pi^{(1)} = P\pi^{(0)} = P\pi = \pi$, and continuing in

the same fashion, $\pi^{(n)} = \pi$ for all n. Thus if the chain starts from its stationary distribution, the marginal distribution of the state at time n is again given by the stationary distribution.

Another important result is that for an irreducible, aperiodic chain with stationary distribution π,

$$\lim_{n\to\infty} \pi^{(n)} = \pi,$$

regardless of the initial distribution $\pi^{(0)}$. That is, the marginal distribution of $X^{(n)}$ converges to the stationary distribution as $n \to \infty$. Thus if an irreducible, aperiodic Markov chain is started from some arbitrary state, then for sufficiently large n, the current state $X^{(n)}$ is essentially generated from the stationary distribution on $\mathcal{S}$. Also, once the distribution $\pi^{(n)}$ converges to the stationary distribution, the marginal distribution of the state at all future times is again given by the stationary distribution, so these values are an identically distributed sample from this distribution (they are generally not independent, though). Thus a way to generate values from a distribution f on a set $\mathcal{S}$ is to construct a Markov chain with f as its stationary distribution, and to run the chain from an arbitrary starting value until the distribution $\pi^{(n)}$ converges to f. Two important problems are how to construct an appropriate Markov chain, and how long the chain needs to be run to reach the stationary distribution.

For an irreducible, aperiodic chain, if

$$\pi_i p(x_i, x_j) = \pi_j p(x_j, x_i) \tag{9.1}$$

for all i, j, then π is the stationary distribution. This follows because from (9.1),

$$\sum_{i=1}^{S} \pi_i p(x_i, x_j) = \sum_{i=1}^{S} \pi_j p(x_j, x_i) = \pi_j,$$

since $\sum_i p(x_j, x_i) = 1$, so by definition, π must the the stationary distribution. (9.1) is called a reversibility condition, since it states that for the stationary distribution, the probability of being in state x_i and moving to state x_j on the next step is the same as the probability of being in state x_j and moving to state x_i. Condition (9.1) is usually easy to check, and will be very useful in helping to construct chains with arbitrary stationary distributions, but not all chains have stationary distributions that satisfy this condition.

An analogous formulation is available for continuous state spaces, and as already noted, applications are often formulated in terms of density functions on continuous spaces, instead of mass functions on discrete spaces, even though the actual sampling is technically done from discrete approximations to the continuous densities.

Some particular methods of constructing Markov Chains will be described next.

9.2 The Metropolis-Hastings Algorithm

Practical application of Markov chain sampling goes back at least to Metropolis et al (1953). Hastings (1970) extended the basic proposal from that paper and gave some of the first applications in the statistical literature. These methods did not become popular until the wide-spread availability of high speed computers in the last decade. Beginning with Tanner and

Wong (1987), and Gelfand and Smith (1990), there has now been an enormous literature developed on Markov chain Monte Carlo.

Hastings' extension of the Metropolis et al algorithm will be referred to as the M-H algorithm in the following. The M-H algorithm gives a general method for constructing a Markov chain with stationary distribution given by an arbitrary mass function (or approximating density) $f(x)$. Let $q(x, y)$ be any Markov chain transition kernel whose state space is the same as the sample space of f. Some specific proposals for $q(x, y)$ will be discussed below. To be useful, $q(x, \cdot)$ should be easy to sample from, but generally q does not have f as its stationary distribution. Define

$$\alpha(x, y) = \min\left\{\frac{f(y)q(y, x)}{f(x)q(x, y)}, 1\right\}. \tag{9.2}$$

Given the state of the chain at time n, $X^{(n)}$, the M-H algorithm samples a trial value $X_t^{(n+1)}$ from $q(X^{(n)}, \cdot)$, sets $X^{(n+1)} = X_t^{(n+1)}$ with probability $\alpha(X^{(n)}, X_t^{(n+1)})$, and sets $X^{(n+1)} = X^{(n)}$ with probability $1 - \alpha(X^{(n)}, X_t^{(n+1)})$. In practice this is accomplished by drawing $U^{(n)} \sim U(0, 1)$ and setting $X^{(n+1)} = X_t^{(n+1)} I\{U^{(n)} \le \alpha(X^{(n)}, X_t^{(n+1)})\} + X^{(n)} I\{U^{(n)} > \alpha(X^{(n)}, X_t^{(n+1)})\}$.

The transition kernel of the resulting chain is given by

$$p(x, y) = \begin{cases} q(x, y)\alpha(x, y) & y \neq x \\ 1 - \sum_{u \neq x} q(x, u)\alpha(x, u) & y = x. \end{cases} \tag{9.3}$$

Roughly, if $\alpha(X^{(n)}, X_t^{(n+1)}) < 1$ then $X^{(n)}$ is underrepresented relative to $X_t^{(n+1)}$ in the chain generated by q, and occasionally rejecting $X_t^{(n+1)}$ and keeping $X^{(n)}$ adjusts for this underrepresentation. More formally, it can be shown that f satisfies the reversibility condition (9.1) for the transition kernel (9.3) for all x and y in the sample space of f, guaranteeing that f is the stationary distribution. For example, suppose $x \neq y$ are such that $f(x)q(x, y) > f(y)q(y, x)$. Then $\alpha(x, y) = f(y)q(y, x)/f(x)q(x, y)$ and $\alpha(y, x) = 1$, so

$$\begin{aligned} f(x)p(x, y) &= f(x)q(x, y)\alpha(x, y) \\ &= f(x)q(x, y)f(y)q(y, x)/f(x)q(x, y) \\ &= f(y)q(y, x) \\ &= f(y)q(y, x)\alpha(y, x) = f(y)p(y, x). \end{aligned}$$

The other cases are left as an exercise.

Note that since f appears in both the numerator and denominator of (9.2), f only needs to be known to within a normalizing constant.

The success of this algorithm depends on how close $q(x, y)$ is to $f(y)$. If $f(\cdot)$ is small where $q(x, \cdot)$ is large, then most trial points sampled will be rejected, and the chain will stay for long periods of time in the same state. Thus choosing an appropriate q is in general not a trivial problem. The ideas discussed in previous contexts, such as using distributions with the same mode and similar spread as f, but easier to sample from, again play an important role.

Some specific methods for constructing transitional kernels $q(x, y)$ will now be described. In the following, let $\hat{x}$ be the mode of $f(x)$, and let $\hat{H} = -\{\partial^2 \log[f(\hat{x})]/\partial x \partial x'\}^{-1}$. If these quantities

cannot be easily approximated, then other approximations to the mean and variance of $f(x)$ could be considered, too.

9.2.1 Random Walk Chain

Let g be a density defined on the same space as f, and set $q(x, y) = g(y - x)$. One choice for g would be the mean 0 normal density with covariance matrix $\hat{H}$. A multivariate t distribution with dispersion matrix $\hat{H}$ might also be an appropriate choice.

If g is symmetric, then

$$\alpha(x, y) = \min\left\{\frac{f(y)g(y-x)}{f(x)g(x-y)}, 1\right\} = \min\{f(y)/f(x), 1\}.$$

This was the original algorithm proposed by Metropolis et al (1953).

9.2.2 Independence Chain

Again let g be a density defined on the same space as f, and set $q(x, y) = g(y)$. That is, trial values are generated independently of the current value. Again logical choices for g are a normal or t distribution with mean $\hat{x}$ and dispersion matrix $\hat{H}$. In this case

$$\alpha(x, y) = \min\left\{\frac{f(y)g(x)}{f(x)g(y)}, 1\right\} = \min\left\{\frac{f(y)/g(y)}{f(x)/g(x)}, 1\right\},$$

so $\alpha(X^{(n)}, X_t^{(n+1)})$ is the ratio of the importance sampling weights at the current and the trial points.

9.2.3 Rejection Sampling Chain

For rejection sampling, it is necessary to find a function g which everywhere dominates the density f. It is often difficult to prove a function dominates everywhere. For example, let g be a t density with small to moderate degrees of freedom, with mean $\hat{x}$ and dispersion matrix $c\hat{H}$ for some $c > 1$, rescaled so that $g(\hat{x}) > f(\hat{x})$. Such a function would often dominate f everywhere, or nearly everywhere, but this would generally be difficult to to prove. Tierney (1994) gave a method to use the general M-H algorithm to correct for possible non-dominance in the proposed dominating function g.

Suppose then that there is a rejection sampling algorithm sampling from a density proportional to a function g, which may not actually dominate f everywhere. In Tierney's method, at each step in the M-H algorithm, the rejection sampling algorithm is run, and the M-H trial value $X_t^{(n+1)}$ is the first value not rejected in the rejection sampling. If g does not actually dominate, then the density/mass function for $X_t^{(n+1)}$ is $h(x) \propto \min\{f(x), g(x)\}$, and the M-H transition kernel $q(x, y) = h(y)$. Then defining

$$\alpha(x, y) = \min\left\{\frac{f(y)h(x)}{f(x)h(y)}, 1\right\}$$

gives an M-H chain that corrects the sample for possible non-dominance of g. If g does dominate f, then $\alpha(x, y) \equiv 1$, and this algorithm is identical to rejection sampling. If g does not dominate, then points where $g(x) < f(x)$ will have $\alpha(x, y) < 1$ for some values of y (assuming g does dominate at some points), so when these non-dominated points do occur, the M-H algorithm will sometimes reject the new trial point, increasing the frequency of the non-dominated points. It is straightforward to verify that f satisfies the reversibility condition (9.1) for the transition kernel of this chain, guaranteeing that f is the stationary distribution.

Given the similarities among importance sampling, independence chain sampling, and rejection chain sampling, it is an interesting question which can be implemented more efficiently in particular applications.

9.3 Block-at-a-Time Algorithms

If the above algorithms were the full extent of what could be achieved with Markov chain sampling, then there might be little to be gained over other methods such as importance sampling. The real power of Markov chain sampling is that when using a chain to generate observations from a vector valued distribution, it is not necessary to update all components simultaneously, so that a complex problem can be broken down into a series of simpler problems.

Suppose $X \sim f(x)$ can be divided into U and V, $X = (U, V)$. (In this section, the orientation of vectors does not matter, although elsewhere vectors will continue to be column vectors.) Both U and V can also be vector valued. Let $f_{U|V}(u|v)$ and $f_{V|U}(v|u)$ be the conditional distributions of $U|V$ and $V|U$. Suppose $p_{1|2}(\cdot, \cdot\,|v)$ is a Markov chain transition kernel with stationary distribution $f_{U|V}(\cdot\,|v)$, and $p_{2|1}(\cdot, \cdot\,|u)$ is a Markov chain transition kernel with stationary distribution $f_{V|U}(\cdot\,|v)$.

Given the current state $X^{(n)} = (U^{(n)}, V^{(n)})$, consider the two step update

1. generate $U^{(n+1)}$ from $p_{1|2}(U^{(n)}, \cdot\,|V^{(n)})$,
2. generate $V^{(n+1)}$ from $p_{2|1}(V^{(n)}, \cdot\,|U^{(n+1)})$.

The two steps above can be thought of as generating a single update $X^{(n+1)}$ of the process. This update has transition kernel

$$p(x, y) = p((u, v), (w, z)) = p_{1|2}(u, w|v)p_{2|1}(v, z|w).$$

This transition kernel generally does not satisfy the reversibility condition (9.1), but it does have f as its stationary distribution. To see this, note that

$$\begin{aligned}
\sum_u \sum_v f(u, v)p((u, v), (w, z)) &= \sum_u \sum_v f_{U|V}(u|v)f_V(v)p_{1|2}(u, w|v)p_{2|1}(v, z|w) \\
&= \sum_v f_{U|V}(w|v)f_V(v)p_{2|1}(v, z|w) \\
&= \sum_v f_{V|U}(v|w)f_U(w)p_{2|1}(v, z|w) \\
&= f_{V|U}(z|w)f_U(w) = f(w, z),
\end{aligned}$$

where the second line follows because $f_{U|V}(w|v)$ is the stationary distribution of $p_{1|2}(u, w|v)$, and the last line because $f_{V|U}(z|w)$ is the stationary distribution of $p_{2|1}(v, z|w)$. Thus to find a Markov chain with stationary distribution f, it is only necessary to find transition kernels for the conditional distributions of blocks of components. These can be simpler to construct and to sample from.

This approach can be extended to any number of blocks. Any Metropolis-Hastings type update can be used within each block, and different types of updates can be used in different blocks.

The advantage of a block-at-a-time algorithm is that it is often much easier to find good approximations to the conditional distributions to use in M-H and other Markov chain updating schemes, leading to simpler methods of generating new values and greater acceptance rates of generated trial values. In some applications the conditional distributions can be sampled from directly, as described in the following subsection. However, separately updating blocks will often induce greater autocorrelation in the resulting Markov chain, leading to slower convergence. Transformations to reduce the correlation between blocks can greatly improve the performance of block-at-a-time algorithms, although there are no simple general recipes for finding appropriate transformations.

9.3.1 Gibbs Sampling

Gibbs sampling is a block-at-a-time update scheme where the new values for each block are generated directly from the full conditional distributions. That is, in terms of the previous notation,

$$p_{1|2}(u, w|v) = f_{U|V}(w|v)$$

and

$$p_{2|1}(v, z|w) = f_{V|U}(z|w).$$

It turns out a variety of problems can be put in a framework where the full conditional distributions are easy to sample from. Gelfand et al (1990) gave several examples. This is especially true of hierarchical normal random effects models, and incomplete data problems involving normal distributions.

The idea of generating data from full conditional distributions was discussed by Geman and Geman (1984), and independently by Tanner and Wong (1987). The origin of the term 'Gibbs sampling' is not completely clear, since neither of these papers used this term. Geman and Geman did use Gibbs distributions in their paper, though.

9.4 Implementation Issues

The goal of Markov chain sampling (in this context) is to generate observations from a specified distribution f. Having constructed a transition kernel with stationary distribution f, the usual approach to is to start from an arbitrary point and run the chain until it is thought to have converged, and to discard the values generated prior to convergence. The values generated before convergence (and discarded) are referred to as the 'burn-in'.

Gelfand and Smith (1990) proposed just keeping a single value from the chain, and starting a new

chain to generate each new observation from f. In this way they obtain an independent series of observations from f. It is now generally recognized that this in inefficient. If the marginal distributions of the Markov chain have converged, then more information would be obtained by continuing to sample from the same chain than from starting a new one, since each time the chain is restarted from an arbitrary value there is a new burn-in period that needs to be discarded.

Empirical monitoring for convergence of a Markov chain is not a simple problem, and examples can be constructed that will fool any particular test. Gelman and Rubin (1992) advocated running several parallel chains, starting from widely dispersed values, and gave a convergence diagnostic based on an analysis of variance procedure for comparing the within chain and between chain variability. At minimum, it seems prudent to run several chains from different starting values and to compare inferences from the different chains. There have been a number of other convergence diagnostics proposed. Cowles and Carlin (1996) gave a comparative review. Simple plots of the trace of generated values over the iterations ($X^{(n)}$ versus n) and of cumulative sums ($\sum(X^{(n)} - \overline{X})$), can reveal serious problems, although apparently good results in selected plots does not guarantee overall convergence.

Perhaps the greatest danger is that a chain run for some finite period of time will completely miss some important region of the sample space. For example, if a density has distinct local modes, with little probability mass between the modes, then the chain will tend to become trapped in one of the modes and could completely miss the other modes. In this case, although technically the chain might be irreducible, practically speaking it is not, since the probability of a transition from the neighborhood of one mode to another is very small. Gelman and Rubin (1992) also advocated doing an extensive search for local modes, and starting chains within the distinct modes. Once the modes are located, M-H sampling based on mixtures of components centered on each mode could be considered.

9.4.1 Precision of Estimates

If $X^{(1)}, X^{(2)}, \ldots, X^{(N)}$ are the values generated from a single long run of a Markov chain with stationary distribution $f(x)$, and if the chain has approximately converged by the Bth iteration, then an approximately unbiased estimator for

$$\int b(x)f(x)\,dx$$

is

$$\tilde{b} = \sum_{i=B+1}^{N} b(X^{(i)})/(N-B),$$

(see eg Tierney, 1994, for a more precise statement of consistency and asymptotic normality of averages of sequences from Markov chains). Since the $X^{(i)}$ are generally not independent, determining the precision of this estimator is not a trivial problem. Let $\sigma^2 = \mathrm{Var}[b(X^{(i)})]$ be the marginal variance and $\rho_k = \mathrm{cor}[b(X^{(i)}), b(X^{(i+k)})]$ be the lag k autocorrelation in the sequence. If the chain is assumed to have converged by the Bth step, σ^2 and the ρ_k should not depend on i for $i > B$. If the autocorrelations die out reasonably fast and can be assumed to be negligible for

$k > K$ for some K, then

$$\mathrm{Var}(\tilde{b}) = \frac{1}{(N-B)^2}\left(\sum_{i=B+1}^{N}\sigma^2 + \sum_{i<j} 2\rho_{j-i}\sigma^2\right) = \frac{\sigma^2}{(N-B)^2}\left(N-B+2\sum_{j=1}^{K}(N-B-j)\rho_j\right).$$

If the autocorrelations do go to zero reasonably fast, then the usual empirical moments will be consistent (although somewhat biased) for σ^2 and the ρ_k, so this quantity is straightforward to estimate.

Since Markov chain Monte Carlo runs are often quite long, a simpler approach is to group the data into blocks and estimate the variance from the block means. That is, suppose $N - B = Jm$, and let

$$\tilde{b}_j = \sum_{i=B+1+(j-1)m}^{B+jm} b(X^{(i)})/m, \quad j = 1, \ldots, J,$$

be the means of groups of m consecutive values. Note that $\tilde{b} = \sum_{j=1}^{J}\tilde{b}_j/J$. If m is large relative to the point at which the autocorrelations die out, then the correlations among the $\tilde{b}_j$ should be negligible, and the variance can be estimated as if the $\tilde{b}_j$ were independent. If the correlation is slightly larger, then it might be reasonable to assume the correlation between $\tilde{b}_j$ and $\tilde{b}_{j+1}$ is some value ρ to be estimated, but that correlations at larger lags are negligible. In this case

$$\mathrm{Var}(\tilde{b}) \doteq \mathrm{Var}(\tilde{b}_j)(1+2\rho)/J, \tag{9.4}$$

and ρ and $\mathrm{Var}(\tilde{b}_j)$ can be estimated using empirical moments. The following function computes (9.4) for a generated vector `x`.

```
varest <- function(x,m=100) {
# estimate standard error in mean of correlated sequence x by using
# variation in means of blocks of size m.  Assumes adjacent blocks are
# correlated, but blocks at greater lags are not
  ng <- floor(length(x)/m)
  if (ng*m < length(x)) x <- x[-(1:(length(x)-ng*m))]
  ind <- rep(1:ng,rep(m,ng))
  gmx <- tapply(x,ind,mean)
  mx <- mean(gmx)
  rho <- cor(gmx[-1],gmx[-length(gmx)])
  c(mean=mx,se=sqrt(var(gmx)*(1+2*rho)/ng),rho=rho)
}
```

Other methods of estimation, such as overlapping batch means, are discussed in Chapter 3 of Chen, Shao and Ibrahim (2000).

9.5 One-way Random Effects Example

The following example uses Gibbs sampling for a Bayesian analysis of a one-way random effects model, which is similar to that used in Section 4 of Gelfand et al (1990). The distribution of the responses Y_{ij} is given by

$$Y_{ij} = \theta_i + e_{ij}, \quad i = 1, \ldots, n, \quad j = 1, \ldots, J,$$

with the θ_i iid $N(\mu, \sigma_\theta^2)$ and the e_{ij} iid $N(0, \sigma_e^2)$. The prior on μ is $N(\mu_0, \sigma_0^2)$, the prior on $1/\sigma_e^2$ is $\Gamma(a_1, b_1)$ and the prior on $1/\sigma_\theta^2$ is $\Gamma(a_2, b_2)$, where $\Gamma(a, b)$ denotes the gamma distribution with density proportional to $x^{a-1}\exp(-bx)$.

Integrating over the random effects, conditional on the parameters $(\mu, \sigma_\theta^2, \sigma_e^2)$, the vectors $Y_i = (Y_{i1}, \ldots, Y_{iJ})'$ are independent with

$$Y_i \sim N(\mu \mathbf{1}_J, \sigma_e^2 I + \sigma_\theta^2 \mathbf{1}_J \mathbf{1}_J').$$

In this simple example, explicit formulas for the determinant and inverse of the covariance matrix are easily given, so an explicit formula for the likelihood is available. With only 3 parameters it is likely that posterior means and variances can be calculated using the Gaussian quadrature methods described earlier. And with only 3 parameters a variety of methods could be used to sample from the posterior. Gibbs sampling provides an especially easy way to sample from the posterior, though, and the Gibbs sampling algorithm is easily generalized to more complex random effects models.

For Gibbs sampling to be easy to implement, it is necessary that each of the full conditional distributions be easy to sample from. Here this is accomplished by thinking of the θ_i as parameters (in the Bayesian sense), and the random effects structure as a hierarchical prior on these parameters. Conditional on $\theta = (\theta_1, \ldots, \theta_n)'$ and σ_e^2, the Y_{ij} are iid $N(\theta_i, \sigma_e^2)$, so the likelihood is just

$$L(\theta, \sigma_e^2) = \sigma_e^{-nJ} \exp\left(-\sum_{i,j} (Y_{ij} - \theta_i)^2 / (2\sigma_e^2) \right),$$

and the posterior is proportional to $L(\theta, \sigma_e^2) g_1(\theta|\mu, \sigma_\theta^2) g_2(\mu) g_3(\sigma_\theta^2) g_4(\sigma_e^2)$, where g_1 is the conditional prior density of the θ_i (iid $N(\mu, \sigma_\theta^2)$) and g_2, g_3 and g_4 are the densities of the prior distributions of μ, σ_θ^2 and σ_e^2 given earlier. The conditional distributions that need to be sampled then are $[\theta|\sigma_e^2, \sigma_\theta^2, \mu]$, $[\mu|\theta, \sigma_\theta^2]$, $[\sigma_\theta^2|\mu, \theta]$ and $[\sigma_e^2|\theta]$ (these are also conditional on the data, but that only directly affects the first and last). The priors were chosen to be conditionally conjugate, so the sampling will only require generating values from normal and gamma distributions. It is straightforward to verify that the conditional distributions are

$$\begin{aligned}
[\theta|\sigma_e^2, \sigma_\theta^2, \mu] &\overset{iid}{\sim} N\left(\frac{J\overline{Y}_i \sigma_\theta^2 + \mu\sigma_e^2}{J\sigma_\theta^2 + \sigma_e^2}, \frac{\sigma_\theta^2 \sigma_e^2}{J\sigma_\theta^2 + \sigma_e^2} \right) \\
[\mu|\theta, \sigma_\theta^2] &\sim N\left(\frac{n\overline{\theta}\sigma_0^2 + \mu_0 \sigma_\theta^2}{n\sigma_0^2 + \sigma_\theta^2}, \frac{\sigma_0^2 \sigma_\theta^2}{n\sigma_0^2 + \sigma_\theta^2} \right) \\
[1/\sigma_\theta^2|\mu, \theta] &\sim \Gamma(n/2 + a_2, b_2 + \sum_i (\theta_i - \mu)^2/2) \\
[1/\sigma_e^2|\theta] &\sim \Gamma(nJ/2 + a_1, b_1 + \sum_{i,j} (Y_{ij} - \theta_i)^2/2),
\end{aligned}$$

where $\overline{Y}_i = \sum_j Y_{ij}/J$ and $\overline{\theta} = \sum_i \theta_i/n$. Starting from arbitrary values, the Gibbs sampler consists of sampling from each of these distributions in turn, where at each iteration the current values are used for any parameters in the corresponding conditional distribution.

In the calculations below, the distributions are sampled in the order given above, but this was arbitrary. The values used in the priors are `m0`= $\mu_0 = 0$, `nu0`= $1/\sigma_0^2 = 10^{-5}$, and `a1, b1, a2, b2`

(a_1, b_1, a_2, b_2) all equal to .001. The variable names used for the parameters are `ths` for θ, `nu.es` for $1/\sigma_e^2$, `nu.ths` for $1/\sigma_\theta^2$, and `mus` for μ. The commands used to generate the data set are given below. The actual data is in the file `gibbs.dat`.

```
> K <- 6; J <- 5; mu <- 5; vt <- 4; ve <- 16;
> .Random.seed
 [1]  1 53 41 59 44  3 59  6 53 42  5  3
> theta <- rnorm(K,mu,sqrt(vt))
> Y <- rnorm(K*J,0,sqrt(ve)) + rep(theta,rep(J,K))
> group <- rep(1:K,rep(J,K))
> ybar <- tapply(Y,group,mean)
> write(rbind(Y,group),"gibbs.dat",2)
> # summary of the data
> ybar
        1        2          3        4         5        6
 5.097592 7.457386 -0.2985133 8.141821 -1.162981 7.066291
> mean(ybar)
[1] 4.383599
>
> # prior distribution parameters
> m0 <- 0; nu0 <- 1.e-5; a1 <- .001; b1 <- .001; a2 <- .001; b2 <- .001;
```

The following function runs the Gibbs sampler, using 0's and 1's as initial values. It is used to generate an initial sequence of 500 values, which are plotted in Figure 9.1.

```
> gibbs.re <- function(nt) {
+   print(.Random.seed)
+ # initial values for Gibbs Sampling
+   mus <- rep(0,nt+1)
+   nu.es <- nu.ths <- rep(1,nt+1)
+   ths <- matrix(0,nrow=K,ncol=nt+1)
+ # The following loop runs the sampler.
+   for (i in 2:(nt+1)) {
+     j <- i-1
+     wt <- 1/(nu.es[j]*J+nu.ths[j])
+     ths[,i] <- rnorm(K,(J*ybar*nu.es[j]+nu.ths[j]*mus[j])*wt,sqrt(wt))
+     wt <- 1/(K*nu.ths[j]+nu0)
+     mus[i] <- rnorm(1,(nu.ths[j]*K*mean(ths[,i])+nu0*m0)*wt,sqrt(wt))
+     nu.ths[i] <- rgamma(1,K/2+a2)/(sum((ths[,i]-mus[i])^2)/2+b2)
+     nu.es[i] <- rgamma(1,J*K/2+a1)/(b1+sum((Y-ths[group,i])^2)/2)
+   }
+   list(mus=mus,nu.es=nu.es,nu.ths=nu.ths,ths=ths)
+ }
> nt <- 500
> run1 <- gibbs.re(nt)
 [1]  9 24 29 51 30  2  6  5 60 30 56  3
```

```
> postscript("fig1.ps",pointsize=20)
> plot(1:nt,1/run1$nu.ths[-1],type="l",xlab="Iteration",main=
+      "Variance of Theta Dist")
> dev.off()
> postscript("fig2.ps",pointsize=20)
> plot(1:nt,1/run1$nu.es[-1],type="l",xlab="Iteration",main=
+      "Variance of Error Dist")
> dev.off()
> postscript("fig3.ps",pointsize=20)
> plot(1:nt,run1$mus[-1],type="l",xlab="Iteration",main=
+      "Mean of Theta Dist")
> dev.off()
> postscript("fig4.ps",pointsize=20)
> plot(1:nt,run1$ths[2,-1],type="l",xlab="Iteration",main="Theta2")
> dev.off()
```

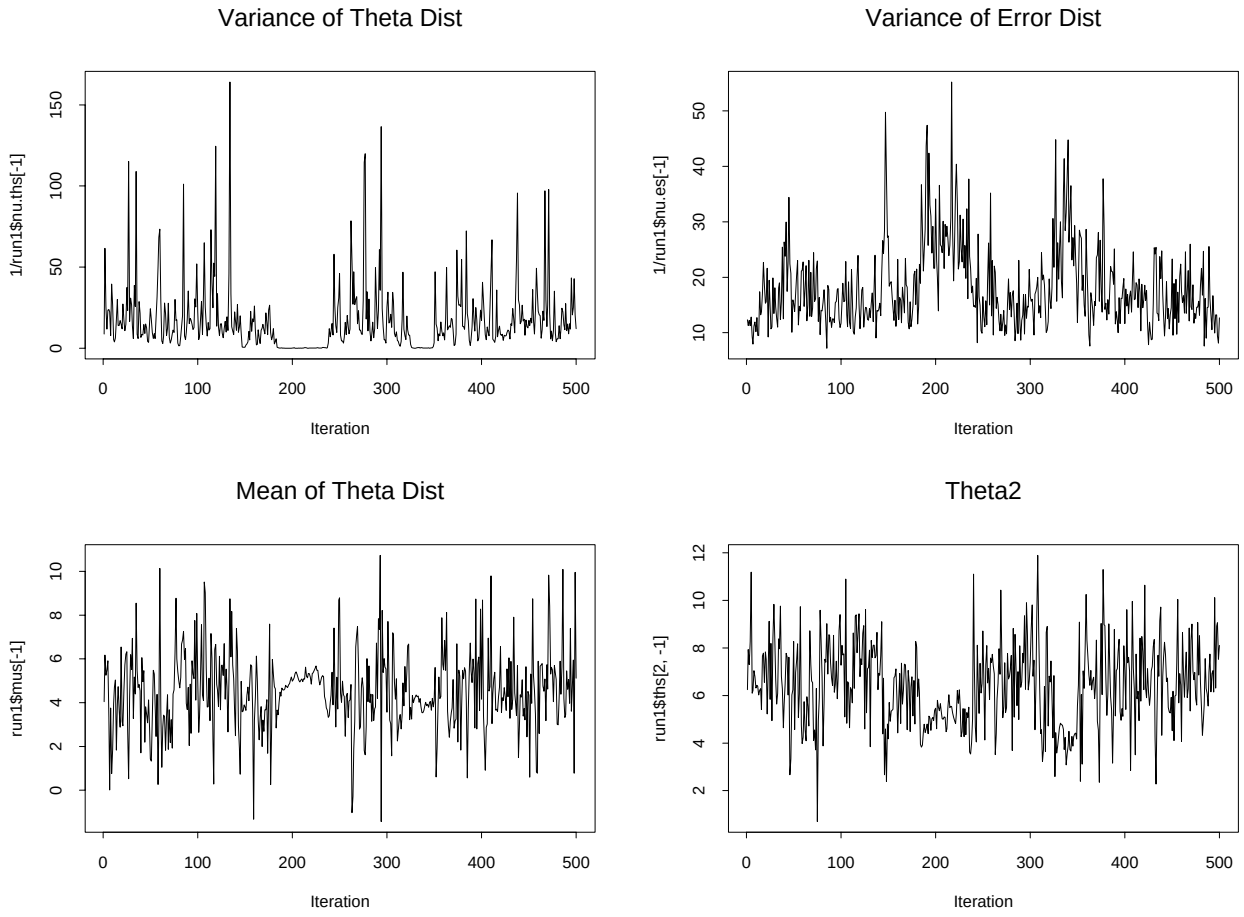

Figure 9.1: Output from the first 500 cycles

After viewing output from the first 500 samples, it was decided (informally, and perhaps conservatively) to discard the first 100 from a full run of 5000 cycles through the sampler.

```
> run2 <- gibbs.re(5000)
 [1]  1 63 42 14 52  3 63 11 22 47 14  3
> ths <- run2$ths[,-(1:101)]
> mus <- run2$mus[-(1:101)]
> nu.ths <- run2$nu.ths[-(1:101)]
> nu.es <- run2$nu.es[-(1:101)]
> # How much correlation is there in the sequence?
> acor <- function(x,k) cor(x[-(1:k)],x[-((length(x)-k+1):length(x))])
> for (k in c(1,10,20,30,40,50,100,200,300)) print(acor(1/nu.ths,k))
[1] 0.2582468
[1] 0.001480924
[1] 0.001870187
[1] -0.006411884
[1] -0.009027028
[1] -0.01824866
[1] -0.001897716
[1] -0.003997428
[1] 0.008303639
> for (k in c(1,10,20,30,40,50,100,200,300)) print(acor(1/nu.es,k))
[1] 0.3178706
[1] 0.1060543
[1] 0.07578219
[1] 0.02398592
[1] 0.003063344
[1] -0.002129605
[1] 0.001628338
[1] 0.06055358
[1] 0.004518483
> # Estimate the sampling error in the posterior means
> varest(1/nu.es)
     mean        se         rho
 16.33774 0.2040973 -0.01361772
> # the following is roughly the se for 4900 independent samples
> sqrt(var(1/nu.es)/length(nu.es))
[1] 0.07907397
> # spread of the posterior
> quantile(1/nu.es,c(.025,.5,.975))
     2.5%    50.0%    97.5%
 8.988528 15.19181 30.26893
> # variance of random effects
> varest(1/nu.ths)
     mean        se         rho
 22.02392 0.6837816 -0.07702611
> sqrt(var(1/nu.ths)/length(nu.ths))
[1] 0.5079442
> quantile(1/nu.ths,c(.025,.5,.975))
```

```
     2.5%    50.0%    97.5%
 1.086735 13.67551 93.38017
> # overall mean
> varest(mus)
     mean         se        rho
 4.444879 0.02229332 -0.3051893
> sqrt(var(mus)/length(mus))
[1] 0.02962804
> quantile(mus,c(.025,.5,.975))
      2.5%    50.0%    97.5%
 0.4691267 4.433664 8.608497
```

To examine the sensitivity to the prior, the runs were repeated with different priors on variance of the random effects. In the first case, the sample generated is almost noninformative, due to extremely high autocorrelation in the μ sequence, as can be seen in Figure 9.2. In the second case, performance was much better.

```
> a2 <- 2 # prior for variance of random effects
> #         has mode at 1, variance of 2000000).
> run3 <- gibbs.re(5000)
 [1] 13 32 24 55 17  0 36 29 20 11  2  0
> ths <- run3$ths[,-(1:101)]
> mus <- run3$mus[-(1:101)]
> nu.ths <- run3$nu.ths[-(1:101)]
> nu.es <- run3$nu.es[-(1:101)]
> for (k in c(1,10,20,30,40,50)) print(acor(1/nu.ths,k))
[1] 0.6487311
[1] -0.01415425
[1] 0.004182847
[1] -0.0003099855
[1] -0.01843221
[1] -0.006256902
> for (k in c(1,10,20,30,40,50,100,200,300)) print(acor(mus,k))
[1] 0.9991632
[1] 0.9912071
[1] 0.9827219
[1] 0.9748673
[1] 0.9662091
[1] 0.9564646
[1] 0.9043435
[1] 0.8012627
[1] 0.6749711
>
> # the following se is not valid, since the correlation in blocks at
> # lag more than 1 is much larger than 0.
> varest(mus)
```

```
     mean        se       rho
 5.085794 0.1066138 0.9319637
> varest(1/nu.ths)
         mean           se       rho
 0.0009894144 7.207767e-05 0.1943027
> postscript("plot2.ps",pointsize=20)
> plot(1:length(mus),mus,type="l",xlab="Iteration",main=
+      "Mean of Theta Dist")
> dev.off()
> ## different prior where the performance is better
> a2 <- .5
> b2 <- 1
> run4 <- gibbs.re(5000)
 [1] 45  8 31 63 55  3  7  2 40 52 54  0
> ths <- run4$ths[,-(1:101)]
> mus <- run4$mus[-(1:101)]
> nu.ths <- run4$nu.ths[-(1:101)]
> nu.es <- run4$nu.es[-(1:101)]
> varest(1/nu.es)
     mean        se       rho
 16.46548 0.1015839 -0.204841
> quantile(1/nu.es,c(.025,.5,.975))
     2.5%    50.0%    97.5%
 8.991402 15.38827 30.09218
> varest(1/nu.ths)
     mean        se        rho
 15.12667 0.3180309 -0.1422266
> quantile(1/nu.ths,c(.025,.5,.975))
     2.5%    50.0%    97.5%
 1.380939 10.51433 56.65054
> varest(mus)
     mean         se       rho
 4.374258 0.03810206 0.1592661
> quantile(mus,c(.025,.5,.975))
      2.5%    50.0%    97.5%
 0.7687718 4.392122 7.927748
```

9.6 Logistic Regression Example

Consider again the example of Bayesian inference in the standard binary logistic regression model from Section 8.3. The likelihood again is

$$L(\beta) = \prod_{i=1}^{n} \frac{\exp(y_i x_i' \beta)}{1 + \exp(x_i' \beta)},$$

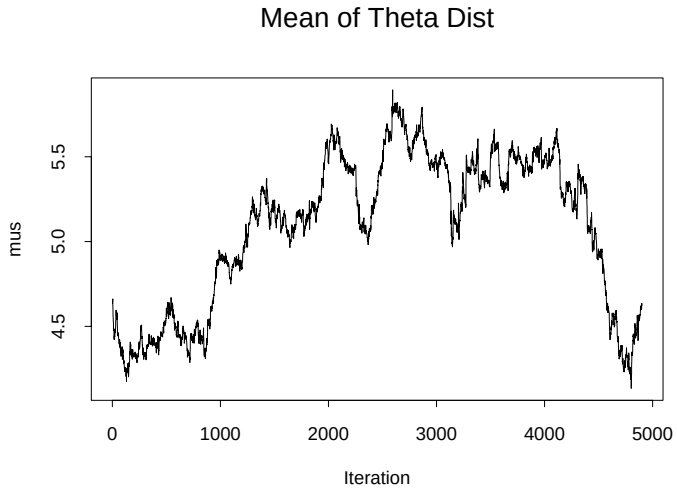

Figure 9.2: Gibbs sampler sequence for μ, from the first alternate prior, where the sampler performed poorly.

where the y_i are binary responses and the x_i are p-dimensional covariate vectors, and the prior on β specifies that the individual components are iid $\mathrm{N}(0, \sigma^2)$, so the joint prior density is

$$\pi(\beta) \propto \exp[-\beta'\beta/(2\sigma^2)].$$

The posterior is then

$$\pi(\beta|y) \propto L(\beta)\pi(\beta).$$

In the following subsections, independence chain sampling and rejection chain sampling are applied to this problem.

9.6.1 Independence Chain Sampling

Here the independence chain M-H algorithm from Section 9.2.2 will be applied.

M-H transition kernel for generating trial values is $q(\beta^{(s)}, \beta) = g(\beta)$, a fixed density independent of the current value $\beta^{(s)}$. In this example the normal approximation to the posterior (as given in Section 8.3) will be used for $g(\beta)$. In this case

$$\alpha(\beta^{(s)}, \beta) = \frac{\pi(\beta|y)/g(\beta)}{\pi(\beta^{(s)}|y)/g(\beta^{(s)})} = \frac{w(\beta)}{w_s},$$

where $w(\beta)$ and $w_s = w(\beta^{(s)})$ are the importance sampling weights as defined in (8.8).

To sample from this chain, trial values of β are sampled from the normal approximation to the posterior, exactly as in importance sampling, and the weights w_s are calculated as before. In fact, the exact set of $\beta^{(s)}$ values used in Section 8.3 could be used, although new values were generated here. Since the commands for generating the $\beta^{(s)}$ are identical to Section 8.3, that part of the output is omitted. Then for each $s = 2, \ldots, S$, a $U_s \sim U(0, 1)$ is generated, and if $U_s < w_s/w_{s-1}$ then the trial value is retained, and if not then $\beta^{(s)}$ is set equal to $\beta^{(s-1)}$ (and $w_s = w_{s-1}$). This is done in a `for()` loop, but since little is being done inside the loop, it turns out to be quite fast.

```
> Hi <- H
> wti <- wt
> ui <- runif(nt)
> l <- 0
> for (j in 2:nt) {
+   if (ui[j] >= wti[j]/wti[j-1]) {
+     l <- l+1
+     Hi[,j] <- Hi[,j-1]
+     wt[,j] <- wt[,j-1]
+   }
+ }
> l
[1] 1275
> apply(Hi,1,mean)
[1]  0.8527268 -0.7810300  1.1719671 -0.4104312  0.5518204 -0.0425595
> bpm
[1]  0.86455126 -0.79344280  1.19705198 -0.40929410  0.56454239 -0.04644663
> bhat
[1]  0.8282945 -0.7588482  1.1381240 -0.3916353  0.5320517 -0.0423460
```

Note that of the 10,000 trial values generated, 1,275 wind up being rejected. The advantage of using the independence chain over importance sampling is that once the chain converges to its stationary distribution, the $\beta^{(s)}$ generated can be viewed as a sample from the posterior, and it is not necessary to use the importance weights to estimate posterior quantities. Above, the posterior mean is estimated with a simple average of the retained values. The disadvantages of the Markov chain are the convergence and dependence issues. Here the normal approximation is a fairly good approximation to the posterior, so convergence should be very fast, and the average of the entire generated sequence was used in calculating the means.

The functions `acor()` and `varest()` below are the same as in the previous section.

```
> for(k in 1:5) print(apply(Hi,1,acor,k=k))
[1] 0.1656941 0.1650504 0.1626723 0.1485015 0.1478927 0.1417715
[1] -0.0001425621  0.0041872393 -0.0128523335  0.0276380796  0.0164524764
[6] -0.0064613996
[1]  0.003057361  0.003440434 -0.005240666  0.019521594  0.007019761
[6] -0.009516269
[1] -0.013895209  0.011933642  0.001239134  0.017269755  0.004064773
[6] -0.010002931
[1] -0.001681361 -0.009623145 -0.001216035 -0.012933311  0.005685681
[6] -0.003769621
> for (i in 1:6) print(varest(Hi[i,],50))
      mean          se        rho
 0.8527268 0.002016549 0.04118475
     mean          se       rho
 -0.78103 0.002453827 0.1459101
```

```
     mean          se        rho
 1.171967 0.002382343 0.01111419
       mean          se        rho
 -0.4104312 0.002299085 0.07305585
      mean          se        rho
 0.5518204 0.002373905 0.01112616
       mean          se        rho
 -0.0425595 0.001710989 -0.1573334
```

The autocorrelations at lags greater than 1 are very small. The standard errors for the posterior means here are a little smaller than from the standard importance sampling algorithm, and overall about the same as for the control variate corrected importance sampling. The differences in the means from the two approaches are larger than would be expected from the standard errors for some of the parameters. The reason for this is not clear.

Quantiles of the posterior distribution are even easier to estimate here than with importance sampling, since all points are given equal weight, so quantiles can be estimated directly from the order statistics.

```
> apply(Hi,1,quantile,probs=c(.025,.5,.975))
          [,1]       [,2]      [,3]        [,4]      [,5]        [,6]
[1,] 0.4911447 -1.1681642 0.7553135 -0.76811032 0.1625419 -0.41527798
[2,] 0.8533511 -0.7790818 1.1640513 -0.40996736 0.5493667 -0.04223188
[3,] 1.2245509 -0.4138491 1.6071968 -0.05675237 0.9480288  0.32381396
```

9.7 Rejection Chain Sampling

In this section the rejection chain variation on the Metropolis-Hastings algorithm will be implemented (see Section 9.2.3).

As in Section 8.3, denote the mean and covariance matrix of the normal approximation to the posterior by $\hat{\beta}$ and $I^{-1}(\hat{\beta})$. The proposal for the function $g(\beta)$ to dominate $L(\beta)\pi(\beta)$ is the density of the normal distribution with mean $\hat{\beta}$ and covariance matrix $1.2I^{-1}(\hat{\beta})$, rescaled so that $L(\hat{\beta})\pi(\hat{\beta})/g(\hat{\beta}) = 1$. This function was deliberately chosen so that it would not dominate everywhere; using a larger multiplier of the variance would lead to fewer rejections of trial values in the M-H algorithm, but perhaps require generating more candidate points within each rejection step. Below 10,000 candidate points for the initial rejection sampling are generated first. Of these 3,846 are rejected, leaving 6,154 candidate points for the M-H rejection sampling chain. Apart from using rejection sampling within each step, the steps in the Markov chain are the same as in the independence chain, except the sampling density is the pointwise minimum of the posterior and the possible dominating function. Functions and quantities not explicitly defined are the same as before.

```
> ## Metropolis-Hastings rejection chain sampling
> nt <- 10000
> Z <- matrix(rnorm(nt*np),nrow=np)
```

```
> # normal approximation to the posterior, with variance inflated by 1.2
> H <- (sqrt(1.2)*B) %*% Z + bhat  # B as before
>
> # calculate ratios for rejection sampling
> w <- rep(1,np) %*% (Z*Z)
> w2 <- apply(H,2,flog)  # flog as before
> # densities not normalized
> f <- exp(flog(bhat)-w2) # proportional to target density
> g <- exp(-w/2)      # 'dominating' function
> ratio <- f/g
> summary(ratio)
      Min. 1st Qu. Median   Mean 3rd Qu.  Max.
 0.0001585  0.4579 0.6527 0.6166  0.7943 4.479
> # candidates for Markov-Chain
> sub <- ratio > runif(nt)
> H <- H[,sub]
> dim(H)
[1]    6 6154
> f <- f[sub]
> q <- pmin(f,g[sub]) # proportional to actual sampling density
> # Markov-Chain
> Hi <- H
> ui <- runif(ncol(H))
> l <- 0 # count # times candidate value rejected
> ll <- 0 # count # times transition probability < 1
> for (j in 2:ncol(H)) {
+   alpha <- f[j]*q[j-1]/(f[j-1]*q[j]) # transition probability
+   if (alpha<1) {
+     ll <- ll+1
+     if (ui[j] >= alpha) {# reject new value and keep old
+       l <- l+1
+       Hi[,j] <- Hi[,j-1]
+       f[j] <- f[j-1]
+       q[j] <- q[j-1]
+     }
+   }
+ }
>
> l
[1] 31
> ll
[1] 179
>
> # Estimated posterior means
> apply(Hi,1,mean)
[1]  0.86187776 -0.79164665  1.19414352 -0.40826093  0.56016282 -0.04155914
```

```
> apply(Hi,1,acor,k=1)
[1] 0.010092943 0.009675866 0.055761542 0.008355903 0.020832825 0.011807339
> apply(Hi,1,acor,k=2)
[1] 0.001331318 0.013919084 0.016189311 0.020072205 0.020893868 0.004500746
> apply(Hi,1,acor,k=3)
[1] -0.0009875959 -0.0127251232  0.0078220516 -0.0106785176 -0.0045123510
[6]  0.0105690761
> for (i in 1:6) print(varest(Hi[i,],50))
      mean          se        rho
 0.8619278 0.002511336 0.02710478
       mean          se        rho
 -0.7915666 0.002672721 0.06601914
     mean         se        rho
 1.194157 0.00337172 0.03359866
       mean          se         rho
 -0.4082298 0.002525831 -0.02808374
      mean          se      rho
 0.5600244 0.003036515 0.161695
        mean          se         rho
 -0.04165937 0.002600647 -0.02406042
```

The tradeoff between this algorithm and the independence chain is that of the initial 10,000 normal vectors generated here, only 6,154 were accepted by the rejection sampling screen. However, for this reduced sample, all but 31 points were accepted by the M-H algorithm, so even the lag 1 autocorrelations were very small, whereas in the independence chain, 1,275 of the 10,000 trial values were rejected by the M-H algorithm, leading to more dependence in the sequence. The standard errors of the posterior means were a little smaller for the independence chain. The standard errors here are mostly a little smaller than for the standard importance sampling algorithm.

9.8 Exercises

Exercise 9.1 Show that f satisfies the reversibility condition (9.1) for the M-H transition kernel (9.3).

Exercise 9.2 Verify that f is the stationary distribution of the rejection sampling chain described in Section 9.2.3.

Exercise 9.3 Consider the same model as in Exercise 8.2, but now suppose $f(\cdot)$ is the standard normal density. Also, suppose that c is not known, but instead has a $\Gamma(.01, .01)$ prior (sometimes called a hyperprior, since it is a prior on a parameter of a prior distribution). (The prior on ν is again $\Gamma(.1, .1)$.)

1. Give the conditional distributions $[\beta|\nu, Y, c]$, $[\nu|\beta, Y, c]$, and $[c|\nu, Y, \beta]$.

2. Using the same data as in Exercise 8.2, implement a Gibbs sampling algorithm to sample from the joint posterior distribution. Estimate the posterior mean (and estimate the precision of the estimated mean) for each of the parameters. Also estimate the .025, .5 and .975 quantiles of the posterior distribution of each of the parameters.

9.9 References

Chen M-H, Shao Q-M and Ibrahim JG (2000). *Monte Carlo Methods in Bayesian Computation.* Springer.

Cowles MK and Carlin BP (1996). Markov chain Monte Carlo convergence diagnostics: a comparative review. *Journal of the American Statistical Association* 91:883–904.

Gelfand AE and Smith AFM (1990). Sampling-based approaches to calculating marginal densities. *Journal of the American Statistical Association* 85:398–409.

Gelfand AE, Hills SE, Racine-Poon A, Smith AFM (1990). Illustration of Bayesian inference in normal data models using Gibbs sampling. *Journal of the American Statistical Association* 85:972–985.

Gelman A and Rubin DB (1992). Inference from iterative simulation using multiple sequences (with discussion). *Statistical Science* 7:457–511.

Geman S and Geman D (1984). Stochastic relaxation, Gibbs distributions and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 6:721-741.

Hastings WK (1970). Monte Carlo sampling methods using Markov chains and their application. *Biometrika* 57:97–109.

Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller AH, and Teller E (1953). Equations of state calculations by fast computing machines. *Journal of Chemical Physics* 21:1087–1092.

Tanner MA and Wong WH (1987). The calculation of posterior distributions by data augmentation (with discussion). *Journal of the American Statistical Association* 82:528–550.

Tierney L (1994). Markov Chains for Exploring Posterior Distributions (with Discussion). *Annals of Statistics* 22:1701–1762.

Chapter 10

Bootstrap Methods

10.1 Variance Estimation

Let $\mathbf{X} = (X_1, \ldots, X_n)$ be a sample and θ a parameter, and suppose θ is estimated by $\hat{\theta} = \hat{\theta}(X_1, \ldots, X_n)$. Once $\hat{\theta}$ is computed, the next step towards drawing inferences about θ would often be estimating the variance of $\hat{\theta}$. The exact variance of $\hat{\theta}$ is defined by

$$\mathrm{E}[\hat{\theta} - \mathrm{E}(\hat{\theta})]^2,$$

where the expectations are over the true sampling distribution of $\mathbf{X}$. There are generally two problems with computing the exact variance. First, the true distribution is usually unknown. Second, for all but the simplest statistics, direct computation of the exact moments is prohibitively complex. Because of the difficulties evaluating the exact variance, large sample approximations are often used.

The bootstrap is a general approach to frequentist inference. The underlying idea of the bootstrap is to first use the sample to estimate the unknown sampling distribution of the data. Then this estimated distribution is used in place of the unknown true distribution in calculating quantities needed for drawing inferences on θ. Exact calculations are used where possible, but often simulations, drawing samples from the estimated distribution, are required to approximate the quantities of interest. The estimate of the sampling distribution may be either parametric or nonparametric.

Example 10.1 Suppose $X_1, \ldots, X_n$ are iid, $\theta = \mathrm{E}(X_i)$, and $\hat{\theta} = \overline{X} = \sum_{i=1}^n X_i/n$, the sample mean. Then

$$\mathrm{Var}(\hat{\theta}) = \mathrm{Var}(X_i)/n,$$

and the problem of estimating the variance of $\hat{\theta}$ is reduced to estimating the variance of X_i.

To apply the bootstrap, the sampling distribution needs to be estimated. Since the data are iid, only the (common) marginal distribution of the X_i needs to be estimated. If a parametric model has been specified, then a parametric estimator can be used. For example, suppose that the data follow an exponential model, with $P(X_i > x) = \exp(-\lambda x) = 1 - F(x; \lambda)$, where $\lambda > 0$ is an unknown parameter. Then the CDF $F(x; \lambda)$ can be estimated by substituting the MLE $\hat{\lambda} = 1/\overline{X}$

for λ. The bootstrap approach uses the true variance under this estimated distribution as an estimate of the variance under the true distribution. Since the estimated distribution is exponential with rate $\hat{\lambda}$, and the variance of an exponential with rate λ is λ^{-2}, the bootstrap estimate of the variance of $\hat{\theta}$ is $\overline{X}^2/n$.

If a parametric model is not assumed, then with iid data a nonparametric estimate of the common marginal distribution of the X_i is given by the empirical CDF

$$\hat{F}(x) = n^{-1} \sum_{i=1}^{n} I(X_i \le x).$$

This distribution places probability mass n^{-1} on each of the observed data points X_i. Thus if X^* has the distribution with CDF $\hat{F}(\cdot)$, then for any function $q(\cdot)$,

$$\mathrm{E}[q(X^*)] = \int q(x)\, d\hat{F}(x) = \sum_{i=1}^{n} q(X_i)/n.$$

In particular,

$$\mathrm{E}[X^*] = \sum_{i=1}^{n} X_i/n = \overline{X}$$

and

$$\mathrm{E}[(X^* - \mathrm{E}(X^*))^2] = n^{-1} \sum_{i=1}^{n} (X_i - \overline{X})^2$$

(the last expression is the usual sample variance multiplied by $(n-1)/n$). Thus the nonparametric bootstrap estimate of the variance of the sample mean is

$$n^{-2} \sum_{i=1}^{n} (X_i - \overline{X})^2.$$

□

In most settings the exact variance of $\hat{\theta}$ under the bootstrap distribution cannot be calculated directly. In these settings, simulations can be used to approximate the exact variance under the bootstrap distribution. In general, suppose $F^*(x_1, \ldots, x_n)$ is the estimated CDF of the *sample* $\mathbf{X}$, and let $\mathbf{X}^{*(1)}, \mathbf{X}^{*(2)}, \ldots, \mathbf{X}^{*(B)}$ be independent simulated samples, each with distribution F^*. Also, let $\mathbf{X}^*$ be a generic random sample with distribution F^*, and write $\hat{\theta}^* = \hat{\theta}(\mathbf{X}^*)$. Then

$$\tilde{\theta}^* = B^{-1} \sum_{b=1}^{B} \hat{\theta}(\mathbf{X}^{*(b)})$$

is the simulation estimate of $\mathrm{E}(\hat{\theta}^*)$, and

$$v^* = (B-1)^{-1} \sum_{b=1}^{B} [\hat{\theta}(\mathbf{X}^{*(b)}) - \tilde{\theta}^*]^2 \tag{10.1}$$

is the simulation estimate of $\mathrm{Var}(\hat{\theta}^*)$ (and thus a bootstrap estimate of $\mathrm{Var}(\hat{\theta})$). The simulated samples $\mathbf{X}^{*(b)}$ are often called resamples. This terminology comes from the nonparametric version, where each component of each $\mathbf{X}^{*(b)}$ is one of the observed values in the original sample.

Example 10.2 Suppose the sample consists of bivariate observations (X_i, Y_i), $i = 1, \ldots, n$, where the pairs are iid. Let $F(x, y)$ be the bivariate CDF of (X_i, Y_i). Consider estimating the correlation coefficient

$$\rho = \mathrm{Cov}(X_i, Y_i)/[\mathrm{Var}(X_i)\mathrm{Var}(Y_i)]^{1/2}.$$

The usual moment estimator is

$$\hat{\rho} = \frac{\sum_{i=1}^n (X_i - \overline{X})(Y_i - \overline{Y})}{\left[\sum_{i=1}^n (X_i - \overline{X})^2 \sum_{i=1}^n (Y_i - \overline{Y})^2\right]^{1/2}}.$$

If F is a bivariate normal distribution, then the Fisher Z transform can be applied, giving that

$$\frac{1}{2}\log\left(\frac{1+\hat{\rho}}{1-\hat{\rho}}\right) \dot{\sim} \mathrm{N}\left(\frac{1}{2}\log\left(\frac{1+\rho}{1-\rho}\right), \frac{1}{n-3}\right).$$

From the delta method, it then follows that $\mathrm{Var}(\hat{\rho}) \doteq (1-\rho^2)^2/(n-3)$. For other distributions, large sample approximations to $\mathrm{Var}(\hat{\rho})$ would generally be substantially more complicated.

Using simulations to approximate the nonparametric bootstrap estimate of $\mathrm{Var}(\hat{\rho})$ is straightforward. The empirical CDF is

$$\hat{F}(x, y) = n^{-1} \sum_{i=1}^n I(X_i \le x, Y_i \le y),$$

which again places mass n^{-1} on each of the observed pairs (X_i, Y_i).

One way to draw a sample $\mathbf{X}^{*(b)}$ from the estimated distribution is to first generate n uniforms $u_1^{(b)}, \ldots, u_n^{(b)}$, and set $i_j = k$ if $k - 1 < nu_j^{(b)} \le k$. Then the resample is

$$\mathbf{X}^{*(b)} = \{(X_{i_1}, Y_{i_1}), \ldots, (X_{i_n}, Y_{i_n})\}.$$

In Splus the indices i_j can be generated directly by using the `sample()` command (with the option `replace=T`).

Figure 10.1 gives a small example data set. The commands used to generate the data, and parametric and nonparametric bootstrap estimates of the variance, are given in the Splus code below.

```
> # generate first data set
> print(.Random.seed)
 [1] 41 58 36 53 44  3 37 11 61 34 14  3
> v <- matrix(c(1,.75,.75,1),2,2)
> vc <- chol(v)
> u <- matrix(rnorm(100),ncol=2) %*% vc + 5
> var(u)
          [,1]      [,2]
[1,] 0.9972487 0.8412334
[2,] 0.8412334 1.2056130
```

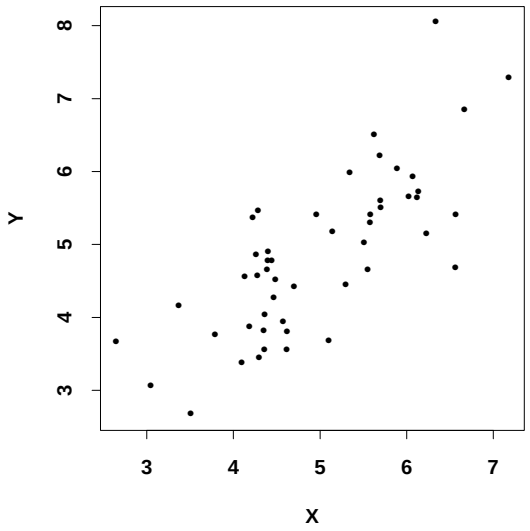

Figure 10.1: First data set, correlation coefficient example. $\hat{\rho} = .77$.

```
>
> postscript(file = "../boot/fig1.ps", horizontal = T, pointsize = 22,
+         ps.preamble = ps.pre)
> par(font=5,lwd=2,pty='s',mar=c(4.1,4.1,0.1,2.1))
> plot(u[,1],u[,2],xlab='X',ylab='Y')
> dev.off()
> r <- cor(u[,1],u[,2])
> r
[1] 0.7672038
> (1-r^2)/(nrow(u)-3)
[1] 0.008753155
>
> normsim <- function(B,n,vc,ff) {
+ # the correlation coefficient is invariant to location shifts, so
+ # don't need the mean
+   w <- t(vc) %*% matrix(rnorm(2*B*n),nrow=2)
+   w <- split(w,rep(1:B,rep((2*n),B)))
+   out <- unlist(lapply(w,ff))
+   print(c(var(out),sqrt(var((out-mean(out))^2)/B)))
+   out
+ }
> ff <- function(w) {w <- matrix(w,nrow=2); cor(w[1,],w[2,])}
> # true variance
> out <- normsim(500,nrow(u),vc,ff)
[1] 0.0040219830 0.0003057766
> # parametric bootstrap
> vce <- chol(var(u))
> vce
          [,1]      [,2]
```

```
[1,] 0.9986234 0.8423930
[2,] 0.0000000 0.7042635
attr(, "rank"):
[1] 2
> unix.time(out <- normsim(500,nrow(u),vce,ff))
[1] 0.0035354108 0.0002697988
[1] 1.360001 0.000000 2.000000 0.000000 0.000000
>
> # nonparametric bootstrap
> npboot <- function(B,u,ff) {
+   index <- sample(nrow(u),B*nrow(u),replace=T)
+   index <- split(index,rep(1:B,nrow(u)))
+   out <- unlist(lapply(index,ff,data=u))
+   print(c(var(out),sqrt(var((out-mean(out))^2)/B)))
+   out
+ }
> ff2 <- function(index,data) cor(data[index,1],data[index,2])
> unix.time(out <- npboot(500,u,ff2))
[1] 0.0026423798 0.0001784584
[1] 1.020004 0.000000 1.000000 0.000000 0.000000
> # compare lapply() and for()
> npboot.2 <- function(B,u,ff) {
+   out <- rep(0,B)
+   for (i in 1:B) {
+     index <- sample(nrow(u),nrow(u),replace=T)
+     out[i] <- ff(index,u)
+   }
+   print(c(var(out),sqrt(var((out-mean(out))^2)/B)))
+   out
+ }
> unix.time(out <- npboot.2(500,u,ff2))
[1] 0.0027813029 0.0001733102
[1] 1.379997 0.000000 1.000000 0.000000 0.000000
```

The difference between the true variance calculation and the parametric bootstrap is that in the true variance the covariance matrix used to generate the original data is used in the simulation, while in the parametric bootstrap the estimated covariance matrix from the original sample is used in generating the new samples. The number of samples is not large enough to give a highly precise estimate of the true variance, but as will be discussed later, 500 bootstrap resamples may be more than adequate for estimating the variance, since the sampling error in the simulation is then usually smaller than the error in bootstrap estimate of the true distribution. The parametric bootstrap appears to be a little smaller than the true variance in this example, and the nonparametric bootstrap even smaller. However, nothing can be concluded about the bias of the bootstrap variance estimate from a single sample. Investigating the bias would require generating many samples, and computing the bootstrap variance estimate on each sample. The large sample variance approximation appears to be less accurate here.

Note the use of `lapply()` and `split()` to avoid explicit use of `for()` loops in the functions above. To compare the time used, `npboot.2()` performed the same calculations as `npboot`, but using explicit `for()` loops. The version with `lapply()` was slightly faster in this example. Also, on a Sparc Ultra 80, only about a second was needed for each of the bootstrap runs.

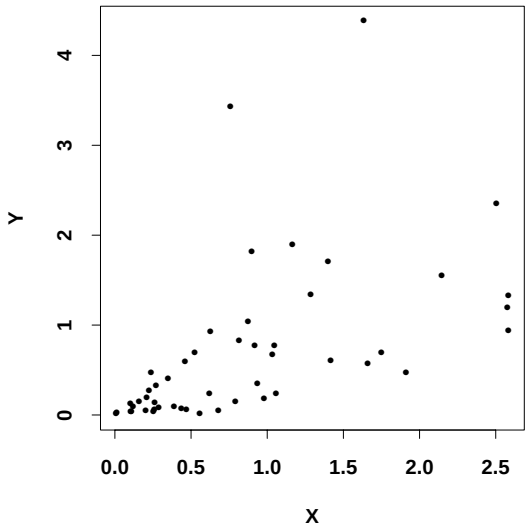

Figure 10.2: Second data set, correlation coefficient example. $\hat{\rho} = .54$.

Figure 10.2 gives a second data set, which is not very close to normal. The commands used to generate the data, and parametric and nonparametric bootstrap estimates of the variance, are given in the Splus code below.

```
> # generate second data set
> print(.Random.seed)
 [1] 49 42 19 60  9  3 25 57  7 25 55  3
> a <- rexp(50)
> a <- cbind(a,rexp(50,1/a))
> var(a)
          a
a 0.5376905 0.3485758
  0.3485758 0.7806216
>
> postscript(file = "../boot/fig2.ps", horizontal = T, pointsize = 22,
+         ps.preamble = ps.pre)
> par(font=5,lwd=2,pty='s',mar=c(4.1,4.1,0.1,2.1))
> plot(a[,1],a[,2],xlab='X',ylab='Y')
> dev.off()
> r <- cor(a[,1],a[,2])
> r
[1] 0.5380353
> (1-r^2)/(nrow(a)-3)
[1] 0.0151174
>
```

```
> # true variance
> B <- 500
> w <- rexp(B*nrow(a))
> w <- rbind(w,rexp(B*nrow(a),1/w))
> w <- split(w,rep(1:B,rep((2*nrow(a)),B)))
> ff <- function(w) {w <- matrix(w,nrow=2); cor(w[1,],w[2,])}
> out <- unlist(lapply(w,ff))
> var(out)
[1] 0.0157834
> sqrt(var((out-mean(out))^2)/B)
[1] 0.0009073865
>
> # parametric bootstrap--wrong distribution
> vce <- chol(var(a))
> vce
          a
a 0.7332738 0.4753692
  0.0000000 0.7447454
attr(, "rank"):
[1] 2
> unix.time(out <- normsim(500,nrow(a),vce,ff))
[1] 0.0105435672 0.0006532394
[1] 1.34999847 0.02999973 2.00000000 0.00000000 0.00000000
> # nonparametric bootstrap
> unix.time(out <- npboot(500,a,ff2))
[1] 0.008679424 0.000570964
[1] 1.050003 0.000000 1.000000 0.000000 0.000000
```

For these data, the true variance is larger than in the previous case. The large sample variance, which is only approximately valid for normal data, is again not very accurate here. The parametric bootstrap, using the wrong (normal) parametric distribution, happens to be fairly close to the true variance. Since the wrong distribution was used, this estimator is not even asymptotically valid, and in general it cannot be relied on. The nonparametric bootstrap is fairly close to the true variance. (It should be noted, though, that the bootstrap always involves sampling from the "wrong" distribution, since only an estimate of the true sampling distribution is available.) □

10.1.1 Regression Models

Consider the linear regression model

$$y_i = x_i'\beta + \epsilon_i, \quad i = 1, \ldots, n,$$

where x_i is a p-vector of covariates (usually including a constant term), β is a vector of unknown parameters, and the errors ϵ_i are iid with mean 0 and finite variance σ^2. The observed data consist of $(y_i, x_i')'$, $i = 1, \ldots, n$. The regression coefficients β can be consistently estimated using

the least squares estimator

$$\hat{\beta} = \left(\sum_i x_i x_i' \right)^{-1} \sum_i x_i y_i,$$

or any of a variety of more robust estimators. In large samples, $\hat{\beta}$ is approximately normal with mean β and variance $\sigma^2 \left(\sum_i x_i x_i' \right)^{-1}$. σ^2 can be estimated by the usual residual mean square

$$\hat{\sigma}^2 = \sum_{i=1}^{n} (y_i - x_i' \hat{\beta})^2 / (n - p).$$

As in the iid case, both parametric and nonparametric bootstrap methods can be applied. However, here there are two distinct ways of proceeding. In one, the x_i are kept fixed, and the distribution of the y_i estimated by estimating the distribution of the ϵ_i and the value of β. In the other, the joint distribution of $(y_i, x_i')'$ is estimated. In the first approach, inferences will be conditional on the observed covariate values. This is particularly appropriate if the covariate values were fixed by design, but is also often appropriate when covariates are random, since usually the covariate values are ancillary. In the second approach inferences are averaged over the distribution of possible covariate values. For the least squares estimator, the two approaches usually lead to the same large sample inferences.

If parametric distributions are specified, either approach is straightforward. The distributions are estimated using standard methods, such as maximum likelihood or more robust methods, and inferences made under the estimated sampling distribution, using simulations if necessary.

The nonparametric bootstrap is easier to implement in the second (unconditional) approach. The joint distribution of $(y, x')'$ can be estimated with the empirical distribution of the observed data points, which again places probability $1/n$ on each of the observed data points. In simulations, observations can be selected for the bootstrap resamples by sampling integers with replacement from the set $\{1, \ldots, n\}$. This approach does not make use of the assumption that the errors are iid, and in fact this method remains valid even if the error distribution depends on x_i. It also does not require that the regression model be correctly specified (to give valid resamples, in the sense that the estimated distribution is consistent, although validity of the inferences drawn may still require that the model be correctly specified).

In the first (conditional) approach, an estimate of the error distribution is needed. One way to estimate this distribution is with the empirical distribution of the estimated residuals $\hat{e}_i = y_i - x_i' \hat{\beta}$. This distribution places mass $1/n$ on each of the observed $\hat{e}_i$. In simulations, to generate a bootstrap resample, errors ϵ_i^* are drawn with equal probability with replacement from the set $\{\hat{e}_1, \ldots, \hat{e}_n\}$. The x_i are kept fixed, and the responses are $y_i^* = x_i' \hat{\beta} + \epsilon_i^*$. The bootstrap resample then is $(y_i^*, x_i')'$, $i = 1, \ldots, n$. Validity of this method requires both that the form of the regression model be correctly specified, and that the errors be iid. It is thus not a true nonparametric procedure, and is more properly referred to as semiparametric.

For the least-squares estimate $\hat{\beta}$, the semiparametric bootstrap procedure gives

$$\mathrm{Var}(\hat{\beta}^*) = \frac{n - p}{n} \hat{\sigma}^2 \left(\sum_i x_i x_i' \right)^{-1},$$

the usual estimator times $(n-p)/n$.

There is another way to approach conditional nonparametric resampling. Instead of relying on the validity of the hypothesized model, a nonparametric estimate of the distribution of $y|x$ can be used. In its simplest form, the collection of nearest neighbors indexed by $J(i) = \{j : \|x_i - x_j\| < h_i\}$ could be used, with y_i^* sampled with replacement from $\{y_j : j \in J(i)\}$. Points within this set could be weighted based on the magnitudes of $\|x_i - x_j\|$. The window size h_i could be chosen in a variety of ways, such as to include a certain number of points. This approach requires minimal assumptions, and if the $h_i \to 0$ at an appropriate rate as $n \to \infty$, then the estimated distribution should be consistent. However, the estimated distribution can have substantial bias for finite n, and the details of implementation, such as the values of the h_i and the weights used within a neighborhood are fairly arbitrary, but can effect the results. A refinement of this procedure is to use a local semiparametric model, for example, fitting a linear model to the points in a neighborhood, and resampling from the residuals of the local linear fit.

The techniques discussed above can generally be applied to nonlinear regression models with independent observations. Applying the nonparametric bootstrap to more general dependent data problems, such as time series and clustered data, is more difficult. If enough structure is assumed, semiparametric sampling is still straightforward. For example, in first order autoregression model, iid residuals can be estimated, and resamples drawn from their empirical distribution. Methods for fully nonparametric resampling of time series data have been proposed, such as the moving blocks bootstrap (which resamples blocks of consecutive values in the time series), but it is not clear how reliable these methods are in general problems. For clustered data, the obvious nonparametric approach is to resample clusters from the empirical distribution of the clusters (possibly also resampling subjects within clusters, depending on the nature of the inference problem). This approach may not perform well, unless the number of clusters is fairly large, though.

10.1.2 How many resamples?

The variance of the bootstrap simulation variance estimator v^*, defined by (10.1), can be decomposed

$$\mathrm{Var}(v^*) = \mathrm{Var}_{\mathbf{X}}[\mathrm{E}(v^*|\mathbf{X})] + \mathrm{E}_{\mathbf{X}}[\mathrm{Var}(v^*|\mathbf{X})]. \tag{10.2}$$

In both terms on the right, the inner moments are over the distribution of the bootstrap resamples, conditional on the original sample $\mathbf{X}$, and the outer moments are over the distribution of $\mathbf{X}$. Since $\mathrm{E}(v^*|\mathbf{X})$ is the value v^* converges to as $B \to \infty$, the first term is the inherent variability from using an estimate of the true distribution. The second term reflects the variability from using a finite number of resamples B to approximate the bootstrap variance.

For the second term in (10.2), from the discussion of the variance of a sample variance estimator on page 18 of the Basic Simulation Methodology handout, it follows that

$$\begin{aligned} \mathrm{Var}(v^*|\mathbf{X}) &\doteq \mathrm{Var}\{[\hat{\theta}^* - \mathrm{E}(\hat{\theta}^*|\mathbf{X})]^2|\mathbf{X}\}/B \\ &= \mathrm{E}\{[\hat{\theta}^* - \mathrm{E}(\hat{\theta}^*|\mathbf{X})]^4|\mathbf{X}\}/B - \mathrm{E}\{[\hat{\theta}^* - \mathrm{E}(\hat{\theta}^*|\mathbf{X})]^2|\mathbf{X}\}^2/B \\ &= \sigma^{*4}[\kappa(\hat{\theta}^*) + 2]/B \end{aligned} \tag{10.3}$$

where $\sigma^{*2} = \mathrm{Var}[\hat{\theta}^*|\mathbf{X}]$, and $\kappa(\cdot)$ is the standardized kurtosis of the distribution of its argument

(here the conditional distribution of $\hat{\theta}^*$, given $\mathbf{X}$). (In general, the standardized kurtosis of a random variable W is $\mathrm{E}[W - \mathrm{E}(W)]^4/\mathrm{Var}(W)^2 - 3$. The value 3 is subtracted so the standardized kurtosis of a normal distribution is 0.)

To go further requires knowledge of the statistic and the sampling distribution of $\mathbf{X}$. The simplest case is when $\hat{\theta} = \overline{X}$, and the original data are an iid sample $X_1, \ldots, X_n$. In this case $\hat{\theta}^* = \sum_{i=1}^n X_i^*/n$, $\mathrm{Var}(\hat{\theta}^*) = \mathrm{Var}(X_i^*)/n$, and $\kappa(\hat{\theta}^*) = \kappa(X_i^*)/n$. Thus in this case

$$\mathrm{E}_{\mathbf{X}}[\mathrm{Var}(v^*|\mathbf{X})] \doteq \mathrm{E}_{\mathbf{X}}\left[\frac{1}{B}\frac{\mathrm{Var}(X_i^*)^2}{n^2}\left(\frac{\kappa(X_i^*)}{n} + 2\right)\right] \doteq \frac{1}{B}\frac{\mathrm{Var}(X_i)^2}{n^2}\left(\frac{\kappa(X_i)}{n} + 2\right), \tag{10.4}$$

since for large n the variance and kurtosis of the bootstrap distribution will be converging to those of the true distribution. Also, from Example 10.1, in this case

$$\mathrm{E}(v^*|\mathbf{X}) = n^{-2}\sum_{i=1}^n (X_i - \overline{X})^2. \tag{10.5}$$

Applying the argument leading to (10.4) to (10.5) gives that

$$\mathrm{Var}_{\mathbf{X}}[\mathrm{E}(v^*|\mathbf{X})] \doteq \frac{\mathrm{Var}(X_i)^2}{n^3}\left[\kappa(X_i) + 2\right]. \tag{10.6}$$

Combining (10.4) and (10.6) gives that

$$\mathrm{Var}(v^*) \doteq \frac{\mathrm{Var}(X_i)^2}{n^2}\left(\frac{\kappa(X_i)}{nB} + \frac{2}{B} + \frac{\kappa(X_i) + 2}{n}\right). \tag{10.7}$$

Only the terms involving B are affected by the number of bootstrap resamples. If B is substantially larger than n, then clearly little would be gained by drawing additional samples, since the other terms are already dominant. Since $\mathrm{E}(v^*) \doteq \mathrm{Var}(X_i)/n$, the coefficient of variation of v^* is approximately

$$\left(\frac{\kappa(X_i)}{nB} + \frac{2}{B} + \frac{\kappa(X_i) + 2}{n}\right)^{1/2} \tag{10.8}$$

For most reasonable values of $\kappa(X_i)$ (in particular, for the normal distribution where $\kappa(X_i) = 0$), considering (10.8) as a function of B, the rate of change is flattening by $B = 50$, and there is generally little gained by increasing B beyond about 200. With large n, it is possible to get more precise estimates with larger values of B, though.

In applications to percentile estimation, such as will be discussed in connection with hypothesis tests and confidence intervals, a substantially larger number of resamples, at least an order of magnitude larger, would often be appropriate.

10.2 Bootstrap Bias Correction

If $\hat{\theta}$ is an estimator of a parameter θ, then the bias of $\hat{\theta}$ is

$$\mathrm{E}_\theta(\hat{\theta}) - \theta.$$

Let $\theta(F)$ denote the value of θ corresponding to an arbitrary distribution F (it may not always be possible to define $\theta(F)$ sensibly, for example, if θ is the shape parameter of a gamma distribution,

and F is the empirical CDF of an observed sample). Also, let F^* be the estimated distribution used for the bootstrap, and let E^* denote the expectation under F^*. In many bootstrap problems, it turns out that $\theta(F^*) = \hat{\theta}$.

The bootstrap estimate of bias is defined to be

$$\mathrm{bias}(\hat{\theta}^*) = \mathrm{E}^*(\hat{\theta}^*) - \theta(F^*), \tag{10.9}$$

and the bias corrected estimator is

$$\hat{\theta} - \mathrm{bias}(\hat{\theta}^*) = \hat{\theta} + \theta(F^*) - \mathrm{E}^*(\hat{\theta}^*).$$

If $\theta(F^*) = \hat{\theta}$, then the bias corrected estimator is

$$2\hat{\theta} - \mathrm{E}^*(\hat{\theta}^*).$$

If simulations are used to estimate $\mathrm{E}^*(\hat{\theta}^*)$, then in the formulas for the bootstrap estimate of bias and the bias corrected estimator, $\mathrm{E}^*(\hat{\theta}^*)$ would be replaced by $B^{-1}\sum_{b=1}^{B}\hat{\theta}^{*(b)}$.

Recall the control variates method from Section 7.3.1. Control variates and closely related adjustment can often be used to greatly improve the efficiency of bootstrap bias estimators, as described in the following example.

Example 10.3 Bias Correction for Ratio Estimators. Suppose pairs $(y_i, z_i)'$, $i = 1, \ldots, n$, are iid from a distribution with mean $(\mu_y, \mu_z)'$. Suppose the parameter of interest is $\theta = \mu_y/\mu_z$, and $\hat{\theta} = \overline{y}/\overline{z}$. Define the distribution F^* of the bootstrap resamples by taking the pairs within a resample to be iid, with each having the empirical distribution $\hat{F}$, which puts mass $1/n$ on each of the observed data pairs. Then $\theta(F^*) = \overline{y}/\overline{z} = \hat{\theta}$.

Linear approximations to statistics often give useful control variates for bias estimation. A first order Taylor series expansion of u/v about $(u, v)' = (\mu_y, \mu_z)'$ gives

$$u/v \doteq \mu_y/\mu_z + (u - \mu_y)/\mu_z - \mu_y(v - \mu_z)/\mu_z^2.$$

Substituting $\overline{y}$ for u and $\overline{z}$ for v gives

$$\hat{\theta} \doteq \theta + (\overline{y} - \mu_y)/\mu_z - \mu_y(\overline{z} - \mu_z)/\mu_z^2.$$

For the estimator $\hat{\theta}^*$ defined from bootstrap resamples, the true means are $(\overline{y}, \overline{z})'$, so the corresponding expansion is

$$\hat{\theta}^* \doteq \hat{\theta} + (\overline{y}^* - \overline{y})/\overline{z} - \overline{y}(\overline{z}^* - \overline{z})/\overline{z}^2.$$

The quantity

$$(\overline{y}^* - \overline{y})/\overline{z} - \overline{y}(\overline{z}^* - \overline{z})/\overline{z}^2$$

has exact mean 0 (in the bootstrap distribution, conditional on the original sample), and is a first order approximation to $\hat{\theta}^* - \hat{\theta}$, and thus is a good candidate for a control variate. The bootstrap estimate of bias incorporating this control variate, based on B resamples, is

$$B^{-1}\sum_{b=1}^{B}\{\hat{\theta}^{*(b)} - (\overline{y}^{*(b)} - \overline{y})/\overline{z} + \overline{y}(\overline{z}^{*(b)} - \overline{z})/\overline{z}^2\} - \hat{\theta} \tag{10.10}$$

(the control variate adjustment generally multiplies the control variate by a constant, but here to first order the optimal constant is 1). An additional modification can be made. Again using a first order expansion of u/v,

$$\frac{\sum_{b=1}^B \overline{y}^{*(b)}}{\sum_{b=1}^B \overline{z}^{*(b)}} \doteq \hat{\theta} + \frac{1}{\overline{z}}\left(\frac{1}{B}\sum_{b=1}^B \overline{y}^{*(b)} - \overline{y}\right) - \frac{\overline{y}}{\overline{z}^2}\left(\frac{1}{B}\sum_{b=1}^B \overline{z}^{*(b)} - \overline{z}\right).$$

Thus to first order, (10.10) equals

$$B^{-1}\sum_{b=1}^B \hat{\theta}^{*(b)} - \frac{\sum_{b=1}^B \overline{y}^{*(b)}}{\sum_{b=1}^B \overline{z}^{*(b)}}. \tag{10.11}$$

This is the version that is usually used. Either (10.10) or (10.11) will often improve the precision of a bias estimator by 1 to 2 orders of magnitude.

Formula (10.10) has an interesting interpretation. Let $N_j^{*(b)}$ be the number of times the point $(y_j, z_j)'$ appears in the bth bootstrap resample, so $\overline{y}^{*(b)} = n^{-1}\sum_{j=1}^n N_j^{*(b)} y_j$. Then

$$\frac{1}{B}\sum_{b=1}^B \overline{y}^{*(b)} = \frac{1}{n}\sum_{j=1}\left(\frac{1}{B}\sum_{b=1}^B N_j^{*(b)}\right) y_j, \tag{10.12}$$

with similar formula for the z's. Since the points are sampled with equal probability, $B^{-1}\sum_{b=1}^B N_j^{*(b)} \to 1$ as $B \to \infty$, so not surprisingly (10.12) converges to $\overline{y}$, which is the true mean in the bootstrap sample. The correction term

$$\frac{\sum_{b=1}^B \overline{y}^{*(b)}}{\sum_{b=1}^B \overline{z}^{*(b)}} - \hat{\theta}$$

to the standard bias estimate (10.9) reflects the extent to which the actual bootstrap resamples deviate from an idealized set where each of the observed data points appears exactly B times in the B bootstrap resamples. □

The technique in the previous example can often be used when the empirical distribution of the data is used as the bootstrap sampling distribution. Suppose that the statistic computed from the bth resample can be written as a function of the proportions $P_j^{*(b)} = N_j^{*(b)}/n$, where again $N_j^{*(b)}$ is the number of times the jth data point appears in the bth resample. That is, $\hat{\theta}^{*(b)} = g(P_1^{*(b)} \ldots, P_n^{*(b)})$, and $\hat{\theta} = g(1/n, \ldots, 1/n)$. In the previous example,

$$g(u_1, \ldots, u_n) = \sum_{j=1}^n u_j y_j / \sum_{j=1}^n u_j z_j.$$

Then the standard bootstrap estimate of bias is

$$B^{-1}\sum_{b=1}^B g(P_1^{*(b)} \ldots, P_n^{*(b)}) - g(1/n, \ldots, 1/n).$$

The adjusted bias estimate corresponding to (10.11) is

$$B^{-1}\sum_{b=1}^{B} g(P_1^{*(b)} \ldots, P_n^{*(b)}) - g(\overline{P}_1^*, \ldots, \overline{P}_n^*),$$

where $\overline{P}_j^* = \sum_{b=1}^{B} P_j^{*(b)}/B$. Again this adjusts for the difference between the actual samples, and a balanced set of resamples where each point appears exactly B times.

Nonlinear estimators generally have bias of $O(n^{-1})$. The bootstrap bias correction generally removes the leading $O(n^{-1})$ term in an expansion for the bias, and the bias corrected estimator will have remaining bias of $O(n^{-2})$. As will be discussed in Section 10.5, it turns out that nested application of the bootstrap can be used to further reduce the order of the bias.

10.3 Bootstrap Hypothesis Tests

Given a sample $\mathbf{X}$, consider testing the hypothesis $H_0 : \theta = \theta_0$ using the statistic $T(\mathbf{X})$, and suppose without loss of generality the test rejects for large values of the statistic.

Let t_0 be the value of the statistic for the observed data. The p-value is defined to be

$$P[T(\mathbf{X}) \geq t_0 | \theta = \theta_0]. \tag{10.13}$$

If the null hypothesis completely specifies the distribution, then the p-value is simply computed under that distribution. If null hypothesis is composite, then the bootstrap can be used to approximate the p-value. In this case the bootstrap consists of using an estimate of in place of the unknown true null distribution. As for other types of bootstrap problems, exact computations are usually not possible, and simulations are used. If the bootstrap resamples are denoted by $\mathbf{X}^{*(b)}$, then the basic bootstrap p-value estimate is given by

$$\frac{1}{B}\sum_{b=1}^{B} I[T(\mathbf{X}^{*(b)}) \geq t_0].$$

Since this is of the same form as used in estimating the rejection probability of a test in the Basic Simulation notes, importance sampling can often be used to improve the efficiency of the resampling, as will be discussed in more detail below.

The main distinction between the bootstrap variance estimation and hypothesis testing problems is that in hypothesis testing it is necessary to use an estimated distribution which satisfies the null hypothesis. This means it is often not appropriate to use the empirical distribution, without some modification. The problem of testing equality of means of two samples will help to illustrate the issues.

Example 10.4 Testing Equality of Means. Suppose the sample $\mathbf{X}$ consists of iid observations $y_1, \ldots, y_n$ drawn from one population, and iid observations $z_1, \ldots, z_m$ drawn from a separate population. Let $\theta = \mathrm{E}(y_i) - \mathrm{E}(z_j)$, and consider $H_0 : \theta = 0$, that is, that the means of the two populations are equal. Let $\overline{y}$ and $\overline{z}$ be the usual sample means, and let s_y^2 and s_z^2 be the usual sample variances. The statistic

$$T(\mathbf{X}) = |\overline{y} - \overline{z}|/(s_y^2/n + s_z^2/m)^{1/2}$$

can be used to test this hypothesis.

Usually the size of the samples from the two populations would be treated as fixed (although even this is not mandatory). Consider drawing resamples by drawing $y_1^*, \ldots, y_n^*$ from the empirical distribution of $y_1, \ldots, y_n$ and $z_1^*, \ldots, z_m^*$ from the empirical distribution of $z_1, \ldots, z_m$. The difficulty is that then $\mathrm{E}(y_i^*) = \overline{y}$ and $\mathrm{E}(z_i^*) = \overline{z}$, and generally these are not equal, so the resampling distribution does not satisfy the null hypothesis.

One way to proceed is to still resample from the empirical distributions, but to approximate the p-value in the original problem with the p-value in the bootstrap simulation for the hypothesis $H_0 : \mathrm{E}(y_i^* - z_j^*) = \overline{y} - \overline{z}$. Since this is the true mean in the simulation, the bootstrap simulation is then conducted under this null hypothesis. Setting

$$S(\mathbf{X}^*) = |\overline{y}^* - \overline{z}^* - \overline{y} + \overline{z}|/(s_y^{*2}/n + s_z^{*2}/m)^{1/2},$$

the bootstrap estimate of the p-value is

$$B^{-1} \sum_{b=1}^{B} I[S(\mathbf{X}^{*(b)}) \geq t_0]. \tag{10.14}$$

Equivalently, this can be thought of as shifting the location of one or both of the distributions so they have the same mean. For example, resample the $y_1^*, \ldots, y_n^*$ from the empirical distribution of $y_1, \ldots, y_n$ (as before), and $\tilde{z}_1^*, \ldots, \tilde{z}_n^*$ from the empirical distribution of $z_1 - \overline{z} + \overline{y}, \ldots, z_m - \overline{z} + \overline{y}$. Then $\mathrm{E}(y_i^*) = \mathrm{E}(\tilde{z}_i^*) = \overline{y}$, so the means are equal and the estimate of the rejection probability is

$$B^{-1} \sum_{b=1}^{B} I[|\overline{y}^* - \overline{\tilde{z}}^*|/(s_y^{*2}/n + \tilde{s}_z^{*2}/m)^{1/2} \geq t_0],$$

which is algebraically identical to (10.14). In this approach, the shape of the y and z distributions are estimated using the empirical distributions of the two samples (and no assumption is made that the shapes are similar), but there is an assumption that the only difference between the null and alternative distributions is shifting the relative locations of the two samples. This would not be appropriate, for example, if both samples had a common range restriction, such as to the interval $(0, \infty)$ or $(0, 1)$.

With more assumptions, a better approach can be given. Suppose it can be assumed that the distributions of the two samples differ only with respect to a location shift, that is, if $P(y_i \leq u) = G(u)$, then $P(z_i \leq u) = G(u - \theta)$. Then under $H_0 : \theta = 0$, the distributions of the two samples are identical, and the common distribution can be estimated with the distribution of the pooled sample. In this case it would be more appropriate to use a pooled estimate of variance in the test statistic, so let

$$s_p^2 = \left(\sum_{i=1}^{n} (y_i - \overline{y})^2 + \sum_{i=1}^{m} (z_i - \overline{z})^2 \right) /(n + m - 2),$$

$$R(\mathbf{X}) = |\overline{y} - \overline{z}|/[s_p(1/n + 1/m)^{1/2}],$$

and r_0 be the observed value of $R(\mathbf{X})$. Sampling $y_1^*, \ldots, y_n^*$ and $z_1^*, \ldots, z_m^*$ from the empirical distribution of the pooled sample, the p-value $P[R(\mathbf{X}) > r_0 | \theta = 0]$ can be estimated by

$$B^{-1} \sum_{b=1}^{B} I[R(\mathbf{X}^{*(b)}) \geq r_0].$$

While this seems like a sensible approach in this case, it is not the only possibility under the location shift model. If the observed sample means $\overline{y}$ and $\overline{z}$ are far apart, so the data are not consistent with the null hypothesis, then the spread in the pooled sample may be larger than would be appropriate for samples obtained under the null. In this case an alternative sampling procedure that would give samples with a smaller variance is to sample $y_1^*, \ldots, y_n^*$ and $z_1^*, \ldots, z_m^*$ from the empirical distribution of the pooled data with one sample shifted, for example, from the empirical distribution of $y_1, \ldots, y_n, z_1 - \overline{z} + \overline{y}, \ldots, z_m - \overline{z} + \overline{y}$. □

The previous example illustrates some of the difficulties in choosing an appropriate sampling distribution under the null hypothesis. This is generally not a trivial problem in using simulations to approximate p-values. Usually one tries to use a good estimate of the distribution obtained under an appropriate model for the data under the null hypothesis, but the choice is usually somewhat arbitrary, and different choices give different results. If the estimate is consistent for the true distribution under the null, then all choices should be asymptotically equivalent, though.

The use of the studentized statistics in the previous example was based on theoretical considerations. Generally if $T(\mathbf{X}) \xrightarrow{\mathcal{D}} N(0,1)$, and the asymptotic normal distribution is used to approximate the p-value, say with $\Phi(-t_0) + 1 - \Phi(t_0)$, where $\Phi(\cdot)$ is the standard normal CDF, then the error in this approximation to the p-value is $O([1/n + 1/m]^{1/2})$. Using the bootstrap distribution of $T(\mathbf{X}^*)$ to approximate the p-value then usually has an error of order $O(1/n + 1/m)$, and so can be substantially more accurate. However, if $T(\mathbf{X}) \xrightarrow{\mathcal{D}} N(0, \sigma^2)$, where σ^2 is unknown, and the bootstrap is applied directly to $T(\mathbf{X})$, then usually the error is $O([1/n + 1/m]^{1/2})$.

10.3.1 Importance Sampling for Nonparametric Bootstrap Tail Probabilities

Suppose that under H_0 the sample $\mathbf{X}$ consists of iid observations $x_1, \ldots, x_n$, which may be vector valued, and the bootstrap distribution F^* samples values independently from $\hat{F}$, the empirical distribution of the $\tilde{x}_i$ (where the $\tilde{x}_i$ consist of the original sample, possibly with some observations shifted to guarantee that $\hat{F}$ satisfies the null hypothesis). To estimate $P(T(\mathbf{X}) \geq t_0)$, independent resamples $\mathbf{X}^{*(1)}, \ldots, \mathbf{X}^{*(B)}$ are drawn, where each $\mathbf{X}^{*(b)}$ consists of n independent observations from $\hat{F}$, and

$$\hat{p} = B^{-1} \sum_{b=1}^{B} I[T(\mathbf{X}^{*(b)}) \geq t_0]$$

computed. In this section importance sampling to improve the efficiency of the bootstrap simulation is considered. The problem and approach taken are quite similar to those in Section 7.3.2.

Consider first the case where $T(\mathbf{X}) = n^{-1} \sum_{i=1}^{n} g(x_i)$, for some known function $g(\cdot)$. To make effective use of importance sampling, some of the mass of $\hat{F}$ should be modified so that it no longer satisfies the null hypothesis. Shifting the points would mean that the actual sampling distribution and the distribution of interest have support on different points, which would make it difficult to compute the importance weights. Instead, the distribution can be modified by keeping the support points the same, but assigning different mass to each of the points. In particular, instead of giving each point equal weight, giving large mass to points with larger values of $g(\tilde{x}_i)$ and less to points with smaller values will increase the mean of $T(\mathbf{X}^*)$ in the bootstrap resampling.

One way to do this is with a one parameter family of weighted discrete distributions. Let

$$p(\tilde{x}_i;\lambda) = \frac{\exp(\lambda[g(\tilde{x}_i) - \tilde{g}])}{\sum_{j=1}^n \exp(\lambda[g(\tilde{x}_j) - \tilde{g}])},$$

where $\tilde{g} = n^{-1}\sum_{i=1}^n g(\tilde{x}_i)$. Note that $p(\tilde{x}_i;0) = 1/n$. If the distribution of x_i^* is given by

$$P(x_i^* = \tilde{x}_j) = p(\tilde{x}_j;\lambda), \tag{10.15}$$

then when $\lambda > 0$, the expectation

$$\mathrm{E}_\lambda^*[g(x_i^*)] = \frac{\sum_{j=1}^n g(\tilde{x}_j)\exp(\lambda[g(\tilde{x}_j) - \tilde{g}])}{\sum_{j=1}^n \exp(\lambda[g(\tilde{x}_j) - \tilde{g}])} > \tilde{g} = \mathrm{E}_{\lambda=0}^*[g(x_i^*)].$$

Thus when $\lambda > 0$, $\mathrm{E}_\lambda^*[T(\mathbf{X}^*)]$ is increased, and its distribution should put more mass into the critical region, so (10.15) might be a suitable candidate for an importance sampling distribution. This procedure can be sensitive to the value of λ. One possibility is to choose λ to solve

$$\mathrm{E}_\lambda^*[g(x_i^*)] = t_0, \tag{10.16}$$

so that about half the bootstrap resamples will have $T(\mathbf{X}^*) > t_0$.

For more general statistics, a similar approach can be used, if the statistic can be approximated by a linear function of the data (most asymptotically normal statistics can be). That is, suppose that $T(\mathbf{X})$ is scaled so that it is asymptotically normal, and that it can be written as

$$T(\mathbf{X}) = n^{-1/2}\sum_{i=1}^n U(x_i, F) + V(\mathbf{X}, F),$$

where $V(\mathbf{X}, F)$ is asymptotically negligible, and F is included as an argument of U and V to indicate that they can depend on the true distribution (including unknown parameters) in addition to the data. Then a possible importance sampling distribution for the bootstrap resampling is

$$P(x_i^* = \tilde{x}_j) = \exp[\lambda U(\tilde{x}_j, \hat{F})]/\sum_k \exp[\lambda U(\tilde{x}_k, \hat{F})]. \tag{10.17}$$

Analogously to (10.16), λ could be chosen to solve $\mathrm{E}_\lambda^*[U(x_i^*, \hat{F})] = t_0/n^{1/2}$. The $U(x_i, F)$ terms in the linear expansion are called influence functions.

Example 10.5 Suppose the sample consists of iid bivariate observations $(s_i, t_i)'$, $i = 1, \ldots, n$, with $\mathrm{E}^*[(s_i, t_i)'] = (\mu_s, \mu_t)'$, $\theta = \mu_t \exp(-\mu_s)$, and the test statistic for $H_0 : \theta = \theta_0$ versus $H_a : \theta > \theta_0$ is

$$T(\mathbf{X}) = \sqrt{n}[\overline{t}\exp(-\overline{s}) - \theta_0].$$

Then using a first order Taylor series expansion,

$$T(\mathbf{X}) \doteq \sqrt{n}[\theta_0 + \theta_0(\overline{t} - \mu_{t_0})/\mu_{t_0} - \theta_0(\overline{s} - \mu_{s_0})],$$

where μ_{t_0} and μ_{s_0} are the means under the true null distribution, so an appropriate choice for $U(x, F)$ is

$$U(x, F) = \theta_0(t - \mu_{t_0})/\mu_{t_0} - \theta_0(s - \mu_{s_0}).$$

In this case the easiest way to perform a nonparametric bootstrap resampling under H_0 is to use the empirical distribution of the data, and to change the null value of the parameter to match this observed value. That is, in the resampling use the statistic

$$T(\mathbf{X}^*) = \sqrt{n}[\overline{t}^* \exp(-\overline{s}^*) - \overline{t}\exp(-\overline{s})].$$

Also,

$$U(x_i, \hat{F}) = \exp(-\overline{s})(t_i - \overline{t}) - \overline{t}\exp(-\overline{s})(s_i - \overline{s}).$$

The importance sampling distribution is given by (10.17), say with λ chosen so that

$$\frac{\sum_i U(x_i, \hat{F}) \exp[\lambda U(x_i, \hat{F})]}{\sum_i \exp[\lambda U(x_i, \hat{F})]} = t_0/n^{1/2},$$

where t_0 is the value of $T(\mathbf{X})$ for the observed data. The bootstrap importance sampling estimate of the p-value is then

$$B^{-1} \sum_{b=1}^{B} I(T(\mathbf{X}^{*(b)}) \geq t_0) n^{-n} \frac{(\sum_j \exp[\lambda U(x_j, \hat{F})])^n}{\exp[\lambda \sum_i U(x_i^{*(b)}, \hat{F})]}$$

(n^{-n} is the mass function for the target resampling distribution, and the last factor is the inverse of the mass function for the actual resampling distribution).

The following code implements this procedure on a simulated data set. Note that in the resampling, the null hypothesis is taken to be the value of the statistic in the observed data. In the first run, using the value of λ suggested above, there is a small reduction in the standard error. In the second run, using a larger value of λ leads to more improvement.

```
> isboot <- function(nboot,x,lstar,U,t00,t0) {
+ # x=data=cbind(s,t);lstar=lambda;U=influence
+ # functions; t00=null value;t0=obs stat
+   probs <- exp(lstar*U)
+   probs <- probs/sum(probs)
+   print(summary(probs))
+   out <- matrix(0,3,nboot)
+   for (i in 1:nboot) {
+     i1 <- sample(1:nrow(x),nrow(x),T)
+     i2 <- sample(1:nrow(x),nrow(x),T,probs)
+     xmb1 <- mean(x[i1,1])
+     xmb2 <- mean(x[i1,2])
+     t1 <- sqrt(nrow(x))*(xmb2*exp(-xmb1)-t00)
+     xmb1 <- mean(x[i2,1])
+     xmb2 <- mean(x[i2,2])
+     t2 <- sqrt(nrow(x))*(xmb2*exp(-xmb1)-t00)
+     wt <- exp(sum(-log(probs[i2]*nrow(x))))
+     out[,i] <- c(t1,t2,wt)
+   }
+   print(apply(out,1,summary))
```

```
+   w <- (out[2,]>=t0)*out[3,]
+   p1 <- mean(out[1,]>=t0)
+   print(c(p1,mean(w)))
+   print(sqrt(c(p1*(1-p1),var(w))/nboot))
+   print(table(out[2,]>t0))
+   invisible()
+ }
> # generate data H0:theta=.11
> set.seed(200)
> x <- matrix(rnorm(100),nrow=2) # s,t
> B <- matrix(c(1,1,0,1),2,2)
> t(B)%*%B
     [,1] [,2]
[1,]    2    1
[2,]    1    1
> x <- t(exp(t(B)%*%x))
> xm <- apply(x,2,mean)
> xm
[1] 2.661298 1.777114
> t00 <- xm[2]*exp(-xm[1])
> t0 <- sqrt(nrow(x))*(t00-.11)
> t0
[1] 0.1000186
> # influence functions
> U <- exp(-xm[1])*((x[,2]-xm[2])-xm[2]*(x[,1]-xm[1]))
> ff <- function(lam) {w <- exp(lam*U)
+   sum(w*U)/sum(w)-t0/sqrt(nrow(x))}
> lstar <- uniroot(ff,c(-5,5))$root
> lstar
[1] 0.07419001
> isboot(1000,x,lstar,U,t00,t0)
    Min. 1st Qu. Median Mean 3rd Qu.    Max.
 0.01651 0.02007 0.0202 0.02 0.02029 0.02053
         [,1]    [,2]   [,3]
[1,] -0.77560 -0.7121 0.6139
[2,] -0.24780 -0.1540 0.8289
[3,]  0.04528  0.1653 0.9447
[4,]  0.09853  0.2090 0.9929
[5,]  0.37280  0.5012 1.1160
[6,]  1.80600  1.8030 2.1850
[1] 0.4470000 0.4529162
[1] 0.01572231 0.01323698
 FALSE TRUE
   455  545
> isboot(1000,x,.25,U,t00,t0) # new lambda
    Min. 1st Qu.  Median Mean 3rd Qu.    Max.
```

```
 0.01044 0.02015 0.02061 0.02  0.0209 0.02176
         [,1]     [,2]   [,3]
[1,] -0.80880 -0.59050 0.1554
[2,] -0.23770  0.06732 0.4837
[3,]  0.04307  0.41530 0.7131
[4,]  0.10040  0.44410 1.0020
[5,]  0.39740  0.77000 1.1630
[6,]  1.74400  2.56600 7.3220
[1] 0.4510000 0.4390412
[1] 0.01573528 0.01048219
 FALSE TRUE
   276  724
```

□

10.3.2 Antithetic Bootstrap Resampling

Recall the basic idea of antithetic sampling from Section 7.3.3. Hall (1989) proposed a method for generating antithetic bootstrap resamples. This is a general method that can be used to varying degrees of effectiveness in all types of bootstrap problems. Let $x_1, \ldots, x_n$ be a sample and $\hat{\theta} = \hat{\theta}(x_1, \ldots, x_n)$ be an estimator of θ, which is invariant to the order of the data points x_i. As was discussed for $T(\mathbf{X})$ in the previous section, generally $\hat{\theta}$ has an expansion

$$\hat{\theta} = \theta + n^{-1} \sum_{i=1}^{n} U(x_i, F) + V(\mathbf{X}, F),$$

where $V(\mathbf{X}, F)$ is asymptotically negligible compared to the linear term.

To define the antithetic sample, relabel the data points $x_1, \ldots, x_n$ so that $U(x_1, \hat{F}) \leq \cdots \leq U(x_n, \hat{F})$. The regular bootstrap sample $x_1^*, \ldots, x_n^*$ is sampled with replacement from the observed data. Suppose that $x_i^* = x_{j(i)}$. The antithetic bootstrap resample is defined to be the resample consisting of the points $x_i^{*a} = x_{n-j(i)+1}$. That is, the antithetic resample takes the opposite value of each point in the original resample.

As with antithetic sampling in the Basic Simulation handout, the quantity of interest is computed both from the regular sample and the antithetic sample, and the average of the two values used as the final estimate. The success of the method depends on there being a fairly large negative correlation between the estimates from the regular and antithetic samples. Hall (1989) shows that the antithetic resampling as defined above maximizes the magnitude of the negative correlation for several types of bootstrap estimation problems.

For estimating the variance of $\hat{\theta}$, it may be more appropriate to order the points based on the linear terms in the expansion of $(\hat{\theta} - \theta)^2$.

Below, this antithetic sampling algorithm is applied to the testing problem in Example 10.5 from the previous section. Antithetic sampling reduces the variance of the p-value estimate by about 60% here, with the trade-off that the test is computed twice within each sample.

Example 10.5 (continued).

```
> atboot <- function(nboot,x,t00,t0) {
+ # x=data=cbind(s,t);t00=null value;t0=obs stat
+   out <- matrix(0,2,nboot)
+   for (i in 1:nboot) {
+     i1 <- sample(1:nrow(x),nrow(x),T)
+     xmb1 <- mean(x[i1,1])
+     xmb2 <- mean(x[i1,2])
+     t1 <- sqrt(nrow(x))*(xmb2*exp(-xmb1)-t00)
+     i2 <- nrow(x)-i1+1 #antithetic sample
+     xmb1 <- mean(x[i2,1])
+     xmb2 <- mean(x[i2,2])
+     t2 <- sqrt(nrow(x))*(xmb2*exp(-xmb1)-t00)
+     out[,i] <- c(t1,t2)
+   }
+   out <- out>=t0
+   print(cor(out[1,],out[2,]))
+   out2 <- (out[1,]+out[2,])/2
+   p <- c(mean(out[1,]),mean(out[2,]))
+   print(c(p,mean(out2)))
+   print(sqrt(c(p*(1-p),var(out2))/nboot))
+   invisible()
+ }
> # x,u as in previous section
> # sort data on influence functions
> o <- order(U)
> U <- U[o]
> x <- x[o,]
> atboot(1000,x,t00,t0)
[1] -0.2098982
[1] 0.423 0.453 0.438
[1] 0.015622772 0.015741379 0.009861706
```

□

10.4 Bootstrap Confidence Intervals

A $100(1-\alpha)\%$ upper confidence limit for a real valued parameter θ is a random variable (a function of the data) $\tilde{\theta}_u$ such that

$$P(\tilde{\theta}_u > \theta|\theta) = 1 - \alpha. \tag{10.18}$$

The notation "$P(\cdot|\theta)$" means that the probability is computed using the distribution of the data when θ is the true value of the parameter. Although the requirement is only that $\tilde{\theta}_u$ give the right coverage probability at the true distribution, since the true distribution is unknown, a good confidence interval procedure should satisfy (10.18) for all possible values of the parameter. That is, if $\tilde{\theta}_u$ only gave the right coverage probability at a particular θ_0, but it is not known whether θ_0

is the true value of θ, then $\tilde{\theta}_u$ may or may not have reasonable coverage. Lower confidence limits can be defined analogously to (10.18). Here only the upper confidence limit will be studied in detail.

Apart from specifying the coverage probability, the definition of an upper confidence limit is completely noninformative about how to construct good confidence limits. Ideally, the confidence set should reflect the values of the parameter which are most likely to have generated the observed data. A brief discussion of some general issues follows, and then some bootstrap procedures will be defined. Some general background is also given in Chapter 12 of Efron and Tibshirani (1993), and several bootstrap procedures defined in Chapters 12–14.

A "pivot" is a function of the parameter of interest and the data whose distribution is independent of the parameter(s) (including any nuisance parameters). Suppose $X_1, \ldots, X_n$ are sampled from a location-scale family, where the location parameter μ is of primary interest and the scale parameter σ is a nuisance parameter. Then $(\overline{X}_n - \mu)/\{\sum_i (X_i - \overline{X}_n)^2\}^{1/2}$ is a pivot, since its distribution is independent of μ and σ.

A function of the data and the parameter of interest whose asymptotic distribution is independent of any unknown parameters is an asymptotic or approximate pivot. For example, if $n^{1/2}(\hat{\theta}_n - \theta) \xrightarrow{\mathcal{D}} N(0, \sigma^2)$, and $\hat{\sigma}_n$ is a consistent estimator for σ, then

$$n^{1/2}(\hat{\theta}_n - \theta)/\hat{\sigma}_n \tag{10.19}$$

is an approximate pivot.

For an exact pivot of the form (10.19), an upper $100(1-\alpha)\%$ confidence limit can be defined by

$$\tilde{\theta}_{un} = \hat{\theta}_n - x_\alpha \hat{\sigma}_n / n^{1/2}, \tag{10.20}$$

where x_α satisfies

$$P\{n^{1/2}(\hat{\theta}_n - \theta)/\hat{\sigma}_n > x_\alpha\} = 1 - \alpha.$$

Since $n^{1/2}(\hat{\theta}_n - \theta)/\hat{\sigma}_n$ is a pivot, its distribution is the same for all parameter values, and a single value x_α gives exact coverage regardless of the true values of the parameters. For an approximate pivot of the form (10.19), a standard approximate upper confidence limit is again given by (10.20), but with the percentile x_α defined from the asymptotic distribution of $n^{1/2}(\hat{\theta}_n - \theta)/\hat{\sigma}_n$. This again gives a single value x_α regardless of the true value of the parameter, but the exact coverage probability

$$P(\hat{\theta}_n - x_\alpha \hat{\sigma}_n / n^{1/2} > \theta)$$

will generally not equal $1 - \alpha$, and usually will vary with the true value of the parameter. However, since $n^{1/2}(\hat{\theta}_n - \theta)/\hat{\sigma}_n$ is an asymptotic pivot, the coverage probability converges to $1 - \alpha$ for all values of the parameter.

Another way to define a confidence limit is to consider testing the null hypothesis $H_0 : \theta = \theta_0$ against the alternative $H_1 : \theta < \theta_0$. Suppose there is a family of valid tests which reject if $\hat{\theta}_n < K(\theta_0)$, where $K(\theta_0)$ satisfies

$$P\{\hat{\theta}_n < K(\theta_0) | \theta_0\} = \alpha. \tag{10.21}$$

Then the set of θ_0 values which are not rejected at level α are an upper $100(1-\alpha)\%$ confidence set. Note that $K(\theta_0)$ should be an increasing function of θ_0, and the upper confidence limit can be thought of as being the solution to $\hat{\theta}_n = K(\tilde{\theta}_{un})$; that is $\tilde{\theta}_{un}$ is the value of θ_0 which puts the observed $\hat{\theta}_n$ on the boundary of the critical region. Then

$$P(\theta < \tilde{\theta}_{un}|\theta) = P\{K(\theta) < K(\tilde{\theta}_{un})|\theta\} = P\{K(\theta) < \hat{\theta}_n|\theta\} = 1-\alpha,$$

verifying that this is a $100(1-\alpha)\%$ confidence limit. If a test has good power properties, then the confidence set obtained by inverting the test should in some sense contain the parameter values most consistent with the observed data.

If the distribution of $\hat{\theta}_n$ depends on other unknown parameters besides θ, then further modification is needed, such as choosing $K(\theta_0)$ so that

$$P\{\hat{\theta}_n < K(\theta_0)|\theta_0\} \leq \alpha$$

for all values of the nuisance parameters. Typically the exact type I error of such a test will vary with the values of the nuisance parameters, and consequently so will the coverage probability of the confidence set determined by inverting the test. However, the coverage probability would be $\geq 1-\alpha$ for all values of the nuisance parameter.

If $n^{1/2}(\hat{\theta}_n - \theta)/\hat{\sigma}_n$ is a pivot, then the natural test would be to reject if $n^{1/2}(\hat{\theta}_n - \theta_0)/\hat{\sigma}_n < x_\alpha$, where x_α satisfies $P\{n^{1/2}(\hat{\theta}_n - \theta_0)/\hat{\sigma}_n < x_\alpha|\theta_0\} = \alpha$. Inverting this test again gives (10.20).

In the testing approach to confidence limits, the behavior of the estimator or test statistic is considered under a variety of values of the parameter. This behavior is then used to determine which values of the parameter, if they were the true value, would be most consistent with the observed data. Proposals for bootstrap confidence limits attempt to construct confidence limits from a single approximate sampling distribution. This generally does not give exact procedures. A possible alternative to the usual bootstrap confidence intervals is to construct a confidence set by inverting the bootstrap hypothesis tests described in Section 10.3 above. This would require an iterative search to determine the null hypothesis which puts the observed data on the boundary of the critical region, and each step in the iteration would require a new set of bootstrap samples.

The bootstrap-t confidence limit is defined in Section 12.5 of Efron and Tibshirani (1993). In this approach, for an approximate pivot $n^{1/2}(\hat{\theta}_n - \theta)/\hat{\sigma}_n$, instead of using the asymptotic distribution to approximate the percentile x_α, the bootstrap distribution is used. This is in accordance with the general bootstrap principle. That is, the true x_α solves

$$P\{n^{1/2}(\hat{\theta}_n - \theta)/\hat{\sigma}_n > x_\alpha\} = 1-\alpha.$$

In the bootstrap, the probability in this expression is approximated using the bootstrap distribution of the statistic, and x_α estimated by determining the appropriate percentile of the bootstrap distribution. Thus the bootstrap estimate of the percentile $\hat{x}_\alpha$ solves

$$P\{n^{1/2}(\hat{\theta}^*_n - \hat{\theta}_n)/\hat{\sigma}^*_n > \hat{x}_\alpha|\mathbf{X}\} = 1-\alpha,$$

where as usual $\hat{\theta}^*_n$ and $\hat{\sigma}^*_n$ are computed from bootstrap samples. (Because the bootstrap distribution is usually discrete, this may not have an exact solution.) The upper confidence limit is then

$$\hat{\theta}_n - \hat{\sigma}_n \hat{x}_\alpha / n^{1/2}; \tag{10.22}$$

the bootstrap is only used to estimate the percentile $\hat{x}_\alpha$. Confidence limits formed using the asymptotic distribution of the approximate pivot will have coverage probabilities that are first order accurate, in the sense that they differ from $1-\alpha$ by $O(n^{-1/2})$, while bootstrap-t confidence limits are second order accurate, and have coverage probabilities that differ from $1-\alpha$ by $O_p(n^{-1})$.

As discussed in Section 12.6 of Efron and Tibshirani (1993), asymptotic-t and bootstrap-t confidence limits are not invariant to transformations of the parameters. That is, if $n^{1/2}(\hat{\theta}_n - \theta)/\hat{\sigma}_n \xrightarrow{\mathcal{D}} N(0,1)$, and g is a smooth monotone increasing function, then from the delta method,

$$n^{1/2}\{g(\hat{\theta}_n) - g(\theta)\}/(\hat{\sigma}_n |g'(\hat{\theta}_n)|) \xrightarrow{\mathcal{D}} N(0,1), \tag{10.23}$$

and so is also an approximate pivot. Any such g can be used to define an upper confidence limit for $g(\theta)$ as before, and then since g is monotone, transforming the upper confidence limit for $g(\theta)$ by g^{-1} will give an upper confidence limit for θ of the form

$$\tilde{\theta}_{gn} = g^{-1}\{g(\hat{\theta}_n) - \hat{\sigma}_n |g'(\hat{\theta}_n)| x_\alpha^{(g)}/n^{1/2}\}$$

where $x_\alpha^{(g)}$ is the asymptotic percentile of (10.23). In the bootstrap-t version, $x_\alpha^{(g)}$ is replaced by the corresponding percentile from the bootstrap distribution. These confidence limits will be different for different g, and while all will have the same order of coverage accuracy, not all will perform as well for a given n. For parameters that have a restricted range, like the correlation coefficient, some transformations will give confidence limits for some data sets which are outside the parameter space, which is not desirable.

To deal with range restrictions and transformations, Efron proposed a bootstrap percentile method, which is described in Chapter 13 of Efron and Tibshirani (1993). If $G_n(u|\mathbf{X}) = P(\hat{\theta}_n^* \le u|\mathbf{X})$, then Efron's percentile method defines the upper confidence limit by

$$G_n^{-1}(1-\alpha|\mathbf{X}). \tag{10.24}$$

If $\hat{\theta}_n$ only takes values in the parameter space, then G_n will have support on the parameter space, and so (10.24) will be in the parameter space. Also, if estimates of functions of θ satisfy $\widehat{g(\theta)}_n = g(\hat{\theta}_n)$, then a confidence limit for θ determined from the distribution of $\widehat{g(\theta)}_n$ will be the same regardless of g. Thus these confidence limits transform properly. However, (10.24) does not appear to be defined in accordance with the general bootstrap principle, where a quantity of interest is defined from the distribution of $\hat{\theta}_n$, and then estimated using the conditional distribution of $\hat{\theta}_n^*$. And at first glance it may not be clear whether (10.24) has any connection with confidence limits.

To make a bit more sense of this, suppose $P(\hat{\theta}_n \le u) = H(u;\theta)$. Consider the quantity $H^{-1}(1-\alpha;\hat{\theta}_n)$. Formula (10.24) could be thought of as a bootstrap analog of this, with the unknown distribution H estimated from the bootstrap distribution. If $H(u;\theta) = F(u-\theta)$, where F is the CDF of a symmetric density, then $H^{-1}(1-\alpha;\hat{\theta}_n)$ is a valid upper confidence limit. That is

$$\begin{aligned} P(H^{-1}(1-\alpha;\hat{\theta}_n) > \theta|\theta) &= P(1-\alpha > H(\theta;\hat{\theta}_n)|\theta) \\ &= P(1-\alpha > F(\theta-\hat{\theta}_n)|\theta) \end{aligned}$$

$$\begin{aligned} &= P(\theta - F^{-1}(1-\alpha) < \hat{\theta}_n|\theta) \\ &= P(\theta + F^{-1}(\alpha) < \hat{\theta}_n|\theta) \\ &= 1 - F([\theta + F^{-1}(\alpha)] - \theta) \\ &= 1 - \alpha, \end{aligned}$$

since $F^{-1}(1-\alpha) = -F^{-1}(\alpha)$ because of symmetry, and since $P(\hat{\theta}_n > u|\theta) = 1 - F(u-\theta)$. Note that without symmetry, even with the location family assumption, the coverage probability would have been $F[-F^{-1}(1-\alpha)]$, which would generally not be $1-\alpha$. In this sense Efron's percentile method uses the percentile from the "wrong tail" of the distribution. When $n^{1/2}(\hat{\theta}_n - \theta)$ is asymptotically normal, then asymptotically $\hat{\theta}_n$ has the distribution of a symmetric location family, and the coverage probability of Efron's percentile method will converge to $1-\alpha$. Since it converges to this distribution at a rate of $n^{-1/2}$, it is not surprising that Efron's percentile method is only first order accurate.

Hall (1992) defines a different percentile method. Hall proposes as a $100(1-\alpha)\%$ upper confidence limit the point $\hat{\theta}_n + t$, where t is chosen so that

$$P(\hat{\theta}_n + t > \theta) = 1 - \alpha. \tag{10.25}$$

The bootstrap is then used to estimate t. Specifically, choose $\hat{t}$ as the solution to

$$P(\hat{\theta}_n^* + \hat{t} > \hat{\theta}_n|\mathbf{X}) = 1 - \alpha.$$

Note that $\hat{\theta}_n - \hat{t} = G_n^{-1}(\alpha|\mathbf{X})$, where again G_n is the conditional CDF of $\hat{\theta}_n^*$, so the upper confidence limit is

$$2\hat{\theta}_n - G_n^{-1}(\alpha|\mathbf{X}), \tag{10.26}$$

in contrast to (10.24). This is based on the general bootstrap principle, in that a quantity t of interest is defined, and the bootstrap distribution used to estimate it. However, since generally $\hat{\theta}_n - \theta$ is not a pivot, the quantity t defined in (10.25) is a function of unknown parameters, so if the exact t could be determined $\hat{\theta}_n + t$ would still not be a useful confidence limit. The bootstrap estimates t under a particular distribution, but does not examine how t changes with the true distribution. Since the bootstrap distribution converges to the true distribution at a rate of $n^{-1/2}$, it is not surprising that Hall's percentile method is also only first order accurate. It is also true that Hall's percentile method can violate range restrictions on the parameter. That is, since it takes the percentile from the opposite tail of the distribution and reflects it, it can go beyond the range of the parameter space when there are restrictions on the range of the parameter.

Various methods have been proposed to modify or correct Efron's percentile method to maintain some of its desirable properties while getting second order accuracy. Efron's BC_a method (Chapter 14 in Efron and Tibshirani (1993)) is an example of this. Hall (1992) Section 3.10 discusses some other possibilities (and elsewhere various other approaches to constructing second order accurate confidence limits). Bootstrap iteration, a general approach to improving accuracy of bootstrap procedures, is discussed in the following section.

10.5 Iterated Bootstrap

Bootstrap iteration is discussed in Sections 1.4 and 3.11 of Hall (1992). It was introduced in various settings by Efron (1983), Hall (1986) and Beran (1987). The basic idea of the iterated

bootstrap is to use a second level of bootstrap sampling to estimate the error in a bootstrap procedure, and to use this estimate of error to correct the original procedure. This procedure can be iterated: use a 3rd level of sampling to estimate the error in the corrected procedure to get a new correction, a 4th level to estimate the error ....

In the setting of an upper confidence limit, correcting the error in the coverage probability of a procedure would be of interest. To be concrete, consider the bootstrap-t procedure. The upper confidence limit is

$$\hat{\theta}_n - \hat{x}_\alpha \hat{\sigma}_n / n^{1/2},$$

with $\hat{x}_\alpha$ chosen to satisfy

$$P(\hat{\theta}_n < \hat{\theta}_n^* - \hat{x}_\alpha \hat{\sigma}_n^* / n^{1/2} | \mathbf{X}) = 1 - \alpha.$$

How can the actual coverage probability of this procedure be estimated? The true distribution is unknown, so the actual coverage probability cannot be computed from the true distribution. Also, if the bootstrap resamples used to construct the upper confidence limit were also used to estimate the error in the coverage probability, then the estimated error would have mean 0, since $\hat{x}_\alpha$ was chosen to be the appropriate quantile of the bootstrap distribution.

Suppose the sample $X_1, \ldots, X_n$, gives an estimator $\hat{\theta}_n$. Next draw a large number B of bootstrap resamples $\mathbf{X}^{*(b)} = \{X_1^{*(b)}, \ldots, X_n^{*(b)}\}$, each giving an estimator $\hat{\theta}_n^{*(b)}$. For each resample, treat it as if it were an original sample, and consider the bootstrap distribution based on the bth resample. That is, let $\hat{F}^{*(b)}(\cdot)$ be the empirical CDF of $\mathbf{X}^{*(b)}$, and draw $X_i^{**(bc)}$ from $\hat{F}^{*(b)}(\cdot)$, $i = 1, \ldots, n$. Use this nested resample to construct a confidence bound on the parameter. In this second level bootstrap (boot^2), $\hat{\theta}_n^{*(b)}$ is the estimator computed from the sample $\mathbf{X}^{*(b)}$, and the bootstrap-t confidence limit is defined by

$$\tilde{\theta}_{un}^{*(b)} = \hat{\theta}_n^{*(b)} - \hat{\hat{x}}_\alpha \hat{\sigma}_n^{*(b)} / n^{1/2}, \tag{10.27}$$

with $\hat{\hat{x}}_\alpha$ chosen to satisfy

$$P(n^{1/2}(\hat{\theta}_n^{**(bc)} - \hat{\theta}_n^{*(b)}) / \hat{\sigma}_n^{**(bc)} > \hat{\hat{x}}_\alpha | \mathbf{X}^{*(b)}) = 1 - \alpha.$$

Note that the true value of the parameter in the "population" $\mathbf{X}$ from which the "sample" $\mathbf{X}^{*(b)}$ is drawn is $\hat{\theta}_n$, so (10.27) is really an upper confidence limit on $\hat{\theta}_n$.

Using this procedure, B upper confidence limits on $\hat{\theta}_n$ are obtained. These confidence limits can then be used to estimate the actual coverage probability of the bootstrap-t procedure. That is, calculate

$$P(\hat{\theta}_n < \tilde{\theta}_{un}^{*(b)} | \mathbf{X}),$$

or really, estimate this from the resamples, say with $\sum_{j=1}^{B} I(\hat{\theta}_n < \tilde{\theta}_{un}^{*(b)})/B$. If the estimated coverage probability is not equal to $1 - \alpha$, then the confidence level used to calculate $\hat{\hat{x}}_\alpha$ can be adjusted, until this bootstrap estimate of the coverage probability equals the nominal coverage probability. To be more precise, instead of using $\hat{\hat{x}}_\alpha$ in (10.27), use $\hat{\hat{x}}_{\alpha+\hat{t}}$, giving

$$\tilde{\theta}_{utn}^{*(b)} = \hat{\theta}_n^{*(b)} - \hat{\hat{x}}_{\alpha+\hat{t}} \hat{\sigma}_n^{*(b)} / n^{1/2},$$

with $\hat{t}$ chosen so that

$$P(\hat{\theta}_n < \tilde{\theta}_{utn}^{*(b)} | \mathbf{X}) = 1 - \alpha.$$

This procedure thus estimates a confidence level $1-\alpha-\hat{t}$ that should be used in the bootstrap-t procedure to give an actual confidence level of $1-\alpha$. The upper confidence limit on θ is then constructed by applying the bootstrap-t procedure to the original data, using the level $1-\alpha-\hat{t}$. That is, set

$$\tilde{\theta}_u = \hat{\theta}_n - \hat{x}_{\alpha+\hat{t}}\hat{\sigma}_n/n^{1/2}, \tag{10.28}$$

with $\hat{x}_{\alpha+\hat{t}}$ chosen to satisfy

$$P(n^{1/2}(\hat{\theta}_n^* - \hat{\theta}_n)/\hat{\sigma}_n > \hat{x}_{\alpha+\hat{t}}|\mathbf{X}) = 1-\alpha-\hat{t}$$

(treating $\hat{t}$ as fixed). It turns out that by using a second level of bootstrapping to adjust the confidence level, the bootstrap-t procedure improves from second order accurate to third order. In principle, one could add a third order of bootstrapping to estimate the error in this 2-stage procedure, a 4th level to look at the error in the 3 stage procedure, etc. In theory, each such additional level improves the order of the approximation. It is probably not possible to iterate this procedure to ∞ to produce exact limits, however. First, it would be prohibitively expensive in most settings. Second, the Edgeworth expansions on which the theory is based usually converge as $n \to \infty$ for a fixed number of terms, but often do not converge for fixed n as the number of terms increases. But it is this latter convergence that would be required to get exact coverage for a fixed sample size.

Example 10.6 Consider computing an upper confidence limit on the square of the mean, based on the estimator $\overline{X}_n^2$. The delta method ($g(x) = x^2$, $g'(x) = 2x$) gives that

$$n^{1/2}[\,\overline{X}_n^2 - \mu^2]/(2|\overline{X}_n|s_n),$$

where $s_n^2 = \sum_{i=1}^n (X_i - \overline{X}_n)^2/(n-1)$, is asymptotically $N(0,1)$, and hence is an approximate pivot. Consider the following data, with $n = 6$.

```
> set.seed(33)
> x <- rexp(6)
> x
[1] 0.36520032 1.10559251 2.32469911 0.21976869 2.16249816 0.01115996
```

The delta method upper 90% confidence limit is calculated by the following commands.

```
> # upper confidence limit for f1(E(X)), using the estimator f1(mean(x))
> # f2 is the derivative of f1
> #asymptotic normal UCL (delta method)
> f1 <- function(x) x^2; f2 <- function(x) 2*x
> f1(mean(x))-qnorm(.10)*abs(f2(mean(x)))*sqrt(var(x))/sqrt(length(x))
> [1] 2.154137
```

`b.ucl` below is a function to compute the bootstrap-t upper confidence limits, given an array of bootstrap resamples. Note that it computes the approximate pivot on each of the resamples, and then computes the appropriate quantile(s) of the vector of approximate pivots (`apvar` is a function to compute the variances of the rows of a matrix, which is significantly faster than `apply(xb,1,var)`).

```
> b.ucl
function(xb, x, probs)
{
## xb=matrix of bootstrap resamples,x=original data,1-probs=conf level
## returns bootstrap-t upper confidence limits of levels 1-probs.
        xbm <- apply(xb, 1, mean)
        xbv <- apvar(xb, xbm)
        xbv <- ifelse(xbv > 0, xbv, 1e-10)
        xbm <- (sqrt(length(x)) * (f1(xbm) - f1(mean(x))))/(abs(f2(xbm)) * sqrt(
                xbv))
        x5 <- quantile(xbm, prob = probs)
        f1(mean(x)) - (x5 * abs(f2(mean(x))) * sqrt(var(x)))/sqrt(length(x))
}
> apvar
function(a, am)
{
        a <- (a - am)^2
        apply(a, 1, sum)/(ncol(a) - 1)
}
> B <- 500
> xb <- sample(x,B*length(x),replace=T)
> dim(xb) <- c(B,length(x))
> b.ucl(xb,x,.10)
      10%
 2.816042
```

Thus the ordinary bootstrap-t upper confidence limit is 2.82, quite a bit larger than the asymptotic value. ($B = 500$ is a little small for estimating percentiles, but to use a second level of sampling, B cannot be too large.) To correct the bootstrap-t confidence limit using a second level of bootstrap sampling to estimate the actual coverage probability requires computing bootstrap-t confidence limits for different nominal coverage probabilities, and then searching for the nominal value that gives the correct actual coverage. To do this, the confidence limits for nominal levels 94% to 75% will be computed for each set of bootstrap^2 resamples. The following commands do this using each of the original 500 resamples as an "original" data set. (The `memory.size()` command is to monitor memory usage, which stayed under about 4MB during this run.)

```
> xbb <- matrix(-1,nrow=B,ncol=length(x))
> out <- matrix(-1,nrow=B,ncol=20)
> for (i in 1:500) {
+   xbb[1:B,1:length(xb[i,])] <- sample(xb[i,],B*length(x),replace=T)
+   out[i,] <- b.ucl(xbb,xb[i,],6:25/100)
+   print(memory.size())
+ }
```

The following commands tabulate the proportion of the 500 resamples where the upper confidence limit was $\geq \overline{X}_n^2$, which is the "true value" of the parameter in the first level of bootstrap

resamples. Thus these are estimates of the coverage probabilities for the confidence limits of nominal size 94% to 75%. The results suggest that a nominal level of about 77% to 78% should be used to get actual coverage of 90% using the bootstrap-t procedure. The 77%, 77.5% and 78% bootstrap-t upper confidence limits are given at the end of the output. These are smaller than both the original delta method confidence limit and the uncorrected bootstrap-t limit.

```
> w <- matrix(-1,20,2)
> for ( i in 1:20) w[i,] <- c((95-i)/100,table(out[,i]>=f1(mean(x)))[2]/B)
> w
      [,1]  [,2]
 [1,] 0.94 0.978
 [2,] 0.93 0.974
 [3,] 0.92 0.964
 [4,] 0.91 0.946
 [5,] 0.90 0.940
 [6,] 0.89 0.928
 [7,] 0.88 0.926
 [8,] 0.87 0.924
 [9,] 0.86 0.924
[10,] 0.85 0.924
[11,] 0.84 0.924
[12,] 0.83 0.924
[13,] 0.82 0.924
[14,] 0.81 0.924
[15,] 0.80 0.920
[16,] 0.79 0.914
[17,] 0.78 0.906
[18,] 0.77 0.892
[19,] 0.76 0.872
[20,] 0.75 0.862
> b.ucl(xb,x,c(.22,.225,.23))
    22.0%    22.5%    23.0%
 1.841403 1.815584 1.782364
```

□

The iterated bootstrap is not often used because of the computational expense. It has been proposed most frequently, though, in connection with Efron's percentile method. Recall that Efron's percentile method transforms properly and stays within range restrictions on the parameter, but is not directly motivated as a frequentist confidence set procedure, and often has poor coverage. Using a second level of bootstrap sampling to correct the coverage probability then gives a second order accurate procedure, which automatically transforms properly and stays within range restrictions on parameters. Beran (1987) proposed a method called prepivoting, slightly different than the method described here, for getting second order accurate confidence limits using a double bootstrap procedure.

10.6 Exercises

Exercise 10.1 Bootstrap iteration is a general technique that can (in principle) be applied to improve other types of bootstrap estimates. In this exercise its application to estimating bias is discussed.

As discussed in Section 10.2 above, the bias of an estimator $\hat{\theta}_n$ of θ is

$$\mathrm{E}(\hat{\theta}_n) - \theta,$$

and in accordance with the general bootstrap principle, the bootstrap estimate of bias is

$$\mathrm{E}(\hat{\theta}^*_n|\mathbf{X}) - \hat{\theta}_n.$$

It is assumed here that $\hat{\theta}_n$ is the true value of the parameter in the bootstrap distribution, and that the expectation over the bootstrap distribution can be computed exactly, or estimated with a sufficiently large number of bootstrap resamples to make it essentially exact, so the control variate type correction discussed in Section 10.2 is not needed. The bias corrected estimate is then

$$\hat{\theta}_n - [\mathrm{E}(\hat{\theta}^*_n|\mathbf{X}) - \hat{\theta}_n] = 2\hat{\theta}_n - \mathrm{E}(\hat{\theta}^*_n|\mathbf{X}).$$

As with confidence intervals and coverage probabilities, a second level of bootstrapping can be used to estimate the bias in the bias corrected estimator. As before $\mathbf{X}^{*(b)} = \{X_1^{*(b)}, \ldots, X_n^{*(b)}\}$ is a resample from the original data, and $X_1^{**(bc)}, \ldots, X_n^{**(bc)}$ a resample from $\mathbf{X}^{*(b)}$. The distribution of the second level of resamples for each first level resample provides an estimate

$$\mathrm{E}(\hat{\theta}^{**(bc)}_n|\mathbf{X}^{*(b)}) - \hat{\theta}^{*(b)}_n$$

of the bias of $\hat{\theta}^*_n$ (the difference between $\hat{\theta}^{*(b)}_n$ and $\hat{\theta}^*_n$ is that the former refers to a particular resample). The actual bias (conditional on $\mathbf{X}$) of $\hat{\theta}^*_n$ is

$$\mathrm{E}(\hat{\theta}^*_n|\mathbf{X}) - \hat{\theta}_n.$$

Thus an estimate of the bias in the bootstrap estimate of bias is

$$\mathrm{E}(\hat{\theta}^{**(bc)}_n|\mathbf{X}^{*(b)}_n) - \hat{\theta}^{*(b)}_n - [\mathrm{E}(\hat{\theta}^*_n|\mathbf{X}) - \hat{\theta}_n].$$

Taking the expectation over $\mathbf{X}^{*(b)}|\mathbf{X}$ reduces this expression to

$$\mathrm{E}(\hat{\theta}^{**(bc)}_n|\mathbf{X}) - 2\mathrm{E}(\hat{\theta}^*_n|\mathbf{X}) + \hat{\theta}_n.$$

An improved estimate of bias is then given by

$$\mathrm{E}(\hat{\theta}^*_n|\mathbf{X}_n) - \hat{\theta}_n - [\mathrm{E}(\hat{\theta}^{**(bc)}_n|\mathbf{X}) - 2\mathrm{E}(\hat{\theta}^*_n|\mathbf{X}) + \hat{\theta}_n] = 3\mathrm{E}(\hat{\theta}^*_n|\mathbf{X}) - \mathrm{E}(\hat{\theta}^{**(bc)}_n|\mathbf{X}) - 2\hat{\theta}_n.$$

The double bias corrected estimator is

$$\hat{\theta}_n - [3\mathrm{E}(\hat{\theta}^*_n|\mathbf{X}) - \mathrm{E}(\hat{\theta}^{**(bc)}_n|\mathbf{X}_n) - 2\hat{\theta}_n].$$

As with confidence intervals, this process can be further iterated.

Many asymptotically normal estimators have a bias of $O(n^{-1})$. Correcting using the bootstrap estimate of bias typically reduces this to $O(n^{-2})$. The double bootstrap correction reduces to $O(n^{-3})$, and each further iteration reduces the order by a factor of n^{-1}. This is discussed in Sections 1.4 and 1.5 of Hall (1992).

The following is a simple example where exact calculations can be given. Suppose $X_1, \ldots, X_n$ are iid with mean μ and variance σ^2, and suppose $\theta = \mu^2$ is estimated with $\hat{\theta}_n = \overline{X}_n^2$. Also let $\hat{\sigma}_n^2 = \sum_i (X_i - \overline{X}_n)^2/n$.

(a) Show $\mathrm{E}(\hat{\theta}_n) = \mu^2 + \sigma^2/n$.

(b) Show that the bias corrected bootstrap estimator is

$$\overline{X}_n^2 - \hat{\sigma}_n^2/n.$$

(c) Give an exact formula for the bias of the estimator in (b).

(d) Show that the kth iterated bootstrap gives a bias^k corrected estimator of the form

$$\overline{X}_n^2 - (n^{-1} + n^{-2} + \cdots + n^{-k})\hat{\sigma}_n^2.$$

(e) Give an exact formula for the bias of the estimator in (d), and show that it is $O(n^{-(k+1)})$.

(f) Identify the limit as $k \to \infty$ of the estimator in (d), and show this limit is unbiased for μ^2.

10.7 References

Beran R (1987). Prepivoting to reduce level error of confidence sets. *Biometrika*, **74**:457–468.

Efron B (1983). Estimating the error rate of a prediction rule: improvement on cross-validation. *Journal of the American Statistical Association*, **78**:316–331.

Efron B and Tibshirani RJ (1993). *An Introduction to the Bootstrap*. Monographs on Statistics and Probability vol. 57. Chapman & Hall, London.

Hall P (1986). On the bootstrap and confidence intervals. *Annals of Statistics*, **14**:1431–1452.

Hall P (1989). Antithetic resampling for the bootstrap. *Biometrika*, **76**:713–724.

Hall P (1992). *The Bootstrap and Edgeworth Expansion*. Springer-Verlag, New York.