Algorithms

Lloyd D. Fosdick, Editor

ALGORITHM 355
AN ALGORITHM FOR GENERATING ISING CONFIGURATIONS [Z]

J. M. S. Simões Pereira (Recd. 20 Dec. 1967 and 10 Mar. 1969)
University of Coimbra, Coimbra, Portugal

KEY WORDS AND PHRASES: Ising problem, zero-one sequences
CR CATEGORIES: 5.39

```
procedure Ising (n, x, t, S);  integer n, x, t;  integer array S;
comment  Ising generates n-sequences (S1, ···, Sn) of zeros and
   ones where x = Σ_{i=1}^{n} S_i and t = Σ_{i=1}^{n-1} | S_{i+1} - S_i | are given.
   The main idea is to interleave compositions of x and n - x
   objects and resort to a lexicographic generation of composi-
   tions. We call these sequences Ising configurations since we
   believe they first appeared in the study of the so-called Ising
   problem (See Hill [1], Ising [2]). The number R(n, x, t) of dis-
   tinct configurations with fixed n, x, t is well known [1, 2]:
```

$$R(n, x, t = 2m + 1) = 2\binom{x - 1}{m}\binom{n - x - 1}{m}$$

$$R(n, x, t = 2m) = \binom{x-1}{m}\binom{n - x - 1}{m - 1} + \binom{x-1}{m-1}\binom{n - x - 1}{m}$$

Now define a block of 1's (or zeros) in the sequence as a set of a maximum number of consecutive 1's (or zeros) eventually consisting of a single element. For given n, x, t, the number p of blocks of 1's may easily be deduced from t, as well as the number q of blocks of zeros. In fact, a block of 1's including either S_1 or S_n yields one variation and each one of the others yields two variations; hence we get $p = q = m + 1$ when $t = 2m + 1$ (t odd requires $S_1 \neq S_n$) and either $p = m + 1$, $q = m$ ($S_1 = S_n = 1$), or $p = m$, $q = m + 1$ ($S_1 = S_n = 0$) when $t = 2m$. Clearly, there is a 1-1 correspondence between the compositions of x with p parts and the distributions of the x 1's into p blocks. And for each distribution of 1's, distinct distributions of the $n - x$ zeros into g blocks correspond to distinct configurations.

The main body of the algorithm is *compose*, which generates compositions of an integer x with k parts and stores them in the array L. The role of *sort* and *bisort* is to form the final sequence $(S_1, \cdots, S_n)$ from the structure of one-blocks L_i and zero-blocks M_i.

The Ising problem was brought to my attention by Dr. B. Dejon during an informal visit to the IBM Research Laboratory in Zurich. Thanks are also due to Prof. Paul Erdös for pointing out to me reference [1] and to Prof. A. A. Zykov for correspondence. The procedure was tested on the NCR 4130 of the Laboratório de Cálculo Automático, Universidade do Porto. Thanks are also due to the Director and his Staff.

References

1. Hill, T. L. *Statistical Mechanics*. McGraw Hill, New York, 1956, p. 318.
2. Ising, E. Beitrag zur Theorie des Ferromagnetismus. *Z. Physik 31* (1925), 253–258;

```
begin
  integer k;  integer array L, M[1 : t÷2+1];
  procedure sort (L, M, z);  integer array L, M;  integer z;
  begin
    integer r, i, j, m, zb;
    for m := 1 step 1 until n do S[m] := z;
    r := i := 1;  zb := 1 - z;
AA: j := r + L[i] - 1;
    for m := r step 1 until j do S[m] := zb;
    if i + 1 ≤ k then
    begin r := j + M[i] + 1;  i := i + 1;  go to AA end;
    comment  Insert here an output procedure such as out-
      array (1, S);
  end sort;
  procedure bisort (L, M);  integer array L, M;
  begin sort (L, M, 0);  sort (M, L, 1) end bisort;
  procedure compose (x, k, L, p);  value x;  integer x, k;
    integer array L;  procedure p;
  begin
    integer i, a;
    if x < k then go to CC;
    L[1] := x - k + 1;
    for i := 2 step 1 until k do L[i] := 1;
    p;
    if k ≤ 1 then go to CC;
    a := 1;
BB: if L[a] > 1 then
    begin
      L[a] := L[a] - 1;  L[a+1] := L[a+1] + 1;  p;
      if a ≠ k - 1 then a := a + 1;  go to BB
    end;
    L[a] := L[a+1];  L[a+1] := 1;  a := a - 1;
    if a ≥ 1 then go to BB;
CC:
  end compose;
  k := t ÷ 2 + 1;
  if t ≠ (t÷2) × 2 then
  begin
    procedure p1;  bisort (L, M);
    procedure p2;  compose (n-x, k, M, p1);
    compose (x, k, L, p2)
  end
  else
  begin
    procedure p3;  sort (L, M, 0);
    procedure p4;  compose (n-x, k-1, M, p3);
    procedure p5;  sort (M, L, 1);
    procedure p6;  compose (n-x, k, M, p5);
    compose (x, k, L, p4);
    compose (x, k-1, L, p6)
  end
end Ising
```

ALGORITHM 356
A PRIME NUMBER GENERATOR USING THE TREESORT PRINCIPLE [A1]

RICHARD C. SINGLETON* (Recd. 28 Jan. 1969 and 11 June 1969)

Stanford Research Institute, Menlo Park, CA 94025

* This research was supported by the Stanford Research Institute out of Research and Development funds.

KEY WORDS AND PHRASES: prime numbers, number theory, sorting

CR CATEGORIES: 3.15, 5.30, 5.31

```
procedure PRIME(IP, m); value m;
  integer m; integer array IP;
```

comment This procedure finds the first $m \geq 4$ elements of the infinite sequence 2, 3, 5, 7, 11, $\cdots$ of prime numbers and stores them in $IP[1]$, $IP[2]$, $\cdots$, $IP[m]$. The method of distinguishing primes from composite numbers is similar to that used by B. A. Chartres [1]. A counter value n is compared with the smallest value in a list *IQ* of odd multiples of primes less than or equal to $\sqrt{n}$. If unequal, n is a prime and is added to the output list *IP*. Otherwise, the matching elements of *IQ* are incremented, based on the corresponding entries in the list *JQ*. Both n and the composite numbers in *IQ* are incremented so as to omit multiples of 2 and 3.

This procedure differs from Algorithm 311 in the method of finding the smallest entry in *IQ*. Here the list *IQ* is kept partially ordered as a tree, i.e.

$$IQ[i] \geq IQ[i \div 2] \quad \text{for } 2 \leq i \leq j,$$

thus the base element $IQ[1]$ is always smallest. The variable *iqi* holds the current value of $IQ[1]$, and *jqi* the negative of $JQ[1]$. If $n = iqi$, then *iqi* is incremented by $jqi + jqi$ if $jqi > 0$ or by $-jqi$ if $jqi < 0$. Then *IQ* is reordered to bring the next smallest element to the base and to return the new value of *iqi* to the tree, using a method similar to Williams' procedure *SWOPHEAP* [3]. The tag list *JQ* is permuted along with *IQ*. The treesort principle, used in *SWOPHEAP*, is well suited to the present task of finding the smallest element of a changing list.

In Algorithm 311, five working-storage arrays serve the function of the two used here, and the information is totally ordered each time a prime is found. Between primes the unordered segment of the information is searched to locate the smallest element. The method used here is both simpler and more efficient.

On the Burroughs B5500 computer, this procedure finds the first 10,000 primes in 53 sec. For other values of m, time is proportional to $m^{1.24}$. Corresponding times for Algorithm 311 were 91 sec for $m = 10{,}000$, with time proportional to $m^{1.35}$ for other values of m. However, another algorithm [2] finds the first 10,000 primes in 14 sec on the B5500 and has times proportional to $m^{1.14}$ for other values of m.

REFERENCES:

1. CHARTRES, B. A. Algorithm 311: Prime number generator 2. *Comm. ACM 10* (Sept. 1967), 570.
2. SINGLETON, R. C. Algorithm 357: An efficient prime number generator. *Comm. ACM 12* (Oct. 1969), 563–564.
3. WILLIAMS, J. W. J. Algorithm 232: Heapsort. *Comm. ACM 7* (June 1964), 347;

```
begin
  integer array IQ, JQ[0 : sqrt(m)];
  integer i, ij, inc, iqi, j, jj, jqi, k, n;
  IP[1] := j := 2;
  IP[2] := k := 3;
  IP[3] := n := 5;
  jj := iqi := 25; jqi := −10;
  IQ[2] := 49; JQ[2] := −14;
  inc := 4;
  go to Lc;
La: iqi := if jqi > 0 then iqi + jqi + jqi else iqi − jqi;
  i := 1;
  comment Reorder the tree, bringing the smallest element to
    the bottom;
  for ij := i + i while ij < j do
  begin
    if IQ[ij] > IQ[ij + 1] then ij := ij + 1;
    if IQ[ij] ≥ iqi then go to Lb;
    IQ[i] := IQ[ij]; JQ[i] := JQ[ij]; i := ij
  end;
  if iqi < jj then go to Lb; jj := IQ[j];
  comment Add a new entry to the top of the tree;
  j := j + 1; ij := IP[j + 2];
  IQ[j] := ij ↑ 2; JQ[j] := ij + ij;
  if (ij−(ij÷3)×3) = 1 then JQ[j] := − JQ[j];
  comment Return iqi and jqi to the tree and fetch a new pair
    from the bottom;
Lb: IQ[i] := iqi; iqi := IQ[1];
  JQ[i] := jqi; jqi := − JQ[1];
  if n = iqi then go to La;
  comment Increment n and compare with the next smallest
    composite number;
Lc: inc := 6 − inc; n := n + inc;
  if n = iqi then go to La;
  k := k + 1; IP[k] := n;
  if k ≠ m then go to Lc;
end PRIME
```

ALGORITHM 357
AN EFFICIENT PRIME NUMBER GENERATOR [A1]

RICHARD C. SINGLETON* (Recd. 28 Jan. 1969 and 11 June 1969)

Stanford Research Institute, Menlo Park, CA 94025

* This research was supported by the Stanford Research Institute out of Research and Development funds.

KEY WORDS AND PHRASES: prime numbers, factoring, number theory

CR CATEGORIES: 3.15, 5.30

```
integer procedure NPRIME(IP, m, jlim); value m, jlim;
  integer m, jlim; integer array IP;
```

comment This procedure finds the next m primes and stores them in $IP[1]$, $IP[2]$, $\cdots$, $IP[m]$. $IP[m+1]$, $IP[m+2]$, $\cdots$, $IP[jlim]$ are used for working storage, where $jlim > m$. On the first entry, $IP[1]$ must have a value less than 0 as a flag to set initial conditions. Also, m must be greater than or equal to 2 on first entry and greater than or equal to 1 on subsequent entries. The arrays *IQ* and *JQ* must be large enough to hold all primes less than or equal to the square root of the maximum number scanned in looking for primes. To generate the first million primes, approximately 550 entries are needed in each of these two lists. The lists are extended as needed, using a secondary prime number generator similar to Wood's [3], and the current upper index is returned as the value of *NPRIME*.

The method used is the familiar sieve of Eratosthenes. The elements of the upper portion of array *IP* are set to zero, and correspond to a sequence of consecutive odd integers. The composite numbers are crossed off by entering the smallest prime factor in the corresponding cell, leaving zeros for primes. (At this point, the array *IP* contains the equivalent of a factor table, i.e. the smallest factor for each composite odd integer.)

The list of primes is then constructed by storing the consecutive prime numbers in the lower portion of *IP*. Whenever the information in the upper portion of *IP* is exhausted, a new sequence of odd numbers is scanned as described above. On exit, the unused portion is left for use in the next call.

As compared with another algorithm [2] based on comparing a counter value with the next smallest composite number, and not working ahead in a scratch storage, the present algorithm was found to be faster, even for $jlim = m + 1$. Efficiency improves with added working storage. The improvement is substantial at first but is slight beyond $jlim = 2m$. For $jlim = 2m$, time to find the first n primes on the Burroughs B5500 or the CDC 6400 computer was proportional to $n^{1.14}$. On the B5500 computer, it took 13.5 sec to find the first 10,000 primes, generating them 500 at a time in an array length of 1022. On the CDC 6400 computer, with the algorithm coded in machine language, it took less than 98 sec to find the first million primes, generating them 1000 at a time in an array of length 10,000. Timing within this run, with $jlim = 10m$, was proportional to $n^{1.094}$. It is interesting to note that Chartres estimated a time of 12 hours on the B5500 for this task, using Algorithm 311 [1].

This algorithm can be expressed in either ALGOL or FORTRAN, and gains no special advantage from machine language coding. However, if we plan to produce very large tables of primes for future use, machine language shift operations may be useful in compressing the data for storage. One method of compression is to use a single bit to indicate that an integer is a prime, e.g. 0 = composite and 1 = prime. By omitting multiples of 2, 3, and 5 from the corresponding sequence of integers, 8 bits suffice to identify the primes in each 30 consecutive integers.

REFERENCES:

1. CHARTRES, B. A. Algorithm 311: Prime number generator 2. *Comm. ACM 10* (Sept. 1967), 570.
2. SINGLETON, R. C. Algorithm 356: A prime number generator using the treesort principle. *Comm. ACM 12* (Oct. 1969), 563.
3. WOOD, T. C. Algorithm 35: Sieve. *Comm. ACM 4* (Mar. 1961), 151;

```
begin
  own integer array IQ, JQ[0 : 600]
  own integer ij, ik, inc, j, nj;
  integer i, jqi, k, ni;
  k := 0;  if IP[1] ≥ 0 then go to Lf;
  comment Set initial conditions;
  IP[1] := JQ[1] := ik := inc := 2;
  IQ[2] := 9;  JQ[2] := IQ[1] := ij := 3;
  IQ[3] := 25;  JQ[3] := nj := 5;  k := 1;
  comment Prepare to delete a sequence of composite numbers;
La: j := k + 1;  ni := IQ[1] − j − j;
  IQ[1] := jlim + jlim + ni;
  for i := j step 1 until jlim do IP[i] := 0;
Lb: i := ij;  if IQ[ij] ≥ IQ[1] then go to Le;
  comment Extend the list of primes in array JQ counting so
    as to omit multiples of 2 and 3;
Lc: nj := nj + inc;  inc := 6 − inc;
  if JQ[ik + 1] ↑ 2 ≤ nj then ik := ik + 1;
  for j := 3 step 1 until ik do
    if (nj ÷ JQ[j]) × JQ[j] = nj then go to Lc;
  ij := ij + 1;  JQ[ij] := nj;  IQ[ij] := nj ↑ 2;
  go to Lb;
  comment If j + j + ni is composite, enter its smallest prime
    factor in IP[j]. If j + j + ni is prime, then IP[j] = 0;
Ld: IP[j] := jqi;  j := j + jqi;
  if j < jlim then go to Ld;
  IQ[i] := j + j + ni;
Le: i := i − 1;  jqi := JQ[i];  j := (IQ[i] − ni) ÷ 2;
  if j < jlim then go to Ld;
  if i ≠ 1 then go to Le;  j := k;
  comment Pack the next m primes in IP[1], ···, IP[m];
Lf: j := j + 1;  if IP[j] ≠ 0 then go to Lf;
  if j = jlim then go to La;
  k := k + 1;  IP[k] := j + j + ni;
  if k ≠ m then go to Lf;
  comment The current length of the tables in arrays IQ and
    JQ is returned;
  NPRIME := ij
end NPRIME
```

ALGORITHM 358
SINGULAR VALUE DECOMPOSITION OF A COMPLEX MATRIX [F1, 4, 5]

PETER A. BUSINGER AND GENE H. GOLUB (Recd. 31 Jan. 1969 and 18 June 1969)
Bell Telephone Laboratories, Inc., Murray Hill, NJ 07974
Stanford University, Stanford, CA 94305

KEY WORDS AND PHRASES: singular values, matrix decomposition, least squares solution, pseudoinverse
CR CATEGORIES: 5.14

CSVD finds the singular values $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_N$ of the complex M by N matrix (M ≥ N) which is given in the first N columns of the array A. The computed singular values are stored in the array S. CSVD also finds the first NU columns of an M by M unitary matrix U and the first NV columns of an N by N unitary matrix V such that $\|A - U\Sigma V^*\|$ is negligible relative to $\|A\|$, where $\Sigma = \text{diag}(\sigma_i)$. (The only values permitted for NU are 0, N, or M; those for NV are 0 or N). Moreover, the transformation U^* is applied to the P vectors given in columns N + 1, N + 2, ···, N + P of the array A. This feature can be used as follows to find the least squares solution of minimal Euclidean length (the pseudoinverse solution) of an overdetermined system $Ax \approx b$: Call CSVD with NV = N and with columns N + 1, N + 2, ···, N + P of A containing P right-hand sides b. From the computed singular values determine the rank r of Σ and define $\Sigma^+ = \text{diag}(\sigma_1^{-1}, \sigma_2^{-1}, \cdots, \sigma_r^{-1}, 0, \cdots, 0)$. Now $x = V\Sigma^+\tilde{b}$, where $\tilde{b} = U^*b$ is furnished by CSVD in place of each right-hand side b.

CSVD can also be used to solve a homogeneous system of linear equations. To find an orthonormal basis for all solutions of the system $Ax = 0$ call CSVD with NV = N. The desired basis consists of those columns of V which correspond to negligible singular values. Further applications are mentioned in the references.

The constants used in the program for ETA and TOL are machine-dependent. ETA is the relative machine precision, TOL the smallest normalized positive number divided by ETA. The assignments made are valid for a GE635 computer (a two's complement binary machine with a signed 27-bit mantissa and a signed 7-bit exponent). For this machine, ETA = $2^{-26} \doteq$ 1.5E-8 and TOL = $2^{-129}/2^{-26} \doteq$ 1.E-31.

The arrays B, C, and T are dimensioned under the assumption that N ≤ 100.

The authors wish to thank Dr. C. Reinsch for his helpful suggestions.

REFERENCES

1. GOLUB, G. Least squares, singular values, and matrix approximations. *Aplikace Matematiky 13* (1968), 44–51.
2. GOLUB, G., AND KAHAN, W. Calculating the singular values and pseudoinverse of a matrix. *J. SIAM Numer. Anal. 2* (1965), 205–224.
3. GOLUB, G., AND REINSCH, C. Singular value decomposition and least squares solutions. *Numer. Math.* (to appear)

```
      SUBROUTINE C S V D
     1   (A, MMAX, NMAX, M, N, P, NU, NV,
     2    S, U, V)
      COMPLEX A(MMAX,1),U(MMAX,1),V(NMAX,1)
      INTEGER M,N,P,NU,NV
      REAL S(1)
      COMPLEX G,R
      REAL B(100),C(100),T(100)
      DATA ETA,TOL/1.5E-8,1.E-31/
      NP=N+P
      N1=N+1
C
C HOUSEHOLDER REDUCTION
      C(1)=0.E0
      K=1
   10 K1=K+1
C
C    ELIMINATION OF A(I,K), I=K+1,...,M
      Z=0.E0
      DO 20 I=K,M
   20    Z=Z+REAL(A(I,K))**2+AIMAG(A(I,K))**2
      B(K)=0.E0
      IF(Z.LE.TOL)GOTO 70
      Z=SQRT(Z)
      B(K)=Z
      W=CABS(A(K,K))
      Q=(1.E0,0.E0)
      IF(W.NE.0.E0)Q=A(K,K)/W
      A(K,K)=Q*(Z+W)
      IF(K.EQ.NP)GOTO 70
      DO 50 J=K1,NP
         Q=(0.E0,0.E0)
         DO 30 I=K,M
   30       Q=Q+CONJG(A(I,K))*A(I,J)
         Q=Q/(Z*(Z+W))
         DO 40 I=K,M
   40       A(I,J)=A(I,J)-Q*A(I,K)
   50    CONTINUE
C
C    PHASE TRANSFORMATION
      Q=-CONJG(A(K,K))/CABS(A(K,K))
      DO 60 J=K1,NP
   60    A(K,J)=Q*A(K,J)
C
C    ELIMINATION OF A(K,J), J=K+2,...,N
   70 IF(K.EQ.N)GOTO 140
      Z=0.E0
      DO 80 J=K1,N
   80    Z=Z+REAL(A(K,J))**2+AIMAG(A(K,J))**2
      C(K1)=0.E0
      IF(Z.LE.TOL)GOTO 130
      Z=SQRT(Z)
      C(K1)=Z
      W=CABS(A(K,K1))
      Q=(1.E0,0.E0)
      IF(W.NE.0.E0)Q=A(K,K1)/W
      A(K,K1)=Q*(Z+W)
      DO 110 I=K1,M
         Q=(0.E0,0.E0)
         DO 90 J=K1,N
   90       Q=Q+CONJG(A(K,J))*A(I,J)
         Q=Q/(Z*(Z+W))
         DO 100 J=K1,N
  100       A(I,J)=A(I,J)-Q*A(K,J)
  110    CONTINUE
C
C    PHASE TRANSFORMATION
      Q=-CONJG(A(K,K1))/CABS(A(K,K1))
      DO 120 I=K1,M
  120    A(I,K1)=A(I,K1)*Q
  130 K=K1
      GOTO 10
C
C TOLERANCE FOR NEGLIGIBLE ELEMENTS
  140 EPS=0.E0
      DO 150 K=1,N
         S(K)=B(K)
         T(K)=C(K)
  150    EPS=AMAX1(EPS,S(K)+T(K))
      EPS=EPS*ETA
C
C INITIALIZATION OF U AND V
      IF(NU.EQ.0)GOTO 180
      DO 170 J=1,NU
         DO 160 I=1,M
  160       U(I,J)=(0.E0,0.E0)
  170    U(J,J)=(1.E0,0.E0)
  180 IF(NV.EQ.0)GOTO 210
      DO 200 J=1,NV
         DO 190 I=1,N
  190       V(I,J)=(0.E0,0.E0)
  200    V(J,J)=(1.E0,0.E0)
C
C QR DIAGONALIZATION
  210 DO 380 KK=1,N
         K=N1-KK
C
C    TEST FOR SPLIT
  220    DO 230 LL=1,K
            L=K+1-LL
            IF(ABS(T(L)).LE.EPS)GOTO 290
            IF(ABS(S(L-1)).LE.EPS)GOTO 240
  230       CONTINUE
C
C    CANCELLATION OF E(L)
  240    CS=0.E0
         SN=1.E0
         L1=L-1
         DO 280 I=L,K
            F=SN*T(I)
            T(I)=CS*T(I)
            IF(ABS(F).LE.EPS)GOTO 290
            H=S(I)
            W=SQRT(F*F+H*H)
            S(I)=W
            CS=H/W
            SN=-F/W
            IF(NU.EQ.0)GOTO 260
            DO 250 J=1,N
               X=REAL(U(J,L1))
               Y=REAL(U(J,I))
               U(J,L1)=CMPLX(X*CS+Y*SN,0.E0)
  250          U(J,I)=CMPLX(Y*CS-X*SN,0.E0)
  260       IF(NP.EQ.N)GOTO 280
            DO 270 J=N1,NP
               Q=A(L1,J)
               R=A(I,J)
               A(L1,J)=Q*CS+R*SN
  270          A(I,J)=R*CS-Q*SN
  280       CONTINUE
C
C    TEST FOR CONVERGENCE
  290    W=S(K)
         IF(L.EQ.K)GOTO 360
C
C    ORIGIN SHIFT
         X=S(L)
         Y=S(K-1)
         G=T(K-1)
         H=T(K)
         F=((Y-W)*(Y+W)+(G-H)*(G+H))/(2.E0*H*Y)
         G=SQRT(F*F+1.E0)
         IF(F.LT.0.E0)G=-G
         F=((X-W)*(X+W)+(Y/(F+G)-H)*H)/X
C
C    QR STEP
      CS=1.E0
      SN=1.E0
      L1=L+1
      DO 350 I=L1,K
         G=T(I)
         Y=S(I)
         H=SN*G
         G=CS*G
         W=SQRT(H*H+F*F)
         T(I-1)=W
         CS=F/W
         SN=H/W
         F=X*CS+G*SN
         G=G*CS-X*SN
         H=Y*SN
         Y=Y*CS
         IF(NV.EQ.0)GOTO 310
         DO 300 J=1,N
            X=REAL(V(J,I-1))
            W=REAL(V(J,I))
            V(J,I-1)=CMPLX(X*CS+W*SN,0.E0)
  300       V(J,I)=CMPLX(W*CS-X*SN,0.E0)
  310    W=SQRT(H*H+F*F)
         S(I-1)=W
         CS=F/W
         SN=H/W
         F=CS*G+SN*Y
         X=CS*Y-SN*G
         IF(NU.EQ.0)GOTO 330
         DO 320 J=1,N
            Y=REAL(U(J,I-1))
            W=REAL(U(J,I))
            U(J,I-1)=CMPLX(Y*CS+W*SN,0.E0)
  320       U(J,I)=CMPLX(W*CS-Y*SN,0.E0)
  330    IF(N.EQ.NP)GOTO 350
         DO 340 J=N1,NP
            Q=A(I-1,J)
            R=A(I,J)
            A(I-1,J)=Q*CS+R*SN
  340       A(I,J)=R*CS-Q*SN
  350    CONTINUE
         T(L)=0.E0
         T(K)=F
         S(K)=X
         GOTO 220
C
C    CONVERGENCE
  360    IF(W.GE.0.E0)GOTO 380
         S(K)=-W
         IF(NV.EQ.0)GOTO 380
         DO 370 J=1,N
  370       V(J,K)=-V(J,K)
  380    CONTINUE
C
C SORT SINGULAR VALUES
      DO 450 K=1,N
         G=-1.E0
         J=K
         DO 390 I=K,N
            IF(S(I).LE.G)GOTO 390
            G=S(I)
            J=I
  390       CONTINUE
         IF(J.EQ.K)GOTO 450
         S(J)=S(K)
         S(K)=G
         IF(NV.EQ.0)GOTO 410
         DO 400 I=1,N
            Q=V(I,J)
            V(I,J)=V(I,K)
  400       V(I,K)=Q
  410    IF(NU.EQ.0)GOTO 430
         DO 420 I=1,N
            Q=U(I,J)
            U(I,J)=U(I,K)
  420       U(I,K)=Q
  430    IF(N.EQ.NP)GOTO 450
         DO 440 I=N1,NP
            Q=A(J,I)
            A(J,I)=A(K,I)
  440       A(K,I)=Q
  450    CONTINUE
C
C BACK TRANSFORMATION
      IF(NU.EQ.0)GOTO 510
      DO 500 KK=1,N
         K=N1-KK
         IF(B(K).EQ.0.E0)GOTO 500
         Q=-A(K,K)/CABS(A(K,K))
         DO 460 J=1,NU
  460       U(K,J)=Q*U(K,J)
         DO 490 J=1,NU
            Q=(0.E0,0.E0)
            DO 470 I=K,M
  470          Q=Q+CONJG(A(I,K))*U(I,J)
            Q=Q/(CABS(A(K,K))*B(K))
            DO 480 I=K,M
  480          U(I,J)=U(I,J)-Q*A(I,K)
  490       CONTINUE
  500    CONTINUE
  510 IF(NV.EQ.0)GOTO 570
      IF(N.LT.2)GOTO 570
      DO 560 KK=2,N
         K=N1-KK
         K1=K+1
         IF(C(K1).EQ.0.E0)GOTO 560
         Q=-CONJG(A(K,K1))/CABS(A(K,K1))
         DO 520 J=1,NV
  520       V(K1,J)=Q*V(K1,J)
         DO 550 J=1,NV
            Q=(0.E0,0.E0)
            DO 530 I=K1,N
  530          Q=Q+A(K,I)*V(I,J)
            Q=Q/(CABS(A(K,K1))*C(K1))
            DO 540 I=K1,N
  540          V(I,J)=V(I,J)-Q*CONJG(A(K,I))
  550       CONTINUE
  560    CONTINUE
  570 RETURN
      END
```

REMARK ON ALGORITHM 304 [S15]
NORMAL CURVE INTEGRAL [I. D. Hill and S. A. Joyce, *Comm. ACM 10* (June 1967), 374]

ARTHUR G. ADAMS* (Recd. 17 Feb. 1969 and 11 June 1969)
Glaxo Research Ltd., Greenford, Middlesex, England
* Deceased 7 July 1969.

KEY WORDS AND PHRASES: normal curve integral, probability, special functions
CR CATEGORIES: 5.5, 5.12

Algorithm 304 may be made faster by using the continued fraction

$$\frac{1}{x}\left(1+\frac{-1}{x^2+3+}\,\frac{-6}{x^2+7+}\,\frac{-20}{x^2+11+}\,\frac{-42}{x^2+15+}\,\frac{-72}{x^2+19+}\cdots\right)$$

whose convergents are equal to alternate convergents of the continued fraction

$$\frac{1}{x+}\,\frac{1}{x+}\,\frac{2}{x+}\,\frac{3}{x+}\,\frac{4}{x+}\,\frac{5}{x+}\cdots$$

used in the original algorithm when x lies in one of the tails. This requires two extra statements in the iteration loop, which, however, will only be performed about half as many times.

The alteration required to implement this improvement is to replace the 19 lines between

```
if x > (if upper then 2.32 else 3.5) then
```

and

```
q1 := q2;  q2 := s;
```

by

```
begin
  real p1, p2, q1, q2, a1, a2, m;
  a1 := 2.0;  a2 := 0.0;
  n := x2 + 3.0;
  p1 := y;  q1 := x;
  p2 := (n − 1.0) × y;  q2 := n × x;
  m := p1/q1;  t := p2/q2;
  if ¬ upper then
  begin
    m := 1.0 − m;  t := 1.0 − t
  end;
  for n := n + 4.0, n + 4.0 while m ≠ t ∧ s ≠ t do
  begin
    a1 := a1 − 8.0;  a2 := a1 + a2;
    s := a2 × p1 + n × p2;
    p1 := p2;  p2 := s;
    s := a2 × q1 + n × q2;
```

This also incorporates the alterations suggested in [1] below.

Comparison of the two versions using an ICL1903 (37-bit floating-point mantissa), showed that the number of iterations was approximately halved, and that the results differed only to the extent to be expected from rounding error.

The original Algorithm 304 contains in its **comment**, "The value 2.32 may be changed to 1.28 $\cdots$ if the full accuracy of the machine is desired." However a test of the two versions taking arguments in the sequence 2.34 **step** −0.01 showed that the original version ran into overflow at 1.44, and the new version at 1.58, on a machine allowing exponents up to 10^{77}.

Reference

1. Bergson, A. Certification of and Remark on Algorithm 304, Normal Curve Integral. *Comm. ACM 11* (Apr. 1968), 271.

REMARK ON ALGORITHM 345 [C6] AN ALGOL CONVOLUTION PROCEDURE BASED ON THE FAST FOURIER TRANSFORM [Richard C. Singleton, *Comm. ACM 12* (Mar. 1969), 179]

Richard C. Singleton (Recd. 15 May 1969)
Stanford Research Institute, Menlo Park, CA 94025

KEY WORDS AND PHRASES: fast Fourier transform, complex Fourier transform, multivariate Fourier transform, Fourier series, harmonic analysis, spectral analysis, orthogonal polynomials, orthogonal transformation, convolution, autocovariance, autocorrelation, cross-correlation, digital filtering, permutation

CR CATEGORIES: 3.15, 3.83, 5.12, 5.14

On page 180, column 2, the 3rd and 2nd lines from the end of procedure *CONVOLUTION* must be interchanged, i.e. the final four lines should read:

```
begin C[n−j] := scale × (C[j] − D[j]);
  C[j] := scale × (C[j] + D[j])
end
end CONVOLUTION;
```

The procedures included in Algorithm 345 were punched from the printed page and tested on the CDC 6400 Algol compiler. After making the one correction the test results agreed with those obtained earlier with this compiler.

Algorithms Policy • Revised September, 1969 (Includes ALGOL, FORTRAN, and PL/I)

A contribution to the Algorithms department should be in the form of an algorithm, a certification, or a remark. Contributions should be sent in duplicate to the editor, typewritten double-spaced. Authors should carefully follow the style of this department paying especial attention to the indentations and to the completeness of references.

An algorithm must normally be written in the ALGOL 60 Reference Language [*Comm. ACM 6* (Jan. 1963), 1–17] or in ASA Standard FORTRAN or Basic FORTRAN [*Comm. ACM 7* (Oct. 1964), 590–625]. Consideration will be given to algorithms written in other languages provided the language has been fully documented in the open literature and provided the author presents convincing arguments that his algorithm is best described in the chosen language and cannot be adequately described in either ALGOL 60 or FORTRAN. For example, an algorithm may be published in PL/I. Until such time as a standard language definition is approved, the language acceptable to any PL/I translator in common use will suffice.

An algorithm written in ALGOL 60 normally consists of a commented procedure declaration. It should be typewritten double-spaced in capital and lowercase letters. Material to appear in **boldface** type should be underlined in black. Blue underlining may be used to indicate italic type, but this is usually best left to the editor.

An algorithm written in FORTRAN normally consists of a commented subprogram. It should be typewritten double-spaced in the form normally used for FORTRAN, or it should be in the form of a listing of a FORTRAN card together with a copy of the card deck.

An algorithm written in PL/I normally consists of a commented procedure declaration. It should be typewritten double-spaced in capital and lowercase letters. Keywords (which will appear in lowercase boldface type) should be underlined in black and should not be abbreviated. Blue underlining may be used to indicate italic type, but this is usually best left to the editor. In order to increase the readability of PL/I programs, the Algorithms department suggests that the following conventions be observed. Variables should all be declared. Default determination of base and scale should be avoided, as should all contextual declarations. Identifiers should be mnemonic; the use of keywords as identifiers should be avoided. Excessive use of **go to** statements should be avoided. A standard amount of indentation (say three spaces) should be used throughout the program as follows: (1) each new statement should begin a new line; (2) labels should appear on a separate line and be "outdented" from the current program position; (3) if a statement extends beyond one line, the continuation on the next line should be indented; (4) the statements within a procedure block, a begin block, or a do group should be indented to the right of the keyword **procedure**, **begin**, or **do**. The matching **end** should explicitly appear directly beneath the beginning keyword.

Each algorithm must be accompanied by a complete driver program in its language which generates test data, calls the procedure, and produces test answers. Moreover, selected previously obtained test answers should be given in comments in either the driver program or the algorithm. The driver program may be published with the algorithm if it would be of major assistance to a user.

For ALGOL 60 programs, input and output should be achieved by procedure statements, using any of the following eleven procedures (whose body is not specified in ALGOL) [See "Report on Input-Output Procedures for ALGOL 60," *Comm. ACM 7* (Oct. 1964), 628–630]:

insymbol	*inreal*	*outarray*	*ininteger*
outsymbol	*outreal*	*outboolean*	*outinteger*
length	*inarray*	*outstring*	

If only one channel is used by the program for output, it should be designated by 1, and similarly a single input channel should be designated by 2. Examples:

```
outstring (1, 'x=');  outreal (1,x);
for i := 1 step 1 until n do outreal (1,A[i]);
ininteger (2, digit [17]):
```

For FORTRAN programs, input and output should be achieved as described in the ASA preliminary report on FORTRAN and Basic FORTRAN. For PL/I programs, input and output should be achieved by means of the commonly used input/output statements.

It is intended that each published algorithm be well organized, clearly commented, syntactically correct, and a substantial contribution to the literature of algorithms. It is necessary but not sufficient that a published algorithm operate on some machine and give correct answers; it must also communicate a method to the reader in a clear and unambiguous manner. All contributions will be refereed both by human beings and by an appropriate compiler. Authors should pay considerable attention to the correctness of their programs, since referees cannot be expected to debug them.

Certifications and remarks should add new information to that already published. Readers are especially encouraged to test and certify previously uncertified algorithms. Rewritten versions of previously published algorithms will be refereed as new contributions and should not be embedded in certifications or remarks.

Galley proofs will be sent to authors; obviously rapid and careful proofreading is of paramount importance.

Although each algorithm has been tested by its author, no liability is assumed by the contributor, the editor, or the Association for Computing Machinery in connection therewith.

The reproduction of algorithms appearing in this department is explicitly permitted without any charge. When reproduction is for publication purposes, reference must be made to the algorithm author and to the issue of *Communications*.—J. G. Herriot