An Introduction to Computational Statistics

STAT 604, Fall 1999

H. Joseph Newton
Department of Statistics
Texas A&M University

Table of Contents

CHAPTER 1

How are Numbers Represented?

Most programming languages have data types for representing integers and numbers that have decimal parts. For example, Fortran has two byte and four byte integers and four byte and eight byte real numbers (also called single and double precision). In C, these four types are called `int`'s, `long`'s, `float`'s, and `double`'s. In this section we describe how these numbers are actually stored in the computer. We describe the situation of Sun workstations and IBM PC's and compatibles.

Integers

Suppose that i is a two byte integer and let $b_{15}, \ldots, b_1, b_0$ represent the 16 bits in the representation of i, that is, b_{15} is the most significant bit and b_0 is the least significant bit. Then:

- If $i \geq 0$, then $b_{15} = 0$ and the remaining bits are the binary representation of i, that is, b_i is the coefficient of 2^i.
- If $i < 0$, then to get the values of the b's:
  1. Write the binary representation of $-i$.
  2. Complement the bits, that is, change the 0's to 1's and the 1's to 0's.
  3. Add 1 to the result of step 2.

For example, if $i = -149$ we have (writing b_{15} through b_0 from left to right):

```
149          = 128 + 16 + 4 +1
             = 0000 0000 1001 0101
complement:    1111 1111 0110 1010
add 1:         1111 1111 0110 1011
bytes:            255       107
```

Four byte integers are done the same way except they have 32 bits. Note that two byte integers can range from $-32,768$ to 32,767, while four byte integers range from -2×10^9 to 2×10^9.

Real (Floating Point) Numbers

For a four byte real number, the most significant bit (b_{31}) for a number x is a sign bit, that is, it is 0 if $x \geq 0$ and 1 if $x < 0$. The remaining 31 bits are divided into a set b_{30} through b_{23} of eight bits to represent the exponent of x while the remaining 23 bits represent the fractional part of x, which is also called the mantissa or significand of x. To determine the representation of x, the following steps are followed (consider $x = -197.625$ for example):

1. Write the binary representation of $|x|$, for example,

```
197.625 D = 128 + 64 + 4 + 1 + 1/2 + 1/8
          = 11000101.101 B
          = 1.1000101101 x 2\^{\ }7 B
```

2. The bits b_{30} through b_{23} are the binary representation of the exponent plus 127 (the 127 is called a bias):

```
134 = 128 + 4 + 2
    = 10000110
```

3. The bits b_{22} through b_0 are the bits to the right of the binary point in step 1 (note that we don't need to store the fact that there is a 1 to the left of the binary point):

```
10001011010000000000000
```

Thus, the representation of -197.625 is:

```
-197.625 = 1100 0011  0100 0101  1010 0000  0000 0000
         =    195         69         160         0
```

This makes it so that

$$8.43 \times 10^{-37} \leq |x| \leq 3.37 \times 10^{38},$$

with about 6 or 7 decimal place accuracy.

For an eight byte real, b_{63} is a sign bit, the next 11 bits are the binary representation of the exponent plus 1023 (which is $2^{10} - 1$), while the last 52 bits are the significand. This makes it so that

$$4.19 \times 10^{-307} \leq |x| \leq 1.67 \times 10^{308},$$

with about 15 or 16 decimal place accuracy.

Character Variables

In Fortran, each character in a character variable occupies one byte of memory so that there can be at most 256 such characters (each character has a numeric code associated with it called its ASCII code and these codes range from 0 to 255). On most computers, the first 32 characters are reserved for special purposes (such as signals to printers), while the second 128 characters are reserved for special purposes. Thus characters numbered 32 through 127 are the ones commonly used in creating text. A file that only has these characters is called an "ASCII file," and any file that has characters other than these cannot be guaranteed to be handled properly in going from one type of computer to another (one of the compelling reasons to use TEX is that it uses only ASCII files which are perfectly portable from one computer to another). In the table below we give the ASCII characters having codes 32 through 127. Note that the capital letters have ASCII codes 65 through 90 while the lower case letters are from 97 through 122.

```
        0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
------------------------------------------------------
 32        !  "  #  $  %  &  '  (  )  *  +  ,  -  .  /
 48     0  1  2  3  4  5  6  7  8  9  :  ;  <  =  >  ?
 64     @  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O
 80     P  Q  R  S  T  U  V  W  X  Y  Z  [  \  ]  ^  _
 96     `  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o
112     p  q  r  s  t  u  v  w  x  y  z  {  |  }  ~
```

An Example

To illustrate what we have been discussing, consider the following Fortran program

```
      integer*2 i
      real*4 x
      character*1 ci(2),cx(4)

      equivalence (i,ci),(x,cx)

      i=-149
      write(*,10) i,(ichar(ci(j)),j=1,2)
 10   format(1x,3i10)

      x=-197.625
      write(*,20) x,(ichar(cx(j)),j=1,4)
 20   format(1x,f10.4,4i10)

      stop
      end
```

which uses two uncommon features of Fortran: 1) the intrinsic function `ichar` which returns the ASCII code of a character, and 2) the `equivalence` statement which tells the compiler to store the two variables it names in the same location. This allows us to see how the numbers we have used as examples (-149 and -197.625) are represented. On a PC, the following results are obtained from the program:

```
      -149       107       255
 -197.6250         0       160        69       195
```

whereas on a Sun Sparcstation we get

```
      -149       255       107
 -197.6250         0       160        69       195
 -197.6250       195        69       160         0
```

This shows that the bytes are stored in reverse order on the two platforms.

CHAPTER 2

Some Basic Algorithms

The Quicksort Algorithm

The problem of sorting (arranging a set of numbers or character variables $X(1), \ldots, X(n)$ into non-decreasing order) occurs in many places in statistics. The basic operations in sorting are comparisons and swaps (exchanges) of elements of the array. Most naive sorting algorithms (such as the infamous bubble sort, see below) require proportional to n^2 operations on the average to sort a vector of n randomly generated distinct numbers.

Bubble Sort

Perhaps the most famous sorting algorithm is bubble sort since it is simple to code and has a catchy name. Unfortunately bubble sort is also a terribly slow method unless n is very small (less than 20 or so). The algorithm starts by defining a boundary $k = n$ and then keeps going through the data for $i = 1, \ldots, k-1$, interchanging $X(i)$ and $X(i+1)$ if they are out of order and each time setting k to be the largest value of i for which a swap is made. The algorithm stops if a pass through the data is made with no swaps.

Quicksort

Quicksort starts by splitting (partitioning) $X(1), \ldots, X(n)$ into three pieces: 1) one element (called the splitting element) which is placed in its proper place in the final sorted array, 2) a piece consisting of all of the elements in the array that are less than or equal to the splitting element (these are placed to the left of the splitting element), and 3) a piece consisting of all of the elements that are greater than the splitting element (these are placed to the right of the splitting element). All of this is done "in place", that is, in the same array that we started with. Once this splitting is done, the algorithm continues by splitting each of the two pieces, and then splitting the resulting four pieces, etc. until there are no pieces left to split (we know this will happen as every split results in an element being in its proper place). The algorithm keeps track of what pieces need to be split by keeping a counter of how many there are as well as two arrays containing the left and right "boundaries" of the pieces. There are two other basic rules that are used in the algorithm: 1) pieces of size three or fewer are sorted, not split (it's faster to sort them than to split them), and 2) the next piece to be split is the shorter of the two pieces obtained from the previous splitting (this makes it so that the "stack", that is, the arrays containing the boundaries of the pieces still to be split, need only be of length k where k is the smallest integer such that $2^k > n$). Notice that it could happen that the splitting element is the largest of smallest element in a piece in which case the splitting only results in a single piece.

Choosing the Splitting Element and Doing the Splitting

Ideally, one would like the splitting element to be approximately the median of the piece to split as then the piece would be split approximately in half. There is no generally agreed upon method for choosing the splitting element, but one popular choice is as the median of the first, last, and "middle" (that is, the ones having indexes i, j, and $[(i+j)/2]$ where i and j are the boundaries of the piece. Another possibility is to just choose one of the elements of the piece at random.

The actual splitting process is difficult to describe. Consult the code below for one possible method.

Extensions

When data come in pairs $(X(i), Y(i)), i = 1, \ldots, n$, it is often desired to sort one of the variables while preserving the pairing. One simple method for doing this is to pass both arrays to the algorithm and merely replace the `SWAP` routine by a `SWAP2` routine that swaps the elements of the second array whenever elements of the first array are swapped. Note that `QKSORT` is written so that swapping is done only in the `SWAP` routine, which makes this extension simple.

Another common task is to find the ranks of the elements of an array. One way to do this is to do the paired version of the quicksort algorithm with an array of X's together with an array of indices (which starts out with ith element being i). The resulting array of indices can then be used to easily find the ranks.

```
      subroutine qsort(x,n)
c**********************************************************************
c
c   Quicksort a real array x of length n.
c
c*********************************************************************
c
      double precision x(n)
      integer ll(20),lr(20)
c
c   ns is # of pieces left to partition
c
c   ll and lr are left and right borders of pieces
c
      ns=1
      ll(1)=1
      lr(1)=n
c
c   After splitting we come here. Stop when no pieces left.
c
    5 if(ns.eq.0) go to 99
c
      i=ll(ns)
      j=lr(ns)
      nl=j-i+1
c
c   Sort pieces of size 1 or 2:
c
      if(nl.le.2) then
             ns=ns-1
             if(nl.eq.2.and.x(i).gt.x(j)) call swap(x(i),x(j))
             go to 5
      endif
c
c   Sort first, ``middle'' and last elements:
c
      nm=(i+j)/2
      if(x(i).gt.x(nm)) call swap(x(i),x(nm))
      if(x(nm).gt.x(j)) call swap(x(nm),x(j))
      if(x(i).gt.x(nm)) call swap(x(i),x(nm))
c
c   If piece of size 3, it's now sorted
c
      if(nl.eq.3) then
             ns=ns-1
             go to 5
      endif
c
c   Put middle (target) into 1st element and keep a copy for comparisons
c
```

```
      ax=x(nm)
      call swap(x(i),x(nm))
c
c   Now we'll look for 1st one from left (starting with 2nd)  > target
c   and first from right < target. If no such pair we end up at 20.
c
      itemp=i
      i=i+1
c
 10   if(x(i).le.ax) then
             if(i.eq.j) go to 20
             i=i+1
             go to 10
      endif
c
 15   if(x(j).ge.ax) then
             if(i.eq.j) go to 20
             j=j-1
             go to 15
      endif
c
c   Can only get to here if we found a pair to swap
c
      call swap(x(i),x(j))
c
c   Only go back to look for another pair if there's a chance to get one
c
      if(j-i.le.1) then
             go to 20
      else
             i=i+1
             j=j-1
             go to 10
      endif
c
c   Get to here when no more pairs to swap. i might be 1 space too far
c   to right
c
 20   if(x(i).gt.ax) i=i-1
      call swap(x(itemp),x(i))
c
c   Now put shorter piece at end of ll and lr (this guarantees small
c   number of pieces at any one time). A piece might be empty, but that
c   is taken care of above.
c
      nleft=i-ll(ns)
      nright=lr(ns)-i
      ns=ns+1
      if(nleft.le.nright) then
             ll(ns)=ll(ns-1)
             ll(ns-1)=i+1
             lr(ns)=i-1
      else
             lr(ns)=lr(ns-1)
             lr(ns-1)=i-1
             ll(ns)=i+1
      endif
      go to 5
c
 99   continue
      return
      end
      subroutine swap(a,b)
c*********************************************************************
c
c   Swap two reals.
c
c********************************************************************
c
      double precision a,b,c
      c=b
      b=a
      a=c
c
      return
      end
```

Vector Operations

The inner product of n-dimensional vectors x and y is given by

$$< x, y >= x^T y = y^T x = \sum_{i=1}^{n} x_i y_i,$$

and x and y are said to be orthogonal (denoted by $y \perp x$) if $< x, y >= 0$. By the operation of "making y orthogonal to x" we mean the finding a vector $z = y - \alpha x$, where the number α is chosen so that $z \perp x$, that is, so that

$$(y - \alpha x)^T x = (y^T - \alpha x^T)x = y^T x - \alpha x^T x = 0,$$

that is $\alpha =< y, x > / < x, x >$.

Matrix Operations

Most matrix operations in statistical computing are concerned with either an $n \times m$ matrix X having columns $x_1, \ldots, x_m$ or with a symmetric positive semidefinite $m \times m$ matrix V. The case where $\mathrm{V} = \mathrm{X}^T\mathrm{X}$ is particularly important. In almost every situation, n is bigger than m, that is, X has more columns than rows. In this case, the matrix X is said to be of full rank if all of its columns are linearly independent, that is, no column of X can be written as a linear combination of the remaining columns. This can be expressed succinctly by saying that for any m dimensional vector h whose elements are not all zero, then $\mathrm{Xh} \neq 0$.

An $(m \times m)$ matrix V is said to be positive definite or positive semidefinite if for any nonzero m-dimensional nonzero vector h, we have $\mathrm{h}^T\mathrm{Vh} > 0$ or $\mathrm{h}^T\mathrm{Vh} \geq 0$ respectively. If $\mathrm{V} = \mathrm{X}^T\mathrm{X}$, then its (i, j)th element is the inner product $< x_i, x_j >=< x_j, x_i >$ and thus it is symmetric. Further, for any nonzero vector h,

$$\mathrm{h}^T\mathrm{Vh} = \mathrm{h}^T\mathrm{X}^T\mathrm{XH} = (\mathrm{Xh})^T(\mathrm{Xh}) = \mathrm{z}^T\mathrm{z} =< \mathrm{z}, \mathrm{z} >,$$

which is nonnegative for any X, and in fact, is positive if and only if X is of full rank.

In statistics it often necessary to find what is called the square root of a positive definite matrix.

Definition. *The square root of a symmetric, positive definite* $(m \times m)$ *matrix* V *is an* $(m \times m)$ *matrix* A *satisfying* $\mathrm{V} = \mathrm{AA}^T$. *We denote such a matrix by* $\mathrm{V}^{1/2}$. *The inverse square root of* V *is the inverse of* $\mathrm{V}^{1/2}$ *and is denoted by* $\mathrm{V}^{-1/2}$. *The transposes of* V^{-1} and $\mathrm{V}^{-1/2}$ *are denoted by* V^{-T} *and* $\mathrm{V}^{-T/2}$ *respectively.*

In this section we consider a variety of operations on matrices like X and $\mathrm{X}^T\mathrm{X}$. We will answer a wide variety of questions, including

1. How do we check whether X is of full rank?
2. How do we check whether V is positive definite?
3. How do we find the square root of V?
4. How do we find the determinant of V?
5. How do we solve a system of equations $\mathrm{V}x = \mathrm{b}$?

The Modified Cholesky Decomposition

The next theorem shows that there exists a unique matrix square root for any positive definite matrix and also shows how to find it. Note that a triangular matrix is called unit if it has ones on the main diagonal.

Theorem 1. MODIFIED CHOLESKY DECOMPOSITION

Let V *be a symmetric* $(m \times m)$ *matrix. Then*

a) V *is positive definite if and only if there exists a unique unit lower triangular* $(m \times m)$ *matrix* L *and a unique diagonal* $(m \times m)$ *matrix* D *having positive diagonal elements, such that*

$$\mathrm{V} = \mathrm{LDL}^T.$$

b) This factorization is called the modified Cholesky decomposition (MCD) of V*, and if the decomposition exists we can calculate the elements of* L *and* D *one row at a time by* $D_{11} = V_{11}$ *and for* $i = 2, \ldots, m$*:*

$$L_{ij} = \frac{V_{ij} - \sum_{l=1}^{j-1} L_{il} D_{ll} L_{jl}}{D_{jj}}, \qquad j = 1, \ldots, i-1$$

$$D_{ii} = V_{ii} - \sum_{l=1}^{i-1} D_{ll} L_{il}^2.$$

c) The MCD of V *is nested; that is, if* V_k, L_k, *and* D_k *represent the upper left-hand* $(k \times k)$ *parts of* V, L, *and* D, *respectively, then*

$$\mathrm{V}_k = \mathrm{L}_k \mathrm{D}_k \mathrm{L}_k^T, \qquad k = 1, \ldots, m,$$

and thus for any k *greater than or equal to* i *and* j*, the* (i,j)*th elements of* V_k, L_k, *and* D_k *can be denoted by* V_{ij}, L_{ij}, and D_{ij}, *respectively.*

d) The unique square root of a positive definite matrix V *is given by* $\mathrm{V}^{1/2} = \mathrm{LD}^{1/2}$ *where* $\mathrm{D}^{1/2} = \mathrm{Diag}(D_{11}^{1/2}, \ldots, D_{nn}^{1/2})$.

e) The inverse square root of a positive definite matrix V *is given by* $\mathrm{V}^{-1/2} = \mathrm{D}^{-1/2}\mathrm{L}^{-1}$ *where* $\mathrm{D}^{-1/2} = \mathrm{Diag}(D_{11}^{-1/2}, \ldots, D_{nn}^{-1/2})$.

Solving Systems of Equations and Checking for Positive Definiteness

The Gram-Schmidt Decomposition

An important operation in many areas of mathematics and statistics is to find what is called an orthogonal basis for a matrix; that is, from one set of vectors, find a new set by some linear transformation such that the inner product of any two different new vectors is zero. We will use the Gram-Schmidt decomposition (see Clayton (1971)) to accomplish this aim.

Theorem 2 GRAM-SCHMIDT DECOMPOSITION

If X *is an* $(n \times m)$ *matrix having full rank* m*, then there exists an* $(n \times m)$ *orthogonal matrix* Q *(that is,* $\mathrm{Q}^T\mathrm{Q}$ *is diagonal) and a unit upper triangular* $(m \times m)$ *matrix* R *such that*

$$\mathrm{X} = \mathrm{QR}.$$

This decomposition of X *is called its Gram-Schmidt decomposition (GSD).*

Note that $\mathrm{X}^T\mathrm{X} = \mathrm{R}^T\mathrm{Q}^T\mathrm{QR} = \mathrm{R}^T\mathrm{VR}$ where R^T is unit lower triangular and V is diagonal with positive diagonal elements. Since the modified Cholesky decomposition $\mathrm{X}^T\mathrm{X} = \mathrm{LDL}^T$ of $\mathrm{X}^T\mathrm{X}$ is unique, we have that $\mathrm{R}^T = \mathrm{L}$ and $\mathrm{V} = \mathrm{D}$. The Modified Gram-Schmidt decomposition algorithm for an $n \times m$ matrix X having columns $x_1, \ldots, x_m$ consists of a series of $m-1$ steps. The nature of these steps is such that the algorithm can be done in place, that is, the matrix Q will be produced in the matrix X.

For $i = 1, \ldots, m-1$, the ith step consists of making the columns $x_{i+1}, \ldots, x_m$ orthogonal to x_i. As each new column is produced it replaces the original column in X. The coefficients that are produced by the orthogonalizations are placed in $R_{i,i+1}, \ldots, R_{im}$.

The Sweep Operator

Many of the computational and theoretical results in regression, multivariate analysis, and time series can be expressed succinctly using what is called the sweep operator.

Definition. *Let* A *be an* $(n \times n)$ *matrix. The process of sweeping* A *on its* kth *diagonal element, denoted by* $\mathrm{B} = \mathrm{SWEEP}(k)\mathrm{A}$, *is the process of forming the matrix* B *by (assuming* $A_{kk} \neq 0$*):*

$$B_{kk} = \frac{1}{A_{kk}}$$

$$B_{ik} = -\frac{A_{ik}}{A_{kk}}, \qquad i \neq k \qquad \textit{(kth column)}$$

$$B_{kj} = \frac{A_{kj}}{A_{kk}}, \qquad j \neq k \qquad \textit{(jth row)}$$

$$B_{ij} = A_{ij} - \frac{A_{ik}A_{kj}}{A_{kk}}, \qquad i \neq k, j \neq k.$$

If $\mathrm{B}_1 = \mathrm{SWEEP}(k_1)\mathrm{A}$, $\mathrm{B}_2 = \mathrm{SWEEP}(k_2)\mathrm{B}_1$, ..., $\mathrm{B} = \mathrm{SWEEP}(k_r)\mathrm{B}_{r-1}$, *we write* $\mathrm{B} = \mathrm{SWEEP}(k_1, \ldots, k_r)\mathrm{A}$, *and say that* B *is the result of sweeping* A *on its diagonals* $k_1, \ldots, k_r$.

The sweep operator has many useful properties as we see in the next theorem.

Theorem 3 PROPERTIES OF THE SWEEP OPERATOR

Let

$$\mathrm{A} = \begin{bmatrix} \mathrm{B} & \mathrm{C} \\ \mathrm{D} & \mathrm{E} \end{bmatrix},$$

where B, C, D, *and* E *are* $(r \times r)$, $(r \times s)$, $(s \times r)$, *and* $(s \times s)$ *respectively. Then*

a) $\mathrm{SWEEP}(k_1, \ldots, k_p)\mathrm{A}$ *can be found by sweeping the diagonals in any order, for example,*

$$\mathrm{SWEEP}(k_1, k_2)\mathrm{A} = \mathrm{SWEEP}(k_2, k_1)\mathrm{A}.$$

b) $\mathrm{SWEEP}(k, k)\mathrm{A} = \mathrm{A}$*; that is, sweeping a second time on a given diagonal undoes the effect of a previous sweeping on that diagonal.*

c) $\mathrm{SWEEP}(1,\ldots,r)\mathrm{A} = \begin{bmatrix} \mathrm{B}^{-1} & \mathrm{B}^{-1}\mathrm{C} \\ -\mathrm{DB}^{-1} & \mathrm{E}-\mathrm{DB}^{-1}\mathrm{C} \end{bmatrix}$ *if* B *is nonsingular.*

d) $\mathrm{SWEEP}(r+1,\ldots,r+s)\mathrm{A} = \begin{bmatrix} \mathrm{B}-\mathrm{CE}^{-1}\mathrm{D} & -\mathrm{CE}^{-1} \\ \mathrm{E}^{-1}\mathrm{D} & \mathrm{E}^{-1} \end{bmatrix}$ *if* E *is nonsingular.*

e) $\mathrm{SWEEP}(1,\ldots,r+s)\mathrm{A} = \mathrm{A}^{-1}$

$$= \begin{bmatrix} \mathrm{B}^{-1}+\mathrm{B}^{-1}\mathrm{C}(\mathrm{E}-\mathrm{DB}^{-1}\mathrm{C})^{-1}\mathrm{DB}^{-1} & -\mathrm{B}^{-1}\mathrm{C}(\mathrm{E}-\mathrm{DB}^{-1}\mathrm{C})^{-1} \\ -(\mathrm{E}-\mathrm{DB}^{-1}\mathrm{C})^{-1}\mathrm{DB}^{-1} & (\mathrm{E}-\mathrm{DB}^{-1}\mathrm{C})^{-1} \end{bmatrix}$$

if B, E, *and* $(\mathrm{E}-\mathrm{DB}^{-1}\mathrm{C})^{-1}$ *are nonsingular.*

The sweep operator can be used to solve many theoretical and computational problems. We consider two examples of its use. First, the basic quantities for the regression of y on X (where X is $(n \times m)$) can be found by applying property (c)

$$\mathrm{SWEEP}(1,\ldots,m) \begin{bmatrix} \mathrm{X}^T\mathrm{X} & \mathrm{X}^T\mathrm{y} \\ \mathrm{y}^T\mathrm{X} & \mathrm{y}^T\mathrm{y} \end{bmatrix} = \begin{bmatrix} (\mathrm{X}^T\mathrm{X})^{-1} & \hat{\boldsymbol{\beta}} \\ -\hat{\boldsymbol{\beta}}^T & \mathrm{RSS} \end{bmatrix}.$$

To illustrate the theoretical utility of sweep, notice for the matrix A in Theorem 3 that sweeping on all of its diagonals is the same as sweeping the matrix $\mathrm{SWEEP}(r+1,\ldots,r+s)\mathrm{A}$ on diagonals $1,\ldots,r$. Thus the upper left-hand corner in part (e) is the inverse of the upper left-hand corner in part (d) by part (c). Doing a similar calculation with $\pm\mathrm{E}^{-1}$ replacing E in A gives the well-known matrix inversion lemma (see Rao (1973), p. 33 for example).

Theorem 4 THE MATRIX INVERSION LEMMA

Let B *and* E *be nonsingular* $(r \times r)$ *and* $(s \times s)$ *matrices and* C *and* D *be* $(r \times s)$ *and* $(s \times r)$ *matrices. Then*

$$(\mathrm{B} \pm \mathrm{CED})^{-1} = \mathrm{B}^{-1} \mp \mathrm{B}^{-1}\mathrm{C}(\mathrm{E}^{-1} \pm \mathrm{DB}^{-1}\mathrm{C})^{-1}\mathrm{DB}^{-1}.$$

This formula is used extensively in recursive regression, that is, when we consider adjusting estimators for the addition or deletion of observations. It can also be used to motivate the Kalman Filter Algorithm.

Finding Zeros of Functions

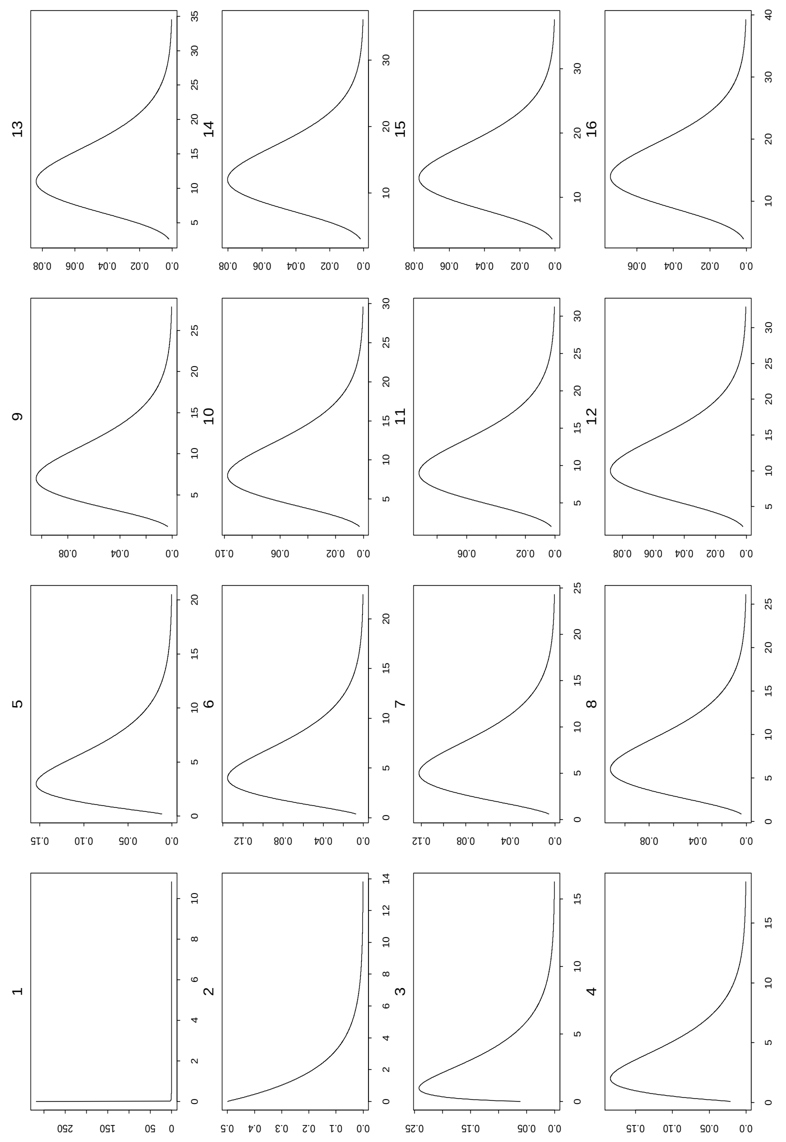
In this section we consider two standard methods for finding a zero of a function f. Note that these methods can be used to find a relative min or max of f by applying them to the derivative of f.

The first method is called Newton's method and consists of the the iterative process

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \qquad i = 0, 1, 2, \ldots,$$

given a starting value x_0. In most cases, if the starting value is close to the zero, the convergence of the process is very rapid.

An alternative method called bisection is very effective if one can bracket a zero of f in an interval $[l, u]$. One starts by checking whether the zero is in $[l, m]$ or $[m, u]$, where $m = (l + u)/2$ is the midpoint of $[l, u]$



χ^2 pdfs for degrees of freedom $\nu = 1, 2, \ldots, 16$.

(the zero is in the left or right half of the interval depending on whether the sign of $f(l)f(m)$ is negative or positive, respectively). If the zero is in the left (right) half of the interval, one sets $u = m$ $(l = m)$ and repeats the process. This continues until the length of the bracketing interval becomes very close to zero.

An example that combines Newton's method and bisection is as follows. A χ^2_ν pdf for $\nu \geq 3$ is unimodal (see the set of graphs of the χ^2 pdf). For a given ν and probability $\alpha < 0.5$, we want to find the value f_α of the pdf

$$f_{\chi^2}(x; \nu) = \frac{1}{2^{\nu/2}\Gamma(\nu/2)} e^{-x/2} x^{(\nu/2)-1}, \quad x > 0,$$

(and the two values x_l and x_u of x that make the pdf equal to f_α) so that the total area under the pdf to the left of x_l and the to the right of x_u is α (this is used in finding the $100(1-\alpha)\%$ confidence interval for the variance σ^2 of a normal population so that the interval has smallest width). Note that this process consists of raising and lowering a horizontal line cutting the pdf until the area in the tails is α. For a specified value of the pdf, we use Newton's method to find x_l and x_u using starting values equal to the $\alpha/2$ and $1 - (\alpha/2)$ quantiles of the distribution. We can bracket the value of the pdf by a small value that will work for all values of ν (like 0.001) and its value at the 0.25 quantile of the distribution.

The Newton-Raphson Procedure

If we want to minimize the scalar function $L(\theta)$ of the r dimensional argument θ and we have an initial guess θ_0 for the value θ^* of θ maximizing L, then the Newton-Raphson algorithm calculates the vectors θ_1, θ_2, and so on until they converge to θ^* (or possibly don't converge) by

$$\theta_{i+1} = \theta_i - H_i^{-1} g_i, \qquad i \geq 0,$$

where g_i and H_i are called the gradient vector and Hessian matrix of L and are the vector of first derivatives of L with respect to each of the r elements of θ and the $r \times r$ matrix of second derivatives, respectively. We can also write this process as

$$\theta_{i+1} = \theta_i - \delta_i, \qquad \text{where } H_i \delta_i = g_i,$$

since typically it is better to solve a system of equations rather than finding an inverse and doing the multiplication.

The Newton-Raphson procedure is the multidimensional version of Newton's root finding algorithm and actually will find a critical point of L, that is, a value of θ for which the elements of the gradient vector are all zero. Such a point can be a maximum or a minimum and care must be taken to ensure that a critical point is in fact what one wants. We will consider minimization since if θ^* is a minimum, the matrices H_1, H_2, and so on can be expected to be positive definite and we can use the Modified Cholesky Decomposition method of solving the system $H_i \delta_i = g_i$.

CHAPTER 3

Algorithms for Distributions

In this chapter we discuss calculating the probability density and mass functions and the cdf and quantile functions for a wide variety of distributions as well as how to generate observations from these distributions. The distributions considered are all listed in a table at the end of the chapter.

The chapter also makes extensive use of the gamma and beta functions and the incomplete gamma and beta functions defined at the beginning of the table of distributions. We will use f, F, and Q to denote the pdf (or pmf), cdf, and quantile functions, respectively. We will put the name of the random variable as a subscript and in parentheses we'll put the argument of the function followed by a semicolon and then the parameters of the distribution. Thus for example, $f_{\chi^2}(x;\nu)$ denotes the pdf of the χ^2 distribution with ν degrees of freedom.

Calculating Mass and Density Functions

Calculating probability mass functions and density functions must be done with care primarily because many of them are expressed as the ratio of quantities that can be very large. The most common example of this is in evaluating factorials or the gamma function (which is the same thing for integer arguments since $\Gamma(n+1) = n!$. For example, in single precision arithmetic, 34! is the largest factorial that can be computed before overflow occurs. Thus most mass or density functions are calculated by exponentiating the logarithm of the function. For example, the pdf of the χ^2 distribution can be written

$$\begin{aligned} f_{\chi^2}(x;\nu) &= \frac{1}{2^{\nu/2}\Gamma(\nu/2)} e^{-x/2} x^{(\nu/2)-1}, \quad x > 0, \\ &= \exp(((\nu/2) - 1)\log x - x/2 - (\nu/2)\log 2 - \log\Gamma(\nu/2)) \end{aligned}$$

This last equation illustrates that we'll often need to be able to calculate the log of the gamma function.

Calculating the Log of the Gamma Function

One way to do this is ("Numerical Recipes", eq. 6.1.5)

$$\log\Gamma(z+1) = (z+\frac{1}{2})\log(z+\gamma+\frac{1}{2}) - (z+\gamma+\frac{1}{2}) + \log(\sqrt{2\pi}) + \log\left[c_0 + \sum_{j=1}^{n}\frac{c_j}{z+j}\right],$$

for certain values of γ, n, and $c_0, c_1, \ldots, c_n$.

Recursive Calculation of Probability Mass Functions

One interesting simplification for many probability mass functions is that $f(x) = \Pr(X = x)$ can be easily calculated recursively in x. As an example, for the binomial distribution with parameters n and p, we have

$$\frac{\Pr(X = x+1)}{\Pr(X = x)} = \frac{n-x}{x+1}\frac{p}{1-p}, \qquad 0 \le x < n,$$

which, if we define $\phi = p/(1-p)$ and $l_x = \log \Pr(X = x)$ gives the recursion

$$l_{x+1} = l_x + \log\phi + \log(n-x) - \log(x+1).$$

This recursion is particularly useful if the entire pmf is desired. Note also that the recursion can start from either end, that is for increasing x starting at $x = 0$ or for decreasing x starting at $x = n$.

Calculating the cdfs

The cdf of many distributions, including the Z, t, χ^2, F, binomial, and Poisson are related to the incomplete gamma and beta functions defined at the beginning of the table of distributions at the end of this chapter. It is not difficult to show that

$$F_Z(x) = \begin{cases} \frac{1}{2} + \frac{1}{2} IG\left(\frac{x^2}{2}; \frac{1}{2}\right), & x \ge 0 \\ 1 - F_Z(-x), & x < 0, \end{cases}$$

$$F_t(x;\nu) = \begin{cases} 1 - \dfrac{IB\left(\frac{\nu}{\nu+x^2}; \frac{\nu}{2}, \frac{1}{2}\right)}{2}, & x \ge 0 \\ 1 - F_t(-x;\nu), & x < 0, \end{cases}$$

$$F_{\chi^2}(x;\nu) = IG\left(\frac{x}{2}; \frac{\nu}{2}\right),$$

$$F_F(x;\nu_1,\nu_2) = 1 - IB\left(\frac{\nu_2}{\nu_2+\nu_1 x}; \frac{\nu_2}{2}, \frac{\nu_1}{2}\right),$$

$$F_{Bin}(k;n,p) = 1 - IB(p; k+1, n-k),$$

$$F_{Poiss}(k;\lambda) = 1 - F_{\chi^2}(2\lambda; 2(k+1)) = 1 - IG(\lambda; k+1).$$

Thus to evaluate these six cdfs we need only be able to calculate IG and IB.

Calculating the Incomplete Gamma and Beta Functions

There is a vast literature on calculating IG and IB. Popular methods include ("Numerical Recipes", eqs. 6.2.5, 6.2.6, and 6.3.5)

$$\mathrm{IG}(x;a) = \begin{cases} e^{-x}x^a \sum_{n=0}^{\infty} \frac{1}{\Gamma(a+1+n)} x^n, & x \le a+1 \\ 1 - \frac{1}{\Gamma(a)} e^{-x} x^a \left(\frac{1}{x+}\frac{1-a}{1+}\frac{1}{x+}\frac{2-a}{1+}\frac{2}{x+}\cdots\right), & x > a+1, \end{cases}$$

$$\mathrm{IB}(x;a,b) = \begin{cases} \frac{x^a(1-x)^b}{a\beta(a,b)}\left[\frac{1}{1+}\frac{d_1}{1+}\frac{d_2}{1+}\cdots\right], & x < \frac{a+1}{a+b+1}, \\ 1 - \mathrm{IB}(1-x; b, a), & x \ge \frac{a+1}{a+b+1}, \end{cases}$$

where for $m = 0, 1, 2, \dots$

$$d_{2m+1} = -\frac{(a+m)(a+b+m)x}{(a+2m)(a+2m+1)},$$

$$d_{2m} = \frac{m(b-m)x}{(a+2m-1)(a+2m)},$$

and we have used the continued fraction notation

$$a_0 + \frac{a_1}{b_1+}\frac{a_2}{b_2+}\frac{a_3}{b_3+}\cdots = a_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cdots}}}.$$

If we let f_n be the result of evaluating such a continued fraction through the terms a_n and b_n, then it can be shown by induction on n that $f_n{=}A_n/B_n$, where the A_n's and B_n's satisfy the recursion

$$A_j = b_j A_{j-1} + a_j A_{j-2} \qquad B_j = b_j B_{j-1} + a_j B_{j-2}, \quad j = 1, \dots, n,$$

where $A_{-1} = 1$, $B_{-1} = 0$, $A_0 = b_0$, and $B_0 = 1$. Often this recursion is rescaled at each step, i.e. consecutive pairs of A's and B's are divided by the latest B (unless it is zero). This does not effect the value of f_n and in fact makes $A_n = f_n$, which can be tested for convergence of the continued fraction.

Calculating the Quantile Functions

Continuous Distributions

The quantile function $Q(u)$ of a random variable X having strictly increasing cdf F is the value q of X having $F(q) = u$, i.e. $Q(u) = F^{-1}(x)$. The usual method for finding the quantile function is to use one of the standard root-finding methods to solve $F(q) - u = 0$ for a specified u. For example, Newton's method finds q as the limit of the sequence

$$q_i = q_{i-1} - \frac{F(q_{i-1}) - u}{f(q_{i-1})}, \quad i = 1, 2, \dots,$$

where the starting value q_0 is obtained by some approximation to q. Notice that since the cdf's of Z, t, χ^2, and F are related to IG and IB, then so are their inverses. Thus we could just write procedures to find these latter two. However, since a great deal of research has been devoted to finding approximations to the quantile functions of various distributions, we have listed them for a variety of distributions in the table at the end of the chapter.

Discrete Distributions

For a discrete random variable X having possible values $x_1, x_2, \dots$ and corresponding probabilities $f(x_1), f(x_2), \dots$ and cumulative probabilities $F(x_1), F(x_2), \dots$, the quantile function $Q(u)$ of X is the smallest x such that $F(x) \geq u$. Thus the straightforward method of finding $Q(u)$ is to successively compare u to $F(x_1)$, $F(x_2)$, and so on until the first x is found for which $F(x) \geq u$.

Calculating Power and Sample Sizes

These operations require the use of the noncentral distributions listed in the table at the end of the chapter.

Random Number Generators

Methods for generating random numbers fall into two classes; 1) special purpose methods that are tailored to a specific distribution, and 2) general methods that can be applied to a wide variety of distributions.

Generating numbers from the uniform distribution plays a particularly important role because of the following theorem.

Theorem 1. PROBABILITY INTEGRAL TRANSFORM

Let X be a random variable having cdf F and quantile function Q, that is $Q(u) = \inf\{x : F(x) \geq u\}$ for $u \in [0,1]$. If U is a random variable having the U[0,1] distribution, then $Q(U)$ has the cdf F.

Thus in theory we have a method for generating random numbers $X_1, \ldots, X_n$ from any distribution, namely to first generate $U[0,1]$'s $U_1, \ldots, U_n$ and then to get $X_i = Q(U_i)$. This is easy to do for several of the distributions in the table at the end of this chapter including the exponential, weibull, and others. Unfortunately, there are many distributions for which it is too difficult to calculate the quantile function (the normal distribution for example which requires the evaluation of the incomplete gamma function) and special generators must be developed on a case by case basis.

Generating Normal Data

For many distributions it is possible to find a simple transformation of other easy to generate variables that has the desired distribution. For the normal distribution, there are two famous examples of this. The first is called the Box-Muller generator which relies on the fact that if U_1 and U_2 are independent $U(0,1)$ random variables, then

$$X_1 = \sqrt{-2\log U_1}\cos(2\pi U_2), \qquad X_2 = \sqrt{-2\log U_1}\sin(2\pi U_2),$$

are independent $N(0,1)$ random variables. Thus to generate a pair of normals, we need two uniforms, a log, a cosine, and a sine.

Another transformation can be found which also generates normals in pairs but reduces the amount of work required. Called Marsaglia's power method, it proceeds as follows. First, it generates a point (V_1, V_2) that is uniformly distributed in the unit circle (the circle of radius one centered at the origin). By uniform over a geometric region, we mean that the probability that a point falls into a subset of the region is equal to the "area" of the subset divided by the "area" of the entire region (for three dimensional regions, area would actually be volume). If U_1 and U_2 are independent $U(0,1)$ variables, then the point (V_1, V_2) where $V_i = 2(U_i - .5)$ is uniformly distributed in the unit square (square having vertices (1,0), (0,1), $(-1,0)$, and $(-1,-1)$ and if $W = V_1^2 + V_2^2 < 1$ then the point is also uniform on the unit circle. Thus we generate pairs of uniforms until we find one in the circle. Note that the chance that a point is in the circle is equal to the ratio of the area of the circle (which is π) to the area of the square (which is four) and this ratio is approximately 0.785.

Once we have a point in the circle, we can get independent $N(0,1)$'s X_1 and X_2 by

$$X_1 = V_1\sqrt{\frac{-2\log W}{W}}, \qquad X_2 = V_2\sqrt{\frac{-2\log W}{W}}.$$

This method avoids the calculation of the cosine and sine in the Box-Muller method at the expense of having to generate a few more uniforms (which we'll see is very simple).

Uniform Random Number Generators

Most authors attribute the first calculation-based (as opposed to mechanical devices such as dice) random number generator to von Neumann (1951) which consisted of generating a sequence of 'pseudo-random' numbers by starting with a four digit integer (called a 'seed') and squaring it and then using the four middle digits of the result as the next number in the sequence. To get a random number in the interval from zero to one, one can divide the elements of the sequence by 10,000. The numbers are called 'pseudo-random' since they are in fact generated by a deterministic method.

After 40 years of experimentation with such recursive methods, most uniform random number generators in common use are now based on an integer recursion of the form

$$X_{i+1} = (aX_i + b)\mathrm{mod}M, \qquad i \geq 0,$$

which is called a mixed linear congruential generator, and a vast literature exists on how the integers a, b, and M should be chosen (see Knuth, etc). The starting value X_0 is called a seed and again needs to be supplied by the user. Values between zero and one are then found by dividing the X's by M.

The basic considerations used in choosing a, b, and M include:

1. Because of the modular arithmetic involved, there are only M values that the X's can assume. Thus the value of M needs to be very large and typically are of the order of 2^{32}.

2. As soon as the recursion encounters an X that has occurred previously, then the sequence will repeat. Thus we want to choose a and b so that we are guaranteed of a long cycle length.

3. We must choose a, b, and M so that in any section of the sequence of uniforms, the numbers will appear to be "uniform" (each number in (0,1) seems to appear equally often) and "independent" (there is no discernible pattern in the numbers in the sequence).

Sampling from Discrete Distributions

Let X be a discrete random variable having possible values $x_1, x_2, \ldots$ and corresponding probabilities $f(x_1), f(x_2), \ldots$ and cumulative probabilities $F(x_1), F(x_2), \ldots$. Then the quantile function $Q(u)$ of X is given by the smallest x such that $F(x) \geq u$. Thus a straightforward method for simulating X is to first generate $U \sim U(0, 1)$ and then successively compare U to $F(x_1)$, $F(x_2)$, and so on, until finding the first x for which $F(x) \geq U$.

Generating Multinomial Data

A wide variety of random number generation problems can be phrased in terms of randomly selecting (with replacement) one element from a set of K elements. For example a random variable X having the $B(n, p)$ distribution can be written as $X = \sum_{i=1}^{n} Y_i$ where each of the Y's is either one (with probability p) or zero (with probability $1-p$). A multinomial random vector $(X_1, \ldots, X_k)$ with n trials and probability vector $(p_1, \ldots, p_K)$ is similar except that X_i is the number of outcomes of type i (out of K possible outcomes).

In this section we consider methods for randomly selecting an outcome from a set of K outcomes when the probabilities of the outcomes are $p_1, \ldots, p_K$.

Equally Likely Outcomes

If the K outcomes are equally likely, then all we need to do is generate a random integer i from one to K and increment X_i. A random integer from 1 to K is easily obtained from $i = [UK] + 1$ where U is $U[0, 1]$. Simple extensions of equally likely outcomes are also simple to implement as well. For example, if we have three outcomes and the probabilities are 1/4, 1/2, and 1/4, then we can generate a random integer M from one to four and select outcome one if $M = 1$, outcome two if $M = 2$ or $M = 3$, and outcome three if $M = 4$.

Table Look-Up Methods

The same idea can be used anytime the probabilities $p_1, \ldots, p_K$ have only a small number of decimal places. For example, if there are three decimal places, that is, $p_i = n_i/1000$, for $i = 1, \ldots, K$, then we can form a vector l of length 1,000 whose first n_1 elements are all one, the next n_2 are all two, and so on. Then we can generate a random integer M from one to 1,000 and obtain an outcome equal to $l(M)$.

Generating Random Permutations

A method for producing permutations $i_1, \ldots, i_n$ of the integers one through n is said to produce random permutations if all $n!$ such permutations are equally likely to occur. An obvious way to generate random permutations is to generate a sequence of integers where at the ith step, the selected integer is chosen randomly from the set of integers that have not been selected so far. One simple method for doing this that doesn't require keeping track of what has been selected so far at any step is to start by constructing a vector x of length n where $x_i = i$, and then for $i = 1, \ldots, n-1$ randomly selecting an integer from i to n to be the index of the x to swap with x_i.

Sampling Without Replacement

There are many problems where we need to sample without replacement from some finite population. One example is generating a hypergeometric variable, that is, the number of 'defectives' in a random sample (without replacement) of n objects from a population of N objects wherein there are D defectives and $N-D$ nondefectives. A simple way to do this is to modify the method for generating random permutations. We start with a vector x of length N whose first D elements are ones and last $N-D$ are zeros. Then we can get the indices of the x's to include in our sample by running the random permutation procedure except running the loop from one to n rather than the full one to N.

The Poisson Distribution

The sampling methods given above can be used to generate observations from all of the common discrete distributions except for the Poisson which because of its infinite support and not being expressible in terms of simple sampling methods, requires special methods. Most methods use the fact that a Poisson variable X having intensity λ is the number of events in the interval [0,1] where the times between events are iid Exponential with parameter λ. Thus we can get a Poisson variable X as one less than the number of exponentials needed to get a sum greater than one. Since an exponential with parameter λ is just $-\log U/\lambda$ where U is uniform on [0,1], this means we need to generate uniforms until $-\sum_i \log U_i/\lambda > 1$ or equivalently until $\prod_i U_i < e^{-1/\lambda}$.

Some General Generation Methods

Rejection Methods

If it is hard to generate realizations from a random variable X having pdf f and we can find another pdf g defined over the same range as f for which it is easy to generate realizations and we can find a constant $c > 1$ such that

$$h(x) = cg(x) \geq f(x), \qquad \text{for all } x,$$

then we can generate a realization X from f via the following algorithm, called a rejection or acceptance-rejection method:

1. Generate Y from g.
2. Let $Z = Uh(Y)$ where U is $U(0,1)$ independent of Y.
3. Let $X = Y$ if $Z < f(Y)$, otherwise go back to 1.

This process generates a point (Y, Z) somewhere under the function h, and the basic idea is to use the abscissa of the point as our random number if the point also falls under f at that abscissa. The variable Y has density g while the conditional distribution of Z given Y is $U(0, h(Y))$ and thus the joint pdf of Y and Z is

$$f_{Y,Z}(y, z) = \frac{1}{h(y)} g(y) = \frac{1}{c}, \qquad y \in \mathcal{R},\ z \in [0, h(y)],$$

that is, the point is uniformly distributed above the region in the plane where $y \in \mathcal{R}$ and $z \in [0, h(y)]$. This means that $c \Pr(X \leq x)$ is the area under f for $Y \leq x$, which is $cF_X(x)$, and thus X has cdf F_X as desired.

The probability that a point will be rejected is the area between h and f divided by the area under h, that is,

$$\Pr(\text{rejection}) = \frac{\int_{-\infty}^{\infty}(h(x) - f(x))dx}{\int_{-\infty}^{\infty} h(x)dx} = 1 - \frac{1}{c},$$

and thus we should try to find a c that is as small as possible.

If we know that we can find an "enveloping function" g, then we can find the "best" value of c by finding

$$c = \max_x \left(\frac{f(x)}{g(x)}\right),$$

since we want $cg(x) \geq f(x)$.

To illustrate the rejection method, consider generating $X \sim \Gamma(\nu, 1)$, that is from the pdf

$$f(x) = \frac{x^{\nu-1}e^{-x}}{\Gamma(\nu)}, \qquad x > 0,\ \nu > 1.$$

Note that if $X \sim \Gamma(\nu, 1)$ then $X/\lambda \sim \Gamma(\nu, \lambda)$ so we need only consider the $\Gamma(\nu, 1)$ case to take care of the two parameter case having $\nu > 1$. It is easy to show that for any ν we can find a c so that c times an exponential pdf having parameter $1/\nu$ is uniformly greater than f. Thus we find c as the maximum of the ratio of $f(x)$ to $g(x) = e^{-x/\nu}/\nu$. To find where this maximum occurs, let $y = f(x)/g(x)$ and note that

$$\log y = (\nu - 1)\log x - x + \frac{x}{\nu} + \log(\nu/\Gamma(\nu)),$$

$$\frac{\partial \log y}{\partial x} = \frac{\nu - 1}{x} - 1 + \frac{1}{\nu}, \quad \frac{\partial^2 \log y}{\partial x^2} = \frac{1 - \nu}{x^2},$$

and thus the maximum of the ratio occurs at $x = \nu$ and has value $c = \nu^\nu e^{1-\nu}/\Gamma(\nu)$. This gives the algorithm:

1. Generate $Y \sim \text{Exp}(1/\nu)$, that is, $Y = -\nu \log(U_1)$, where $U_1 \sim U(0, 1)$.
2. Let $Z = ce^{-Y/\nu}U_2/\nu)$, where $U_2 \sim U(0, 1)$ independent of U_1.
3. Let $X = Y$ if and only if $Z < f(Y)$.

The Decomposition Method

Many times it is possible to decompose a difficult-to-simulate-from pdf f into a mixture of k pdf's, that is, we can write

$$f(x) = \sum_{i=1}^{k} \alpha_i f_i(x), \qquad \alpha_i \in [0, 1],\ \sum_{i=1}^{k} \alpha_i = 1.$$

Then if we first pick pdf f_i with probability α_i and then generate a realization X from f_i, it is easy to see that in fact X is a realization from f, since

$$\begin{aligned}\Pr(X \leq x) &= \sum_{i=1}^{k} \Pr(X \leq x | X \text{ from } f_i) \Pr(X \text{ from } f_i) \\ &= \sum_{i=1}^{k} \alpha_i \int_{-\infty}^{x} f_i(z)dz = \int_{-\infty}^{x} \sum_{i=1}^{k} \alpha_i f_i(z)dz = \int_{-\infty}^{x} f(z)dz = F_X(x),\end{aligned}$$

as desired. Note that the method works for discrete distributions as well and that the method is very effective if the most likely pdf's are easy to simulate from.

To illustrate the method, consider simulating from the standard exponential pdf $f(x) = e^{-x}$ for $x > 0$. Since $Q(u) = -\log(1-U)$, it is easy to generate data from f directly using the quantile method. However, by using a very simple decomposition method, it is possible to avoid the log an appreciable percent of the time. If we decompose the region under the exponential pdf into 1) the rectangle having lower left corner (0,0) and upper right corner $(1, 1/e)$, 2) the "wedge" above the rectangle, and 3) the tail to the right of $x = 1$, we can decompose $f(x)$ into $\alpha_1 f_1(x) + \alpha_2 f_2(x) + \alpha_3 f_3(x)$ where $\alpha_1 = \alpha_3 = 1/e \doteq 0.368$, $\alpha_2 = 1 - 2/e \doteq 0.264$, and

$$
\begin{aligned}
f_1(x) &= 1, \qquad x \in [0,1], \\
f_2(x) &= \frac{e^{-x} - e^{-1}}{1 - 2/e}, \qquad x \in [0,1], \\
f_3(x) &= e^{-(x-1)}, \qquad x \geq 1.
\end{aligned}
$$

Thus, approximately 37% of the time we need only generate a $U(0,1)$, while approximately 26% of the time we need a realization from f_2, and 37% of the time from f_3. Now the quantile function of f_3 is $1 - \log(1-u)$, and thus we can generate a standard exponential via the quantile method. Finally, it remains to see how to simulate from f_2 the approximately 26% of the time we need to and be sure that this is simple enough that the decomposition method does in fact result in a savings over the quantile method.

It is easy to show that f_2 is the pdf of the minimum of N $U(0,1)$'s where N has the truncated Poisson distribution with parameter one, that is,

$$
\Pr(N = n) = \frac{1}{n!(e-2)}, \qquad n = 2, 3, \ldots.
$$

To three decimal places, the cumulative probabilities for $n = 2$ through $n = 6$ are .696, .928, .986, .998, and 1.000. Thus we can easily generate the Poisson using the direct quantile method for discrete variables, that is, generate a $U(0,1)$ and find the smallest value of n so that $\Pr(N \leq n) \geq U$. Almost always this will give a small value of N. In fact,

$$
\mathrm{E}(N) = \sum_{n=2}^{\infty} n \frac{1}{n!(e-2)} = \frac{1}{e-2} \sum_{n=2}^{\infty} \frac{1}{(n-1)!} = \frac{1}{e-2} \sum_{n=1}^{\infty} \frac{1}{n!} = \frac{e-1}{e-2} \doteq 2.392.
$$

This means that on the average we will need to generate approximately 3.392 $U(0,1)$'s to get one standard exponential (one to get N and 2.392 to get X), as well as do the comparisons in getting N and the minimum of N uniforms.

While this does not lead to great savings, it does illustrate the idea of the decomposition method.

Combining Rejection and Decomposition

Using the decomposition method leads to the need to generate data from several distributions and it is natural to in turn use rejection and/or decomposition on these distributions.

Generating Multivariate Data

Multivariate Normal

To generate a p-dimensional normal random vector X with given mean vector μ and covariance matrix Σ, we can first generate a p-dimensional normal random vector e having mean vector zero and identity covariance matrix, that is, e is a vector of p iid $N(0,1)$'s, and then calculate

$$
X = \Sigma^{1/2} e + \mu,
$$

where $\Sigma^{1/2}$ is the positive definite square root of Σ, that is Σ can be written as the product of $\Sigma^{1/2}$ times the transpose of $\Sigma^{1/2}$. This method relies on the facts that if Z is an r dimensional random vector and A is an $(n \times r)$ matrix of constants, then

$$\mathrm{Var}(AZ) = A\mathrm{Var}(Z)A^T,$$

and that a matrix times a normal random vector is also normally distributed.

An important special case is a bivariate normal vector $(X, Y)^T$ having means μ_X and μ_Y, variances σ_X^2 and σ_Y^2, and correlation coefficient ρ. Thus

$$\Sigma = \begin{bmatrix} \sigma_X^2 & \rho\sigma_X\sigma_Y \\ \rho\sigma_X\sigma_Y & \sigma_Y^2 \end{bmatrix}, \qquad \Sigma^{1/2} = \begin{bmatrix} \sigma_X & 0 \\ \rho\sigma_Y & \sigma_Y\sqrt{1-\rho^2} \end{bmatrix},$$

which gives

$$X = \sigma_X e_1 + \mu_X, \qquad Y = \rho\sigma_Y e_1 + \sigma_Y\sqrt{1-\rho^2}e_2 + \mu_Y,$$

where e_1 and e_2 are independent N(0,1)'s.

The Gibbs Sampler

Simulating data of possibly very high dimension (such as simulating visual images) has been made possible in recent years by the creation of methods collectively called Monte Carlo Markov Chain (MCMC) methods, the most famous of which is called Gibbs sampling, the basic idea of which is the following. Suppose we want to generate n observations from a random vector $X = (X_1, \ldots, X_p)^T$ of dimension p. In situations other than the multivariate normal case, this is usually a very difficult problem. Further, there are many situations where we want to get observations when we don't even know the complete joint distribution of X. If we do know the conditional distribution of each X_i given the other $p-1$ X's, then we can do a series of "Gibbs steps." We start with some initial values of all the X's and then one first generates an X_1 from its conditional distribution given the others (using the inital values for the others), then generate an X_2 from its conditional distribution given the just obtained value of X_1 and the initial values for the other X's, then an X_3 from X_1, X_2, and the inital values of the others, and so on. The second Gibbs step is similar except that one uses the X's from the first step the same way the initial values were used during the first step. Thus, each time one is generating an individual X_i, it is from a univariate distribution. Further, there are many situations where the conditional distributions are known when the complete p-dimensional distribution is unknown.

The remarkable fact about this Gibbs sampling procedure is that in a great many important situations, after a reasonable number of Gibbs steps, the distribution of the generated vectors is indistinguishable from the desired joint distribution. See Casella and George (1992, *American Statistician*, pg. 167–174) for details.

Table of Distributions

The table makes extensive use of the beta and gamma functions:

$$\beta(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}, \quad \Gamma(a) = \int_0^\infty z^{a-1}e^{-z}\,dz, \quad a,b>0,$$

and the incomplete gamma and beta functions:

$$\mathrm{IG}(x;a) = \frac{1}{\Gamma(a)}\int_0^x e^{-t}t^{a-1}\,dt, \quad \mathrm{IB}(x;a,b) = \frac{1}{\beta(a,b)}\int_0^x t^{a-1}(1-t)^{b-1}\,dt,$$

for $a,b>0,\ 0<x<1$.

Four Discrete Distributions

Binomial with Parameters n and p

X is 1) the number of 1's in a sample (with replacement) from a 0–1 population having proportion p of 1's, or 2) the number of "successes" in n independent "trials" where each trial can be a success (with probability p) or a failure (with probability $1-p$).

$$\mathrm{E}(X) = np, \qquad \mathrm{Var}(X) = np(1-p).$$

$$f_{Bin}(x; n, p) = \binom{n}{p} p^x (1-p)^{n-x}, \quad x = 0, 1, \ldots, n.$$

Hypergeometric with Parameters N, n, and M

X is the number of 1's in a sample (without replacement) of size n from a 0–1 population of size N, where the number of 1's in the population is M.

$$\mathrm{E}(X) = np, \quad p = M/N, \qquad \mathrm{Var}(X) = \frac{N-n}{N-1} np(1-p).$$

$$f_{Hyp}(x; N, n, M) = \frac{\binom{M}{x}\binom{N-M}{n-x}}{\binom{N}{n}}, \qquad \max(0, M-(N-n)) \leq x \leq \min(n, M).$$

Negative Binomal with parameters n and p

X is the number of the binomial trial where the nth success occurs when on each trial the probability of a success is p.

$$\mathrm{E}(X) = n/p, \qquad \mathrm{Var}(X) = \frac{n(1-p)}{p^2}.$$

$$f_{Nbin}(x; n, p) = \binom{x-1}{n-1} p^n (1-p)^{x-n}, \qquad x = n, n+1, n+2, \ldots.$$

Poisson with parameter λ

X arises in two ways: (1) If we have a series of events in which the times between events are independent and have the exponential distribution with mean μ, then the number of events in a time interval of length T has the Poisson distribution with $\lambda = T/\mu$. (2) If a binomial parameter n is large and p is small, then the binomial distribution can be approximated by the Poisson distribution with $\lambda = np$.

$$\mathrm{E}(X) = \lambda, \qquad \mathrm{Var}(X) = \lambda.$$

$$f_{Poiss}(x; \lambda) = \frac{\lambda^x e^{-\lambda}}{x!}, \qquad x = 0, 1, 2, \ldots, \quad \lambda > 0.$$

Normal and Those Derived from Normal

Chi-Square with ν Degrees of Freedom

$\chi^2_\nu = \sum_{i=1}^{\nu} X_i^2$, where $X_1, \ldots, X_\nu$ are iid $N(0,1)$.

$$\mathrm{E}(\chi^2_\nu) = \nu, \qquad \mathrm{Var}(\chi^2_\nu) = 2\nu.$$

$$f_{\chi^2}(x; \nu) = \frac{1}{2^{\nu/2}\Gamma(\nu/2)} e^{-x/2} x^{(\nu/2)-1}, \quad F_{\chi^2}(x; \nu) = \mathrm{IG}\left(\frac{x}{2}; \frac{\nu}{2}\right), \quad x > 0.$$

Define the Wilson-Hilferty approximation

$$x_0 = \nu \left[z \left(\frac{2}{9\nu} \right)^{1/2} + 1 - \frac{2}{9\nu} \right]^3, \qquad z = Q_Z(u).$$

Then (Kennedy and Gentle, pg. 118):

$$q_0(u;\chi^2_\nu) = \begin{cases} \left(u\nu\Gamma\left(\frac{\nu}{2}\right)2^{(\nu-2)/2}\right)^{2/\nu}, & \nu < -1.24\log(u), \\ x_0, & \nu \geq -1.24\log(u) \text{ and } x_0 \leq 2.2\nu+6 \\ -2\log\left(\frac{(1-u)\Gamma(\nu/2)}{(x_0/2)^{(\nu/2)-1}}\right), & \nu \geq -1.24\log(u) \text{ and } x_0 > 2.2\nu+6. \end{cases}$$

Noncentral Chi-Square with ν Degrees of Freedom and Noncentrality Parameter λ

$\chi^2_{\nu,\lambda} = \sum_{i=1}^{\nu} X_i^2$, $X_1,\dots,X_\nu$ independent, $X_i \sim N(\mu_i,1)$, $\lambda = \sum_{i=1}^{\nu}\mu_i^2$

$$\mathrm{E}(\chi^2_{\nu,\lambda}) = \nu+\lambda, \quad \mathrm{Var}(\chi^2_{\nu,\lambda}) = 2(\nu+2\lambda).$$

$$f_{\chi^2}(x;\nu,\lambda) = \sum_{j=0}^{\infty} f_{\text{Poiss}}(j;\lambda/2) f_{\chi^2}(x;\nu+2j)$$

$$F_{\chi^2}(x;\nu,\lambda) = \sum_{j=0}^{\infty} f_{\text{Poiss}}(j;\lambda/2) F_{\chi^2}(x;\nu+2j)$$

$$q_0(u;\chi^2_{\nu,\lambda}) = \frac{1+b}{2}\left[Q_Z(u) + \sqrt{\frac{2a}{1+b}-1}\right]^2, \quad \text{(A\&S, 26.4.31)}$$

where $a = \nu+\lambda$, $b = \lambda/(\nu+\lambda)$.

F with ν_1 and ν_2 Degrees of Freedom

$$F_{\nu_1,\nu_2,\lambda} = \frac{Z_1/\nu_1}{Z_2/\nu_2}, \qquad Z_1 \sim \chi^2_{\nu_1},\ Z_2 \sim \chi^2_{\nu_2},\ Z_1, Z_2 \text{ independent.}$$

$$\mathrm{E}(F_{\nu_1,\nu_2}) = \frac{\nu_2}{\nu_2-2}, \quad \nu_2 > 2, \quad \mathrm{Var}(F_{\nu_1,\nu_2}) = \frac{2\nu_2^2(\nu_1+\nu_2-2)}{\nu_1(\nu_2-2)^2(\nu_2-4)}, \quad \nu_2 > 4.$$

$$f_F(x;\nu_1,\nu_2) = \frac{\left(\frac{\nu_1}{\nu_2}\right)^{\nu_1/2} x^{(\nu_1/2)-1}}{\beta\left(\frac{\nu_1}{2},\frac{\nu_2}{2}\right)\left(1+\frac{\nu_1 x}{\nu_2}\right)^{(\nu_1+\nu_2)/2}}, \qquad x > 0$$

$$F_F(x;\nu_1,\nu_2) = 1 - \mathrm{IB}\left(\frac{\nu_2}{\nu_2+\nu_1 x}; \frac{\nu_2}{2}, \frac{\nu_1}{2}\right)$$

For $\nu_1 > 1$ and $\nu_2 > 1$, with $z = Q_Z(u)$, $a = \nu_2/2$, and $b = \nu_1/2$, we have (A & S, 26.5.22, 26.6.16):

$$q_0(u;F_{\nu_1,\nu_2}) = e^{2w},$$

where

$$w = \frac{z(h+\lambda)^{1/2}}{h} - \left(\frac{1}{2b-1} - \frac{1}{2a-1}\right)\left(\lambda + \frac{5}{6} - \frac{2}{3h}\right)$$

$$h = 2\left(\frac{1}{2a-1} + \frac{1}{2b-1}\right)^{-1}, \quad \lambda = \frac{z^2-3}{6}.$$

If $\nu_1 = 1$ or $\nu_2 = 1$, we have

$$Q_F(u;1,\nu) = \left[Q_t\left(\frac{1+u}{2};\nu\right)\right]^2, \qquad Q_F(u;\nu,1) = \left[\frac{1}{Q_t\left(1-\frac{u}{2};\nu\right)}\right]^2.$$

Noncentral F with ν_1 and ν_2 Degrees of Freedom and Noncentrality Parameter λ

$$F_{\nu_1,\nu_2,\lambda} = \frac{Z_1/\nu_1}{Z_2/\nu_2}, \qquad Z_1 \sim \chi^2_{\nu_1,\lambda},\ Z_2 \sim \chi^2_{\nu_2},\ Z_1, Z_2 \text{ independent}$$

$$\mathrm{E}(F_{\nu_1,\nu_2,\lambda}) = \frac{\nu_2(\nu_1+\lambda)}{\nu_1(\nu_2-2)}, \quad \nu_2 > 2,$$

$$\mathrm{Var}(F_{\nu_1,\nu_2,\lambda}) = 2\left(\frac{\nu_2}{\nu_1}\right)^2 \frac{(\nu_1+\lambda)^2 + (\nu_1+2\lambda)(\nu_2-2)}{(\nu_2-2)^2(\nu_2-4)}, \quad \nu_2 > 4.$$

$$f_F(x;\nu_1,\nu_2,\lambda) = \sum_{j=0}^{\infty} \frac{(\lambda/2)^j e^{-\lambda/2}\left(\frac{\nu_1}{\nu_2}\right)^{(\nu_1+2j)/2} x^{(\nu_1+2j-2)/2}}{j!\beta\left(\frac{\nu_2}{2},\frac{\nu_1+2j}{2}\right)\left(1+\frac{\nu_1 x}{\nu_2}\right)^{(\nu_1+\nu_2+2j)/2}},\ x > 0$$

$$F_F(x;\nu_1,\nu_2,\lambda) = \sum_{j=0}^{\infty} f_{\mathrm{Poiss}}(j;\lambda/2)\mathrm{IB}\left(\frac{\nu_1 x}{\nu_1 x+\nu_2};\frac{\nu_1}{2}+j,\frac{\nu_2}{2}\right)$$

Normal Distribution with Mean μ and Variance σ^2

$$f_N(x;\mu,\sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2}, \qquad x \in \mathcal{R},\ \mu \in \mathcal{R},\ \sigma^2 > 0$$

$$F_Z(x) = \begin{cases} \frac{1}{2} + \frac{1}{2}\mathrm{IG}\left(\frac{x^2}{2};\frac{1}{2}\right), & x \geq 0 \\ 1 - F_Z(-x), & x < 0, \end{cases}$$

Defining $t = \sqrt{-2\log u}$, we have (A & S, eq. 26.2.22):

$$q_0(u;Z) = \begin{cases} \frac{2.30753 + .27061t}{1 + .99229t + .04481t^2} - t, & 0 < u \leq 0.5, \\ -q_0(1-u;Z), & .5 < u < 1. \end{cases}$$

Student t with ν Degrees of Freedom

$$t_{\nu,\lambda} = \frac{X}{\sqrt{Y/\nu}}, \qquad X \sim N(0,1),\ Y \sim \chi^2_\nu,\ X, Y \text{ independent}$$

$$\text{Odd moments are zero}, \mathrm{Var}(t_\nu) = \frac{v}{\nu-2}, \quad \nu > 2.$$

$$f_t(x;\nu) = \frac{1}{\sqrt{\nu}\beta(1/2,\nu/2)}(1+(x^2/\nu))^{-(\nu+1)/2}, \quad -\infty < x < \infty$$

$$F_t(x;\nu) = \begin{cases} 1 - \mathrm{IB}\left(\nu/(\nu+x^2);\nu/2,1/2\right)/2, & x \geq 0 \\ 1 - F_t(-x;\nu), & x < 0, \end{cases}$$

Defining $z = q_0(u;Z)$, one approximation is given by (A & S, eq. 26.7.5)

$$q_0(u;t_\nu) = \begin{cases} z + \sum_{j=1}^4 \frac{g_j(z)}{\nu^j}, & .5 \leq u < 1, \\ -q_0(1-u;t_\nu), & 0 \leq .5, \end{cases}$$

$$g_1(z) = \frac{1}{4}(z^3+z), \qquad g_2(z) = \frac{1}{96}(5z^5+16z^3+3z),$$

$$g_3(z) = \frac{1}{384}(3z^7+19z^5+17z^3-15z),$$

$$g_4(z) = \frac{1}{92160}(79z^9+776z^7+1482z^5-1920z^3-945z).$$

Noncentral t with ν Degrees of Freedom and Noncentrality Parameter λ

$$t_{\nu,\lambda} = \frac{X}{\sqrt{Y/\nu}}, \qquad X \sim N(\lambda,1),\ Y \sim \chi^2_\nu,\ X,Y \text{ independent}$$

$$\mathrm{E}(t_{\nu,\lambda}) = \left(\frac{\nu}{2}\right)^{1/2}\frac{\Gamma((\nu-1)/2)}{\Gamma(\nu/2)}\lambda, \quad \mathrm{Var}(t_{\nu,\lambda}) = \frac{\nu}{\nu-2}(1+\lambda^2) - \mathrm{E}^2(t_{\nu,\lambda}).$$

$$f_t(x;\nu,\lambda) = \frac{\nu^{\nu/2}}{\sqrt{\pi}\Gamma(\nu/2)}\frac{e^{-\lambda^2/2}}{(\nu+t^2)^{(\nu+1)/2}}\sum_{s=0}^{\infty}\Gamma\left(\frac{\nu+s+1}{2}\right)\left(\frac{\lambda^s}{s!}\right)\left(\frac{2t^2}{\nu+t^2}\right)^{s/2}$$

$$F_t(x;\nu,\lambda) = 1 - \sum_{j=0}^{\infty} e^{-\lambda^2/2}\frac{\left(\lambda^2/2\right)^j}{2j!}\mathrm{IB}\left(\frac{\nu}{\nu+x^2};\frac{\nu}{2},j+\frac{1}{2}\right) \doteq F_Z(a/b)$$

where $a = x\left(1-\frac{1}{4\nu}\right) - \lambda$ and $b = \left(1+\frac{x^2}{2\nu}\right)^{1/2}$.

Other Continuous Distributions

Beta(α,β)

$$f_\beta(x;\alpha,\beta) = \frac{1}{\beta(\alpha,\beta)}x^{\alpha-1}(1-x)^{\beta-1}, \qquad x \in [0,1],\ \alpha > 0,\ \beta > 0$$

$$\mu_X = \frac{\alpha}{\alpha+\beta}, \qquad \sigma^2_X = \frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$$

Cauchy(θ,σ)

$$f_{Cauch}(x;\theta,\sigma) = \frac{1}{\pi\sigma}\frac{1}{1+\left(\frac{x-\theta}{\sigma}\right)^2}, \qquad x \in \mathcal{R},\ \theta \in \mathcal{R},\ \sigma > 0$$

Gumbel(α, β)

$Y = \alpha - \beta X$, where X is exponential with parameter 1.

$$f(x;\alpha,\beta) = \frac{1}{\beta} e^{-e^{-(x-\alpha)/\beta}} e^{-(x-\alpha)/\beta}, \qquad x \in \mathcal{R},\ \alpha \in \mathcal{R},\ \beta > 0$$

$$F(x;\alpha,\beta) = e^{-e^{-(x-\alpha)/\beta}}$$

$$Q(u) = \alpha - \beta \log(-\log u), \qquad 0 \le u \le 1$$

$$\mu_X = \alpha + \beta\gamma,\ \gamma \approx .577216,\ \sigma_X^2 = \frac{\pi^2\beta^2}{6}$$

Laplace(μ, σ)

$$f(x;\mu,\sigma) = \frac{1}{2\sigma} e^{-|x-\mu|/\sigma}, \qquad x \in \mathcal{R},\ \mu \in \mathcal{R},\ \sigma > 0$$

$$F(x;\mu,\sigma) = \begin{cases} \frac{1}{2} e^{(x-\mu)/\sigma}, & \text{if } x < \mu \\ 1 - \frac{1}{2} e^{-(x-\mu)/\sigma}, & \text{if } x \ge \mu \end{cases}$$

$$Q(u) = \begin{cases} \sigma \log(2u) + \mu, & \text{if } 0 < u < \frac{1}{2} \\ -\sigma \log 2(1-u) + \mu, & \text{if } \frac{1}{2} \le u < 1 \end{cases}$$

Logistic(μ, β)

For $x \in \mathcal{R}$, $\mu \in \mathcal{R}$, and $\beta > 0$,

$$f(x;\mu,\beta) = \frac{1}{\beta} \frac{e^{-(x-\mu)/\beta}}{[1+e^{-(x-\mu)/\beta}]^2}, \quad F(x;\mu,\beta) = \frac{1}{1+e^{-(x-\mu)/\beta}}, \quad Q(u) = \mu - \beta \log \frac{1-u}{u}.$$

Lognormal(μ, σ^2)

$$f(x;\mu,\sigma^2) = \frac{1}{\sqrt{2\pi}\sigma x} e^{(-\log x - \mu)^2/(2\sigma^2)}, \qquad x > 0,\ \mu \in \mathcal{R},\ \sigma > 0$$

$$\mu_X = e^{\mu + (\sigma^2/2)},\ \sigma_X^2 = e^{2(\mu+\sigma^2)} - e^{2\mu+\sigma^2}$$

Weibull(α, β)

For $x > 0$, $\alpha > 0$, and $\beta > 0$,

$$f(x) = \alpha\beta x^{\beta-1} e^{-\alpha x^\beta}, \quad F(x) = 1 - e^{-\alpha x^\beta}, \quad Q(u) = \left[\frac{-\log(1-u)}{\alpha}\right]^{1/\beta}.$$

CHAPTER 4

Iterative Methods for Parameter Estimation

A wide variety of parameter estimation techniques require the ability to minimize or maximize a complicated function of the parameters. In this chapter we look at several general methods for optimization.

Newton-Raphson for Maximum Likelihood Estimation

Let $X_1, \ldots, X_n$ be n independent random variables such that the probability density or mass function of X_i is $f_i(x;\theta)$ which depends on a vector θ of r parameters. Then the maximum likelihood estimator $\hat{\theta}$ of θ is that value that maximizes the log likelihood function

$$l(\theta) = \sum_{i=1}^{n} \log f_i(X_i;\theta),$$

or, equivalently, minimizes $L(\theta) = -l(\theta)$. Given an initial guess θ_0 for $\hat{\theta}$, Newton-Raphson can be used to find the MLE. The jth element of the gradient and the (j,k)th element of the Hessian are given by

$$\begin{aligned} g_j &= \frac{\partial L}{\partial \theta_j} = -\sum_{i=1}^{n} \frac{\partial \log f_i(X_i;\theta)}{\partial \theta_j}, \qquad j = 1,\ldots,r, \\ H_{jk} &= \frac{\partial^2 L}{\partial \theta_j \partial \theta_k} = -\sum_{i=1}^{n} \frac{\partial^2 \log f_i(X_i;\theta)}{\partial \theta_j \partial \theta_k}, \qquad j,k = 1,\ldots,r. \end{aligned} \tag{1}$$

To perform inferences about θ based on $\hat{\theta}$, we can use the fact that

$$\hat{\theta} \sim \mathrm{AN}(\theta, V(\theta)), \quad V(\theta) = I^{-1}(\theta),$$

where the (j,k)th element of the information matrix $I(\theta)$ is

$$I_{jk}(\theta) = -\mathrm{E}\left(\frac{\partial^2 \log l(\theta)}{\partial \theta_j \partial \theta_k}\right) = \mathrm{E}\left(\frac{\partial^2 \log L(\theta)}{\partial \theta_j \partial \theta_k}\right) = -\sum_{i=1}^{n} \mathrm{E}\left(\frac{\partial^2 \log f_i(X_i;\theta)}{\partial \theta_j \partial \theta_k}\right)$$

If in addition to being independent, the X's are identically distributed with common density or mass function f, then all of the expectations in this last expression will be the same and we can write

$$I_{jk}(\theta) = n i_{jk}(\theta) = -n\mathrm{E}\left(\frac{\partial^2 \log f(X;\theta)}{\partial \theta_j \partial \theta_k}\right),$$

where the matrix $i(\theta)$ is called the information in a single observation.

It is often difficult to evaluate the expectations in $i(\theta)$ so a natural procedure is to approximate them by their sample averages, which from (1) are

$$\hat{i}_{jk}(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\frac{\partial^2 \log f(X_j;\theta)}{\partial\theta_j\partial\theta_k} = \frac{1}{n}H_{jk}(\theta),$$

and so a popular method of finding standard errors of $\hat{\theta}$ is to use covariance matrix $H^{-1}(\hat{\theta})$, that is, the inverse of the Hessian matrix at the last Newton-Raphson iteration.

The Method of Scoring

The method of scoring (see Rao, 1973, p. 366, for example) is a procedure that is very similar to Newton-Raphson and consists of iterations of the form

$$\theta_{i+1} = \theta_i + I^{-1}(\theta_i)g_i.$$

MLE's for the Weibull Distribution

To illustrate the process, we consider the two dimensional Weibull distribution which is an often-used distribution in survival analysis and has density, distribution, and quantile functions

$$f(x) = \begin{cases} \alpha\beta x^{\beta-1}e^{-\alpha x^{\beta}}, & x > 0, \alpha > 0, \beta > 0 \\ 0, & \text{otherwise,} \end{cases}$$

$$F(x) = 1 - e^{-\alpha x^{\beta}}, \quad x > 0,$$

$$Q(u) = \left[\frac{-\log(1-u)}{\alpha}\right]^{1/\beta}, \quad 0 \le u \le 1.$$

Note that the Weibull distribution becomes the exponential distribution when $\beta = 1$. We consider estimating the parameters α and β given a sample $X_1, \ldots, X_n$.

The MLE's $\hat{\alpha}$ and $\hat{\beta}$ are the values of α and β maximizing the log likelihood

$$l(\alpha,\beta;X) = n\log\alpha + n\log\beta + (\beta-1)\sum_{i=1}^{n}\log X_i - \alpha\sum_{i=1}^{n}X_i^{\beta}.$$

It is easy to verify that the gradient and Hessian corresponding to minimizing $L(\theta) = -l(\theta)$ are given by

$$g = \begin{bmatrix} \sum_{i=1}^{n} X_i^{\beta} - \dfrac{n}{\alpha}, \\ \alpha\sum_{i=1}^{n} X_i^{\beta}\log X_i - \sum_{i=1}^{n}\log X_i - \dfrac{n}{\beta}, \end{bmatrix},$$

$$H = \begin{bmatrix} \dfrac{n}{\alpha^2} & \sum_{i=1}^{n} X_i^{\beta}\log X_i, \\ \text{Sym} & \alpha\sum_{i=1}^{n} X_i^{\beta}(\log X_i)^2 + \dfrac{n}{\beta^2}. \end{bmatrix}.$$

From $Q(u) = (-\log(1-u)/\alpha)^{1/\beta}$, we have $y = \beta_0 + \beta_1 x$, where $\beta_0 = \log\alpha$, $\beta_1 = \beta$, $y = \log(-\log(1-u))$, and $x = \log Q(u)$. Thus if we let

$$u_i = \frac{i-1/2}{n}, \quad y_i = \log(-\log(1-u_i)), \quad x_i = \log\hat{Q}(u_i) = \log X_{(i)}, \quad i = 1, \ldots, n,$$

where $X_{(i)}$ is the ith order statistic of the random sample, we can regress the y_i's on the x_i's to get estimators $\hat{\beta}_0$ and $\hat{\beta}_1$, from which we get starting values for the Newton-Raphson procedure as $(e^{\hat{\beta}_0}, \hat{\beta}_1)$.

To find standard errors for $\hat{\alpha}$ and $\hat{\beta}$ we need to find

$$I = -\begin{bmatrix} \frac{1}{\alpha^2} & \mathrm{E}(X^\beta \log X) \\ \mathrm{Sym} & \frac{1}{\beta^2} + \mathrm{E}(\alpha X^\beta (\log X)^2) \end{bmatrix},$$

which is very difficult analytically, so the final Hessian would be used.

Estimating Survival Probabilities

An important quantity in survival analysis is the probability of surviving at least x time units, that is, p_x. The MLE of p_x is just $\hat{p}_x = e^{-\hat{\alpha}x^{\hat{\beta}}}$, and another standard result for MLE's gives that

$$\hat{p}_x \sim \mathrm{AN}(p_x, r^T(\theta)V(\theta)r(\theta)/n),$$

where $r(\theta) = (\partial p_x/\partial\alpha, \partial p_x/\partial\beta)^T$. Thus the asymptotic standard error of $\hat{p}_x$ is given by

$$\mathrm{ase}(p_x) = \sqrt{r^T(\theta)I^{-1}(\theta)r(\theta)/n},$$

which is estimated by substituting $\hat{\theta}$ in $r(\theta)$ and using $\hat{I}^{-1}(\hat{\theta})$ instead of $I^{-1}(\theta)$.

Newton-Raphson and Nonlinear Regression

If we have a set of n observations $y_1, \ldots, y_n$ on some dependent variable y and associated with the ith y we have a vector X_i of measurements on k independent variables, and the y's and X's satisfy the nonlinear regression model

$$y_i = f(X_i; \theta) + \epsilon_i, \qquad i = 1, \ldots, n$$

where f is a function that is nonlinear in the r elements of θ and the ϵ's are uncorrelated with mean zero and constant variance σ^2, then the least squares estimators $\hat{\theta}$ of θ minimize the sum of squares of residuals, that is,

$$S(\theta) = \sum_{i=1}^n (y_i - f(X_i; \theta))^2,$$

and we can again use Newton-Raphson to find $\hat{\theta}$. In this case, it is easy to derive the fact that the elements of the gradient and Hessian can be written as

$$g_j = \frac{\partial S}{\partial \theta_j} = -2\sum_{i=1}^n (y_i - f_i)\frac{\partial f_i}{\partial \theta_j}$$
$$H_{jk} = \frac{\partial^2 S}{\partial \theta_j \partial \theta_k} = -2\sum_{i=1}^n \left[(y_i - f_i)\frac{\partial^2 f_i}{\partial \theta_j \partial \theta_k} - \frac{\partial f_i}{\partial \theta_j}\frac{\partial f_i}{\partial \theta_k}\right],$$

where $f_i = f(X_i; \theta)$. If we define the $(n \times r)$ Jacobian matrix J to have (i,j)th element $\partial f_i/\partial\theta_j$, and the $(r \times r)$ matrices $G_1, \ldots, G_n$ so that the (j,k)th element of G_i is $\partial f_i^2/\partial\theta_j\partial\theta_k$, and finally the vector r of length n to have ith element $r_i = y_i - f_i$, then the gradient and Hessian are given by

$$g = -2J^T r, \qquad H = 2J^T J - 2\sum_{i=1}^n r_i G_i. \tag{2}$$

Often, the Hessian is approximated by the first term in this sum, which gives what is called the Gauss-Newton algorithm. In this case, we need to solve at each iteration the 'normal equations' $J_i^T J_i d_i = -J_i^T r$. In either case, inferences about θ based on $\hat{\theta}$ follow from the fact that

$$\hat{\theta} \sim \mathrm{AN}(\theta, \sigma^2 (J^T J)^{-1}),$$

and estimating σ^2 by $s^2 = S(\hat{\theta})/(n-r)$ and J by $J(\hat{\theta})$.

If the errors in the model are normally distributed, then so are the y's and it is easy to show that the negative of the log likelihood function of θ and σ^2 is

$$L(\theta, \sigma^2) = -\frac{1}{2\sigma^2} S(\theta) - \frac{n}{2} \log 2\pi\sigma^2,$$

which means that

$$\frac{\partial L}{\sigma^2} = \frac{1}{2\sigma^4} S(\theta) - \frac{n}{2\sigma^2}, \qquad \frac{\partial L}{\partial \theta} = \frac{1}{2\sigma^2} \frac{\partial S}{\partial \theta}, \qquad \frac{\partial^2 L}{\partial \theta^2} = \frac{1}{2\sigma^2} \frac{\partial^2 S}{\partial \theta^2},$$

which in turn means that $\hat{\sigma}^2 = S(\hat{\theta})/n$ and that the Newton-Raphson iterations to minimize $S(\theta)$ are exactly the same as the Newton-Raphson iterations to maximize the log likelihood function with respect to θ, and thus the least squares estimates and MLE's are the same.

One other interesting thing to notice is that in the Gaussian error case, the method of scoring is the same as the Gauss-Newton method since

$$\left[-\mathrm{E}\left(\frac{\partial^2 l}{\partial \theta^2}\right)\right]^{-1} \frac{\partial l}{\partial \theta} = -\left[\mathrm{E}\left(\frac{\partial^2 S}{\partial \theta^2}\right)\right]^{-1} \frac{\partial S}{\partial \theta} = -\left[\mathrm{E}(H)\right]^{-1} g = -\left[J^T J\right]^{-1} J^T r,$$

since from (2), we have

$$\mathrm{E}(H) = 2J^T J - 2\sum_{i=1}^{n} \mathrm{E}(y_i - f_i) G,$$

and this last expectation is zero.

A Nonlinear Regression Example

To illustrate the procedure above, we consider an example discussed in Draper and Smith's regression book (p. 475), where y represents the percent of chlorine in a unit of some product and the single explanatory variable X is the length of time since the product was produced. The 44 data points in the example are listed at the beginning of the Fortran program at the end of this section. The nonlinear model used to explain y in terms of X has two parameters α and β and is given by

$$y_i = \alpha + (0.49 - \alpha) e^{-\beta(X_i - 8)} + \epsilon_i, \qquad i = 1, \ldots, 44,$$

that is, $f_i = \alpha + (0.49 - \alpha) e_i$, where $e_i = e^{-\beta x_i}$, with $x_i = X_i - 8$. From this we have

$$\frac{\partial f_i}{\partial \alpha} = 1 - e_i, \quad \frac{\partial f_i}{\partial \beta} = -\alpha^* x_i e_i, \quad \frac{\partial^2 f_i}{\partial \alpha^2} = 0, \quad \frac{\partial^2 f_i}{\partial \beta^2} = \alpha^* x_i^2 e_i, \quad \frac{\partial^2 f_i}{\partial \alpha \partial \beta} = x_i e_i,$$

where $\alpha^* = 0.49 - \alpha$, which gives

$$g = \begin{bmatrix} -2\sum_{i=1}^{n} r_i (1 - e_i) \\ 2\alpha^* \sum_{i=1}^{n} r_i x_i e_i \end{bmatrix}, \qquad H = 2J^T J - 2\begin{bmatrix} 0 & \text{symmetric} \\ \sum_{i=1}^{n} r_i x_i e_i & \alpha^* \sum_{i=1}^{n} r_i x_i^2 e_i \end{bmatrix}.$$

In the Fortran program discussed at the end of this chapter, we have written the subroutine that evaluates the Hessian so that it is easy to include the second term or not. The program calls the Newton-Raphson subroutne twice, the first time does Gauss-Newton and the second time it uses the full Hessian. The output is as follows:

```
Draper and Smith example: Gauss-Newton:

         alpha          beta        SS Res
------------------------------------------
       .300000       .020000       .026315
       .841649       .100682      4.488134
       .390135       .100419       .005006
       .390135       .101605       .005002
       .390140       .101632       .005002
ier =     0

Draper and Smith example: Full Hessian:

         alpha          beta        SS Res
------------------------------------------
       .300000       .020000       .026315
       .259564       .024729       .012877
ier =     2
```

Note that Gauss-Newton converges quickly, while using the full Hessian immediately results in a Hessian that is not positive definite (this is what `ier=2` indicates; see the discussion at the end of the chapter for more details). This is a common phenomenon in nonlinear regression and using Newton-Raphson in general.

Iteratively Reweighted Least Squares

We have seen that the Gauss-Newton procedure for nonlinear regression (equivalent to scoring for the Gaussian case) led to solving a set of ordinary least squares normal equations at each iteration. It turns out that there are a number of estimation procedures that lead at each iteration to solving weighted least squares normal equations. Such an iterative procedure is called iteratively reweighted least squares and it naturally arises when we have m independent random variables $y_1, \ldots, y_m$ whose likelihood depends on a set of parameters η which in turn depends functionally on a smaller set of parameters θ. In Gaussian nonlinear regression for example, the dependent variables $y_1, \ldots, y_n$ are normally distributed with means $\mu_1, \ldots, \mu_n$ which in turn depend on the vector θ by $\mu_i = f(X_i; \theta)$. Another example is logistic regression wherein we observe $y_1, \ldots, y_m$ which are assumed to be independent random variables with y_i having the binomial distribution with parameters n_i and

$$p_i = \frac{1}{1 + e^{-X_i^T \beta}}, \qquad i = 1, \ldots, m,$$

where for each i, $X_i = (X_{i1}, \ldots, X_{ip})^T$ is a vector of covariates, and the vector $\beta = (\beta_1, \ldots, \beta_p)^T$ contains the parameters of interest. For example, a negative value of β_j indicates that the jth covariate has a negative partial effect on the success probability. Thus we have a vector $p = (p_1, \ldots, p_m)^T$ of parameters which are a nonlinear function of a usually much smaller set β of parameters.

Thus in general we consider a negative log likelihood L which depends on a vector η of n parameters which are functionally related to a vector θ of p parameters. If we use the chain rule when differentiating L and if we use the expectation of the Hessian instead of the Hessian itself (that is, use the method of scoring), then the iterations one obtains are of the form

$$\theta_{i+1} = \theta_i - d_i, \qquad W_i^T A_i W_i d_i = W_i^T z_i,$$

where W is $(n \times p)$, A is $(n \times n)$, z is $(n \times 1)$, and

$$W_{jk} = \frac{\partial \eta_j}{\partial \theta_k}, \qquad A_{jk} = -\mathrm{E}\left(\frac{\partial^2 L}{\partial \eta_j \partial \eta_k}\right) = \mathrm{E}\left(\frac{\partial L}{\partial \eta_j}\frac{\partial L}{\partial \eta_k}\right), \qquad z_j = \frac{\partial L}{\partial \eta_j}.$$

Thus at each iteration, one needs to solve weighted least squares normal equations for the regression of z on AW with error covariance matrix A. This can be done via the modified Cholesky decomposition method described in Section ????.

For Gaussian nonlinear regression, the matrix W is the same as the Jacobian J, while since $\partial L/\partial f_i = -(y_i - f_i)/\sigma^2$, we have that $z = -r/\sigma^2$ and the matrix A is σ^2 times the identity matrix, and thus iteratively reweighted least squares is also identical to Gauss-Newton and the method of scoring.

For logistic regression, if we let L be the negative of the log likelihood of β, then

$$L = -\sum_{i=1}^{m} \left[\log \binom{n_i}{y_i} + y_i \log p_i + (n_i - y_i) \log(1 - p_i) \right],$$

from which we get

$$z_j = \frac{\partial L}{\partial p_j} = \frac{n_j - y_j}{1 - p_j} - \frac{y_j}{p_j}, \qquad W_{jk} = \frac{\partial p_j}{\partial \beta_k} = \frac{X_{jk} e^{-X_j^T \beta}}{1 + e^{-X_j^T \beta}} = X_{jk}(1 - p_j),$$

and

$$\frac{\partial^2 L}{\partial p_j \partial p_k} = \delta_{j-k} \left[\frac{n_j - y_j}{(1 - p_j)^2} + \frac{y_j}{p_j^2} \right],$$

where δ_v is the Kronecker delta function, that is, δ_v is one if $v = 0$ and zero otherwise. Thus we have

$$W = \text{Diag}\{1 - p_j\} X, \qquad A = -\text{E}\left(\frac{\partial^2 L}{\partial p \partial p} \right) = \text{Diag}\left\{ \frac{n_j}{p_j(1 - p_j)} \right\},$$

and thus

$$\begin{aligned} W^T A W &= X^T \text{Diag}\{1 - p_j\} \text{Diag}\left\{ \frac{n_j}{p_j(1 - p_j)} \right\} \text{Diag}\{1 - p_j\} X \\ &= X^T \text{Diag}\left\{ \frac{n_j(1 - p_j)}{p_j} \right\} X = X^T \text{Diag}\{n_j e^{-X_j^T \beta}\} X. \end{aligned}$$

To find starting values, we note that

$$r_i = -\log\left(\frac{1 - p_i}{p_i} \right) = \sum_{j=1}^{p} \beta_j X_{ij}, \qquad i = 1, \ldots, m,$$

and thus if we let $\hat{p}_i = y_i/n_i$, we can do ordinary least squares regression of the resulting $\hat{r}$'s on the $(m \times p)$ matrix X of covariates.

A Fortran Implementation

We have listed below a Fortran implementation of the two examples we have described above. There are a number of important features of Fortran used in this program, in particular, the use of the `external` statement and named `common`. The program calls a series of subprograms that are not in the listing, including `runif` for generating U(0,1) variables, `qsort` to sort a vector, `slr` to do simple linear regression, and `solvch` to solve a system of equations via the modified Cholesky decomposition.

Passing a Subroutine Name as an Argument: The External Statement

At the end of the listing is a subroutine called `nr` that is designed to do the Newton-Raphson procedure for any function L. It calls a subroutine called `grdhss` which has input arguments `theta`, `n`, and `ndim` and output arguments `g` and `h` which are the gradient and Hessian of the function being minimized. The subroutine `grdhss` is actually a 'dummy name' that is passed to `nr` through its argument list. In the example, `nr` gets called twice, once for the Draper and Smith nonlinear regression example in which case the gradient and Hessian subroutine is called `dsgh`, and once for the Weibull example in which case the subroutine is called `weibgh`. When one passes the name of a subroutine or function to a subroutine or function, it must be declared in an `external` statement (see the beginning of the main program in the listing).

The Common and Named Common Statements

The `nr` subroutine has no way of knowing what information other than a vector `theta` the gradient and Hessian subroutine it is calling needs to know in order to do its work. Thus this information can't be passed to `grdhss` in its argument list (you can't have varying numbers of arguments in a call to a subroutine either). Thus the information the real gradient and Hessian routine needs (as opposed to the dummy `grdhss`) gets passed in a `common` statement (see the beginning of the main program and the beginning of each of `dsgh` and `weibgh` to see how this works).

If we were only using one gradient and Hessian routine, the `common` statement would not have to be 'named', that is, the first statement would just be `common dsx(44),dsy(44),nds,iopt,ssr`. In this case we have to have two such statements with each one named.

There is one other thing to be sure to notice. We wanted to define the Draper and Smith data in a `data` statement rather than reading it in. Unfortunately, one is not allowed to have `common` variables in a data statement so we defined another pair of vectors and defined them in a `data` statement and then copied them to the vectors in the `common`.

Checking for Convergence

In the `nr` subroutine we have checked for convergence of successive iterations by seeing if all of the elements of successive iterations are within `eps` of each other in terms of 'relative absolute error,' which for numbers x and y is defined to be $|x - y|/|x|$. Since this denominator can be close to zero, we actually check whether $|x - y| < \epsilon|x|$.

```
      subroutine nr(theta,n,ndim,maxit,eps,grdhss,g,h,al,d,wk,
     1              del,ier)
c---------------------------------------------------------------
c
c  Subroutine to perform Newton-Raphson minimization of a
c  function of n variables.
c
c  Input:
c        theta:  vector of length n with starting values
c        n
c        ndim:   row dimension of h and al in calling routine
c        maxit:  maximum number of iterations to perform
c        eps:    convergence criterion, how close successive
c                iterations must be to conclude that
c                convergence has been reached.
c        grdhss: name of the subroutine that will calculate the
c                gradient and Hessian (must be declared
c                external in calling routine).
c
c  Work:
c       al:      (n by n) matrix having ndim rows
c       d, wk, del: vectors of length n
c
c  Output:
c       theta:   value of variables at last iteration
c       g, h:    gradient and Hessian at final iteration
c       ier:     error indicator (0 means none, -1 means no
c                convergence, + means H not positive definite).
c
c  Subprograms Called: grdhss, solvch (mchol)
c
c---------------------------------------------------------------

      double precision theta(n),g(n),h(ndim,1),al(ndim,1),d(n),
     1          wk(n),del(n)

      double precision eps

      do 100 it=1,maxit

              call grdhss(theta,n,ndim,g,h)
```

```
               call solvch(h,g,n,ndim,al,d,wk,del,ier)

               if(ier.ne.0) return

               do 10 i=1,n
 10            if(abs(del(i)).ge.(eps*abs(theta(i)))) go to 20

               ier=0
               return

 20            if(it.ge.maxit) then
                      ier=-1
                      return
               endif

               do 30 i=1,n
 30            theta(i)=theta(i)-del(i)

100     continue

        return
        end
```

The Output

The computer program produces the following output:

```
Draper and Smith example: Gauss-Newton:

         alpha          beta        SS Res
------------------------------------------
      0.300000      0.020000      0.026315
      0.841649      0.100682      4.488132
      0.390135      0.100419      0.005006
      0.390135      0.101605      0.005002
      0.390140      0.101632      0.005002
ier =      0

Draper and Smith example: Full Hessian:

         alpha          beta        SS Res
------------------------------------------
      0.300000      0.020000      0.026315
      0.259564      0.024729      0.012877
ier =      2

For Weibull MLE example:

         alpha          beta        loglik
------------------------------------------
      3.667686      1.321373      9.309643
      5.299813      1.648922     11.631004
      6.708105      1.811216     12.248871
      7.504770      1.883775     12.345123
      7.697009      1.899736     12.348808
      7.706138      1.900465     12.348829
ier =      0
```

Another Example: MLE's for the Logistic Distribution

Let $X_1, \ldots, X_n$ be a random sample from the logistic distribution having pdf

$$f(x) = \frac{e^{-y}}{(1 + e^{-y})^2},$$

where $y = (x - \alpha)/\beta$ for $-\infty < x < \infty$, $-\infty < \alpha < \infty$, and $\beta > 0$. Then the negative of the log likelihood of $\theta = (\alpha, \beta)^T$ given the X's is given by

$$L(\theta) = n \log \beta + \sum_{i=1}^{n} y_i + 2 \sum_{i=1}^{n} \log(1 + e^{-y_i}),$$

where $y_i = (X_i - \alpha)/\beta$. From this we have that the gradient and Hessian are given by

$$g = \begin{bmatrix} -\dfrac{1}{\beta} \displaystyle\sum_{i=1}^{n} \dfrac{1 - z_i}{1 + z_i} \\ \dfrac{n}{\beta} - \dfrac{1}{\beta} \displaystyle\sum_{i=1}^{n} \dfrac{y_i(1 - z_i)}{1 + z_i} \end{bmatrix},$$

and

$$H = \begin{bmatrix} \dfrac{2}{\beta^2} \displaystyle\sum_{i=1}^{n} \dfrac{z_i}{(1 + z_i)^2} & \text{Sym} \\ \dfrac{1}{\beta^2} \displaystyle\sum_{i=1}^{n} \left[\dfrac{2 z_i y_i}{(1 + z_i)^2} + \dfrac{1 - z_i}{1 + z_i} \right] & \dfrac{2}{\beta^2} \displaystyle\sum_{i=1}^{n} \left[\dfrac{z_i y_i^2}{(1 + z_i)^2} + \dfrac{y_i(1 - z_i)}{1 + z_i} \right] - \dfrac{n}{\beta^2} \end{bmatrix},$$

where $z_i = e^{-y_i}$.

The subroutine `lgstgh` calculates g and H for a given θ. One thing to notice is that the line `zi=exp(-yi)` can lead to an error of `yi` is too large. Subroutine `lgstgh` handles this by skipping any term in the sum for which the absolute value of `yi` is too large.

```
      subroutine lgstgh(theta,nr,ndim,g,h)
c------------------------------------------------------------------
c
c  Evaluate the gradient and Hessian for the negative
c  log likelihood of the prameters of a Logistic
c  Distribution based on a sample x of size n.
c
c------------------------------------------------------------------

      dimension theta(nr),g(nr),h(ndim,nr)
      double precision yi,zi,opzi,omzi,ri,si
      common /logist/ x(100),n

      alpha=theta(1)
      beta=theta(2)

      g1=0.0
      g2=0.0
      h11=0.0
      h12=0.0
      h22=0.0

      do 10 i=1,n

              yi=(x(i)-alpha)/beta
              if(abs(yi).gt.60) go to 10
              zi=exp(-yi)
              opzi=1.+zi
              omzi=1.-zi
              ri=omzi/opzi
              si=zi/(opzi*opzi)

              g1=g1+ri
              g2=g2+yi*ri

              h11=h11+si
              h12=h12 + 2.*si*yi+ri
              h22=h22 + si*yi*yi+ri*yi

   10 continue

```

```
      g(1)=-g1/beta
      g(2)=(n-g2)/beta

      beta2=beta*beta
      h(1,1)=2*h11/beta2
      h(1,2)=h12/beta2
      h(2,1)=h(1,2)
      h(2,2)=(2*h22-n)/beta2

      return
      end
```

There are two natural methods for finding starting values for the Newton-Raphson procedure. Since $\mathrm{E}(X) = \alpha$ and $\mathrm{Var}(X) = \beta^2\pi^2/3$, it is very easy to find the method of moments estimators

$$\hat{\alpha}_1 = \bar{X}, \qquad \hat{\beta}_1 = \sqrt{\frac{3\hat{\sigma}^2}{\pi^2}},$$

where $\hat{\sigma}^2 = \sum_{i=1}^n (X_i - \bar{X})^2/n$.

Since the quantile function of X is given by

$$Q(u) = \alpha - \beta \log\left(\frac{1-u}{u}\right),$$

we can estimate $Q(u)$ at $u_i = (i - 1/2)/n$ for $i = 1, \ldots, n$ by the order statistics $X_{(1)} < \cdots < X_{(n)}$ and then get another set of starting values $\hat{\alpha}_2$ and $\hat{\beta}_2$ by the simple linear regression (with intercept) of $X_{(i)}$ on $-\log((1-u_i)/u_i)$.

A Program Comparing Logistic Parameter Estimators

In the Fortran program listed below we do a simulation of three estimators; the method of moments, the 'regression of quantiles,' and the Newton-Raphson MLE's using the method of moments estimates as starting values. The program asks the user for values of α and β, for a random number generator seed, for a number of sample sizes, and the values of sample sizes, and the number of samples to generate for each sample size. It also asks for a value of an integer called `iopt` which should be 1 if the user wants the program to display the which sample it is doing every 100th sample and a 0 otherwise. This is useful since the program may execute for a long time if the number of replciations is large and printing every 100th sample number allows the user to monitor the program's progress.

For each sample size, the program calculates the average and mean square error (average squared distance from the true parameter value) for each of the three estimators of the two parameters. It also calculates the average of the standard errors of the MLE's of α and β where the standard error is calculated in two ways; first by evaluating the inverse of the information matrix

$$I(\theta) = \frac{n}{3\beta^2}\begin{bmatrix} 1 & 0 \\ 0 & 1 + \pi^2/3 \end{bmatrix}.$$

at the MLE of β, and secondly from the inverse of the final Hessian matrix in the Newton-Raphson algorithm. For each sample these two methods of finding standard errors are used to see if the confidence interval for the parameters include the true values and the number of times it does is reported.

Finally, since it is possible for the Newton-Raphson method to not converge or for a Hessian to be not positive definite, the program keeps generating samples until the desired number of samples that have no Newton-Raphson errors are actually generated. Th program keeps track of the number of times te two types of errors occur (the variable `ier1` is the number of times the iterations didn't converge while `ier2` is the number of times a Hessian wasn't positive definite.

```
      dimension g(10),h(10,10),theta(2),al(10,10),d(10),
```

```
     1            wk(10),del(10)
      dimension u(100),nn(10)
      character*40 infile
      real*4 smean,svar
      double precision dseed
      external lgstgh
      common /logist/ x(100),n
      data ndim/10/

      pi=4.0*atan(1.0)

c----------------------------------------
c   Get user input and open output file:
c-----------------------------------------

      write(*,1)
    1 format(' Enter alpha, beta, dseed, nsamps, nreps, iopt: ',$)
      read(*,*) alpha,beta,dseed,nsamps,nreps,iopt
      write(*,3)
    3 format(' Enter sample sizes: ',$)
      read(*,*) (nn(i),i=1,nsamps)
      write(*,4)
    4 format(' Enter output file name: ',$)
      read(*,5) infile
    5 format(a40)

      open(1,file=infile,status='new')

      write(1,6)
    6 format(1x,'LOGISTIC SIMULATION SUMMARY'/,1x,27(1h-))
      write(1,7) alpha,beta,dseed
    7 format(1x,'True alpha = ',f10.6/,1x,'True beta =',f10.6/,
     1        1x,'seed = ',f12.0)
      write(1,8)
    8 format(/,27x,5hALPHA,17x,4hBETA/1x,55(1h-))
      write(1,9)
    9 format(1x,21x,4hMEAN,7x,3hMSE,6x,4hMEAN,7x,3hMSE/1x,55(1h-))

c--------------------------
c   Loop over sample size:
c--------------------------

      do 300 in=1,nsamps

      n=nn(in)
      write(1,10) n
   10 format(1x,'n = ',i3)

      do 11 i=1,n
      u(i)=(i-0.5)/n
   11 u(i)=-log((1.-u(i))/u(i))

      ier0=0
      ier1=0
      ier2=0

      a1bar=0.0
      a2bar=0.0
      a3bar=0.0
      a1mse=0.0
      a2mse=0.0
      a3mse=0.0
      b1bar=0.0
      b2bar=0.0
      b3bar=0.0
      b1mse=0.0
      b2mse=0.0
      b3mse=0.0
      vhabar=0.0
      vhbbar=0.0
      vvabar=0.0
      vvbbar=0.0
      ncova1=0
      ncova2=0
      ncovb1=0
      ncovb2=0

c------------------------
c   Loop over reps:
c------------------------
```

```

        do 100 ir=1,nreps

                if(iopt.eq.1.and.(100*(ir/100).eq.ir)) write(*,*) n,ir

 12             continue

                call rlogist(dseed,n,alpha,beta,x)

c   Method of Moments:

                a1=smean(x,n)
                b1=sqrt(3.*svar(x,n))/pi

c   Regression of quantiles:

                call qsort(x,n)
                call slr(u,x,n,a2,b2)

c   Newton-Raphson MLE:

                theta(1)=a1
                theta(2)=b1
                call nr(theta,2,ndim,20,.0001,lgstgh,g,h,al,d,wk,
     1                  del,ier)
                a3=theta(1)
                b3=theta(2)

                if(ier.eq.0) ier0=ier0+1
                if(ier.lt.0) ier1=ier1+1
                if(ier.gt.0) ier2=ier2+1

c   Go back to get another sample if ier.ne.0:

                if(ier.ne.0) go to 12

c   Update means and mse's for the three estimators:

                a1bar=a1bar+a1
                b1bar=b1bar+b1
                a1mse=a1mse+(a1-alpha)**2
                b1mse=b1mse+(b1-beta)**2

                a2bar=a2bar+a2
                b2bar=b2bar+b2
                a2mse=a2mse+(a2-alpha)**2
                b2mse=b2mse+(b2-beta)**2

                a3bar=a3bar+a3
                b3bar=b3bar+b3
                a3mse=a3mse+(a3-alpha)**2
                b3mse=b3mse+(b3-beta)**2

c   Update mean of two estimates of standard errors:

                hd=h(1,1)*h(2,2)-h(1,2)*h(2,1)
                vha=h(2,2)/hd
                vhb=h(1,1)/hd
                vhabar=vhabar+vha
                vhbbar=vhbbar+vhb

                vva=3.*b3*b3/n
                vvb=3.*b3*b3/(n*(1.+pi*pi/3.))
                vvabar=vvabar+vva
                vvbbar=vvbbar+vvb

c   See if confidence intervals for two standard errors cover:

                a3i=abs(a3-alpha)
                b3i=abs(b3-beta)

                if(vha.le.0.0.or.vhb.le.0.0) go to 20

                if(a3i.lt.1.96*sqrt(vva))   ncova1=ncova1+1
                if(a3i.lt.1.96*sqrt(vha))   ncova2=ncova2+1
                if(b3i.lt.1.96*sqrt(vvb))   ncovb1=ncovb1+1
                if(b3i.lt.1.96*sqrt(vhb))   ncovb2=ncovb2+1

 20             continue

```

```
c------------------
c   End reps loop:
c------------------

 100    continue

c-------------------------------------
c   Get true asymptotic standard error:
c-------------------------------------

        v1=3.*beta*beta/n
        v2=3.*beta*beta/(n*(1.+pi*pi/3.))

c-------------------------
c   Write out information:
c-------------------------

        write(1,110) a1bar/nreps,a1mse/nreps,b1bar/nreps,b1mse/nreps
 110    format(1x,12x,3hMOM,4f10.6)
        write(1,111) a2bar/nreps,a2mse/nreps,b2bar/nreps,b2mse/nreps
 111    format(1x,11x,4hQREG,4f10.6)
        write(1,112) a3bar/nreps,a3mse/nreps,b3bar/nreps,b3mse/nreps
 112    format(1x,12x,3hMLE,4f10.6)
        write(1,113) v1,v2
 113    format(1x,8x,7HTRUE SE,10x,f10.6,10x,f10.6)
        write(1,114) vvabar/nreps,vvbbar/nreps
 114    format(1x,4x,11hAVE INFO SE,10x,f10.6,10x,f10.6)
        write(1,115) vhabar/nreps,vhbbar/nreps
 115    format(1x,4x,11hAVE HESS SE,10x,f10.6,10x,f10.6)
        write(1,116) ier0,ier1,ier2
 116    format(1x,1x,14hIER0,IER1,IER2,3i10)
        write(1,117) ncova1,ncovb1
 117    format(1x,2x,13hINFO CI COVER,10x,i10,10x,i10)
        write(1,118) ncova2,ncovb2
 118    format(1x,2x,13hHESS CI COVER,10x,i10,10x,i10/1x,55(1h-))

 300    continue

        stop
        end
```

An Example Output

Here is the output for a sample run having 100,000 samples for samples of size 10, 20, 40, and 100. Note that the MLE seems to be slightly better than the other two estimators and that the observed MSE agrees quite well with the theoretical asymptotic variance even for the small sample sizes. Further, the confidence interval coverages seem close to the 95% nominal confidence coefficient.

```
LOGISTIC SIMULATION SUMMARY
---------------------------
True alpha =    .000000
True beta =  1.000000
seed =       12345.

                          ALPHA                BETA
-------------------------------------------------------
                     MEAN       MSE      MEAN       MSE
-------------------------------------------------------
n =  10
            MOM   .002560   .327587   .961811   .076593
           QREG   .002560   .327587   .971834   .075911
            MLE   .002341   .306029   .940377   .071973
        TRUE SE             .300000             .069932
    AVE INFO SE             .285823             .066628
    AVE HESS SE             .291413             .066353
 IER0,IER1,IER2    100000        19        49
  INFO CI COVER               91016               86581
  HESS CI COVER               91222               86521
-------------------------------------------------------
n =  20
            MOM  -.001721   .164009   .980222   .038741
           QREG  -.001721   .164009   .982977   .037710
            MLE  -.001445   .151161   .969890   .035712
        TRUE SE             .150000             .034966
    AVE INFO SE             .146324             .034110
    AVE HESS SE             .147668             .034058
 IER0,IER1,IER2    100000        31         6
  INFO CI COVER               93052               90705
  HESS CI COVER               93146               90703
-------------------------------------------------------
n =  40
            MOM   .000132   .081978   .990527   .019696
           QREG   .000132   .081979   .990653   .019131
            MLE  -.000376   .075247   .985780   .017673
        TRUE SE             .075000             .017483
    AVE INFO SE             .074193             .017295
    AVE HESS SE             .074533             .017280
 IER0,IER1,IER2    100000        59         0
  INFO CI COVER               94065               92892
  HESS CI COVER               94119               92891
-------------------------------------------------------
n = 100
            MOM   .000353   .033357   .996293   .007935
           QREG   .000353   .033357   .995616   .007743
            MLE   .000466   .030556   .994402   .007008
        TRUE SE             .030000             .006993
    AVE INFO SE             .029874             .006964
    AVE HESS SE             .029927             .006962
 IER0,IER1,IER2    100000        97         0
  INFO CI COVER               94429               94148
  HESS CI COVER               94462               94146
-------------------------------------------------------
```

CHAPTER 5

Introduction to Computer Graphics

In this chapter, we consider how graphics is done on a computer. We begin by considering the hardware involved; monitors, printers, and so on. Then we consider various graphics primitives (drawing lines and writing text, for example) and then how to design fairly sophisticated graphs such as two and three dimensional plots. Finally, we show how to use widely available programming tools (MS Windows, X-Windows, and Postscript) to create full purpose graphics packages.

Computer Monitors

Most computer screens are composed of a rectangular grid of dots called picture elements (popularly called "pixels" or "pels"). These dots are such that if they are all turned on in the same color in a region of the screen, then that region looks as though it were solidly drawn in that color. Typically, a computer will have a special set of high speed memory chips which contain a description of what color each pixel is on the screen (a bitmap) and a special processing unit that constantly redraws (refreshes) the screen. The more pixels a screen has, the better geometric shapes look on the screen. For example, If one wants to draw a line staring with one pixel and ending with another, the line will not look solid because of the discreteness of the pixels. This effect is commonly called the "jaggies" and can be visualized by thinking of a page of graph paper and drawing a line from one square to another by filling in those squares and "the ones in between."

The inadaquacy of lines on a pixel device is of course less pronounced if the pixels are very small than if they are large, that is, a device having small pixels is said to have "higher resolution." For example, a standard VGA monitor on a PC has 640 rows of pixels with each row having 480 pixels.

There are two prices to pay for high resolution. First, they can be much more expensive to produce, and second, the amount of storage required to represent what is to be displayed on the device is much more. For example, if one only wanted to use black and white on a VGA screen, the first eight pixels (white or black) can be represented by the eight bits (1 or 0) of the first byte of a region in memory, the next byte could be the next eight pixels, and so on. Thus it would take 80 bytes of memory to store each row of pixels or a total of $80 \times 480 = 38,400$ bytes to represent the screen. To have $256 = 2^8$ colors for each pixel would require an entire byte for each pixel, that is a total of 307,200 bytes.

Printers

Many printers work in much the same way as a screen except of course there need be no refreshing. The storage requirement for a printer can be much worse, due to its very high resolution. For an HP laserjet, there are 300 dots per inch in both the horizontal and vertical directions. Thus an entire page (8 by 10.5 inches say) would require 3,150 rows of dots with each row having 2,400 dots. A row would require 300 bytes for a total of 945,000 bytes for a bitmap ofthe entire page. Thus to do full page graphics on a Laserjet requires the printer to have at least one MByte of memory.

Some printers do not require bitmaps be sent to the printer. The most notable example is a Postscript printer. To draw a figure made up of a series of lines, one need only send a series of line drawing instructions (each requiring only a few bytes) and the printer can understand and execute the instructions. Thus the information sent to a postscript printer is typically much smaller than the bitmap that has to be sent to other printers. Further, postscript instructions can be easily modified so that the resulting figure can be of almost any size, that is, scaling is very easily done as opposed to bitmaps which can seldom be scaled.

Compressing Bitmaps

One solution to the problem of needing very large files to store bitmaps is to store them in compressed form. In many cases this is done using the idea of run length encoding (RLE). While there are many forms of this idea, most of them consist of using the fact that most mathematical and statistical graphic figures have a large amount of blank space.

The simplest form of RLE is of the following form. Suppose we have a black and white image on the screen that we want to compress. The process starts by noting the color of the pixel at the left of the topmost line of pixels and works across and then down the lines (the process need not go in this order but for convenience we look at it this way), counting the length of each "run" of pixels before the color changes (only the color of the upper left corner pixel need be recorded). Thus if we were compressing a standard $X - Y$ plot of a curve, many rows of pixels would all be the same color and a huge compression would be achieved.

Drawing Text on a Graphics Screen

Each letter on a screen is actually represented by a bitmap telling the colors of all the pixels in the rectangular area occupied by the character (typically, telling whether each pixel is in the "foreground" or "background" color). For example, in the high resolution graphics mode on an original IBM PC, each character occupies an 8 by 8 pixel region. The letter A is represented by

```
. . 1 1 . . . .    48
. 1 1 1 1 . . .   128
1 1 . . 1 1 . .   204
1 1 . . 1 1 . .   204
1 1 1 1 1 1 . .   252
1 1 . . 1 1 . .   204
1 1 . . 1 1 . .   204
. . . . . . . .     0
```

where a '.' represents a 0 in the representation, and we have listed the decimal number corresponding to the binary representation given by each row of pixels. PC's have a special place in memory where the bitmaps for the first 128 ASCII characters are stored.

Turning on Pixels

All graphics operations on a monitor (and any bitmap printer) consist ultimately of the operation of "turning on a pixel," that is, making a specified pixel (or printing dot) a specified color. As we saw above this includes drawing text. In the rest of this chapter we discuss how turning on pixels can be used to draw lines and ultimately very complicated graphs, including 3D and contour plots.

Some computers have special hardware for drawing lines or circles or polygons, but we will only consider basic algorithms for these operations.

Bresenham's Line Drawing Algorithm

Bresenham's line drawing algorithm is a method for determining which pixels between the end points of a line should be made the color of the line. The algorithm is remarkable in that it only requires arithmetic operations on integers, not on floating point numbers.

Before describing the algorithm in detail, there are a number of things to notice.

1. Unless the line happens to be horizontal, vertical, or falls on a diagonal of the grid of pixels, the line will not appear perfectly straight on the screen.
2. The line should be "connected", that is, there should be no discernible gaps in the set of pixels that make up the line. This means that if the line is "steep", that is, it travels more rows of pixels than it does columns, then a pixel in each row between the starting and ending row must be "turned on". On the other hand, a pixel in each column of a shallow line must be turned on. Thus steep and shallow lines are handled separately, and one loops over rows for steep lines and over columns for shallow lines. The
3. For a fixed row or column in a loop, one has a choice of two pixels that can be turned on. For a steep line with a positive slope for example, when going up from one row to the next in the loop, one has to choose which of two columns contains the pixel to be turned on, either the same column as the one for the previous row or the column to the right of the one for the previous row. These two choices are referred to as a "nondiagonal move" or a "diagonal move" and one must make a choice between such moves for each of four possible cases: 1) steep line with positive slope, 2) steep line with negative slope, 3) shallow line with positive slope, and 4) shallow line with negative slope.

The Algorithm

We will consider the pixel in the lower left hand corner of the screen to be in column zero and row zero and use the notation (c, r) to represent the pixel in column c and row r. Suppose we want to draw a line from pixel (c_0, r_0) to pixel (c_1, r_1), where $c_0 < c_1$, that is, (c_0, r_0) is to the left of (c_1, r_1). If we consider these pixels to be points in Euclidean space, then the true line connecting these two points is the set of (x, y) satisyfing

$$f(x, y) = (y - r_0)\delta_c - (x - c_0)\delta_r = 0,$$

where $\delta_c = c_1 - c_0$ and $\delta_r = r_1 - r_0$. Notice that $\delta_c > 0$ and that δ_r has the same sign as the slope of the line. Further, (x, y) is a point above (below) the line if and only if $f(x, y) > 0$ ($f(x, y) < 0$), while the line is steep (shallow) if $\delta_r > \delta_c$ ($\delta_r < \delta_c$).

In the shallow line case, we will loop over the columns from c_0 to c_1 (that is, from left to right), at each step determining if the row stays the same or increases (in the positive slope case) or decreases (in the negative slope case) by one. In the steep line case, we loop over the rows (from r_0 up to r_1 for positive slope or from r_0 down to r_1 for negative slope), at each step determining if the column stays the same or increases by one.

The Algorithm for a Shallow Line with Positive Slope

We consider how Bresenham's algorithm decides if a diagonal or nondiagonal move is to be made in the shallow line with positive slope case. The other three cases are done similarly and are summarized later. Suppose at the end of one step, the pixel (x_i, y_i) was turned on. Then the next one to be turned on is either $p_N = (x_i + 1, y_i)$, a nondiagonal move, or $p_D = (x_i + 1, y_i + 1)$, a diagonal move. Consider the midpoint m_i of these two pixels, that is $m_i = (x_i + 1, y_i + 1/2)$. Then the true line is closer to p_N (p_D) if it is below (above) the point m_i, that is, if $f(m_i) > 0$ ($f(m_i) < 0$). Thus deciding which move is to be made at a step requires only knowing whether $f(m_i)$ is positive or negative.

The first key idea in Bresenham's algorithm is the fact that $f(m_i)$ can be calculated recursively, that is

$$\begin{aligned} f(m_i) &= f(x_i + 1, y_i + 1/2) = (y_i + 1/2 - r_0)\delta_c - (x_i + 1 - c_0)\delta_r \\ &= \begin{cases} (y_{i-1} + 1 + 1/2 - r_0)\delta_c - (x_{i-1} + 1 + 1 - c_0)\delta_r, & \text{if last step diag} \\ (y_{i-1} + 1/2 - r_0)\delta_c - (x_{i-1} + 1 + 1 - c_0)\delta_r, & \text{if last step nondiag} \end{cases} \\ &= \begin{cases} f(m_{i-1}) + (\delta_c - \delta_r), & \text{if last step diagonal} \\ f(m_{i-1}) - \delta_r, & \text{if last step nondiagonal} \end{cases}. \end{aligned}$$

Since all we care about is the sign of $f(m_i)$, we can just as well keep track of $d_i = 2f(m_i)$ which has the advantage that it is guaranteed to be integer valued. Thus we have

$$d_i = 2f(m_i) = \begin{cases} d_{i-1} + 2(\delta_c - \delta_r), & \text{if last step diagonal} \\ d_{i-1} - 2\delta_r, & \text{if last step nondiagonal} \end{cases}.$$

Another key fact is that the terms $2(\delta_c - \delta_r)$ and $-2\delta_r$, referred to as the diagonal and nondiagonal increments respectively, are the same at every step. Thus they can be determined before starting the loop, and whenever a move is determined, the value of d_i can be updated for the next step. The final part of the recursion is to notice that the starting value d_0 is given by

$$d_0 = 2f(c_0 + 1, r_0 + 1/2) = \delta_c - 2\delta_r.$$

Summary for all Four Cases

Inspection of the four cases (remembering to always work from left to right) shows that a nondiagonal move is chosen when $f(m_i) > 0$ for the shallow line with positive slope and steep line with negative slope cases and when $f(m_i) < 0$ for the other two cases. So that we will always be checking for positivity of whatever we're keeping track of in the recursion, we will keep track of

$$a_i = \begin{cases} 2f(m_i), & \text{in the shallow positive and steep negative cases} \\ -2f(m_i), & \text{in the shallow negative and steep positive cases} \end{cases}.$$

If we work through the algebra for all four cases, we find that a_0 and the diagonal and nondiagonal increments (call them δ_D and δ_N respectively) are given by

$$a_0 = \begin{cases} \delta_c - 2|\delta_r|, & \text{for shallow lines} \\ |\delta_r| - 2\delta_c, & \text{for steep lines} \end{cases}$$
$$\delta_D = \begin{cases} 2(\delta_c - |\delta_r|), & \text{for shallow lines} \\ 2(|\delta_r| - \delta_c), & \text{for steep lines} \end{cases}.$$
$$\delta_N = \begin{cases} -2|\delta_r|, & \text{for shallow lines} \\ -2\delta_c, & \text{for steep lines} \end{cases}$$

In the next section we give a Fortran implementation of this general algorithm.

A Fortran Implementation of the Algorithm

```
      subroutine line(icol1,irow1,icol2,irow2,kolor)
c------------------------------------------------------
c
c   Draw a line connecting pixel (icol1,irow1) and
c   (icol2,irow2) in color kolor.
c
c------------------------------------------------------
c
      integer delc,delr,rinc,dinc,a
c
c   Get left point in (ic0,ir0), right point in (ic1,ir1):
c
      ic0=min0(icol1,icol2)
      ic1=max0(icol1,icol2)
      ir0=irow1
      ir1=irow2
      if(icol1.gt.icol2) then
             ir0=irow2
             ir1=irow1
      endif
c
c   Horizontal, vertical (diagonal is handled in shallow
c   case below):
c
      if(ir0.eq.ir1) then
             do 10 i=ic0,ic1
 10          call wrdota(i,ir0,kolor)
             return
      endif
      if(ic0.eq.ic1) then
             do 20 i=min0(ir0,ir1),max0(ir0,ir1)
 20          call wrdota(ic0,i,kolor)
             return
      endif
c
c   Other cases:
c
      delc=ic1-ic0
      delr=ir1-ir0
      idelc=iabs(delc)
      idelr=iabs(delr)
      rinc=idelr/delr
      ix=ic0
      iy=ir0
c
c   Shallow:
c
      if(delc.ge.idelr) then
             dinc=2*(delc-idelr)
             ndinc=-2*idelr
             call wrdota(ic0,ir0,kolor)
             a=delc-2*idelr
             do 40 ix=ic0+1,ic1
                    if(a.gt.0) then
                           a=a+ndinc
                    else
                           a=a+dinc
                           iy=iy+rinc
                    endif
                    call wrdota(ix,iy,kolor)
 40          continue
             return
c
c   Steep:
c
      else
             dinc=2*(idelr-delc)
             ndinc=-2*delc
             call wrdota(ic0,ir0,kolor)
             a=idelr-2*delc
             do 50 iy=ir0+rinc,ir1,rinc
                    if(a.gt.0) then
                           a=a+ndinc
                    else
                           ix=ix+1
                           a=a+dinc
                    endif
```

```
                    call wrdota(ix,iy,kolor)
 50             continue
        endif
        return
        end
```

The Basic $X - Y$ Plot

Statistics is filled with $X-Y$ plots, that is, graphs that have horizontal and vertical axes with tic marks and numbers labeling the values on the tic marks, a graph of some kind (barplot, scatterplot, function plot, and so on), character strings labeling the meaning of the horizontal and vertical axes, and a character string above the plot providing a caption for the plot, and possibly a variety of other features. Typically, one constructs a plotting routine in such a way that features such as points, lines, text, arrows, and so on can be added to the plot later.

In this section we consider the basic numerical tools one needs to construct such plots.

From Real Coordinates to Pixels

In most graphics, the user thinks of the screen or the printer in real coordinates, that is, as a portion of the two dimensional Cartesian plane. Thus a graphics routine must eventually convert real coordinates to pixel coordinates. In most cases this consists of finding the pixel (c, r) that corresponds to a real coordinate (x, y) for a portion of the screen whose lower left hand corner has real coordinate (x_0, y_0) and pixel coordinate (c_0, r_0) and upper left hand corner having real coordinate (x_1, y_1) and pixel coordinate (c_1, r_1).

We consider finding c from c_0, c_1, x_0, and x_1 and note that finding r is done in the same way. We divide the interval $[x_0, x_1]$ into $m = c_1 - c_0 + 1$ intervals (each interval corresponding to a pixel) of length $h = (x_1 - x_0)/m$, where the ith interval is given by

$$[x_0 + (i-1)h, x_0 + ih), \qquad i = 1, \ldots,$$

and transform x to pixel $c = c_0 + (i-1)$ if x falls in the ith interval, that is, if $x \in [x_0 + (i-1)h, x_0 + ih)$, or equivalently, if $[(x - x_0)/h] = i - 1$.

If $x \geq x_1$, then x will be converted to a pixel greater than c_1, while if $x < x_0$ it will be converted to a pixel less than c_0. Thus one typical conversion method is given by

$$c = c_0 + \max(0,\ \min(\ \mathrm{int}((x - x_0)/h),\ m-1)).$$

Finding Nice Numbers for Axis Tic Mark Labels

In most $X - Y$ plots, one desires tic mark labels that are "nice" (S-Plus calls them "pretty"), that is, labels that are not something like 1.459 followed by 1.645, and so on. Suppose we want to construct a scatterplot of n pairs $(x_1, y_1), \ldots, (x_n, y_n)$ of numbers and we'd like to do this by calling a routine such as `scatter(x,y)`, that is, without having to specify the number of tic marks and the values at the tic marks. The simplest way to do this would be to have the ends of the axes be the minimum and maximum values of the x's and y's and then have the tic mark labels be the equally spaced values between the minima and maxima. This of course could easily lead to "ugly" labels.

Given values x_0 and x_1 of the smallest and largest elements of an array and a desired number N of intervals (and thus $N + 1$ is the desired number of tic marks), we describe an algorithm for finding a new minimum X_0 and maximum X_1 so that the range $[X_0, X_1]$ is the smallest range which contains the interval $[x_0, x_1]$ and simultaneously result in approximately N intervals each of length h where h is the product of 1, 2, or 5 and an integer power of 10 and X_0 and X_1 are integer multiples of h.

The Fortran program given below contains the method and produces the following output:

```
      xmin      xmax     n     xmin1     xmax1         h    n1
  -------------------------------------------------------------
      0.53      0.79     7      0.50      0.80      0.05     6
```

10.00	47.00	11	10.00	50.00	5.00	8
-0.60	-0.40	5	-0.60	-0.40	0.05	4
119.00	9875.00	10	0.00	10000.00	1000.00	10
2.00	19.00	6	2.00	20.00	2.00	9

The Nice Numbers Program

```
      dimension xmin(5),xmax(5),n(5)

      data xmin/ .53, 10., -.6,  119.,  2./
      data xmax/ .79, 47., -.4, 9875., 19./
      data n/      7, 11,    5,   10,   6/

      write(*,10)
 10   format(1x,5x,4hxmin,5x,4hxmax,4x,1hn,4x,5hxmin1,
     1       4x,5hxmax1,8x,1hh,3x,2hn1/,1x,55(1h-))

      do 20 i=1,5
      call nice(xmin(i),xmax(i),n(i),xmin1,xmax1,n1,h)
 20   write(*,30) xmin(i),xmax(i),n(i),xmin1,xmax1,h,n1
 30   format(1x,2f9.2,i5,3f9.2,i5)

      stop
      end
      subroutine nice(xmin,xmax,n,xmin1,xmax1,n1,dist)
c-------------------------------------------------------------
c
c   Given the minimum (xmin) and maximum (xmax) of an array,
c   and a desired number of intervals (n), this subroutine
c   finds a number of intervals (n1) and a new minimum (xmin1)
c   and a new maximum (xmax1) such that [xmin1,xmax1] contains
c   [xmin,xmax] and the length of the intervals (dist) is the
c   product of 1, 2, or 5 with an integer power of 10 and xmin1
c   and xmax1 are multiples of dist.
c
c-------------------------------------------------------------

      dimension vint(4),sqr(3)

      data vint/1.,2.,5.,10./
      data sqr/1.414214,3.162278,7.071068/

      del=.00002

c   Find interval width dist:

      a=(xmax-xmin)/n
      al=log10(a)
      nal=al
      if(a.lt.1.) nal=nal-1
      b=a/(10.**nal)

      do 10 i=1,3
 10   if(b.lt.sqr(i)) go to 30
      i=4

 30   continue

      dist=vint(i)*10.**nal

c   Get xmin1 = largest multiple of dist .le. xmin:

      fm1=xmin/dist
      m1=fm1
      if(fm1.lt.0.0) m1=m1-1
      if(abs(m1+1.0-fm1).lt.del) m1=m1+1
      xmin1=dist*m1

c   Get xmax1 = smallest multiple of dist .ge. xmax:

      fm1=xmax/dist
      m1=fm1
      m1=m1+1
      if(fm1.lt.-1.0) m1=m1-1
      if(abs(fm1+1.0-m1).lt.del) m1=m1-1
      xmax1=dist*m1
```

```
      if(xmin1.gt.xmin) xmin1=xmin
      if(xmax1.lt.xmax) xmax1=xmax

c   Get new number of intervals:

      n1=((xmax1-xmin1)/dist)+0.05

      return
      end
```

Contour Plots

One popular plot is the contour plot of a function $f(x, y)$ of two variables x and y. Typically, we are given values of the function at a grid of (x, y) points, where the grid is made up of all combinations of $x_1 < \cdots < x_{n_x}$ and $y_1 < \cdots, y_{n_y}$. Usually, the values of f are stored in an $(n_x \times n_y)$ matrix Z where $Z_{ij} = f(x_i, y_j)$.

A contour plot is made up of several contours, where a contour having level c traces a curve where the horizontal plane having height c intersects the three dimensional solid object determined by the function f, that is, the set of (x, y) such that $f(x, y) = c$. Since we only have the values of f at a grid of points, we have no way of knowing the behavior of f within a rectangle in the grid.

To see how a contour plot is drawn, consider a contour having height c and the (i, j)th box in the grid of (x, y) values and let

$$v_1 = (x_i, y_j), \quad v_2 = (x_i, y_{j+1}), \quad v_3 = (x_{i+1}, y_{j+1}), \quad v_4 = (x_{i+1}, y_j),$$

be the vertices of the box starting at the lower left corner and proceding clockwise around the box (see below). Let f_1, f_2, f_3, and f_4 be the corresponding values of f at the vertices. If c is greater than all four f's, then we will assume that the contour plane is entirely above the function in this box (and any contour plane above this one is also entirely above the function). If c is less than all four f's, then the contour plane is assumed below the function in this box (and so is any contour plane below this one).

```
v2 .------------. v3
   |            |
   |            |
   |            |
   |            |
v1 .------------. v4
```

If some of the four f's are above c and some are below, then we can assume that the contour plane 'cuts' the function inside the box. For example if f_1 and f_3 are below c while f_2 is above, then we will draw a line in this box representing the intersection of the contour plane and the plane drawn through the points f_1, f_2, and f_3. This will result in a line from the left side to the top of the box where the endpoints of the line are determined by interpolation. Thus if c is close to f_2, then the left endpoint of the line will be close to v_2. For the (i, j)th box, the x value of the endpoint on the top of the box is given by

$$x = x_i + \frac{f_2 - c}{f_2 - f_3}(x_{i+1} - x_i).$$

There are actually 16 combinations of $+$'s and $-$'s of the values of f at the four vertices. Each of the combinations leads to no lines, one line, or two lines being drawn. In the table below, we list each of the results.

```
      subroutine cntour(x,y,z,nx,ny,ndim,npx,npy,nx1,ny1,
     1                  nctrs,ctrs,ix,iy)
```

Table of Configurations in Contour Plotting

```
Line(s)           Configuration(s)  Line(s)        Configuration(s)
-----------------------------------------------------------------
None:             +   +      -   -

                  +   +      -   -

Left -> Bottom    -   -      +   +   Left -> Top   +   -     -   +

                  +   -      -   +                 -   -     +   +

Left -> Right     +   +      -   -   Top -> Bottom +   -     -   +

                  -   -      +   +                 +   -     -   +

Right ->Bottom    -   -      +   +   Right -> Top  -   +     +   -

                  -   +      +   -                 -   -     +   +

Left  -> Top      -   +              Left-> Bottom +   -
   and                                  and
Right ->Bottom    +   -              Right -> Top  -   +
-----------------------------------------------------------------
```

```
c----------------------------------------------------------------
c
c   Subroutine to draw a contour plot on a raster device.
c
c   Input:
c       nx, ny  : Number of grid points in x and y directions
c       x, y    : Vectors containing x and y values of grid pts.
c                 Each vector must be in increasing order.
c       z       : Matrix whose (i,j)th element is value of the
c                 function at grid point (x(i),y(j)).
c       ndim    : Row dimension of z in calling program.
c       npx, npy: Size in pixels of plotting region.
c       nx1, ny1: Pixel column and row numbers of lower left hand
c                 corner of plotting region. Note that the lower
c                 left corner of the screen is column 0, row 0.
c       nctrs   : Number of contours.
c       ctrs    : Vector containing contour values in
c                 increasing order.
c
c   Output:
c       ix, iy  : Vectors defined by: ix(i) = pixel col of x(i).
c                                     iy(i) = pixel row of y(i).
c
c   Notes:
c       1) The outline of plotting region is drawn, but no grid.
c       2) The contours are drawn in the same color. To change
c          this, one need only modify the line subroutine.
c       3) The user is responsible for setting the graphics mode
c          before calling cntour and for making sure that the
c          values of npx, npy, nx1, ny1 are selected so that
c          the plotting region fits on the screen.
c       4) The only graphics routine called by cntour is line.
c
c----------------------------------------------------------------

        dimension x(nx),y(ny),z(ndim,1),ctrs(nctrs),ix(nx),iy(ny)
c
c   Draw lines outlining plotting region:
c
        call line(nx1,ny1,nx1,ny1+npy-1,1)
        call line(nx1,ny1+npy-1,nx1+npx-1,ny1+npy-1,1)
        call line(nx1+npx-1,ny1+npy-1,nx1+npx-1,ny1,1)
        call line(nx1+npx-1,ny1,nx1,ny1,1)
```

```
c
      rx=x(nx)-x(1)
      ry=y(ny)-y(1)
      xmin=x(1)
      ymin=y(1)
c
c  Find pixel coordinates of grid points:
c
      do 5 i=1,nx
    5 ix(i)=ifpix(x(i),npx,nx1,xmin,rx)
      do 6 i=1,ny
    6 iy(i)=ifpix(y(i),npy,ny1,ymin,ry)
c
c  Loop over i = column of boxes, j = row of boxes, k = which
c  contour
c
      do 20 i=1,nx-1
c
      xleft=x(i)
      xright=x(i+1)
      xdiff=xright-xleft
      ixl=ix(i)
      ixr=ix(i+1)
c
      do 20 j=1,ny-1
c
      ylow=y(j)
      yup=y(j+1)
      ydiff=yup-ylow
      iyl=iy(j)
      iyu=iy(j+1)
c
c     z2 .-----------. z3
c         |          |
c         |          |
c         |          |
c         |          |
c     z1 .-----------. z4
c
      z1=z(i,j)
      z2=z(i,j+1)
      z3=z(i+1,j+1)
      z4=z(i+1,j)
c
c
      do 10 k=1,nctrs
      cc=ctrs(k)
c
c  nzg = # of z's greater or equal to current contour
c
      nzg=0
      if(z1.ge.cc) nzg=nzg+1
      if(z2.ge.cc) nzg=nzg+1
      if(z3.ge.cc) nzg=nzg+1
      if(z4.ge.cc) nzg=nzg+1
c
c  If nzg = 0, then all later contours are above too so jump out
c  of contour loop:
c
      if(nzg.eq.0) go to 15
      if(nzg.eq.4) go to 10
c
c  ns = sum of vertex numbers of z's that are greater or
c  equal to cc
c
      ns=0
      if(z1.ge.cc) ns=ns+1
      if(z2.ge.cc) ns=ns+2
      if(z3.ge.cc) ns=ns+3
      if(z4.ge.cc) ns=ns+4
c
c  nzg = 3 and ns = j is the same as nzg = 1 and ns = 10 - j
c
      if(nzg.eq.3) then
              ns=10-ns
              nzg=1
      endif
c
      if(nzg.eq.1) then
c
```

```
                go to(11,12,13,14) ns
c
c   left to bottom:
c
  11            ixx=intpix(xleft,xdiff,cc,z1,z4,npx,nx1,xmin,rx)
                iyy=intpix(ylow,ydiff,cc,z1,z2,npy,ny1,ymin,ry)
                call line(ixl,iyy,ixx,iyl,1)
                go to 10
c
c   left to top:
c
  12            ixx=intpix(xleft,xdiff,cc,z2,z3,npx,nx1,xmin,rx)
                iyy=intpix(ylow,ydiff,cc,z1,z2,npy,ny1,ymin,ry)
                call line(ixl,iyy,ixx,iyu,1)
                go to 10
c
c   top to right:
c
  13            ixx=intpix(xleft,xdiff,cc,z2,z3,npx,nx1,xmin,rx)
                iyy=intpix(ylow,ydiff,cc,z4,z3,npy,ny1,ymin,ry)
                call line(ixx,iyu,ixr,iyy,1)
                go to 10
c
c   right to bottom:
c
  14            ixx=intpix(xleft,xdiff,cc,z1,z4,npx,nx1,xmin,rx)
                iyy=intpix(ylow,ydiff,cc,z4,z3,npy,ny1,ymin,ry)
                call line(ixx,iyl,ixr,iyy,1)
                go to 10
c
        endif
c
        if(nzg.eq.2) then
c
c   top to bottom:
c
                if(ns.eq.3.or.ns.eq.7) then
                ixx1=intpix(xleft,xdiff,cc,z2,z3,npx,nx1,xmin,rx)
                ixx2=intpix(xleft,xdiff,cc,z1,z4,npx,nx1,xmin,rx)
                call line(ixx1,iyu,ixx2,iyl,1)
                go to 10
                endif
c
c   left to right:
c
                if(ns.eq.5) then
                iyy1=intpix(ylow,ydiff,cc,z1,z2,npy,ny1,ymin,ry)
                iyy2=intpix(ylow,ydiff,cc,z4,z3,npy,ny1,ymin,ry)
                call line(ixl,iyy1,ixr,iyy2,1)
                go to 10
                endif
c
c   two lines with negative slopes:
c
                if(ns.eq.4) then
                ixx=intpix(xleft,xdiff,cc,z1,z4,npx,nx1,xmin,rx)
                iyy=intpix(ylow,ydiff,cc,z1,z2,npy,ny1,ymin,ry)
                call line(ixl,iyy,ixx,iyl,1)
                ixx=intpix(xleft,xdiff,cc,z2,z3,npx,nx1,xmin,rx)
                iyy=intpix(ylow,ydiff,cc,z4,z3,npy,ny1,ymin,ry)
                call line(ixx,iyu,ixr,iyy,1)
                go to 10
                endif
c
c   two lines with positive slopes:
c
                if(ns.eq.6) then
                ixx=intpix(xleft,xdiff,cc,z2,z3,npx,nx1,xmin,rx)
                iyy=intpix(ylow,ydiff,cc,z1,z2,npy,ny1,ymin,ry)
                call line(ixl,iyy,ixx,iyu,1)
                ixx=intpix(xleft,xdiff,cc,z1,z4,npx,nx1,xmin,rx)
                iyy=intpix(ylow,ydiff,cc,z4,z3,npy,ny1,ymin,ry)
                call line(ixx,iyl,ixr,iyy,1)
                go to 10
                endif
        endif

c
  10    continue
  15    continue
```

```
  20    continue
c
c
        return
        end
        integer function intpix(x1,xd,c,z1,z2,npx,nx1,xmin,rx)
c---------------------------------------------------------------
c
c   Interpolate and find pixel. x1 is lower x value, xd is
c   difference in x values.
c
c---------------------------------------------------------------

        xx=x1+xd*(z1-c)/(z1-z2)
        intpix=ifpix(xx,npx,nx1,xmin,rx)
        return
        end
        integer function ifpix(x,npx,nx1,xmin,rx)
c---------------------------------------------------------------
c
c   Convert real world coordinate x to pixel coordinate ifpix.
c
c---------------------------------------------------------------

        ifpix=nx1+min(npx-1,max(0,nint(npx*(x-xmin)/rx)))
c
        return
        end
```

CHAPTER 6

Introduction to Unix

The Unix operating system is an incredibly powerful and complex system that is ideal for running a distributed system such as ours, particularly since we use computers primarily for program development. A wide variety of tools are available for doing the kinds of work we do and the ability to have a large number of users communicate with each other is built in.

File Systems

Our system has one very large and complicated hierarchical file system, that is, the files are organized into an inverted tree structure with a root directory at the top of the structure and then a set of subdirectories from the root directory and these subdirectories themselves have subdirectories, and so on. At the root of the system are "system files," that is, files needed to operate the system. Down a couple of levels from the root directory is a directory containing subdirectories `grad`, `stat`, and `staff`. These directories have subdirectories for each graduate student, each faculty member, and each staff member, respectively. Thus each user in the system has their own subdirectory (called their "home directory.") It is possible to organize a set of users into a "group." Most users try to organize their files into a set of subdirectories within their home diectory so they can be easily found.

Permissions

Each file in the system has an owner (typically the user who created the file) and a set of permissions attached to it, that is information on who can read, modify, or execute that file (if it is executable). Entering the command `ls -l` will show a list of files and the attached permissions. This list will look something like the following

```
drwx------  2 jnewton       512 Sep  5 11:33 exams
-rw-rw-r--  1 jnewton       305 Aug 26 09:01 hald.dat
-rw-rw-r--  1 jnewton     32959 Sep  5 11:19 introsys.tex
-rwxrwxr-x  1 jnewton    204800 Aug 24 10:18 regswp
-rw-rw-r--  1 jnewton      8271 Aug 24 10:18 regswp.f
```

Each line has the permissions followed by the owner, the size of the file, the date and time it was last modified, and finally the name of the file. There are 10 fields in the permissions. The first is a `d` if the entry is a directory. The remaining nine entries are in three groups of three each with the three groups being owner, group, and world. Thus `jnewton` can read, write, and execute the directory `exams` and no one else can, everyone can read `hald.dat`, `introsys.tex`, and `regswp.f`, while `regswp` is an executable file that anyone can execute. Note that only `jnewton` and his group can modify all these files (they have write permission as seen by the w in the permissions).

To change the permissions on a file (or a directory) belonging to you, issue the command `chmod xyz` `filename` where `x`, `y`, and `z` are each numbers between 0 and 7 summarizing the permissions for owner, group, and world. They are the sum of the read (4 points for yes, 0 for no), write (2 points for yes, 0 for no), and execute (1 point for yes, 0 for no). Thus to totally protect a directory, you can issue the command `chmod 700`.

Man Pages

Briefly documenting (or remembering) all the commands available in Unix (and all their options) is impossible. Thus Unix supplies the command `man` which will display the manual pages for each command on the system. For example, `man ls` will show you all you ever wanted to know about the `ls` command (and probably more than you want to know.)

Commonly Used Commands

Command	Purpose
`cd`	Change directory
`mkdir`	Create a new directory
`ls`	List the files and directories in the current directory (short version)
`ls -l`	Long version
`cat`	List the contents of a file or concatenate files
`more`	List the contents of a file one screenful at a time
`grep`	Display all lines in a file containing a certain character string
`awk`	Perform operations on specified lines in a file
`wc`	Show the number of lines, words, and bytes in a file
`cut`	Select from a file certain fields or sets of columns
`paste`	Horizontally concatenate files
`lpr`	Print a file
`lpq`	See what's printing on a specified printer
`sort`	Sort a file
`uniq`	Find unique lines in a file
`ps`	See what jobs you are currently executing
`kill`	Kill a job that is executing

Redirection and Pipes

Redirection is the idea of using a file as input to a program that expects something from the keyboard or sending output to a file instead of the screen. For example `sort fname > sfname` will sort the file called `fname` and redirect the output to the file named `sfname` instead of to the screen. To concatenate the output to the end of a file, used two greater than signs, for example, `sort fname >> sfname` will append the result of sorting `fname` to the end of `sfname`.

One can connect a series of commands on a single command line using pipes, for example

```
grep alias .mailrc | sort | uniq > new.mailrc
```

will pull out all lines in the `.mailrc` file having the string `alias`, then sort them, then eliminate any duplicates, and then send the result to a file called `new.mailrc`.

Some Examples

I have a set of files called `fal97.ros`, `spr88.ros`, `sum88.ros`, and so on, up to `sum94.ros`, where the lines in a file consist of information about each student who took a statistics course during that semester, including: 1) their ID in columns 1–9, 2) their name in columns 10–41, 3) their classification in 42–43, 4) their major in 44–47, 5) the course number in 48–50, and 6) the section number in 51–53. Thus the following line (which I've displayed on two lines so it will fit)

```
awk '{print substr($0,48,3) substr($0,44,4)}' spr94.ros | sort |
   uniq -c > bymaj
```

will form a file containing a list of all majors (and how many in each major) taking each course from the statistics department in the Spring of 1994.

Aliases and Shell Scripts

If you create a file called `.alias` in your home directory with lines like

```
alias 604 'cd  /tex/classes/60494'
```

and then say `source  /.alias`, you can use the aliases in the `.alias` file.

You can also create a file called a shell script with one or more unix commands that can be executed as though it were a unix command itself. For example, if I create the file called `texprt` containing

```
tex $1
dvips -Php1 $1
```

and then say `chmod +x texprt`, then I can TeX and print (on `hp1`, see below) a file by saying `texprt filename`, where `filename` is the name of the TeX file.

Printers

The network has two public printers; one in room 421 (named `hp1`) and one in the main office (named `hp2`). To print a file on one of these printers, enter the command `lpr -Pprintername filename` where `printername` is either `hp1` or `hp2`. Graduate students should never print anything longer than a few pages unless they have permission from Drs. Newton or Schmiediche. In particular, do not print documentation or books!

When printing computer code (or any ASCII file), please use the Unix program `a2ps`. This program prints the code sideways on the paper in two columns. A total of 132 lines can be printed on one page and still be very readable. Use the command `a2ps -Pprintername filename` where `printername` is the name of the printer you want the file sent to.

Using the System via a Modem

See Dr. Schmiediche.

The emacs Text Editor

Almost everything one does on the system requires the use of a text editor, including writing C or Fortran programs, using TeX and Splus, and just doing ordinary operating system activities. Almost all Unix systems have a text editor called `vi` (see `man vi`) but the editor of choice among most users is `emacs`, a very sophisticated editor with many features.

Starting emacs

Typically, when one logs into the system from a workstation or X-terminal, one should open an emacs window by entering the command `emacs &` (the ampersand tells Unix to run emacs in the background, that is, allow you to do other things while the emacs window is there.)

The Control and Meta Keys

While in emacs, if you type a series of characters they will appear on the screen. Thus one needs a way to tell emacs that what is to be typed next is not text but a command to emacs. There are two keys that signal to emacs that what comes next is a command. These are called the "control key" (the key with the word control on it) and the "meta key" (which key this is on a keyboard varies from keyboard to keyboard; on our X-terminals it is the key marked "Alt" on the keyboard.)

These keys are used typically by holding it down and then typing some other key. For example, while in emacs, one tells emacs to begin editing a file by typing `Ctrl-x-f`, that is, holding down the control key and striking the x and f keys. Then emacs will ask you for the name of the file to edit (the question will appear in the blank line at the bottom of the screen called the "minibuffer") and you answer with the name of the file.

Buffers

One can edit several files simultaneously. Each file is contained in a "buffer," and one buffer is active at all times.

Editing Commands

Like any other full-screen text editor, one needs to know only a few things in order to edit a file. These things include:

	Keystroke	Meaning
Moving the cursor		
	`arrow keys`	Up or down one line or left or right one character
	`meta-v or Ctrl-v`	Up or down one screen
	`meta-< or meta->`	Beginning or end of file
	`meta-x-goto-line`	Go to a specified line number (emacs prompts for number)
Deleting things (see block operations for deleting a block)		
	`Ctrl-d`	Delete the current character
	`Ctrl-k`	Delete to the end of the current line
Searching and replacing		
	`Ctrl-s`	Search for character string (emacs prompts for string)
	`meta-%`	Global search and replace emacs will prompt for old and new strings and prompt yes or no for each ocurrence; answer with ! to replace all remaining ocurrences
Block operations		
	`Ctrl-@, Ctrl-w`	Mark a set of lines (`Ctrl-@` at start and `Ctrl-w` at end) and delete that set
	`Ctrl-@, meta-w`	Mark a set of lines but don't delete that set
	`Ctrl-y`	Copy a marked set of lines to current cursor position
Miscellaneous		
	`Ctrl-x-s`	Save the file in the current buffer

`Ctrl-x i`	Insert a file where the cursor is (emacs will prompt for file name)

Introduction to email

The Unix `mail` program can be used to either read mail or send mail.

Sending Mail

To send a message to someone, you type `mail address` where `address` is the address (or list of addresses) of the person (or persons) to receive the mail. The `mail` program will then ask you for a subject, which you enter. Then `mail` will ask you if you want carbon copies sent to anyone (by giving you a `CC:` prompt). If you want to, you can enter an address (or a list of addresses) of the person (or persons) to receive the carbon copy. If you don't want carbon copies, just hit enter in response to the `CC:` prompt. Then you type your message, hitting the enter key at the end of each line. When you have finished the message, you type `Ctrl-d` when the cursor is at the beginning of a blank line.

To email a file, enter the command `mail address < filename`.

Reading Your Mail

When mail for you comes to our system, it goes into a file called

```
/var/spool/mail/acct
```

where `acct` is the name of your account. This file is called your "mail spooler." If you enter the command `mail` from the Unix prompt, Unix looks to see if there is anything in your mail spooler. If not, it gives you the message, `no mail for acct` where again `acct` is the name of your account. If your spooler is not empty, `mail` reads it, puts a summary of its contents on your screen, and then issues a prompt (`mail&`). The summary will look something like

```
U  1 rondoes@fwi.uva.nl Fri Jul 22 01:28   42/1403  Test
U  2 ruth               Wed Aug  3 10:25   15/521   PPRI
U  3 gerig@stat.ncsu.edu Wed Aug  3 12:35   20/824  Sign On
U  4 newsl              Wed Aug  3 12:51   19/498   Newsletter
```

that is, the line for each message contains the number of the message as well as the address of the sender an the date and time it was received and finally the subject of the message.

If you enter the number of a message, its contents will be displayed on the screen. You can delete a message from the spooler (by `d number`), or save a message to a file (by `s number filename`), or get a list of all the things you can do (by `?`).

Leaving the Mail Program

When you want to return to the Unix prompt, you have two choices of how to leave `mail`. The first is to type `x` in which case your spoller will be preserved exactly how it was when you entered the `mail` program. If you enter `q`, then any messages you read or deleted will be removed from the spooler. If you have read any messages and exit `mail` with a `q`, then the contents of the read messages will be appended to a file called `mbox` that is in your home directory (this is called your "mbox.") To look at the contents of your mbox, enter the command `mail -f` from the Unix prompt.

Mail Aliases

In your home directory is a file called `.mailrc`. In this file you can create mail aliases such as

```
alias scott         scott@stat.rice.edu
alias wegman        ewegman@endor.galaxy.gmu.edu
alias kettenring    jon@bellcore.com
alias board         scott wegman kettenring
```

There are also a series of "system mail aliases," that are available for all users to use, including `all`, `grads`, `faculty`, `staff`, and so on.

Folders

If you put the line `set folder=.Mail` in your `.mailrc` file and create the directory `.Mail` in your home directory, then you can use what are called "mail folders." For example, you might like to save all email messages for a particular class in a folder having the name of the class. You do this by `s number +folder` from the mail prompt, where `number` is the number of the message and `folder` is the name of the folder. To look at the messages in a particular folder, enter the command `mail -f +folder` from the Unix prompt, where again, `folder` is the name of the folder.

CHAPTER 7

Introduction to Fortran

The Fortran Programming Language

The Fortran programming language was one of the first (if not the first) "high level" languages developed for computers. It is referred to as a high level language to contrast it with machine language or assembly language which communicate directly with the computer's processor with very primitive instructions. Since all that a computer can really understand are these primitive machine language instructions, a Fortran program must be translated into machine language by a special program called a Fortran compiler before it can be executed. Since the processors in various computers are not all the same, their machine languages are not all the same. For a variety of reasons, not all Fortran compilers are the same. For example, more recent Fortran compilers allow operations not allowed by earlier versions. In this chapter, we will only describe features that one can expect to have available with whatever compiler one may have available.

Fortran was initially developed almost exclusively for performing numeric computations (Fortran is an acronym for "Formula Translation"), and a host of other languages (Pascal, Ada, Cobol, C, etc.) have been developed that are more suited to nonnumerical operations such as searching databases for information. Fortran has managed to adapt itself to the changing nature of computing and has survived, despite repeated predictions of its death. It is still the major language of science and is heavily used in statistical computing.

The most standard version of Fortran is referred to as Fortran 77 since it is based on a standard established in 1977. A new standard was developed in 1990 that incorporates some of the useful ideas from other languages but we will restrict ourselves to Fortran 77.

The Basic Elements of Fortran

A Fortran program consists of a series of lines of code. These lines are executed in the order that they appear unless there are statements that cause execution to jump from one place in the series of lines to another. At the end of this chapter, we have listed the code of a complete program and we will refer to this program as we go along. Learning to write Fortran programs is usually done by imitating programs written by others. This chapter cannot possibly provide all of the details of Fortran, but rather to point out those features that are particularly important in statistical computing.

1. Statements begin in column 7 of a line and end before column 72 (if a statement will not fit in columns 7–72, it can be continued onto the next line (or lines) as long as a character is placed in column 6 of each continuation line). Columns 1–5 of a line can contain "a statement number" so that the statement on that line can be referenced by other statements. Lines that begin with the letter c in column one are interpreted as comments and are not executed. It is important to include enough comments in any program in any language so that you (or somebody else) can later understand what the lines of code are intended to do.

2. A file containing a Fortran program begins with a main program which may be followed by subprograms.

Subprograms come in two kinds; functions and subroutines. The main program and the subprograms it calls can be in different files which can be compiled separately and then linked together to form an executable file.

4. A main program begins with a series of statements declaring the names of the variables that are going to be used by the program and what type of variables they are (see "data types" below). Note that variable names must begin with letters of the alphabet, contain letters and the digits 0 through 9, and (to be safe as different compilers allow different numbers of characters) contain 6 characters or fewer. The program ends with the lines `stop` and `end`.

5. Between the declarations and the `stop` and `end` there are a wide variety of operations that can be performed, but usually a program consists of 1) reading some information from somewhere (for example, from a file or from the keyboard from the person using the program 2) performing some numerical or graphical task, and 3) placing the results of the task somewhere so that the user of the program can use them (perhaps on the user's screen or into a file).

6. Often there is some subtask that needs to be performed several times in a program. For example, it may be necessary to form the plot of one vector versus another. Rather than have the same set of lines of code in several different places in the main program, one can form them into what is called a subprogram. A subprogram begins and ends with special lines of code that tell the compiler that it is in fact a subprogram and not part of the main program. Another use of subprograms is in modular programming where one breaks a complicated task into a series of tasks that are more managable and writing a subprogram for each one.

Data Types

Fortran performs numerical and logical operations on variables of a few types, including:

1. **Integers**, that is, numbers that have no decimal part. There are two kinds of integers; long integers (also called four-byte integers or `integer*4`'s, they range from roughly -2^{31} to 2^{31} and use four bytes of memory) and short integers (also called two-byte integers or `integer*2`'s, they range from roughly 2^{-15} to 2^{15} and use two bytes of memory). One declares variables to be short or long integers by statements of the form
`integer*2 n,m,k`

```
integer*2 n,m,k
```

or

```
integer*4 n,m,k
```

respectively.

2. **Floating point numbers**, that is, numbers that can contain decimal parts. There are two kinds of floating point numbers; double precision (also called `real*8` reals, they range from roughly 10^{-300} to 10^{300}, take eight bytes of memory, and are accurate to roughly 13 or 14 decimal places) and single precision (also called `real*4` reals, they range from roughly 10^{-38} to 10^{38}, take four bytes of memory, and are accurate to roughly six or seven decimal places). Single and double precision variables are declared by lines of the form

```
real*4 x,y
```

or

```
real*8 x,y
```

respectively.

3. **Complex numbers**, that is variables that can take on complex values. Actually, complex variables can be thought of as a pair of `real*4` or `real*8` floating point numbers. Single and double precision complex variables are declared by lines of the form

```
complex*8 x,y
```

or

```
complex*16 x,y
```

respectively.

4. **Logical variables**, that is, variables that can only take on the "values" `.TRUE.` or `.FALSE.`.

5. **Character variables**, that is, variables that contain characters such as letters of the alphabet or numerical digits. A character variable can be as long as desired. To declare variables called `st1` and `st2` to be of length 10 and 20 respectively, one uses the statement

```
character st1*10,st2*20
```

6. **Arrays of variables**. In addition to scalar variables, one can declare a variable to be a vector (one dimensional array) or matrix (two dimensional array), or a higher dimensional array. To declare a single precision 100 by 3 matrix called `x`, one would have the statement

```
real*4 x(100,3)
```

while the statement

```
character st1(100)*10,st2(200)*20
```

would declare character arrays of length 100 and 200 called `st1` and `st2` where each of the 100 elements of `st1` can contain 10 characters, while each of the 200 elements of `st2` can contain as many as 20 characters. Note that Fortran can only perform operations on individual elements of arrays, that is, it has no built-in ability to perform operations (such as addition) on all the elements of arrays. Most computer systems have libraries of subprograms that perform operations on entire arrays.

Undeclared Names

It is very good programming pactice to declare every variable that a program uses (most compilers have a way to warn a programmer of any variables that have not been declared). However, one should be aware of the fact that any variable whose name begins with one of the letters i, j, k, l, m, or n, is considered to be an integer (usually an `integer*4`) unless otherwise declared explicitly, while any variable whose name begins with one of the other letters of the alphabet is taken to be a real (usually `real*4`). Often it is useful to follow these conventions in the naming of variables (although they should also be explicitly declared) so that someone reading the code will know what type the variables are without having to look at the declarations.

Assignments and Arithmetic Operations

An important part of a Fortran program is assigning values to variables and performing arithmetic operations on variables. There are five arithmetic operations, namely addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and exponentiation (`**`). If more than one operation is contained in a statement, then exponentiation is done first, then multiplication and division (if there are more than one of these they are done from left to right), and finally addition and subtraction (again if there are more than one of these, they are done from left to right). Expressions within parentheses are done first.

When all operands of an arithmetic expression are of the same type, then the final type of the value of the expression will be of that type. When they are of different types, the value of the expression will be that of the last binary operation performed where the value of a binary operation of data of different types takes on the type of the higher of the two types, where types are ranked from lowest to highest by `integer*2`, `integer*4`, `real*4`, `real*8`, `complex*8`, and `complex*16`. One must be careful of expressions such as `x*(i/j)` or `x**(i/j)` when `i` and `j` are integers since the result of the division is truncated and not rounded.

Once an expression has been evaluated, it is converted to the type of the variable on the left hand side of the expression.

Intrinsic Functions

Fortran has a wide variety of functions built into it. These functions are called intrinsic functions. Examples include trigonometric and logarithm functions. The table below contains a list of the functions that a Fortran programmer can expect to have available when writing code. Note that these functions need not be declared before use. Also, starting with Fortran 77 the concept of generic functions was developed. For example, in previous versions of Fortran there were four functions for finding the absolute value of a number depending on its type. Now there is only the single function `abs` which will return a value having the same type as its input type. The old functions `iabs`, `abs`, `dabs`, and `cabs` are still available as well. The prefixes `i, a, d, c` are used in these functions for integer, real, double precision, and comples, respectively.

Intrinsic Functions[1]

`functions`	Definition
Numerical functions:	
`sdc=COS(sdc), sdc=SIN(sdc), sd=TAN(sd)`	Trigonometric functions
`sd=ACOS(sd), sd=ASIN(sd), sd=ATAN(sd)`	Inverse trig functions[2]
`sd=ATAN2(sd1,sd2)`	
`sd=COSH(sd), sd=SINH(sd), sd=TANH(sd)`	Hyperbolic trigonometric functions
`sdc=LOG(sdc), sd=LOG10(sd)`	Log functions
`sdc=SQRT(sdc)`	Square root
`isds=ABS(isdc)`	Absolute value
`sdc=EXP(sdc)`	Exponentiation
`isd=MOD(isd1,isd2)`	Remainder[3]
`isd=SIGN(isd1,isd2)`	Sign function[4]
`isd=MAX(isd1,isd2,...), isd=MIN(isd1,isd2,...)`	Maximum and minimum
`s=AMAX0(i1,i2,...), s=AMIN0(i1,i2,...)`	Integer input, single output
`i=MAX1(s1,s2,...), i=MIN1(s1,s2,...)`	Single input, integer output
`isd=DIM(isd1,isd2)`	Positive difference: `MAX(isd1-isd2,0)`
`d=DPROD(s1,s2)`	Double precision product of arguments
Type conversion:	
`i=INT(isdc), s=REAL(isdc), d=DBLE(isdc)`	
`c=CMPLX(isd)`	Input is real part, imaginary part is 0
`c=CMPLX(c)`	Identity

`c=CMPLX(isd1,isd2)`	First argument is real part, 2nd is imaginary part
Complex numbers:	
`s=REAL(c),   s=AIMAG(c)`	Real and Imaginary parts
`c=CONJG(c),  s=CABS(c)`	Complex conjugate and modulus
Rounding and truncating5:	
`sd=AINT(sd), sd=ANINT(sd)`	Truncation and rounding
`i=NINT(sd)`	Rounding to nearest integer
Operations with characters:	
`C=CHAR(i)`	Character having input ASCII code
`i=ICHAR(C)`	ASCII code of input character
`i=INDEX(str1,str2)`	Compare strings[6]
`i=LEN(str)`	Number of characters in the string
`L=LGE(str1,str2), L=LGT(str1,str2),` `L=LLE(str1,str2), L=LLT(str1,str2)`	Lexicographical ordering[7]

[1]The table lists the input and output type of the function, for example, the notation `isds=ABS(isdc)` means that for integer, single, double, and complex input, the output is integer, single, double, and single respectively (note that `CABS(c)` is single). For characters, `C` represents a single character while `str` represents a string of characters. Logical variables are denotd by `L`.

[2]The function `ATAN2` has two arguments x_1 and x_2 and is defined to be ?????

[3]The remaindering function applied to arguments x_1 and x_2 is defined to be $x_1 - \mathtt{INT}(x_1/x_2) * x_2$.

[4]The sign function having two arguments x_1 and x_2 is $|x_1|$ if $x_2 \geq 0$ and $-|x_1|$ if $x_2 < 0$.

[5]Truncation means, for example, `REAL(INT(s))`. Rounding means, for example, `REAL(INT(s+.5)` if `s`≥ 0 or `REAL(INT(s-0.5))` if `s`< 0.

[6]If `str2` is contained in `str1` then the result is the position of the first character of `str2` in `str1`; otherwise the result is 0.

[6]For example, `LGE(str1,str2)` is true if the two strings are the same or if `str1` follows `str2` in lexicographical order (comparing ASCII codes character by character).

Input and Output

Perhaps the most difficult part of Fortran programming are the operations of inputting information into the program and outputting information from the program to a place where it can be used later. These operations are performed using the `read` and `write` statements respectively. There are four basic parts of these statements:

1. A description of where the information is coming from or going to,
2. A description of the format that the information is in if it is being read or a description of the format that is desired for the information if it is being written,
3. The statement number of the statement to be executed next if something goes wrong during the reading or writing,
4. The names of the variables that are to receive the information or the names of the variables whose values are to be written.

It is almost impossible to describe all of the variations on these four basic ideas. Most Fortran programmers learn them by imitating what other programmers have done. An example which contains all four elements is provided by the statements

```
      read(2,10,err=20) ((x(i,j),j=1,10),i=1,5)
 10   format(10x,6f10.0)
```

The read statement says to read 50 numbers from the file having “logical unit number” 2 according to the format statement having statement number 10, and assign these numbers to the upper left hand corner of the matrix `x` (upper five rows and 10 columns). The `err=20` means that if something goes wrong during the reading process (such as if a nonnumeric character were mistakenly entered in the file) then the next statement to be executed by the program is the one having statement number 20. The `6f10.0` in the format statement says that the 50 numbers should be read six per line, with the first of the six in columns 11–20, the next in 21-30, and so on, and the the numbers can be anywhere in these columns as long as there is a decimal point (note that the `10x` says to skip the first 10 columns on each line—this makes it posible to have extraneous information in the first 10 columns (such as a date if one were reading information collected over time and wanted the date in the file but didn't want to input it into the program).

Opening a File for Reading or Writing

Before reading from or writing to a file, one must first alert the computer's operating system what the name of the file is using the `open` statement. One also uses the `open` statement to attach a logical unit number to the file (such as the `2` in the write statement above) and then uses this number in future `read` statements. Note that one can have several files open at the same time, each with its own logical unit number.

Looping and Branching

An important part of Fortran are the statements that allow one to perform loops, that is executing sets of lines of code repeatedly. The most famous example is a “do loop”, which is of the form

```
      do 10 i=i1,i2,i3
         ....
         ....
   10 continue
```

where `i1`, `i2`, and `i3` are integers (or integer expressions). The statements between the `do` and the `continue` are all executed with the variable `i` having value `i1`, then they are done again with `i` having value `i1+i3`, and so on, until the value of `i` is outside of the range `i1` to `i2`. Note that do loops can be nested, and that the do loop counter (the variable `i` in the example above) can not be changed during the loop.

Another example of a loop is the so-called “implied loop” which is of the form

```
      i=1
    5 continue
      ....
      ....
      if(i.eq.n) go to 10
      i=i+1
      go to 5
   10 continue
```

There are a number of branching constructs available. The `go to` used above is an unconditional branch. The line

```
    if(i.eq.3) go to 10
```

is an example of a conditional branch. In addition to `.eq.` we have `.ne.`, `.le.`, `.lt.`, `.ge.`, and `.gt.`. The use of `go to`'s is discouraged as they make reading the code of a program difficult. Other, more readable branches are available such as

```
      if(i.eq.1) then
```

```
         ...
         ...
      elseif(i.eq.2) then
         ...
         ...
      else
         ...
         ...
      endif
```

Subroutines and Functions

There are two kinds of subprograms in Fortran; subroutines and functions. We have already seen examples of functions in the intrinsic functions above, for example the statement `x=sqrt(8.0*atan(1.0))` uses two intrinsic functions; `atan` and `sqrt`. Functions return single values "through their names" and any number of values (scalars or arrays) through their argument list. A subroutine on the other hand returns nothing through its name, can return any number of values through its argument list, and is invoked by a statement of the form

```
call subname(x,n,m,y,z)
```

To illustrate writing functions and subroutines, consider the following code:

```
      double precision function inprod(x,y,n)
      dimension x(n),y(n)

      inprod=0.0d0
      do 10 i=1,n
 10   inprod=inprod+dble(x(i))*dble(y(i))

      return
      end

      subroutine mtmult(a,b,n,ndim,c)
      dimension a(ndim,n),b(ndim,n),c(ndim,n)

      do 10 i=1,n
      do 10 j=1,n
         c1=0.0
         do 20 k=1,n
 20      c1=c1+a(i,k)*b(k,j)
 10   c(i,j)=c1

      return
      end
```

Thus functions and subroutines both end with the statements `return` and `end`, and begin with a line that gives their name (note how functions and subroutines name themselves differently and that functions declare their own type) and specifies a list of arguments, that is variables that are to be used in the function or subroutine.

There are a few simple rules that one must use in writing and calling subroutines:

1. Subprograms can call other subprograms but a subprogram can never call itself either directly or indirectly (by calling another subprogram which in turn calls the original subprogram). Some modern compilers allow this recursive calling, but most still do not so we will not discuss it.
2. A subroutine begins with a statement beginning with the word `subroutine` followed by a space, followed by the name of the subroutine, followed (optionally) by a list of variable names in parentheses and

separated by commas. These variables are referred to as the arguments of the subroutine. The variables in the calling program are not available to the subprogram unless they are passed to it through the argument list (one can also use a `common` statement but we ignore that for now). We consider arguments in more detail later in the section discussing arrays in Fortran.

3. One of the most common errors in Fortran occurs if there is a mismatch between the type of a variable in a main program and the type expected by the subroutine. For example, if a subroutine expects a single precision variable and is passed a double precision number, then the number it actually receives will not be what is expected and in many cases this will cause a problem (note that the compiler does not check for "type mismatches.")

Compiling and Linking

The first program considered later in this chapter (which is stored in the file `regswp.f`) is self-contained except that it uses a subroutine called `tcdf` which is in another file called `tcdf.f`. To compile and link `regswp.f` and `tcdf.f` and create an executable file called `regswp`, one does the following (the `-c` switch on the first line says to only compile `tcdf.f` while the `-o` switch on the second line says to name the executable file `regswp`).

```
f77 -c tcdf.f
f77 -o regswp regswp.f tcdf.o
```

Arrays in Fortran

Arrays are an often poorly understood part of Fortran, particularly in relation to how they are used in subprograms. Unfortunately, arrays and subprograms are often covered at the end of introductory Fortran courses, if at all. We will consider one dimensional arrays (vectors), two dimensional arrays (matrices having row and column indices), and three dimensional arrays (often visualized as a vector of matrices, the first two indices being the row and column numbers and the third index being the matrix number). The extension to higher dimensional arrays should be obvious.

Arrays are Statically Allocated

With a few notable exceptions, Fortran compilers force the programmer to specify the size of an array in a main program before it is compiled and linked. This is done with a `dimension` statement (such as `dimension a(10,10,4)`, which allocates a three dimensional array `a` of four 10 by 10 matrices). Thus the memory for an array is said to be "statically allocated." Many programming languages allow "dynamic memory allocation," that is, the program can ask the user to specify the size of a problem and then allocate exactly the right amount of space for the arrays that are involved. In contrast, the Fortran programmer must anticipate the maximum size that a user might want and do the `dimension` accordingly.

Arrays are Stored in Memory in "Column Order"

Most features of a language such as Fortran do not require the programmer to keep in mind how the program interacts with a computer's memory. However, the ordering of the elements of an array is a notable exception to this rule. The elements are stored in consecutive locations in memory in what is called "column order." For a vector this just means that consecutive elements are stored in consecutive locations in memory, but how the ordering is done for higher dimensional arrays is not obvious. For a matrix, the basic rule is that the elements of the first column are stored in order followed by the elements of the second column, and so on. Thus code such as

```
      dimension a(4,4)
      data a/11,21,31,41,12,22,32,42,13,23,33,43,14,24,34,44/
```

```
      do 10 i=1,4
   10 write(*,20) (a(i,j),j=1,4)
   20 format(1x,4f4.0)
```

leads to

```
11. 12. 13. 14.
21. 22. 23. 24.
31. 32. 33. 34.
41. 42. 43. 44.
```

that is, the first four elements in the `data` statement are placed into the first column of `a`, the next four in the second column, and so on.

For a three dimensional array, we have rows, columns, and matrix number. If we had dimensioned `a` above by `dimension a(2,2,4)`, then the four matrices would be

```
11  31     12  32     13  33     14  34
21  41     22  42     23  43     24  44,
```

that is, the first four elements go into the first matrix (with the first two in the first column and the next two in the second), the next four in the second matrix, and so on.

Perhaps the best way to visualize this is to think of all arrays as vectors, with the dimensions specifying how that vector gets reshaped into the array. The important point is that in the assignment of elements into the array so that "the first index varies most rapidly." Thus if we added the line

```
      write(*,20) a
```

to the code above, we would get

```
11. 21. 31. 41.
12. 22. 32. 42.
13. 23. 33. 43.
14. 24. 34. 44.
```

Dimensioning Arrays in a Subprogram

An important part of programming is to create subprograms (subroutines or functions) to perform often needed calculations (such as adding or multiplying matrices). In Fortran, variables defined in one "module" (main program or subprogram) are unknown to any other module unless the programmer does something special to "pass" the values from one module to the other. The standard method to do this in Fortran is to have an argument in the subprogram list and in the call to the subprogram for the variable of interest. This is called "passing a variable" to the subprogram. It is still necessary to dimension any passed array in the subroutine. Further, it is posible to use variables to dimension arrays in a subprogram (note that such a variable must also be in the argument list).

A simple example of this is to write a subroutine which writes out the elements of an n by n matrix

```
      subroutine matprt(b,n)
      dimension b(n,n)
      do 10 i=1,n
   10 write(*,20) (b(i,j),j=1,n)
   20 format(1x,6f12.5)
      return
      end
```

Note that the variables in such a subprogram are called "dummy variables" and need not have the same names as the corresponding variables in the calling subprogram.

In keeping with what we said above, our subroutine will take the first n^2 elements being passed from the array in the calling routine, form the matrix b by column and then write out the rows of the result. But what are the first n^2 elements of the array in the calling routine? In the following main program,

```
      dimension a(4,4)
      data a/11,21,31,41,12,22,32,42,13,23,33,43,14,24,34,44/
      call matprt(a,3)
      stop
      end
```

the first 9 elements of `a` get passed to `matprt` and thus the output of the program will be

```
  11.00000    41.00000   32.00000
  21.00000    12.00000   42.00000
  31.00000    22.00000   13.00000
```

which is probably not at all what the programmer intended (presumably they wanted the upper left hand 3 by 3 portion of the matrix displayed).

The "ndim problem"

This last example illustrates a problem which causes the vast majority of hard-to-find Fortran bugs. What's worse is that many times the programmer has no idea that there is a problem (in the example we knew what the "correct" values should have been). One can only hope that the incorrect passing leads to a big enough problem (such as division by zero) so that the subtle passing problem gets diagnosed.

I refer to this problem as the "ndim problem" since the solution is to include a third argument in the subroutine (often called `ndim`) corresponding to the actual row dimension of the array in the calling routine (in this case it would be 4). In our example, `ndim` is the natural name for this argument as it is the actual dimension corresponding to the argument `n` in the calling routine. Thus the subroutine would be modified to be

```
      subroutine matprt(b,n,ndim)
      dimension b(ndim,n)
      do 10 i=1,n
   10 write(*,20) (b(i,j),j=1,n)
   20 format(1x,6f12.5)
      return
      end
```

and the call would be

```
     call matprt(a,3,4)
```

which would give the intended output.

A Bonus

A very powerful (but often dangerous) feature of Fortran is that one can pass an array having one number of indices (vector, matrix, three dimensional, etc.) to a subroutine which is expecting an array having a different number. This allows us to do something such as

```
      dimension x(8,2)
      data x/11,21,31,41,12,22,32,42,13,23,33,43,14,24,34,44/
      call mean(x(1,2),8,xbar)
      write(*,10) xbar
   10 format(1x,'xbar = ',f12.5)
      stop
      end
      subroutine mean(x,n,xbar)
      dimension x(n)
      xbar=0.0
      do 10 i=1,n
   10 xbar=xbar+x(i)
      xbar=xbar/n
      return
      end
```

which would find the sample mean of the second column of the matrix `x`. This example also illustrates another important fact about array passing. We've seen that a subroutine takes consecutive elements being passed from the calling routine, shapes them into an array according to the dimension in the subprogram, and then operates on the array. In the line `call mean(x(1,2),8,xbar)`, we see that the argument in the call is actually only "pointing" to the first element in `x` to be passed (note that if the argument had just been `x`, the pointer would be at the first element in `x`, that is, `x(1,1)`). This is useful in something such as

```
      dimension x(100),xbar(100)
      n=100
      do 10 i=1,n
   10 x(i)=i
      do 20 i=1,n
   20 call mean(x(n-i+1),i,xbar(i))
```

which finds the mean of the last i x's for $i = 1, \ldots, 100$.

A final example of this is given by writing a subroutine to calculate $X^T X$ for an $n \times m$ matrix X:

```
      subroutine xprx(x,n,m,ndim,mdim,xtx)
      dimension x(ndim,m),xtx(mdim,mdim)
      double precision dbprod
c
      do 10 i=1,m
      do 10 j=1,i
      xtx(i,j)=dbprod(x(1,i),x(1,j),n)
   10 xtx(j,i)=xtx(i,j)
      return
      end
c
      double precision function dbprod(x,y,n)
      dimension x(1),y(1)
      dbprod=0.0d0
c
      do 10 i=1,n
   10 dbprod=dbprod+dble(x(i))*dble(y(i))
      return
      end
```

There are several things to note here:

1. We need both an `ndim` and an `mdim` in this case.

2. The $(i, j)th$ element of $X^T X$ is the inner product of its ith column with its jth column so we are using the inner product function `dbprod` and just pointing to the beginning of the two columns of interest in the call to `dbprod`.

3. Since $X^T X$ is symmetric, `xprx` only calculates the the elements in the lower triangle and puts each element into the corresponding place in the upper triangle as well.

4. The variables `x` and `y` have been "dimensioned with a 1" in the function `dbprod`. This is consistent with what we have said above about the role of dimensions. This sort of thing is sometimes very useful in avoiding extra arguments. For example, if we were writing a subroutine to find the zeros of the polynomial $g(z) = \sum_{j=0}^{p} \alpha_j z^j$ we would probably pass $\alpha_0, \alpha_1, \ldots, \alpha_p$ in the vector `alpha` of length $p+1$ (since there is no zero index in standard Fortran) and to avoid passing the value of $p+1$, we can just dimension `alpha` with a 1.

An Example

We consider a program to do basic multiple linear regression which illustrates reading and writing and how to use subroutines.

Review of Regression Analysis

A very common situation in statistics is to have a set of n "objects" with measurements on a set of $m+1$ "variables" for each object and we wish to explain the behavior of one of the variables y (the "dependent" variable) in terms of the other m variables $X_1, \ldots, X_m$ (the "independent" or "regressor" variables) according to

$$y_i = \beta_0 + \beta_1 X_{i1} + \cdots + \beta_m X_{im} + \epsilon_i, \qquad i = 1, \ldots, n,$$

where y_i and $X_{i1}, \ldots, X_{im}$ are the measurements for the ith object, $\beta_0, \beta_1, \ldots, \beta_m$ are a set of unknown parameters, and ϵ_i is called the error for the ith object. The ϵ's are assumed to be statistically independent for the different objects, have expectation 0, all have the same variance σ^2, and are included in the model to explain the fact that not all objects whose X values are the same will have the same y value.

The aims of the regression analysis are twofold: 1) to determine which of the X's are important in explaining the behavior of y, and 2) to develop a prediction formula that can be used to find reasonable values of y for a new object for which we only know the X's.

An example of such a situation is the Hald data set included at the end of this report, wherein $n = 13$, $m = 4$, y is the amount of heat generated during the hardening of cement, and the X's are the amount of four different chemicals used in the production of the cement. In the data file, y is listed first followed by the four X's.

If we rewrite the multiple linear regression model above in matrix form as

$$\mathrm{y} = \mathrm{X}\boldsymbol{\beta} + \boldsymbol{\epsilon},$$

where y is the $n \times 1$ vector of the y values, X is the $n \times (m+1)$ matrix of the X values with a column of 1's appended at the beginning, $\boldsymbol{\beta}$ is the $(m+1) \times 1$ vector of coefficients, and $\boldsymbol{\epsilon}$ is the $n \times 1$ vector of ϵ's, then it is easy to write down the basic results of a regression analysis. First, the best linear unbiased estimators $\hat{\boldsymbol{\beta}}$ of the β's and an unbiased estimator s^2 of σ^2 are given by

$$\hat{\boldsymbol{\beta}} = (\mathrm{X}^T\mathrm{X})^{-1}\mathrm{X}^T\mathrm{y}, \quad s^2 = \frac{RSS}{n-m-1}, \quad RSS = \mathrm{e}^T\mathrm{e}, \quad \mathrm{e} = \mathrm{y} - \hat{\mathrm{y}}, \quad \hat{\mathrm{y}} = \mathrm{X}\hat{\boldsymbol{\beta}},$$

where the vectors $\hat{\mathrm{y}}$ and e are called the vectors of fitted values and residuals, and RSS is the residual sum of squares.

If we further assume that the ϵ's are normally distributed, then the hypothesis that β_j is zero while the other β's are not is rejected at the α significance level if $|t_j| > t_{\alpha/2,n-m-1}$ where $t_{\alpha/2,n-m-1}$ is the upper $\alpha/2$ critical point of a t distribution having $n-m-1$ degrees of freedom and

$$t_j = \frac{\hat{\beta}_j}{\sqrt{s^2(\mathrm{X}^T\mathrm{X})^{-1}_{j+1,j+1}}}.$$

Note that the denominator of t_j is called the standard error of $\hat{\beta}_j$. Typically, instead of just reporting whether the hypothesis is rejected or not, one determines the p-value of $\hat{\beta}_j$, that is, the probability that a t with $n-m-1$ degrees of freedom is outside the interval $\pm t_j$, in which case the null hypothesis is rejected if $p_j < \alpha$.

A measure of how much of the variability in y is explained by its linear dependence on the X's is

$$R^2 = 1 - \frac{s_e^2}{s_y^2},$$

where s_e^2 and s_y^2 are the sample variances of the e's and the y's, respectively.

A Multiple Linear Regression Program Using SWEEP

In this section we give as an example a regression program which

1. Asks the user for a file name containing a regression data set to analyze and a file name to receive the results of the analysis. The program assumes that the file starts with a line containing a label to be displayed on the output and a line with the number of observations, n, and the number of independent variables, m, in the regression (not counting an intercept term). Each of the remaining n lines in the input file is assumed to contain the value of the dependent variable followed by the values of the independent variables. An example input file is given below which contains the so-called Hald Data which has 13 observations and four independent variables:

```
Hald Data (From Draper and Smith, pg. 296, 304, 630)
13 4
 78.5   7 26  6 60
 74.3   1 29 15 52
104.3  11 56  8 20
 87.6  11 31  8 47
 95.9   7 52  6 33
109.2  11 55  9 22
102.7   3 71 17  6
 72.5   1 31 22 44
 93.1   2 54 18 22
115.9  21 47  4 26
 83.8   1 40 23 34
113.3  11 66  9 12
109.4  10 68  8 12
```

2. Reads the data into the vector y and the matrix X (putting a column of 1's into the beginning of X so that it is $n \times (m+1)$ after reading the data), forms the augmented cross-product matrix

$$A = \begin{bmatrix} X^T X & X^T y \\ y^T X & y^y \end{bmatrix},$$

and then gets the regression information by using the matrix SWEEP algorithm.

3. The least squares estimates, their standard errors, t-statistics, and corresponding p-values are then written out, as well as the value of R^2. The result for the Hald Data are:

```
Hald Data (From Draper and Smith, pg. 296, 304, 630)

Regression results, R-Squared = 0.98237562, d.f. =   8

      i     beta(i)       se(i)        t(i)     pval(i)
  -----------------------------------------------------
      0   62.405369   70.070959    0.890602    0.399134
      1    1.551103    0.744770    2.082660    0.070822
      2    0.510168    0.723788    0.704858    0.500901
```

3	0.101909	0.754709	0.135031	0.895923
4	-0.144061	0.709052	-0.203174	0.844071

The Program

```
c  -------------------------------------------
c   REGSWP.F: Regression program using SWEEP
c
c   Uses functions inprod and smean and subroutines
c   tcdf, tpdf, betai, and gama
c  -------------------------------------------

c  ---------------------------------------------
c   Declare variables and external functions:
c  ---------------------------------------------

      character*60 fnin,fnout,label

      real*8 x(1000,20),y(1000),a(21,21),beta(21),se(21),t(21),pval(21)

      real*8 cdf,pdf,inprod,rss,s2,ybar,ssy,rsq

      real*8 smean

      logical fnexist

      data ndim/21/

c  ------------
c   Get data:
c  ------------

      write(*,10)
 10   format(' Enter file name containing the regression data: '$)

      read(*,20) fnin
 20   format(a60)

      inquire(file=fnin,exist=fnexist)

      if(.not.fnexist) go to 100

      write(*,30)
 30   format(' Enter the file name to get the output: '$)
      read(*,20) fnout

      open(2,file=fnout,status='new')
      open(1,file=fnin,status='old')

c   Read first two lines of input file:

      read(1,20,err=110) label
      read(1,*,err=120) n,np

c   Read data, putting X's in columns 2,...,m+1 and 1's in 1st column:

      m=np+1
      do 40 i=1,n
      x(i,1)=1.0
 40   read(1,*,err=130) y(i),(x(i,j),j=2,m)

c  -----------------------------------------
c   Form the augmented crossproduct matrix:
c  -----------------------------------------

      mp1=m+1
      do 60 i=1,m
             a(i,mp1)=inprod(x(1,i),y,n)
             a(mp1,i)=a(i,mp1)
             do 50 j=1,i
                    a(i,j)=inprod(x(1,i),x(1,j),n)
 50                 a(j,i)=a(i,j)
 60   continue
      a(mp1,mp1)=inprod(y,y,n)

c  ---------------------------------------------------------
c   Sweep it on first m diagonals and check for singularity:
c  ---------------------------------------------------------
```

```

      call sweep(a,ndim,mp1,1,m,ier)

      if(ier.eq.1) go to 90


c  --------------------------------------------------------------
c   Now get LSE's, RSS, R squared, se's, t-statistics, and p-values:
c  --------------------------------------------------------------

      rss=a(mp1,mp1)
      s2=rss/(n-m)

      ybar=smean(y,n)

      do 70 i=1,n
   70 ssy=ssy+(y(i)-ybar)**2

      rsq=1.-(rss/ssy)

      do 80 i=1,m
             beta(i)=a(i,mp1)
             se(i)=sqrt(s2*a(i,i))
             t(i)=beta(i)/se(i)
             call tcdf(dabs(dble(t(i))),n-m,cdf,pdf)
   80        pval(i)=2.*(1.-cdf)

c  --------------------
c   write out results:
c  --------------------

      write(2,81) label
   81 format(1x,a60)

      write(2,82) rsq,n-m
   82 format(/,' Regression results, R-Squared = ',f10.8,
     1          ', d.f. = ',i3)

      write(2,83)
   83 format(/,10x,'i',5x,'beta(i)',7x,'se(i)',8x,'t(i)',5x,'pval(i)',/
     1        6x,53(1h-))

      do 84 i=1,m
   84 write(2,85) i-1,beta(i),se(i),t(i),pval(i)
   85 format(6x,i5,4f12.6)
      go to 99

c  -----------------
c   Error handling:
c  -----------------

   90 write(*,*) ' Regression matrix singular'
      go to 99

  100 write(*,*) ' File doesn''t exist'
      go to 99

  110 write(*,*) ' Error reading label'
      go to 99

  120 write(*,*) ' Error reading n and m'
      go to 99

  130 write(*,135) i
  135 format(' Error reading row ',i5)

   99 continue
      stop
      end
      subroutine sweep(a,mdim,m,k1,k2,ier)
c  -------------------------------------------------------------
c
c   Subroutine to sweep the mxm matrix a on its k1 st thru k2 th
c   diagonals.  ier is 1 if a is singular. mdim is row dimension
c   of a in calling routine.
c
c  -------------------------------------------------------------

      double precision a(mdim,1)
      double precision d
```

```

      ier=1

      do 50 k=k1,k2

             if(abs(a(k,k)).lt.1.e-20) return

             d=1./a(k,k)
             a(k,k)=1.
             do 10 i=1,m
             a(k,i)=d*a(k,i)
 10          if(i.ne.k) a(i,k)=-a(i,k)*d

             do 20 i=1,m
             do 20 j=1,m
 20          if((i.ne.k).and.(j.ne.k)) a(i,j)=a(i,j)+a(i,k)*a(k,j)/d

 50     continue

        ier=0

        return
        end

        double precision function inprod(x,y,n)
        double precision x(n),y(n)

        inprod=0.0d0
        do 10 i=1,n
 10     inprod=inprod+x(i)*y(i)

        return
        end

        double precision function smean(x,n)
        double precision x(n)

        smean=0.0d0
        do 10 i=1,n
10      smean=smean+x(i)
        smean=smean/n

        return
        end

```

The External Statement and Numerical Integration

The Common and Named Common Statements and Newton-Raphson

See the chapter on “Iterative Methods for Parameter Estimation” for examples of using the common and named common statements.

CHAPTER 8

Introduction to S-Plus

S-Plus is a very powerful program for doing statistics research and data analysis, including the ability to easily generate random numbers, manipulate arrays of various dimensions, and to produce very high quality graphics.

The standard reference for using S-Plus is "An Introduction to S and S-Plus" by Phil Spector (a PhD graduate of our department, now at UB-Berkeley, `spector@stat.berkeley.edu`) published by Duxbury Press in 1994.

Starting, Stopping, and Using S-Plus

To start S-Plus, one need only enter the command `S+` from the Unix prompt. When it starts, the user sees a prompt (a `>` sign). To exit S-Plus and return to the Unix prompt, one enters the command `q()`.

An example of what can be done using S-Plus is (anything following a "pound sign" is a comment and is ignored by S-Plus):

```
> X11()                             # open a graphics window on the screen
> x <- rnorm(100)                   # generate N(0,1) sample of size 100
> x                                 # saying the name of a variable or fctn
                                    # causes it to be displayed on the screen
> plot(x,type="l")                  # plot x(i) versus i
> postscript("out.ps")              # open a file to get postscript graph
> plot(x,type="l")                  # send plot to file
> dev.off(dev.clr())                # close postscript file
> !lpr -Php2 out.ps                 # send postscript file to hp2 printer
> X <- matrix(rnorm(1000),100,10)   # form 100 by 10 matrix of N(0,1)'s
> XTX <- t(X) %*% X                 # get X'X
```

Keeping Track of an S-Plus Session

Every variable or function you define during a session is saved in a special form in a directory called `.Data` somewhere in your directory structure. If you have created such a directory in the directory where you start S-Plus, then the program will use that one. If not, it will create a `.Data` in your home directory or use the one that is already there if there is one. This is a very nice feature as everything you've done is still there the next time you use S-Plus (unless of course you've redefined something).

Subscripting

One of the most important parts of S-Plus is subscripting. Let `X` be a matrix and `A` be a three dimensional array. Then `X[i,j]` is the (i, j)th element of `X`, while `X[i,]` and `X[,j]` are the ith row and jth column respectively. This can be extended to three (and higher) dimensional arrays as well, eg, `A[,,i]` is the ith matrix in the three dimensional array and `A[3,,4]` is the third row of the fourth matrix.

You can pick out parts of arrays more generally by specifying each index you want. For example, if `U` is a vector of U(0,1)'s, you can pick out the elements that are less than 0.25 by saying `U[U<0.25]`, since `U<0.25` is a vector of T's and F's.

There are a million tricks one can learn about subscripting and they are crucial to using S-Plus efficiently.

S-Plus Functions

S-Plus has an on-line help system. To get help about a particular command, enter `help(name)` where `name` is the name of the command. In the following table, we list the names of some of the more useful functions contained in S-Plus.

`function(s)`	Purpose
Distributions: Random numbers (prefix `r`), pdf (prefix `d`), cdf (prefix `p`), and quantile (prefix `q`):	
`beta, binom, cauchy, chisq, exp, f, gamma, geom, lnorm`	
`lnorm, logis, nbinom, norm, pois, t, unif, weibull`	
Numerical operators	
`+, -, *, /, ^`	
`%*%, %/%, %%`	Matrix multiplication, integer divide, modulus
Form arrays	
`c`	`c(x,y,0,4,z)` forms vector
`rep`	`x <- rep(0,100)` gives (0,...,0)
`:`	`1:n` gives (1,...,n); `n:1` gives (n,...,1), etc.
`seq`	`seq(from=0,to=1,by=.1)` or `seq(from=0,to=1,len=11)`
`sample`	`sample(n)` gives random permutation of (1,...,n)
`diag`	Create diagonal matrix or extract diagonal elements
`cbind`	Combine vectors into matrix having the vectors as columns
`rbind`	Combine vectors into matrix having the vectors as rows
`col`	`col(A)` same size as `A` but with column numbers
`row`	`row(A)` same size as `A` but with row numbers
Comparison operators	
`>, <, <=, ==, !=`	
Logical operators	
`&, |, &&, ||, !`	Elementwise and, or; control and, or; unary not
Mathematical functions	
`abs, exp, gamma, lgamma, log, log10, sign, sqrt`	
Trigonometric functions	
`cos, sin, tan, acos, asin, atan, cosh, sinh, tanh, acosh, asinh, atanh`	
Complex number functions	
`Arg, Conj, Mod, Re, Im`	

Rounding and truncating

`round, signif, trunc, ceiling, floor`

Looping

`if, for, while, repeat, next, break`

Sorting and ranking

`sort`	Sort an array
`rev`	Reverse the order of an array
`order`	ith element of `order(x)` is index of ith smallest `x`
`rank`	Return ranks of an array

Functions for simple statistics

`cor, cumsum, mean, median, min, max, prod, range, sum, var`

`quantile`	$X_{(i)} = Q((i-1)/(n-1))$
`sample`	`sample(n)` gives a random permutation of $1, \ldots, n$

Functions for categorical data

`cut`	Create categories from continuous variables
`pretty`	Creates convenient break points from continuous variable
`split`	Break up an array based on value of categorical variable
`table`	Count by categories

Data management

`append, duplicated, match, pmatch, replace, unique`

Matrix or array functions

`nrow, ncol`	Number of rows and columns of matrix
`dim`	Vector of dimensions of a multiway array
`chol`	Cholesky decomposition
`crossprod`	Kronecker product
`eigen`	Eigenanalysis
`outer`	Outer product of two vectors
`scale`	Scale the columns of a matrix
`solve`	Solve system of equations or invert a matrix
`qr`	QR Orthogonalization
`t`	Matrix transpose

Character manipulation

`abbreviate`	Generate abbreviations of character values
`cat`	Display values on screen or send to file
`grep`	Search for patterns in characters
`nchar`	Number of characters in a string
`paste`	Put numbers into character strings
`substring`	Extract parts of character values

Statistical inference

```
binom.test, chisq.test, cor.test, fisher.test, friedman.test, kruskal.test
mantelhaen.test, mcnemar.test, prop.test, t.test, var.test, wilcox.test
```

Statistical Modeling

`lm`	Linear models and regression
`aov`	Analysis of variance

`glm`	Generalized linear models
`gam`	Generalized additive models
`loess`	Local regression models
`tree`	Tree-based models
`nls`	Nonlinear regression

Time series analysis

Multivariate analysis

Reading from files

`scan, dget`

Writing to screen or printer

`print, sink, cat, round, paste, format, write, data.dump, data.restore`

Open graphics devices

`X11, motif, tek4014, postscript`	First two for workstations, next for over modem, last for printer

Manipulating graphics devices

`dev.cur, dev.list, dev.next, dev.prev, dev.off, dev.set`

Graphics parameters

High level graphics functions

`barplot, boxplot, contour, hist, pie, plot, qqnorm, qqplot, tsplot, usa`

`coplot`	Separate plots for different ranges
`dotchart, faces`	Plot multivariate data
`pairs`	Scatterplos for all possible pairs of columns in a matrix
`persp`	3-D perspective plot
`plclust`	Plot of cluster trees from `hlclust`

Low level graphics (add things to existing plot)

`lines, points, segments, polygon, arrows, symbols, text`
`axis, axes, box, legend, mtext, title`

`tslines, tspoints`	Lines and points on time series plots
`abline`	Add regression line to plot
`perspp`	Project 3-D points to 2-D points to be used with `persp`
`frame`	Advance to next figure
`labclust`	Add labels to cluster plot
`stamp`	Add time stamp to a plot

An Example

I have files called `oz80.dat`, `oz81.dat`, and so on, up to `oz93.dat`, which contain hourly readings of ozone concentrations for one of the years 1980 through 1993 at between 11 and 13 monitoring stations around Harris County, Texas. Each line in each file starts with the numbers of the month, day, and hour for that line's observation, followed by the ozone level at the different stations. Thus the file for 1993 (a non-leap year) contains 8,760 lines (365 days times 24 hours per day). In 1993 there were 11 monitoring stations active. To read the 8,760 by 14 matrix, we would say

```
oz <- matrix(scan("oz93.dat"),8760,14,byrow=T)
```

The `scan` function reads all 8,760 times 14 values in the file `oz93.dat` into a single vector (it will read across the lines) and then the `matrix` function will put the elements of that vector into the 8760 by 14 matrix `oz` "by row."

If a monitoring site was malfunctioning at any time, the value 8888 or 9999 was inserted into the data file. Thus to count the number of missing values at each site and display them on the screen, we could do

```
for(i in 1:11) print(sum(oz[,i+3]>=8888))
```

which shows the power and terseness of S-Plus syntax. The `oz[,i+3]` will form a vector containing the `i+3`rd column of `oz` (remember that the first three columns of `oz` are month, day, hour). Then `oz[,i+3]>=8888` will form a vector of `T`'s and `F`'s corresponding to elements that are greater than or equal to 8888 (ie, missing) or less than 8888 (not missing). Then applying the `sum` function to that vector of `T`'s and `F`'s will treat the `T`'s as 1's and the `F`'s as 0's, and thus we get the number of missing values.

Arrays Are Stored By Column

If you define a matrix by `X <- matrix(1:9,3,3)`, you get

```
  1  4  7
  2  5  8
  3  6  9
```

Similarly, if you do `A <- array(1:16,c(2,2,4))` the four matrices are

```
 a[,,1]    A[,,2]    A[,,3]     A[,,4]
  1  3       5  7      9  11     13  15
  2  4       6  8     10  12     14  16
```

Writing Your Own Functions

An important part of using S-Plus is writing our own functions. If we include the following code in a file (maybe called `myfuncs.S`)

```
plot4 <- function(x)
{
  par(mfrow=c(2,2))        # divide graphics window into 2 by 2 regions
  hist(x)                  # put histogram in upper left
  boxplot(x)               # boxplot in upper right
  f <- density(x)          # get kernel density estimate
  plot(fx,fy,type="l")   # plot it
  qqnorm(x)                # qq plot in lower left
}
```

and we said `source("myfuncs.S")` from the S-Plus prompt, the fucntion `plot4` would be available from then on to be used anytime we used S-Plus.

In writing functions, you need to keep the following points in mind:

1. Note the structure of a function. The first line has the function name followed by the `<-`, followed by a list of arguments. The "body" of the function is contained within matching left and right curly brackets.
2. A function can return a single object (which often is a "list," see the next section) "in its name." There is a `return` statement (see the example below) but the last thing defined in the function is returned by default.
3. If an argument is changed in the function, its value in the calling function is not changed.

4. Arguments can have default values, eg, `rnorm <- function(n,mean=0,sd=1)` can be used by saying `x <- rnorm(100)` or `x <- rnorm(100,5)` to get $\mu = 5$ and $\sigma = 1$, or `x <- rnorm(100,sd=4)` to get $\mu = 0$ and $\sigma = 4$.

The Concept of a List

Like most programming languages (other than fortran), S-Plus has the ability to group together into one logical structure several variables. The structure in S-Plus is called a "list." For example, if we write the following function

```
myreg <- function(X,y)
{
  beta   <- solve(t(X) %*% X, t(X) %*% y)
  fitted <- X %*% beta
  res    <- y - fitted
  ssq    <- sum(res * res) / (nrow(X) - ncol(X))
  Rsq    <- 1 - var(res)/var(y)
  tstat  <- beta / sqrt(ssq * diag(solve(t(X) %*% X)))
  pval   <- 2 * pt(1 - abs(tstat))
  return(list(coeff=beta, yhat=fitted, res=res, ssq=ssq, Rsq=Rsq,
              tstat=tstat, pval=pval))
}
```

and we have the Hald Data on a file called `hald1.dat` without the title and n and m lines, we could say

```
  zz      <- scan("hald1.dat",list(y=0,x1=0,x2=0,x3=0,x4=0))
  n       <- length(zz$y)
  haldout <- myreg( cbind( rep(1,n),zz$x1,zz$x2,zz$x3,zz$x4)), zz$y)
```

and then, for example, we could plot the residuals by saying

```
plot(haldout$res)
```

Interfacing with C or Fortran

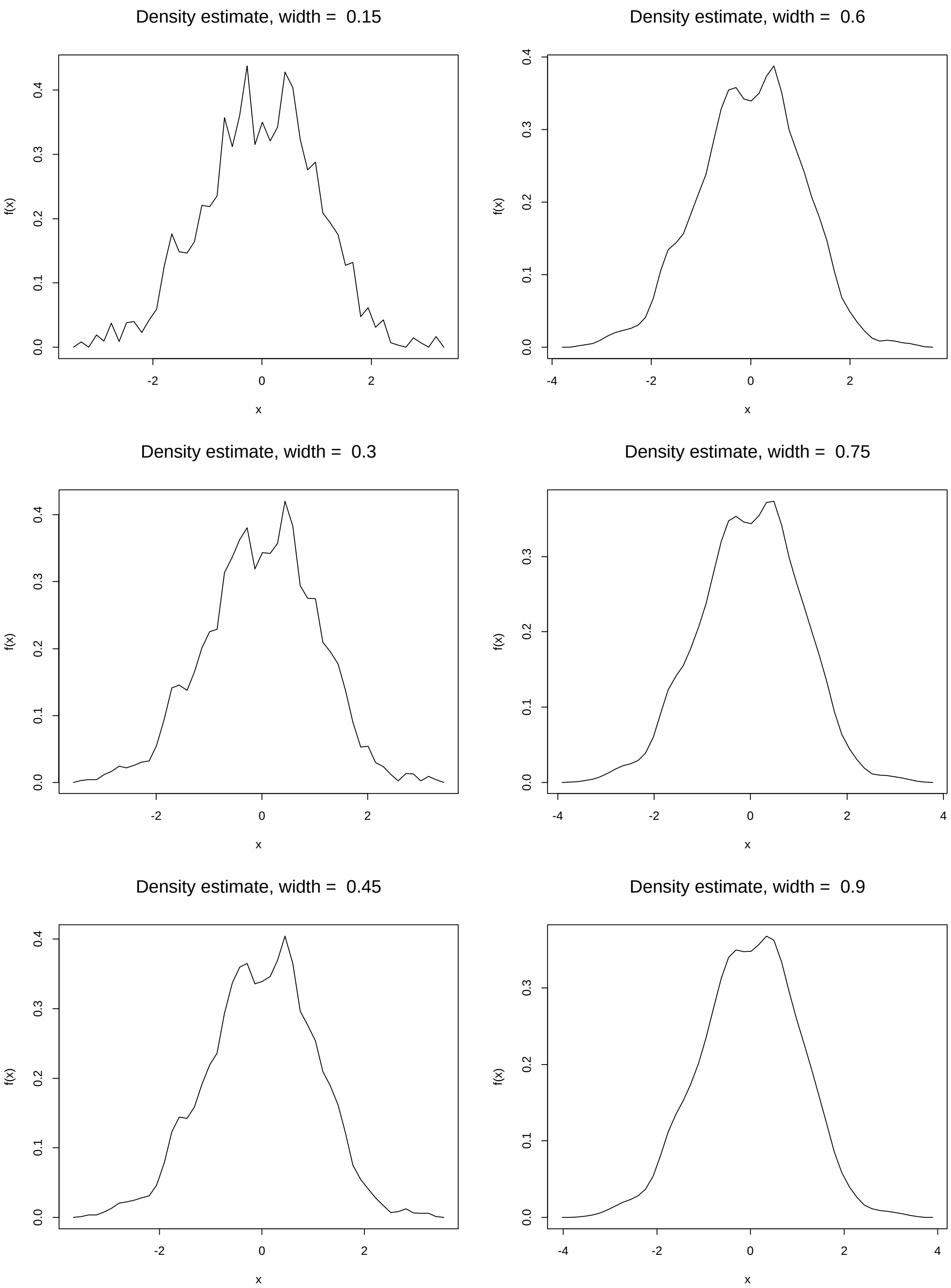
It is also possible to create our own functions in Fortran or C (see pg. 130 of Spector's book for example).

Two Graphics Examples

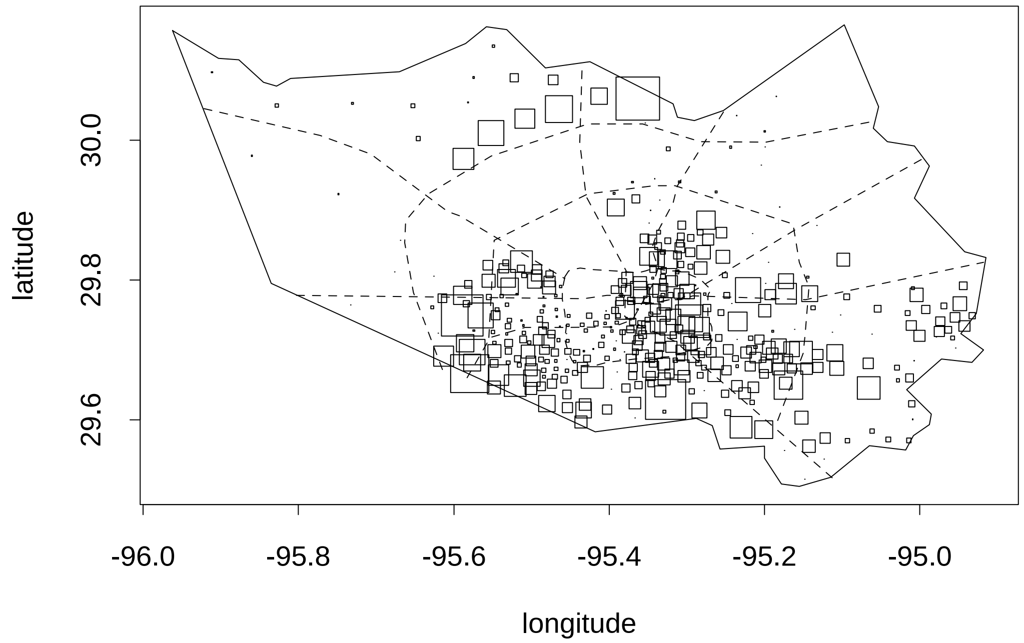
Kernel Density Estimate

The following function produces the page of density estimate plots.

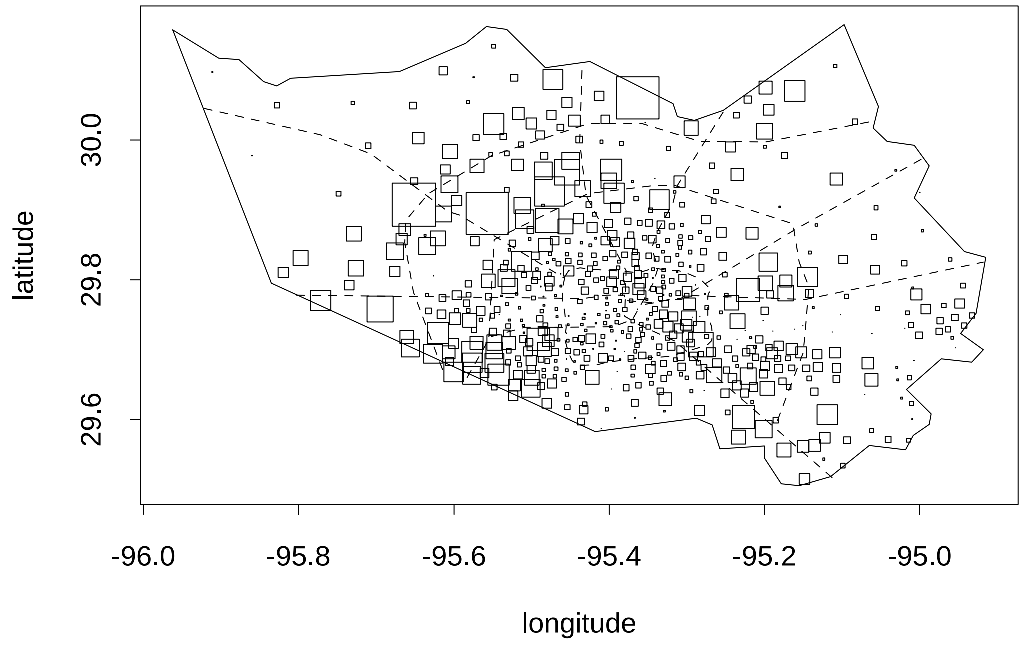
```
plot.dens <- function()
{
  par(mfcol=c(3,2))
  x <- rnorm(1000)
  for(i in 1:6) {
    plot(density(x,width=.15*i),type='l',xlab='x',ylab='f(x)',
         main=paste("Density estimate, width = ",round(.15*i,3)))
  }
}
```


Density estimate, width = 0.15
Density estimate, width = 0.6
Density estimate, width = 0.3
Density estimate, width = 0.75
Density estimate, width = 0.45
Density estimate, width = 0.9
f(x)
x

1980 Census Tracts and Children 5 Or Under



1990 Census Tracts and Children 5 Or Under



Census Data Example

The code given below produces the plot given on the previous page. Here are the first ten lines of the three files the example uses:

```
First 10 lines of kids80.dat
----------------------------
 121.00    29.7576  -95.3633   69.
 201.01    29.7729  -95.3339  295.
 201.02    29.7655  -95.3277  622.
 202.00    29.7567  -95.2899 1098.
 203.01    29.7807  -95.3107  464.
 203.02    29.7835  -95.2997  707.
 203.03    29.7774  -95.2997  286.
 204.00    29.7785  -95.3211  260.
 205.01    29.7865  -95.3433  511.
 205.02    29.7850  -95.3351  664.

First 10 lines of kids90.dat
----------------------------
 12100  29.7576 -95.3633    12
 20101  29.7729 -95.3339   273
 20102  29.7655 -95.3277   502
 20210  29.7657 -95.2966   833
 20220  29.7477 -95.2833     1
 20301  29.7807 -95.3107   337
 20302  29.7835 -95.2997   689
 20303  29.7774 -95.2997   312
 20400  29.7785 -95.3211   172
 20501  29.7865 -95.3433   340

First 10 lines of roads.dat
----------------------------
-95.962 30.157
-95.835 29.795
-95.418 29.583
-95.288 29.602
-95.267 29.592
-95.257 29.558
-95.200 29.562
-95.200 29.545
-95.178 29.508
-95.155 29.505
```

```
plot.census <- function()
{

  kids80 <- scan("kids80.dat",list(tract=0,y=0,x=0,kids=0))
  kids90 <- scan("kids90.dat",list(tract=0,y=0,x=0,kids=0))

  par(mfrow=c(2,1),mar=c(4,4,4,4))

  drawroads(main="1980 Census Tracts and Children 5 Or Under")

  symbols(kids80$x,kids80$y,squares=kids80$kids,add=T,inches=.3)

  drawroads(main="1990 Census Tracts and Children 5 Or Under")

  symbols(kids90$x,kids90$y,squares=kids90$kids,add=T,inches=.3)

}

drawroads <- function(lty=3,cex=1.0,main="")
{

  nl <- c(47,11,13,11,8,6,11,40,6)

  nstart <- c(1,cumsum(nl)[1:8]+1)    # 1,48,59,etc.
  nend   <- c(nstart[2:9]-1,sum(nl))  # 47,58,etc

  plot(roads$x[nstart[1]:nend[1]],roads$y[nstart[1]:nend[1]],type="l",
       xlab="longitude",ylab="latitude",cex=cex,main=main)

  for(i in 2:9) lines(roads$x[nstart[i]:nend[i]],
                      roads$y[nstart[i]:nend[i]],lty=lty)

}
```

```
read.roads <- function()
{

  roads <- scan("roads.dat",list(x=0,y=0))

  return(roads)

}
```

CHAPTER 9

An Introduction to TEX

TEX is a computer program for creating beautiful manuscripts, particularly those containing mathematics. This report was created using TEX, while many new books are being created using TEX. All publications of the American Mathematical Society are now being produced using TEX. While there are many programs for document preparation, TEX has a number of advantages, including a) it is used all over the world on all types of computers, b) to use TEX the user creates a file containing no 'special characters' (usually referred to as an 'ASCII file'), which allows easy communication of documents via electronic means (such as e-mail), and c) most observers feel that documents created using TEX are superior in quality than those created by other software.

In this chapter, we introduce the reader to how TEX can be used to produce mathematical manuscripts. The basic reference for the use of TEX is the *TEXbook* by Donald Knuth (the author of TEX) which is widely available in book stores.

1. Introduction

TEX is not a WYSIWYG (what you see is what you get) program, that is, the user does not see what the paper will look like until it is printed or previewed on the computer screen. One must first produce a file that the TEX program can process. This file is referred to as a TEX file and it should have `.tex` as its extension. On an X-Station, one would have an emacs window open to type in the TEX file. An example of such a file is

```
The pdf $f$ and cdf $F$ of an exponential distribution
having parameter $\lambda$ are given by
$$
f(x)=\lambda e^{-\lambda x},\qquad F(x)=1-e^{-\lambda x},\qquad x>0.
$$

\bye
```

which produces the output

The pdf f and cdf F of an exponential distribution having parameter λ are given by

$$f(x)=\lambda e^{-\lambda x},\qquad F(x)=1-e^{-\lambda x},\qquad x>0.$$

After producing a TEX file, the user must do the following steps to produce a printed document:

1. Use the TEX program to process the TEX file that was created in step (1). This is done by the command `tex filename` where `filename` need not include the `.tex` extension. If there are no mistakes in the TEX file, this step will produce a file called a 'DVI' file since it is 'device independent', that is, it will be the same no matter what computer one is using. The DVI file will have the same name as the TEX file except it will have extension `.dvi`. Using the TEX program will also create a 'log file' having the same

name as the TEX file except extension `.log`. If there are mistakes in the TEX file, the TEX program will describe what they are. The user must then correct the mistakes in the TEX file.

2. Once there are no more mistakes in the TEX file, the user executes a 'printer driver' program which will convert the DVI file to another file which is in a form that is understandable to the particular printer that is connected to the user's computer. In many cases, this printer driver program will also actually send the final file to the printer. Also, many computers have a program called a 'screen previewer' program that can be used to see a low resolution version of the final document on their screen. On many X-Stations, the printer driver programs are called `dvihp` for Laserjet printers and `dvips` for postscript printers, while the screen previewer is called `xdvi`.

3. If the printer driver does not send the final output to the printer, the user executes a printing program to send the file produced in step (2) to the printer.

For more information on this process, see your local system administrator.

Commands and Macros

To typeset ordinary text, the user merely types the text and TEX determines line and page breaks automatically. To tell TEX to do something other than just process text, one types what is called a 'command' (also called a macro), that is, a backslash (like this: \), followed by the name of the command, followed by a space. This space is needed so that TEX knows that the name of the command has ended. For example, typing `\TeX` produces TEX. The names of macros contain only latin letters (no numerical digits for example). Thus if the name of a macro is followed by a character that is not a letter, the usually required space following the name is not required. Note that throughout this report, we will indicate something that is to be typed by the user by having it appear in typewriter font (`like this`).

To tell TEX to end the current paragraph, one issues the command `\par` or leaves a blank line in the file.

TEX comes with about 300 commands built in and one can create their own set of commands. Almost all computers that have TEX installed also have several files containing sets of macros that users have created. These files are usually referred to as 'style files', and usually also include commands specifying what margins and fonts are to be used. The best thing to do is to ask around and find out what style files there are and what they contain.

Sizes and Fonts

1. By default, TEX leaves a one inch margin all around the page. This is controlled by the parameters `\hsize`, `\vsize`, `\hoffset`, and `\voffset`, which are 6.5, 9.0, 0.0, and 0.0 by default. To have a 5 by 7 inch page, still centered on the paper, one would use `\hsize=5in`, `\vsize=7in`, `\hoffset=0.75in`, `\voffset=1in`.

2. By default, TEX numbers the pages of a document starting with the number 1 and places the numbers centered at the bottom of each page. One can have the numbers start with something other than 1 by issuing the command (to start with 10, for example) `\pageno=10`. To get Roman numerals for page numbers, set `\pageno` equal to a negative value. One can also place the page numbers somewhere else and get a running headline.

3. The nonmathematical characters that are produced are in what is called ten point Computer Modern Roman font unless the user does something to change this default setting. This font is called ten point because the height of the tallest character (which is a parenthesis) is ten points high (1 inch is 72.27 points). Each line of text has what is called a baseline (the bottom of the letters that don't have 'descenders'), and by default these baselines are twelve points apart. This can be changed to 14 points for example by the command `\baselineskip=14pt`. When typing math (see below), latin characters are in what is called math italics (*like this*). Note that there is no concept of 'doublespaced' in TEX. In fact, there is nothing like 'lines' on a typewriter. How much space is left between lines of what is printed is determined by the `baselineskip`.

4. Several other types of fonts are available, such as italics (*like this*), or bold face (**like this**), or any combination of Roman, *italics*, **bold face**, `typewriter`, *slanted*, or SMALL CAPS, by typing for example

```
or any combination of Roman, {\it italics}, {\bf bold face},
{\tt typewriter}, {\sl slanted}, or {\smc small caps}
```

This illustrates the very important concept of 'grouping.' Note that we have surrounded each of the words in the various fonts by left and right curly brackets. This makes it so that the font only lasts inside of the brackets. There are a number of situations when you will want something to apply only temporarily and using the brackets will allow this.

5. At any one time, TEX uses fonts of three sizes. One size is for regular characters, while the other two sizes are for sub or superscripts and sub or superscripts applied to sub or superscripts. The default sizes are 10, 7, and 5 points. For example, in X_{n_i}, the default size for the X is ten points, the n is seven points, and the i is five points. Usually a computer installation will have established a style file that will give other sizes since many people find 10, 7, and 5 points to be too small.

6. One can magnify all sizes in a document by five different factors called 'magsteps' which are half, 1, 2, 3, and 4, for example if you include the line

```
\magnification=\magstep1
```

at the beginning of your document, then all sizes will be multiplied by 1.2. The five factors are $\sqrt{1.2}$, 1.2, 1.2^2, 1.2^3, and 1.2^4. You can only apply a magnification once in a document and it applies throughout the document. One is tempted to use magnification to get fonts of sizes other the default 10, 7, and 5 points (for producing overheads for example), but it is important to understand that fonts do not come in all sizes. To define a 12 point bold face font called `bfone`, one includes the line

```
\font\bfone=cmbx10 scaled\magstep1
```

in the TEX file prior to using the font and then invoking the font by saying its name in a way similar to using `\bf` in item 4 above. The important part of this line above is the font name `cmbx10`. The other names are `cmti10` for *italics*, `cmtt10` for `typewriter`, `cmsl10` for *slanted*, `cmcsc` for SMALL CAPS. There are many other families of fonts as well.

7. The first line of each paragraph is indented by an amount controlled by the parameter `\parindent`, the default value of which is 20 pts. You can avoid the indenting of the first line by typing `\noindent`.

Vertical and Horizontal Skips

To skip space vertically one types (to skip two millimeters for example; there are also `in`, `cm`, and `pt` for inches, centimeters, and points) `\vskip 2mm`. Note that doing this will end the current paragraph. To skip space horizontally, one uses `hskip` instead of `vskip`. For a variety of important reasons, these skips are not obeyed at the beginning of a page or the beginning of a line, respectively. To get some blank space at the top of a page, one can type `\vtop{ }\vskip 4mm`.

Using Comments in the TEX File

Anything following a percent sign on a line in the TEX file is treated as a comment and is ignored by the TEX program (see the last page of this chapter for an example of using comments). As with any other programming language, it is important to put comments into the file. This is particularly important in TEX as TEX files are notoriously difficult to read and you will use previous files as templates for future documents and you will need to be able to find useful examples. Further, while you are 'debugging' your TEX file (getting all the syntax errors out of it) you will need to be able to read the file easily. One very useful way to do this is to use blank lines liberally between paragraphs.

Some Special Characters

The characters $, %, &, #, _, and \ have special meaning to TEX and must be typed in a special way:

```
\$, \%, \&, \#, \_, and $\backslash$
```

There are three kinds of dashes: 1) the hyphen as in the phrase ‘math-in-text’, 2) the en-dash as in the phrase ‘read sections 10–12’, and 3) the em-dash as in the phrase ‘now is the time—I mean right now’. These three dashes are produced by typing one, two, or three minus signs, respectively.

You should note the quote marks in this document (‘like this’). The one on the right is familiar to everyone, but pay particular attention to the one on the left. It is different than the one on the right and there is a key on all keyboards that produces this left quote.

One can apply accents to any letter as in ò, ó, ô, ö, õ, ō, ȯ, ŏ, ǒ, ő, o͡o, o̧, ọ, o̲, and there are the special characters œ, Œ, æ, Æ, å, Å, ø, Ø, ł, Ł, ß. These are obtained by typing

```
\‘o \’o \^o \"o \~o \=o \.o \u o \v o \H o \t oo \c o \d o \b o
```

and

```
\oe  \OE  \ae  \AE  \aa  \AA  \o  \O  \l  \L  \ss
```

The Item and Itemitem Macros

Several times in this chapter we have displayed lists of items (see the first page of the chapter for example). The items are characterized by the fact that all lines in an item are indented by the current value of the parameter `\parindent` and the first line of an item is preceded by an identifier (such as the 1, 2, and so on). For example,

1. Item 1.
    a. Subitem a.
    b. Subitem b.
2. Item 2.
3. Item 3.

is obtained by typing

```
\vskip 2mm\item{1.} Item 1.
\vskip 2mm\itemitem{a.} Subitem a.
\vskip 2mm\itemitem{b.} Subitem b.
\vskip 2mm\item{2.} Item 2.
\vskip 2mm\item{3.} Item 3.
```

Note that an item ‘stays active’ until a new paragraph is encountered (by a blank line, a `\vskip`, or another `item` or `itemitem`.

Typing Math

There are two ways that mathematical symbols appear in a manuscript. The first is when they are mixed into a regular line of text such as in ‘Let $S = X_1 + \cdots + X_n$, where $X_1, \ldots, X_n$ is a random sample from the distribution having pdf $f(x) = \lambda e^{-\lambda x}$ for $x > 0$,’ which is obtained by typing

```
‘Let $S=X_1+\cdots+X_n$, where $X_1,\ldots,X_n$ is a
random sample from the distribution having pdf
$f(x)=\lambda e^{-\lambda x}$ for $x>0$,’
```

Thus 'math in text' is obtained by surrounding the math with single dollar signs. The second way math is used is when an expression is displayed on a separate line, that is, 'display math,' such as in

$$
F(x)=\int_{-\infty}^x f(y)dy, \qquad
\bar X=\sum_{i=1}^nX_i, \qquad
\tilde X=\hbox{\rm median}\{X_i\}, \qquad
\hat\beta=S_{XY}/S_{XX},
$$

which is obtained by surrounding the math in 'double dollar signs':

```
$$
F(x)=\int_{-\infty}^x f(y)dy, \qquad
\bar X=\sum_{i=1}^nX_i, \qquad
\tilde X=\hbox{\rm median}\{X_i\}, \qquad
\hat\beta=S_{XY}/S_{XX},
$$
```

These examples illustrate a number of important features of typing math:

1. They illustrate how to get subscripts (using the underscore character) and superscripts (using a caret). The important thing to notice is that if a subscript or a superscript is a single character, then one types the underscore or caret followed by the single character. If it is more than a single character, then the characters must be placed inside curly brackets following the underscore or caret (another example of grouping via the curly brackets). This makes it possible to get extremely complicated expressions very simply.

2. Integrals and summations are obtained similarly, with the lower and upper limits being obtained in the same way as subscripts and superscripts.

3. Notice the difference between `\ldots` and `\cdots` and when they are used.

4. Notice how $\bar X$, $\tilde X$, and $\hat\beta$ are typed.

5. Most mathematical functions (such as 'median') are typed in Roman font while all characters are by default produced in math italics (*like this*). Notice how we typed 'median.'

6. We were able to display in a readable way what we typed to get the displayed equation because of the fact that spaces that you type while in the math mode (except for those required after macros) are ignored by $\TeX$. In order to get spaces, you have to tell $\TeX$ to put them in. For example, the macro `\qquad` leaves a 'qquad' of space, where a 'quad' is the width of a capital M and a 'qquad' is two quads.

Alignments

Many times one needs to line up a series of expressions, such as in

$$
\begin{aligned}
&f(x)=\dfrac{1}{\sqrt{2\pi}}e^{-x^2/2}\\
&F(x)=\int_{-\infty}^xf(y)dy,
\end{aligned}
$$

which is obtained from

```
$$
\eqalign{
   &f(x)=\dfrac{1}{\sqrt{2\pi}}e^{-x^2/2}\cr
   &F(x)=\int_{-\infty}^xf(y)dy,\cr
}
$$
```

or the same kind of thing but with equation numbers:

$$\begin{aligned} X(t) &= \sum_{k=1}^n a_k \cos \frac{2\pi(t-1)}{n} + b_k \sin \frac{2\pi(t-1)}{n} \qquad (2) \\ &= \sum_{k=1}^n c_k \cos \left(\frac{2\pi(t-1)}{n} - \phi_k \right). \end{aligned}$$

which one gets from

```
$$
\eqalignno{
  X(t)&=\sum_{k=1}^na_k\cos\dfrac{2\pi(t-1)}{n}+
                      b_k\sin\dfrac{2\pi(t-1)}{n}&(2)\cr
       &=\sum_{k=1}^nc_k\cos\left(\dfrac{2\pi(t-1)}{n}-\phi_k\right).
\cr}
$$
```

All you have to do is put an & before the characters you want lined up and then put a `\cr` after each line. Note that `eqalignno` is the same as `eqalign` except it allows equation numbers. There is an `leqaligno` too that puts equation numbers on the left.

In the last expression above, the outer parentheses are larger than the inner parentheses. Typing `\left` and `\right` before the outer left and right parentheses told TEX to figure out how large they should be. This can also be done with square and curly brackets.

Typing Cases

Often we have to define functions whose values vary according to different conditions as in

$$(x+y)^n = \begin{cases} x^2+2xy+y^2, & \text{if } n=2 \\ x^3+3x^2y+3xy^2+y^3, & \text{if } n=3 \\ \sum_{j=0}^n \binom{n}{j} x^j y^{n-j}, & \text{in general.} \end{cases}$$

which is obtained by typing

```
$$
(x+y)^n=\cases{
                 x^2+2xy+y^2,&if $n=2$\cr
                 x^3+3x^2y+3xy^2+y^3,&if $n=3$\cr
                 \sum_{j=0}^n{n\choose j}x^jy^{n-j},&in general.
                }
$$
```

This is the `cases` command. Note that it is very similar to `eqalign` in that you put ampersands before the conditions for each case and a `\cr` at the end of each line.

Typing Matrices

Typing matrices is done similarly, for example,

$$\mathbf{X} = \begin{bmatrix} X_{11} & X_{12} & \dots & X_{1p} \\ X_{21} & X_{22} & \dots & X_{2p} \\ \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ X_{n1} & X_{n2} & \dots & X_{np} \end{bmatrix}, \mathbf{L} = \begin{bmatrix} L_{11} & & & & \\ L_{21} & L_{22} & & & \\ \vdots & \vdots & \ddots & & \\ \vdots & \vdots & & \ddots & \\ L_{p1} & L_{p2} & \dots & \dots & L_{pp} \end{bmatrix}.$$

This is obtained by

```
$$
\hbox{{\bf X}}=\left[\matrix{X_{11}&X_{12}&\ldots&X_{1p}\cr
                            X_{21}&X_{22}&\ldots&X_{2p}\cr
                            \vdots&\vdots&      &\vdots\cr
                            \vdots&\vdots&      &\vdots\cr
                            X_{n1}&X_{n2}&\ldots&X_{np}\cr}\right],
\hbox{{\bf L}}=\left[\matrix{L_{11}\cr
                            L_{21}&L_{22}\cr
                            \vdots&\vdots&\ddots\cr
                            \vdots&\vdots&      &\ddots\cr
                     L_{p1}&L_{p2}&\ldots&\ldots&L_{pp}\cr}\right].
$$
```

Thus you type each row of the matrix with ampersands between the elements and a `\cr` at the end of each line.

Log-Like Functions

All latin characters in the math modes are formed in math italics. It is conventional that mathematical functions such as log should be formed in Roman font. Thus TEX has the following macros for producing the desired Roman font for commonly used expressions:

`\arccos`	`\cos`	`\csc`	`\exp`	`\ker`	`\limsup`	`\min`	`\sinh`
`\arcsin`	`\cosh`	`\deg`	`\gcd`	`\lg`	`\ln`	`\Pr`	`\sup`
`\arctan`	`\cot`	`\det`	`\hom`	`\lim`	`\log`	`\sec`	`\tan`
`\arg`	`\coth`	`\dim`	`\inf`	`\liminf`	`\max`	`\sin`	`\tanh`

We can easily obtain similar effects for expressions not in the above list, for example, $\sigma^2 = \sqrt{\mathrm{Var}(X)}$, which is obtained by typing

```
$\sigma^2=\sqrt{\hbox{\rm Var}(X)}$
```

rather than $\sigma^2 = \sqrt{Var(X)}$. Some of the expressions in the table given above will achieve special effects when they have subscripts, for example:

$$\lim_{n\to\infty} a_n = 0, \quad \min_{n\le k} X_n = 10$$

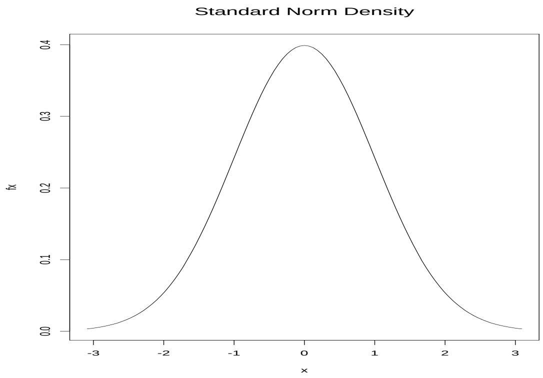
which is obtained from

```
$$
\lim_{n\to\infty}a_n=0,\quad\min_{n\le k}X_n=10
$$
```

Typing Tables

One of the more challenging parts of TEX is typing tables. For example, at the end of this chapter on a separate page is a table and the code that produced the table. Note that to get things on a separate page, one need only put `\pageinsert` at the beginning of the material and `\endinsert` at the end. There is also a `\topinsert` and `\midinsert`. Note the comment lines in the code.

Tables are formed using the `\halign` command. One creates a 'template' describing how each element of each line in the table is to appear in the table. In the template you have to say whether the elements of a column are to be left justified (by typing `#\hfil`; note that the `\hfil` being on the right of the `#` 'pushes' the entry (which is represented by the `#`) to the left), or right justified (by typing `\hfil#`) or centered (by typing `\hfil#\hfil`). Then each row of the table is entered with ampersands separating the entries and a `\cr` at the end of each row. While in the middle of the alignment process, one can use `\noalign` to put in horizontal lines and other effects (see the example).



Inserting Postscript Graphs

To insert postscript graphs into a document (such as the one at the top of this or the next page—with inserts one doesn't always know where the graph will go), one must insert the line

```
\include psfig
```

at the beginning of the file, and then lines such as

```
\topinsert
\centerline{\psfig{file=example.ps,height=2in,width=3in}}
\endinsert
\vskip 4mm
```

in the file.

Writing Your Own Macros

There are two main reasons to write your own macros. First, if you find yourself typing the same long character string over and over again, you can define a macro and then only type the macro. For example, a time series book would include the phrase 'covariance stationary time series' repeatedly. If you include the line

```
\def\CSTS{covariance stationary time series}
```

somewhere in your file, then you can type `\CSTS` anytime after you defined it. Perhaps the most important use of macros is when they have arguments. Here are three examples:

```
\def\sechd#1{\vskip 4mm\noindent{\bf#1}\vskip 4mm}
\def\dfrac#1#2{\displaystyle{{#1}\over{#2}}}
\def\binom#1#2{{{#1}\choose{#2}}}
```

The first example defines a section heading by skipping four millimeters but not indenting, then put the 'argument' of the macro in bold face, and then skip four more millimeters. The macro would be used by typing

```
\sechd{Writing Your Own Macros}
```

for example. Note how the argument is defined using the `#1`. The second example makes it easy to produce fractions, and the third produces binomial coefficients, for example, typing

```
$$
M_{nk}=\binom{n}{k/2},\qquad N_{nk}=\dfrac{M_{nk}}{n^3}
$$
```

gives

$$M_{nk} = \binom{n}{k/2}, \qquad N_{nk} = \frac{M_{nk}}{n^3}$$

Lining Things Up

One often needs to create special forms where things are aligned in unusual ways. One obtains

MEMORANDUM

TO:	John Doe
FROM:	Fred Smith
SUBJECT:	Making Forms

____________________ ____________________

John Doe Fred Smith

by typing

```
\def\hitem#1#2#3{\line{\hbox to #1{#2\hfil}#3\hfil}}
\noindent MEMORANDUM\vskip 2mm
\hitem{1in}{TO:}{John Doe}
\hitem{1in}{FROM:}{Fred Smith}
\hitem{1in}{SUBJECT:}{Making Forms}

\vskip 6mm\noindent
\line{\hbox to 2in{\hrulefill}\hfil\hbox to 2in{\hrulefill}}
\line{\hbox to 2in{\hfil John Doe\hfil}\hfil
      \hbox to 2in{\hfil Fred Smith\hfil}}
```

Running Headers

Here is what I used in this report to do the running headline:

```
\newcount\pgno

\headline={\ifnum\pageno=\pgno\blank \else\ifodd\pageno\rhl
 \else\lhl\fi\fi}

\footline={\ifnum\pageno=\pgno\hss\tenrm\folio\hss
  \else\nopagenumbers\fi}

\def\blank{\tenrm\hfil {}\hfil}
\def\lhl{\tenrm\folio\hfil {\smc Intro to \TeX}\hfil {\smc ch.} ??}
\def\rhl{\tenrm {\smc sec} ??\hfil {\smc Intro to \TeX}\hfil\folio}
```

List of Special Symbols

There are a host of special symbols that TEX can produce. These symbols fall into several categories and are listed below.

Lower Case Greek Letters

`\alpha`	α	`\iota`	ι	`\varrho`	ϱ
`\beta`	β	`\kappa`	κ	`\sigma`	σ
`\gamma`	γ	`\lambda`	λ	`\varsigma`	ς

\delta	δ	\mu	μ	\tau	τ
\epsilon	ϵ	\nu	ν	\upsilon	υ
\varepsilon	ε	\xi	ξ	\phi	ϕ
\zeta	ζ	o	o	\varphi	φ
\eta	η	\pi	π	\chi	χ
\theta	θ	\varpi	ϖ	\psi	ψ
\vartheta	ϑ	\rho	ρ	\omega	ω

Upper Case Greek Letters

\Gamma	Γ	\Xi	Ξ	\Phi	Φ
\Delta	Δ	\Pi	Π	\Psi	Ψ
\Theta	Θ	\Sigma	Σ	\Omega	Ω
\Lambda	Λ	\Upsilon	Υ		

Calligraphic Capitals

In the math mode, typing `\cal A`, `\cal B`, and so on gives

$$\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}, \mathcal{F}, \mathcal{G}, \mathcal{H}, \mathcal{I}, \mathcal{J}, \mathcal{K}, \mathcal{L}, \mathcal{M}, \mathcal{N}, \mathcal{O}, \mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathcal{S}, \mathcal{T}, \mathcal{U}, \mathcal{V}, \mathcal{W}, \mathcal{X}, \mathcal{Y}, \mathcal{Z}.$$

Miscellaneous Symbols of Type Ord

\aleph	$\aleph$	\prime	$\prime$	\forall	$\forall$
\hbar	$\hbar$	\emptyset	$\emptyset$	\exists	$\exists$
\imath	$\imath$	\nabla	∇	\neg	$\neg$
\jmath	$\jmath$	\surd	$\surd$	\flat	$\flat$
\ell	ℓ	\top	$\top$	\natural	$\natural$
\wp	$\wp$	\bot	$\bot$	\sharp	$\sharp$
\Re	Re	\\|	$\Vert$	\clubsuit	$\clubsuit$
\Im	Im	\angle	$\angle$	\diamondsuit	$\diamondsuit$
\partial	∂	\triangle	$\triangle$	\heartsuit	$\heartsuit$
\infty	∞	\backslash	$\backslash$	\spadesuit	$\spadesuit$

Large Operators

\sum	$\sum$	\bigcap	$\bigcap$	\bigodot	$\bigodot$
\prod	$\prod$	\bigcup	$\bigcup$	\bigotimes	$\bigotimes$
\coprod	$\coprod$	\bigsqcup	$\bigsqcup$	\bigoplus	$\bigoplus$
\int	$\int$	\bigvee	$\bigvee$	\biguplus	$\biguplus$
\oint	$\oint$	\bigwedge	$\bigwedge$		

These symbols are bigger in the display math mode than in the math within text mode, and the limits on them are treated differently, for example, $\sum_{j=1}^n$ and $\int_0^1$ versus

$$\sum_{j=1}^n \qquad \int_0^1 .$$

Binary Operations

`\pm`	$\pm$	`\cap`	$\cap$	`\vee`	$\vee$
`\mp`	$\mp$	`\cup`	$\cup$	`\wedge`	$\wedge$
`\setminus`	$\setminus$	`\uplus`	$\uplus$	`\oplus`	$\oplus$
`\cdot`	$\cdot$	`\sqcap`	$\sqcap$	`\ominus`	$\ominus$
`\times`	$\times$	`\sqcup`	$\sqcup$	`\otimes`	$\otimes$
`\ast`	$\ast$	`\triangleleft`	$\triangleleft$	`\oslash`	$\oslash$
`\star`	$\star$	`\triangleright`	$\triangleright$	`\odot`	$\odot$
`\diamond`	$\diamond$	`\wr`	$\wr$	`\dagger`	$\dagger$
`\circ`	$\circ$	`\bigcirc`	$\bigcirc$	`\ddagger`	$\ddagger$
`\bullet`	$\bullet$	`\bigtriangleup`	$\bigtriangleup$	`\amalg`	$\amalg$
`\div`	$\div$	`\bigtriangledown`	$\bigtriangledown$		

Relations

`\leq`	$\leq$	`\geq`	$\geq$	`\equiv`	$\equiv$
`\prec`	$\prec$	`\succ`	$\succ$	`\sim`	$\sim$
`\preceq`	$\preceq$	`\succeq`	$\succeq$	`\simeq`	$\simeq$
`\ll`	$\ll$	`\gg`	$\gg$	`\asymp`	$\asymp$
`\subset`	$\subset$	`\supset`	$\supset$	`\approx`	$\approx$
`\subseteq`	$\subseteq$	`\supseteq`	$\supseteq$	`\cong`	$\cong$
`\sqsubseteq`	$\sqsubseteq$	`\sqsupseteq`	$\sqsupseteq$	`\bowtie`	$\bowtie$
`\in`	$\in$	`\ni`	$\ni$	`\propto`	$\propto$
`\vdash`	$\vdash$	`\dashv`	$\dashv$	`\models`	$\models$
`\smile`	$\smile$	`\mid`	$\mid$	`\doteq`	$\doteq$
`\frown`	$\frown$	`\parallel`	$\parallel$	`\perp`	$\perp$

Many of these symbols can be negated by typing `\not` in front of the command for a symbol, for example, typing `\not\leq` gives $\not\leq$.

Arrows

`\leftarrow`	$\leftarrow$	`\longleftarrow`	$\longleftarrow$	`\uparrow`	$\uparrow$
`\Leftarrow`	$\Leftarrow$	`\Longleftarrow`	$\Longleftarrow$	`\Uparrow`	$\Uparrow$
`\rightarrow`	$\rightarrow$	`\longrightarrow`	$\longrightarrow$	`\downarrow`	$\downarrow$
`\Rightarrow`	$\Rightarrow$	`\Longrightarrow`	$\Longrightarrow$	`\Downarrow`	$\Downarrow$
`\leftrightarrow`	$\leftrightarrow$	`\longleftrightarrow`	$\longleftrightarrow$	`\updownarrow`	$\updownarrow$
`\Leftrightarrow`	$\Leftrightarrow$	`\Longleftrightarrow`	$\Longleftrightarrow$	`\Updownarrow`	$\Updownarrow$
`\mapsto`	$\mapsto$	`\longmapsto`	$\longmapsto$	`\nearrow`	$\nearrow$
`\hookleftarrow`	$\hookleftarrow$	`\hookrightarrow`	$\hookrightarrow$	`\searrow`	$\searrow$
`\leftharpoonup`	$\leftharpoonup$	`\rightharpoonup`	$\rightharpoonup$	`\swarrow`	$\swarrow$
`\leftharpoondown`	$\leftharpoondown$	`\rightharpoondown`	$\rightharpoondown$	`\nwarrow`	$\nwarrow$
`\rightleftharpoons`	$\rightleftharpoons$				

By typing `\buildrel\alpha \beta \over \longrightarrow` one gets $\xrightarrow{\alpha\beta}$.

Table 1.1. Distributions in the `WN` Command

dist	Distribution	Mean	Variance
1	$N(0,1):\ Z$	0	1
2	$U(0,1):\ U$	$1/2$	$1/12$
3	Exponential: $-\ln(1-U)$	1	1
4	Logistic: $\ln(U/1-U)$	0	$\pi^2/3=3.28987$
5	Cauchy: $\tan(\pi(U-\dfrac12))$	∞	∞
6	Extreme Value: $\ln(-\ln(1-U))$	$-\gamma=-.5772$	$\pi^2/6=1.6449$
7	Lognormal: e^Z	$e^{1/2}=1.6487$	$e^2-e=4.67077$
8	Double Exponential	0	2
	$\begin{cases}\ln(2U), & U\le .5\\ -\ln(2(1-U)), & U>.5\end{cases}$		

```
\centerline{{\bf Table 1.1.} Distributions in the {\tt WN} Command}
%
%   Skip a little space, draw the dark line and skip a little more:
%
\vskip 2mm\hrule height1pt\vskip 2mm
%
%   The next line sets up space between entries in table.
%
\tabskip=1em plus2em minus.5em
%
%   Template (1st, 3rd, 4th centered, 2nd left justified):
%
\halign to \hsize{\hfil#\hfil&#\hfil&\hfil#\hfil&\hfil#\hfil\cr
%
%   The column headings and the line under the heading:
%
dist&Distribution&Mean&Variance\cr
\noalign{\vskip 2mm\hrule\vskip 2mm}
%
%   Now enter each line of the table:
%
1 & $N(0,1):\ Z$& 0 & 1\cr
2 & $U(0,1):\ U$& $1/2$& $1/12$\cr
3 & Exponential: $-\ln(1-U)$& 1 &1\cr
4 & Logistic: $\ln(U/1-U)$& 0 & $\pi^2/3=3.28987$\cr
5 & Cauchy: $\tan(\pi(U-\dfrac12))$ & $\infty$ & $\infty$ \cr
6 & Extreme Value: $\ln(-\ln(1-U))$ & $-\gamma=-.5772$
& $\pi^2/6=1.6449$ \cr
7 & Lognormal: $e^Z$ & $e^{1/2}=1.6487$ & $e^2-e=4.67077$ \cr
8 & Double Exponential & 0 & 2\cr
  & $\cases{\ln(2U),&$ U\le.5$\cr-\ln(2(1-U)),&$U>.5$\cr}$ & &\cr
%
%   End the \halign with the }; draw the line under the table:
%
}
\vskip 2mm\hrule
```