Collected Algorithms from ACM

Volume II
Algorithms 221–492

A collection of Algorithms 221–492 including Certifications, Remarks, and Translations from the Algorithms Department of Communications of the ACM, 1964–1974.

acm

1980

A Publication of the Association for Computing Machinery, Inc.
1133 Avenue of the Americas
New York, New York 10036

Price: ACM members $40, others $55. This price includes Algorithms 221–492 in this volume, a looseleaf compilation of Algorithms 493 ff and two looseleaf binders, and a one year free subscription of quarterly supplements to the Collected Algorithms. Prices subject to change without notice. For latest prices refer to the current ACM Publications Catalog available free of charge from ACM Order Department, P.O. Box 64145, Baltimore, MD 21264.

ISBN: 0-89791-026-5

Preface

The Algorithms department of Communications of the ACM (CACM) was established in February 1960, with J. H. Wegstein as editor, for the purpose of publishing algorithms, consisting of procedures and programs, in the Algol language. In 1975 the publication of ACM algorithms material was transferred to ACM Transactions on Mathematical Software (TOMS). A wide variety of algorithms have been published and many of them have been used heavily—either in original form or as translated into other languages. Recognizing the general acceptance of the algorithm material published in CACM and TOMS, the Association for Computing Machinery (ACM) has collected and reprinted the algorithms to make them more readily accessible and more serviceable to a larger group of users.

This collection contains Algorithms 221–492; these appeared in the Algorithms department of CACM from 1964–1974.

Algorithms 221–492 were originally published as received—without any refereeing whatever. Many of these have since been certified and/or corrected by their authors or by other contributors.

To facilitate the updating and to make this volume convenient to use, an understanding of the page numbering scheme for the algorithms is helpful. The page designation is in a three-part format: the left part is the algorithm number; the middle part is the page number within the algorithm (the first page of each algorithm is P1); and the right part is the number of the revision of that page. All sheets in the original, or first, insertion of an algorithm have "0" for the right part. The first revision of a page will have a page number having the left and middle parts identical with those on the page to be replaced, but the right part will be "R1" instead of "0." The second revision of the same page would read R2, and so on. For example, 123-P2-R1 would mean the first revision of page 2 of Algorithm 123.

Information on submitting algorithms for publication may be found in the introductory section located in the front of the current loose-leaf collection. Included in this material is a cumulative index to all the algorithms published since 1960 as well as the ACM Algorithms Policy, which guides the publication of all algorithms submitted to ACM.

Webb Miller
ACM Algorithms Editor
Department of Mathematics
University of California, Santa Barbara
Santa Barbara, CA 93106

ALGORITHM 221
GAMMA FUNCTION

WALTER GAUTSCHI (Recd 10 Aug. 63)
Oak Ridge National Laboratory,* Oak Ridge, Tenn.
* Now at Purdue University, Lafayette, Ind.

```
real procedure gamma (z);   value z;   real z;
comment  This is an auxiliary procedure which evaluates Γ(z)
  for 0 < z ≦ 3 to 10 significant digits. It is based on a polynomial
  approximation given in H. Werner and R. Collinge, Math.
  Comput. 15 (1961), 195–197. This procedure must be replaced
  by a more accurate one if more than 10 significant digits are de-
  sired in Algorithm 222 below. Approximations to the gamma
  function, accurate up to 18 significant digits, may be found in
  the paper quoted above;
begin
  integer k;   real p, t;   array A[0:10];
  A[0] := 1.0;   A[1] := .4227843370;   A[2] := .4118402518;
  A[3] := .0815782188;   A[4] := .0742379076;
  A[5] := − .0002109075;   A[6] := .0109736958;
  A[7] := − .0024667480; A[8] := .0015397681;
  A[9] := −.0003442342;   A[10] := .0000677106;
  t := if z ≦ 1 then z else if z ≦ 2 then z−1 else z−2;
    p := A[10];
  for k := 9 step −1 until 0 do p := t × p + A[k];
  gamma := if z ≦ 1 then p/(z × (z +1)) else if z ≦ 2 then
    p/z else p
end gamma
```

CERTIFICATION OF ALGORITHM 221 [S14]
GAMMA FUNCTION [Walter Gautschi, *Comm. ACM 7* (Mar. 1964), 143]

VAN K. MCCOMBS (Recd. 10 Apr. 1964 and 1 Jun. 1964)
General Electric Co., Huntsville, Ala.

The algorithm was translated into FORTRAN IV for the IBM 7094. Computations were performed in double precision to take advantage of the ten significant digits given by the polynomial coefficients. The function $\Gamma(z)$ was evaluated for the range $0 < z \leqq 10$ with an increment of 0.1, and the results were checked with the values published in Table of the Gamma Function for Complex Arguments, *NBS Applied Mathematics Series 34* (1954). The algorithm gave ten-digit accuracy for the range indicated.

REMARKS ON:
ALGORITHM 34 [S14]
GAMMA FUNCTION
[M. F. Lipp, *Comm. ACM 4* (Feb. 1961), 106]
ALGORITHM 54 [S14]
GAMMA FUNCTION FOR RANGE 1 TO 2
[John R. Herndon, *Comm. ACM 4* (Apr. 1961), 180]
ALGORITHM 80 [S14]
RECIPROCAL GAMMA FUNCTION OF REAL ARGUMENT
[William Holsten, *Comm. ACM 5* (Mar. 1962), 166]
ALGORITHM 221 [S14]
GAMMA FUNCTION
[Walter Gautschi, *Comm. ACM 7* (Mar. 1964), 143]
ALGORITHM 291 [S14]
LOGARITHM OF GAMMA FUNCTION
[M. C. Pike and I. D. Hill, *Comm. ACM 9* (Sept. 1966), 684]

M. C. PIKE AND I. D. HILL (Recd. 12 Jan. 1966)
Medical Research Council's Statistical Research Unit,
University College Hospital Medical School,
London, England

Algorithms 34 and 54 both use the same Hastings approximation, accurate to about 7 decimal places. Of these two, Algorithm 54 is to be preferred on grounds of speed.

Algorithm 80 has the following errors:

(1) *RGAM* should be in the parameter list of *RGR*.

(2) The lines

```
  if x = 0 then begin RGR := 0;   go to EXIT end
```

and

```
  if x = 1 then begin RGR := 1;   go to EXIT end
```

should each be followed either by a semicolon or preferably by an **else**.

(3) The lines

```
  if x = 1 then begin RGR := 1/y;   go to EXIT end
```

and

```
  if x < − 1 then begin y := y × x;   go to CC end
```

should each be followed by a semicolon.

(4) The lines

```
  BB:  if x = −1 then begin RGR := 0;   go to EXIT end
```

and

```
  if x > −1 then begin RGR := RGAM(x);   go to EXIT end
```

should be separated either by **else** or by a semicolon and this second line needs terminating with a semicolon.

(5) The declarations of **integer** *i* and **real array** *B*[0:13] in *RGAM* are in the wrong place; they should come immediately after

```
  begin real z;
```

With these modifications (and the replacement of the array *B* in *RGAM* by the obvious nested multiplication) Algorithm 80 ran successfully on the ICT Atlas computer with the ICT Atlas ALGOL compiler and gave answers correct to 10 significant digits.

Algorithms 80, 221 and 291 all work to an accuracy of about 10 decimal places and to evaluate the gamma function it is therefore on grounds of speed that a choice should be made between them. Algorithms 80 and 221 take virtually the same amount of computing time, being twice as fast as 291 at $x = 1$, but this advantage decreases steadily with increasing x so that at $x = 7$ the speeds are about equal and then from this point on 291 is faster—taking only about a third of the time at $x = 25$ and about a tenth of the time at $x = 78$. These timings include taking the exponential of *loggamma.*

For many applications a ratio of gamma functions is required (e.g. binomial coefficients, incomplete beta function ratio) and the use of algorithm 291 allows such a ratio to be calculated for much larger arguments without overflow difficulties.

ALGORITHM 222
INCOMPLETE BETA FUNCTION RATIOS

WALTER GAUTSCHI (Recd 10 Aug. 63)
Oak Ridge National Laboratory,* Oak Ridge, Tenn.
 * Now at Purdue University, Lafayette Ind.

```
comment  Let $B_x(p, q) = \int_0^x t^{p-1} (1 - t)^{q-1} dt$ $(p > 0, q > 0,$
  $0 \leqq x \leqq 1)$ denote the incomplete beta function. The objective
  of this algorithm is to evaluate a sequence of ratios $I_x(p, q) =$
  $B_x(p, q)/B_1(p, q)$, as one of the parameters $p, q$ varies in steps of
  unity while the other remains fixed. The procedure incomplete beta
  q fixed evaluates $I_x(p + n, q)$ for $n = 0, 1, \cdots, nmax$, assuming
  $0 < p \leqq 1, q > 0$, whereas the procedure incomplete
  beta p fixed evaluates $I_x(p, q + n)$ for $n = 0, 1, \cdots, nmax$, assuming
  $0 < q \leqq 1, p > 0$. The number $d$ of significant digits
  desired can be specified, but is only guaranteed when $x \leqq \frac{1}{2}$.
  When $x > \frac{1}{2}$, the complements $1 - I_x$ will be accurate to $d$ significant
  digits. In the region $0 < p \leqq 1, 0 < q \leqq 2$, $I_x(p, q)$ is
  calculated from a power series expansion. The sequences $f(n)$
  $= I_x(p + n, q)$ and $g(n) = I_x(p, q + n)$, including initial values,
  are generated recursively by means of the recurrence relations
  $f(n + 1) - (1 + (n + p + q - 1) x/(n + p)) f(n) + ((n + p + q$
  $- 1) x/(n + p)) f(n - 1) = 0, g(n + 1) - (1 + (n+p+q - 1)\cdot$
  $(1 - x)/(n + q)) g(n) + ((n + p + q - 1) (1 - x)/(n + q))$
  $\cdot g(n - 1) = 0$. Since the former is mildly unstable, a variant
  of the backward recurrence algorithm of J. C. P. Miller is applied
  to it. A global real procedure gamma (z) must be available
  (see Algorithm 221);
real procedure Isubx p and q small (x, p, q, d);
  value x, p, q, d;
  integer d;  real x, p, q;
comment  This procedure evaluates $I_x(p, q)$ to $d$ significant
  digits when $0 < p \leqq 1$ and $0 < q \leqq 2$. It first calculates
  $B_x(p, q)$ by a series expansion in powers of $x$, and then divides
  the result by $B_1(p, q) = \Gamma(p)\Gamma(q)/\Gamma(p + q)$, using the real
  procedure gamma;
begin integer k;  real epsilon, u, v, s;
  epsilon := $.5 \times 10 \uparrow (-d)$;
  $u := x \uparrow p$;  $s := u/p$;  $k := 0$;
L0: $u := (k-q+1) \times (k+p) \times x \times u/(k+1)$;
  $v := u/(k+p+1)$;  $s := s + v$;  $k := k + 1$;
  if abs(v)/s > epsilon then go to L0;
  Isubx p and q small := $s \times$ gamma$(p+q)$/(gamma$(p) \times$
    gamma$(q)$)
end Isubx p and q small;
procedure forward (x, p, q, I0, I1, nmax, I);
  value x, p, q, I0, I1, nmax;
  integer nmax;  real x, p, q, I0, I1;  array I;
comment  Given $I0 = I_x(p, q)$, $I1 = I_x(p, q+1)$, this procedure
  generates $I_x(p, q+n)$ for $n = 0, 1, 2, \cdots, nmax$, and stores the
  results in the array I;
begin integer n;
  I[0] := I0;  if nmax > 0 then I[1] := I1;
  for n := 1 step 1 until nmax - 1 do
    $I[n+1] := (1+(n+p+q-1) \times (1-x)/(n+q)) \times I[n]$
      $- (n+p+q-1) \times (1-x) \times I[n-1]/(n+q)$
end forward;
procedure backward (x, p, q, I0, nmax, d, I);
  value x, p, q, I0, nmax, d;
  integer nmax, d;  real x, p, q, I0;  array I;
```

```
comment  Given $I0 = I_x(p, q)$, this procedure generates
  $I_x(p+n, q)$ for $n = 0, 1, 2, \cdots, nmax$ to $d$ significant digits,
  using a variant of J. C. P. Miller's backward recurrence algorithm.
  The results are stored in the array I;
begin
  integer n, nu, m; real epsilon, r; array Iapprox,
  Rr [0:nmax];
  I[0] := I0;  if nmax > 0 then
  begin
    epsilon := $.5 \times 10 \uparrow (-d)$;
    for n := 1 step 1 until nmax do Iapprox[n] := 0;
    nu := $2 \times nmax + 5$;
L1:  n := nu;  r := 0;
L2:  $r := (n+p+q-1) \times x/(n+p+(n+p+q-1) \times x$
      $- (n+p) \times r)$;
    if $n \leqq nmax$ then Rr[n-1] := r;  n := n - 1;
      if $n \geqq 1$ then go to L2;
    for n := 0 step 1 until nmax - 1 do
      $I[n+1] := Rr[n] \times I[n]$;
    for n := 1 step 1 until nmax do
      if abs ((I[n] - Iapprox[n])/I[n]) > epsilon then
      begin
        for m := 1 step 1 until nmax do Iapprox[m] := I[m];
        nu := nu + 5;  go to L1
      end
  end
end backward;
procedure Isubx qfixed(x, p, q, nmax, d, I);  value x, p, q, nmax, d;
  integer nmax, d;  real x, p, q;  array I;
comment  This procedure generates $I_x(p+n,q)$, $0 < p \leqq 1$, for
  $n=0, 1, \cdots, nmax$ to $d$ significant digits, using the procedure
  backward. In order to calculate the initial value $I0=I_x(p,q)$, it
  first reduces $q$ modulo 1 to $q_0$, where $0 < q_0 \leqq 1$, then obtains
  $I_x(p,q_0)$ and $I_x(p,q_0+1)$ by the real procedure Isubx p and q small,
  and finally uses these as initial values for the procedure forward,
  which connects with $I_x(p,q)$ by the recurrence in $q$;
begin integer m, mmax;  real s, q0, Iq0, Iq1;
  m := entier(q);  s := q - m;
  q0 := if s > 0 then s else s + 1;
  mmax := if s > 0 then m else m - 1;
  Iq0 := Isubx p and q small(x, p, q0, d);
  if mmax > 0 then Iq1 := Isubx p and q small(x, p, q0+1, d);
  begin array Iq[0:mmax];
    forward (x, p, q0, Iq0, Iq1, mmax, Iq);
    backward(x, p, q, Iq[mmax], nmax, d, I)
  end
end Isubx q fixed;
procedure Isubx pfixed(x, p, q, nmax, d, I);  value x, p, q, nmax, d;
  integer nmax, d;  real x, p, q;  array I;
comment  This procedure generates $I_x(p, q+n)$, $0 < q \leqq 1$, for
  $n=0, 1, \cdots, nmax$ to $d$ significant digits, using the procedure
  forward. The initial values $I0=I_x(p,q)$, $I1=I_x(p,q+1)$ are obtained
  by twice applying the procedure backward. The initial
  values for the latter are provided by the real procedure Isubx p
  and q small;
begin integer m, mmax;  real s, p0, I0, I1, Iq0, Iq1;
  m := entier(p);  s := p - m;
  p0 := if s > 0 then s else s + 1;
  mmax := if s > 0 then m else m - 1;
  I0 := Isubx p and q small(x, p0, q, d);
  I1 := Isubx p and q small(x, p0, q+1, d);
  begin array Ip[0:mmax];
    backward(x, p0, q, I0, mmax, d, Ip); Iq0 := Ip[mmax];
    backward(x, p0, q+1, I1, mmax, d, Ip); Iq1 := Ip[mmax]
  end;
  forward(x, p, q, Iq0, Iq1, nmax, I)
end Isubx p fixed;
procedure incomplete beta q fixed(x, p, q, nmax, d, I);
  value x, p, q, nmax, d;
  integer nmax, d; real x, p, q; array I;
comment This procedure obtains the final results I_x(p+n,q),
  0 < p ≦ 1, n=0, 1, ··· , nmax, directly from the procedure
  Isubx q fixed, if x ≦ 1/2, or via the relation I_x(p+n,q) =
  1 − I_{1−x}(q,p+n) and the procedure Isubx p fixed, if x > 1/2. The
  indicated substitution in the case x > 1/2 is made to ensure fast
  convergence of both the power series used in the real procedure
  Isubx p and q small, and the backward recurrence algorithm used
  in the procedure backward. If the parameters x, p, q, nmax are
  not in the intended range, control is transferred to a nonlocal
  label called alarm;
begin integer n;
  if x < 0∨x > 1∨p ≦ 0∨ p > 1∨q ≦ 0∨ nmax < 0 then go to
  alarm;
  if x=0∨x=1 then for n := 0 step 1 until nmax do I[n] := x else
  begin
    if x ≦ .5 then Isubx q fixed(x, p, q, nmax, d, I) else
    begin
      Isubx p fixed(1−x, q, p, nmax, d, I);
    for n := 0 step 1 until nmax do I[n] := 1 − I[n]
    end
  end
end incomplete beta q fixed;
procedure incomplete beta p fixed(x, p, q, nmax, d, I);
  value x, p, q, nmax, d; integer nmax, d; real x, p, q; array I;
comment This procedure, the exact analogue to the procedure
  incomplete beta q fixed, generates the final results I_x(p,q+n),
  0 < q ≦ 1, n=0, 1, ··· , nmax. For the setup of the procedure,
  see the comment in incomplete beta q fixed;
begin integer n;
  if x < 0 ∨ x > 1 ∨ q ≦ 0 ∨ q > 1 ∨p ≦ 0∨ nmax < 0 then go to
  alarm;
  if x=0∨x=1 then for n := 0 step 1 until nmax do I[n] := x else
  begin
    if x ≦ .5 then Isubx p fixed(x, p, q, nmax, d, I) else
    begin
      Isubx q fixed(1−x, q, p, nmax, d, I);
      for n := 0 step 1 until nmax do I[n] := 1 − I[n]
    end
  end
end incomplete beta p fixed
```

REFERENCE: WALTER GAUTSCHI, Recursive computation of special functions. U. of Michigan, Eng. Summer Conf., Numerical Analysis, 1963.

CERTIFICATION OF ALGORITHM 222 INCOMPLETE BETA FUNCTION RATIOS [Walter Gautschi, *Comm. ACM 7* (March 1964), 143]

WALTER GAUTSCHI (Recd 2 Jan. 1964)
Purdue Univ., Lafayette, Ind.

```
begin integer n; array I1, I2, I3[0: 10];
comment This program calls the procedures Incomplete beta q
  fixed and Incomplete beta p fixed to calculate test values of
  I.4(.5+n, 7), I.4(5, 1+n), I.8(5, 1+n) for n = 0(1)10 to 6
  significant digits. The following results were obtained on the
  CDC 1604-A computer, using the Oak Ridge ALGOL compiler:
```

n	$I_{.4}(.5 + n, 7)$	$I_{.4}(5, 1 + n)$	$I_{.8}(5, 1 + n)$
0	.99143646185	.010239999997	.32768000004
1	.93951533330	.040959999972	.65536000000
2	.83567307612	.096255999927	.85196799999
3	.69444760641	.17367039987	.94371839999
4	.54111709640	.26656767980	.98041856000
5	.39800862042	.36689674211	.99363061758
6	.27831789503	.46722580441	.99803463679
7	.18624810627	.56182177742	.99941875711
8	.11995785836	.64695815314	.99983399319
9	.074724512738	.72074301208	.99995395031
10	.045203802963	.78272229360	.99998753828

```
  All results are in agreement with those tabulated in [1];
  Incomplete beta q fixed (.4, .5, 7, 10, 6, I1);
  Incomplete beta p fixed (.4, 5, 1, 10, 6, I2);
  Incomplete beta p fixed (.8, 5, 1, 10, 6, I3);
  for n := 0 step 1 until 10 do
    write ( I1[n], I2[n], I3[n])
  end Driver incomplete beta function ratios
```

In the original publication of the algorithm, the following correction of a printer's error is needed in the real procedure *Isubx p and q small*. The statement labelled $L0$ should read as follows:

$$u := (k - q + 1) \times x \times u/(k + 1);$$

[1] PEARSON, K. *Tables of the Incomplete Beta-Function*. Cambridge University Press, London, 1934.

ALGORITHM 223
PRIME TWINS

M. SHIMRAT (Recd 7 June 1963; in final form 2 Jan. 1964)
University of Alberta, Calgary, Alberta, Canada

```
procedure Prime Twins (t, Twin1, Twin2, Storage, Act):
  value Storage;  integer t, Twin1, Twin2, Storage;
  procedure Act;
comment This procedure will generate successive "prime
  twins," i.e. pairs of primes Twin1, Twin2 which differ by 2.
  Storage is the maximum number of primes that can be stored.
  Act is any procedure for recording, examining, or utilizing each
  pair of twins as it is generated.  t is a serial number for the
  twins. P[Storage] ↑ 2 is the last number examined;
begin integer array P[1: Storage]; integer j, m, previous,
    current;
  comment P[j] is the jth prime;
  P[1] := 2;  P[2] := 3;  j := 2;  previous := 3;  t := 0;
  for current := 5 step 2 until P[j] × P[j] do
  begin m := 1;  for m := m + 1 while P[m] × P[m] ≦ cur-
    rent do
    if current = (current ÷ P[m]) × P[m] then go to NoPrime;
    comment If this point is reached, current is not divisible by
      any prime up to sqrt(current) and so is a prime. We now
      record the new prime, if storage permits, then check if it
      is the second of twins;
    if j < Storage then
      begin j := j + 1;  P[j] := current
      end;
    if current = previous + 2 then
      begin t := t + 1;  Twin1 := previous;  Twin2 := current;
        Act (t, Twin1, Twin2)
      end;
    previous := current;
    NoPrime:
  end;
end procedure Prime Twins
```

ALGORITHM 224
EVALUATION OF DETERMINANT

Leo J. Rotenberg

(Recd 7 Oct. 1963; in final form 20 Dec. 1963)

Box 2400, 362 Memorial Dr., Cambridge, Mass.

```
real procedure determinant (a, n);
  value n;  real array a;  integer n;
comment This procedure evaluates a determinant by triangulari-
  zation. The matrix supplied by the calling procedure is modified
  by this program. This procedure is an extensive revision and
  correction of Algorithm 41;
begin real product, factor, temp, div, piv, abpiv, maxpiv;
  integer ssign, i, j, r, imax;
  ssign := 1;  product := 1.0;
  for r := 1 step 1 until n−1 do
  begin maxpiv := 0.0;
    for i := r step 1 until n do
    begin piv := a[i, r];
      abpiv := abs(piv);
      if abpiv > maxpiv then
      begin maxpiv := abpiv;
        div := piv;
        imax := i
      end
    end;
    if maxpiv ≠ 0.0 then
    begin if imax = i then go to resume else
      begin for j := r step 1 until n do
        begin temp := a[imax, j];
          a[imax, j] := a[r, j];
          a[r, j] := temp
        end;
        ssign := − ssign;
        go to resume
      end
    end;
    determinant := 0.0;
    go to return;
resume: for i := r+1 step 1 until n do
    begin factor := a[i, r]/div;
      for j := r+1 step 1 until n do
      a[i, j] := a[i, j] − factor × a[r, j]
    end
  end;
  for i := 1 step 1 until n do
  product := product × a[i, i];
  comment Exponent overflow or underflow will most likely
    occur here if at all. For large or small determinants the user
    is cautioned to replace this with a call to a machine-language
    product routine which will handle extremely large or small
    real numbers;
  determinant := ssign × product;
return:
end
```

CERTIFICATION OF ALGORITHM 224 [F3]
EVALUATION OF DETERMINANT

[Leo J. Rotenberg, *Comm. ACM 7* (Apr. 1964), 243]

Vic Hasselblad and Jeff Rulifson (Recd. 17 July 1964)

Computer Center, U. of Washington, Seattle, Wash.

The "Evaluation of Determinant" program was tested on an Algol 60 compiler for an IBM 709 (Share distribution #3032). When the 10th line on page 244 was changed to read:

begin if *imax* = *r* **then go to** *resume* **else**

correct results were obtained. It was tested up through 4 × 4 matrices.

ALGORITHM 225
GAMMA FUNCTION WITH CONTROLLED ACCURACY

S. J. Cyvin and B. N. Cyvin (Recd. 25 Oct. 1963)
Technical University of Norway, Trondheim, Norway

```
real procedure GAMMA (m, x); value m, x; integer m;
  real x;
comment Γ(x) is calculated with at least m significant figures
  (disregarding the machine's roundoff). The range of x is reduced
  by recursion to 5 ≦ x ≦ 6, for which Γ(x) is found (with m−2
  significant decimals) according to

      $\Gamma(x) = \int_0^T t^{x-1}e^{-t}\,dt + \int_T^\infty t^{x-1}e^{-t}\,dt.$

  Simpson's formula is applied to the former integral, which is
  divided into 2n parts. Here n, as well as T, are chosen auto-
  matically to give a result with the required accuracy. For x
  near zero or a negative integer, Γ(x) is put equal to a large value,
  $10^{60}$. The procedure is slower than other algorithms for Γ(x)
  [see Nos. 31, 34, 54, 80], but has the advantage of controlled
  accuracy;
begin integer i,n,f,T; real y,h,S;
   h := 1;  y := x;
A: if abs(y) < 10−60 then begin GAMMA := 1060; go to
      E end else
   if y > 6 then begin y := y−1;  h := h×y;  go to A end else
   if y < 5 then begin h := h/y;  y := y+1;  go to A end else
   begin real a;
      T := 20;
U:    if (T↑5 + 4×T↑4 + 16×T↑3 + 48×T↑2 + 96×T +
        96)×exp(−T)>.25×10↑(2−m) then begin T := T+5;
        go to U end;
      n := 1 + entier(sqrt(sqrt(T↑5×10↑(m−2)/30)));
      S := 0;  f := 4;
      for i := 1 step 1 until 2×n do
      begin
        a := .5×i×T/n;  S := S + f×a↑(y−1)×exp(−a);
        f := if i = 2×n−1 then 1 else if f = 4 then 2 else 4
      end
   end;
   GAMMA := (S×T/(6×n) + (.5×T↑5 + 3×T↑4 + 12×T↑3
      + 36×T↑2 + 72×T + 72)×exp(−T))×h;
E:
end of GAMMA
```

CERTIFICATION OF ALGORITHM 225 [S14]
GAMMA FUNCTION WITH CONTROLLED ACCURACY [S. J. Cyvin and B. N. Cyvin, *Comm. ACM* 7 (May 1964), 295]

T. A. Bray (Recd. 25 May 1964 and 18 Jun. 1964)
Boeing Scientific Research Laboratories, Seattle, Wash.

Algorithm 225 was coded in Fortran II and run on the IBM 1620. No corrections were necessary and the following results were obtained for $m = 2$:

x	*GAMMA* (m, x)	x	*GAMMA* (m, x)
.01	99.44362100	3.50	3.32349920
.05	19.47214000	4.00	6.00067550
.10	9.51444650	4.50	11.63224700
.50	1.77253280	5.00	24.00270200
1.00	1.00011250	5.50	52.34511500
1.50	.88626644	10.00	$0.36286974_{10}\ 6$
2.00	1.00011250	25.00	$0.62043066_{10}\ 24$
2.50	1.32939960	50.00	$0.60826434_{10}\ 63$
3.00	2.00022510		

These results are correct to at least two significant digits. The following results and times were obtained for $x = 0.5$:

m	*GAMMA* (m, x)	*TIME* (*in seconds*)
2	1.77253280	58
3	1.77254230	105
4	1.77245370	200
5	1.77244430	405
6	1.77244020	885

The correct result is 1.7724539. Note that the accuracy decreased as m increased and the result for $m = 6$ is incorrect in the sixth significant digit.

This algorithm is extremely slow as compared to some others available. Algorithm 31 was used for the above set of arguments and gave seven-digit accuracy in 250 milliseconds per argument.

ALGORITHM 226
NORMAL DISTRIBUTION FUNCTION

S. J. Cyvin (Recd. 15 Oct. 1963)
Technical University of Norway, Trondheim, Norway

```
real procedure Fi(m,x);  value m,x;  integer m; real x;
comment  $\Phi(x) = (1/\sqrt{2\pi})\int_{-\infty}^{x} \exp(-\frac{1}{2}u^2)\,du$ is found by com-
  puting $\int_0^x \exp(-\frac{1}{2}u^2)\,du$ with aid of Simpson's formula. The
  latter integral is divided into $2n$ parts, where $n$ automatically
  is adjusted to give a result with at least $m$ significant decimals
  (disregarding the machine's roundoff). The error function is
  obtainable as $\mathrm{erf}(x) = 2\Phi(x/\sqrt{2}) - 1$. The practical use of the
  present method is not restricted to small or large ranges of $x$.
  Probably the method has some advantages compared to Algo-
  rithms 123, 180, and 209;
begin integer i,n,f;  real b,S;
  b := abs(x);
  n := 1 + entier(sqrt(sqrt(b ↑ 5×10 ↑ m/
     (480×sqrt(2×3.14159265)))));
  if n < 4 then n := 4;  S := 1;  f := 4;
  for i := 1 step 1 until 2×n do
  begin
    S := S + f × exp(−(i×b/n) ↑ 2/8);
    f := if i = 2×n−1 then 1 else if f=4 then 2 else 4
  end;
  Fi := .5 + sign(x)×S×b/(6×n×sqrt(2×3.14159265))
end Fi
```

REMARKS ON:

ALGORITHM 123 [S15]
REAL ERROR FUNCTION, ERF(x)
[Martin Crawford and Robert Techo *Comm. ACM 5* (Sept. 1962), 483]

ALGORITHM 180 [S15]
ERROR FUNCTION—LARGE X
[Henry C. Thacher Jr. *Comm. ACM 6* (June 1963), 314]

ALGORITHM 181 [S15]
COMPLEMENTARY ERROR FUNCTION—LARGE X
[Henry C. Thacher Jr. *Comm. ACM 6* (June 1963), 315]

ALGORITHM 209 [S15]
GAUSS
[D. Ibbetson. *Comm. ACM 6* (Oct. 1963), 616]

ALGORITHM 226 [S15]
NORMAL DISTRIBUTION FUNCTION
[S. J. Cyvin. *Comm. ACM 7* (May 1964), 295]

ALGORITHM 272 [S15]
PROCEDURE FOR THE NORMAL DISTRIBUTION FUNCTIONS
[M. D. MacLaren. *Comm. ACM 8* (Dec. 1965), 789]

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

I. D. Hill and S. A. Joyce (Recd. 21 Nov. 1966)
Medical Research Council,
Statistical Research Unit, 115 Gower Street, London
W.C.1., England

These algorithms were tested on the ICT Atlas computer using the Atlas Algol compiler. The following amendments were made and results found:

ALGORITHM 123

(i) **value** x; was inserted.
(ii) $abs(T) \leqslant {}_{10}{-10}$ was changed to $Y - T = Y$
both these amendments being as suggested in [1].
(iii) The labels 1 and 2 were changed to $L1$ and $L2$, the **go to** statements being similarly amended.
(iv) The constant was lengthened to 1.12837916710.
(v) The extra statement $x := 0.707106781187 \times x$ was made the first statement of the algorithm, so as to derive the normal integral instead of the error function.

The results were accurate to 10 decimal places at all points tested except $x = 1.0$ where only 2 decimal accuracy was found, as noted in [2]. There seems to be no simple way of overcoming the difficulty [3], and any search for a method of doing so would hardly be worthwhile, as the algorithm is slower than Algorithm 304 without being any more accurate.

ALGORITHM 180

(i) $T := -0.56418958/x/exp(v)$ was changed to
$T := -0.564189583548 \times exp(-v)/x$. This is faster and also has the advantage, when v is very large, of merely giving 0 as the answer instead of causing overflow.
(ii) The extra statement $x := 0.707106781187 \times x$ was made as in (v) of Algorithm 123.
(iii) **for** $m := m + 1$ was changed to **for** $m := m + 2$. $m+1$ is a misprint, and gives incorrect answers.

The greatest error observed was 2 in the 11th decimal place.

ALGORITHM 181

(i) Similar to (i) of Algorithm 180 (except for the minus sign).
(ii) Similar to (ii) of Algorithm 180.
(iii) m was declared as **real** instead of **integer**, as an alternative to the amendment suggested in [4].

The results were accurate to 9 significant figures for $x < 8$, but to only 8 significant figures for $x = 10$ and $x = 20$.

ALGORITHM 209

No modification was made. The results were accurate to 7 decimal places.

ALGORITHM 226

(i) $10 \uparrow m/(480{\times}sqrt(2{\times}3.14159265))$ was changed to
$10 \uparrow m \times 0.000831129750836$.
(ii) **for** i := 1 **step** 1 **until** 2 × n **do** was changed to

$m := 2 \times n$; **for** $i := 1$ **step** 1 **until** m **do**.

(iii) $-(i\times b/n) \uparrow 2/8$ was changed to $-(i\times b/n) \uparrow 2 \times 0.125$.

(iv) **if** $i = 2 \times n - 1$ was changed to **if** $i = m - 1$

(v) $b/(6\times n\times sqrt(2\times 3.14159265))$ was changed to $b/(15.0397696478\times n)$.

Tests were made with $m = 7$ and $m = 11$ with the following results:

x	*Number of significant figures correct*		*Number of decimal places correct*	
	$m = 7$	$m = 11$	$m = 7$	$m = 11$
−0.5	7	11	7	11
−1.0	7	10	7	10
−1.5	7	10	8	10
−2.0	7	9	8	10
−2.5	6	9	8	11
−3.0	6	7	8	9
−4.0	5	7	10	11
−6.0	2	1	12	10
−8.0	0	0	11	9

Perhaps the comment with this algorithm should have referred to decimal places and not significant figures. To ask for 11 significant figures is stretching the machine's ability to the limit, and where 10 significant figures are correct, this may be regarded as acceptable.

ALGORITHM 272

The constant .99999999 was lengthened to .9999999999.

The accuracy was 8 decimal places at most of the points tested, but was only 5 decimal places at $x = 0.8$.

ALGORITHM 304

No modification was made. The errors in the 11th significant figure were:

$abs(x)$	$x > 0 \equiv upper$	$x > 0 \not\equiv upper$
0.5	1	1
1.0	1	2
1.5	21[a](5)	2
2.0	25[a](0)	4
3.0	0	0
4.0	2	3
6.0	6	0
8.0	14	0
10.0	23	0
20.0	35	0

[a] Due to the subtraction error mentioned in the comment section of the algorithm. Changing the constant 2.32 to 1.28 resulted in the figures shown in brackets.

To test the claim that the algorithm works virtually to the accuracy of the machine, it was translated into double-length instructions of Mercury Autocode and run on the Atlas using the EXCHLF compiler (the constant being lengthened to 0.398942280401432677939946). The results were compared with hand calculations using Table II of [5]. The errors in the 22nd significant figure were:

$abs(x)$	$x > 0 \equiv upper$	$x > 0 \not\equiv upper$
1.0	2	3
2.0	7	1
4.0	2	0
8.0	8	0

Timings. Timings of these algorithms were made in terms of the Atlas "Instruction Count," while evaluating the function 100 times. The figures are not directly applicable to any other computer, but the relative times are likely to be much the same on other machines.

INSTRUCTION COUNT FOR 100 EVALUATIONS

$abs(x)$	Algorithm number							
	123	180	181	209	226 $m = 7$	272	304[a]	304[b]
0.5	58			8	97	24	25	24
1.0	65[c]			8	176	24	29	29
1.5	164	128	127	9	273	25	35	35
2.0	194	78	90	8	387	24	39	39
2.5	252	54	68	10	515	24	131	44
3.0		42	51	9	628	25	97	50
4.0		27	39	9	900[d]	25	67	44
6.0		15	30	6	1400[d]	16	49	23
8.0		9	28	7	2100[d]	18	44	11
10.0		10	25	5	2700[d]	16	38	11
20.0		9	22	5	6500[d]	16	32	11
30.0		9	9	5	10900[d]	16	11	11

[a] Readings refer to $x > 0 \equiv upper$.

[b] Readings refer to $x > 0 \not\equiv upper$.

[c] Time to produce incorrect answer. A count of 120 would fit a smooth curve with surrounding values.

100 times Instruction Count for 1 evaluation.

Opinion. There are advantages in having two algorithms available for normal curve tail areas. One should be very fast and reasonably accurate, the other very accurate and reasonably fast. We conclude that Algorithm 209 is the best for the first requirement, and Algorithm 304 for the second.

Algorithms 180 and 181 are faster than Algorithm 304 and may be preferred for this reason, but the method used shows itself in Algorithm 181 to be not quite as accurate, and the introduction of this method solely for the circumstances in which Algorithm 180 is applicable hardly seems worth while.

Acknowledgment. Thanks are due to Miss I. Allen for her help with the double-length hand calculations.

REFERENCES:

1. THACHER, HENRY C. JR. Certification of Algorithm 123., *Comm. ACM 6* (June 1963), 316.

2. IBBETSON, D. Remark on Algorithm 123. *Comm. ACM 6* (Oct. 1963), 618.

3. BARTON, STEPHEN P., AND WAGNER, JOHN F. Remark on Algorithm 123. *Comm. ACM 7* (Mar. 1964), 145.

4. CLAUSEN, I., AND HANSSON, L. Certification of Algorithm 181. *Comm. ACM 7* (Dec. 1964), 702.

5. SHEPPARD, W. F. *The Probability Integral.* British Association Mathematical Tables VII, Cambridge U. Press, Cambridge, England, 1939.

ALGORITHM 227
CHEBYSHEV POLYNOMIAL COEFFICIENTS

S. J. Cyvin (Recd. 15 Oct. 1963)
Technical University of Norway, Trondheim, Norway

```
procedure Tcheb(n,A); value n; integer n; integer array A;
comment This procedure finds (by recursion) the coefficients
  of $T_n(x)$, rather than the value of the polynomial, which is the
  subject of Algorithms 10 and 36. The (n+2)÷2 nonvanishing
  coefficients are stored in one-dimensional integer array A in
  the following way:

  $T_{2p}(x) = \sum_{i=0}^{p} A[i+1]\, x^{2i}$   (n even),

                    $T_{2p+1}(x) = \sum_{i=0}^{p} A[i+1] x^{2i+1}$   (n odd);
begin integer i,j; integer array B[1:(n+2)÷2]; Boolean
  EVEN;
  A[1] := B[1] := 1; EVEN := n÷2×2 = n; if n > 1 then
  for i := 2 step 1 until (n+2)÷2 do
  for j := i step −1 until 1 do
  begin
    A[j] := if j=i then 2×B[j−1] else if j=1 then −A[1]
    else 2 ×B[j−1] − A[j];
    B[j] := if j=i then 2 × A[i] else 2 × A[j] − B[j]
  end i loop;
  for i := 1 step 1 until (n+2)÷2 do
    A[i] := if EVEN then A[i] else B[i]
end Tcheb
```

ALGORITHM 228
Q-BESSEL FUNCTIONS $\bar{I}_n(t)$

J. M. S. Simões Pereira (Recd. 21 Sept. 63 and 6 Jan. 64)
Gulbenkian Scientific Computing Ctr, Lisboa, Portugal

```
procedure qBesselbar (t,q,n,j,s); integer n, j; real t,q,s;
comment This procedure computes values of any q-Bessel
  function $\bar{I}_n(t)$ for n integer (positive, negative or zero) by the
  use of the expansion $\bar{I}_n(t) = \sum_{k=0}^{\infty} (t^{n+2k}/((q)_k(q)_{n+k}))$ where
  $(q)_n = (1-g)(1-q^2)\cdots(1-q^n)$, $(q)_0 = 1$ and $(1/(q)_{-n})=0$ $(n=1,
  2, \cdots)$. This series is convergent for $t \in (-\infty, +\infty)$ if $|q| > 1$
  and for $|t| < 1$ if $|q| < 1$.  j+1 denotes the number of terms
  (at least 2) retained in the summation and s stands for the sum
  of these first terms. See L. Carlitz, The product of q-Bessel
  functions, Portugaliae Mathematica, vol. 21;
begin integer k,m,p;  real c,u;  m := abs(n);  c := 1;
   if n = 0 then go to A;
   for p := 1 step 1 until m do c := c×(1−q↑p);
   if n < 0 then go to B;
A: u := (t↑n)/c;  s := u;
   for k := 1 step 1 until j do
   begin u := u×(t↑2)/((1−q↑k)×(1−q↑(n+k)));  s :=
     s + u end;
   go to C;
B: u := t ↑ (n+2×m)/c;  s := u;
   for k := m + 1 step 1 until j do
   begin u := u×(t↑2)/((1−q ↑ k)×(1−q ↑ (n+k)));  s :=
     s + u end;
C: end
```

ALGORITHM 229
ELEMENTARY FUNCTIONS BY CONTINUED FRACTIONS

JAMES C. MORELOCK (Recd. 1 Oct. 63 and in final form 24 Jan. 64)
Computation Lab., Marshall Space Flight Ctr, NASA, Huntsville, Ala.

```
procedure CONFRAC (x, n, parm, answer);
  integer parm, n;  real x, answer;
comment This procedure utilizes a continued fraction which is
  equivalent to the diagonal of the Padé table for exp z, with
  error in the computed convergent less than x↑(2n)/(2 × 6↑2 × (10)↑2
  × ··· × (4n − 2)↑2(4n + 2)). This fraction was developed by
  J. C. Morelock, Note on Padé Table Approximation, Internal
  Note MIN-COMP-62-9, Marshall Space Flight Center, Hunts-
  ville, Alabama, 1962. For source reference see Nathaniel Macon,
  On the computation of exponential and hyperbolic functions
  using continued fractions, J. ACM, 2(1955), 262-266. The argu-
  ment, x, is assumed to be less than π/4. For such x any desired
  level of accuracy is quickly computed for each function specified
  as follows:

parm := 1,  answer := sin x     parm := 5,  answer := sinh x
parm := 2,  answer := cos x     parm := 6,  answer := cosh x
parm := 3,  answer := tan x     parm := 7,  answer := tanh x
parm := 4,  answer := exp x
```

The body of this procedure has been tested using extended ALGOL for the B-5000 Computer. It gave the following results:

```
x = 0.50  n = 1  parm = 1  answer = 0.47938 801530
x = 0.50  n = 2  parm = 1  answer = 0.47942 547125
x = 0.50  n = 3  parm = 1  answer = 0.47942 553854
x = 0.50  n = 4  parm = 1  answer = 0.47942 553860
x = 0.50  n = 1  parm = 2  answer = 0.87760 305992
x = 0.50  n = 2  parm = 2  answer = 0.87758 259869
x = 0.50  n = 3  parm = 2  answer = 0.87758 256193
x = 0.50  n = 4  parm = 2  answer = 0.87758 256189
x = 0.50  n = 1  parm = 3  answer = 0.54624 697337
x = 0.50  n = 2  parm = 3  answer = 0.54630 239019
x = 0.50  n = 3  parm = 3  answer = 0.54630 248974
x = 0.50  n = 4  parm = 3  answer = 0.54630 248985
x = 0.50  n = 1  parm = 4  answer = 1.64864 864865
x = 0.50  n = 2  parm = 4  answer = 1.64872 139973
x = 0.50  n = 3  parm = 4  answer = 1.64872 127057
x = 0.50  n = 4  parm = 4  answer = 1.64872 127070
x = 0.50  n = 1  parm = 5  answer = 0.52104 563580
x = 0.50  n = 2  parm = 5  answer = 0.52109 539374
x = 0.50  n = 3  parm = 5  answer = 0.52109 530541
x = 0.50  n = 4  parm = 5  answer = 0.52109 530549
x = 0.50  n = 1  parm = 6  answer = 1.12760 301285
x = 0.50  n = 2  parm = 6  answer = 1.12762 600598
x = 0.50  n = 3  parm = 6  answer = 1.12762 596516
x = 0.50  n = 4  parm = 6  answer = 1.12762 596521
x = 0.50  n = 1  parm = 7  answer = 0.46208 251473
x = 0.50  n = 2  parm = 7  answer = 0.46211 721881
x = 0.50  n = 3  parm = 7  answer = 0.46211 715720
x = 0.50  n = 4  parm = 7  answer = 0.46211 715726
```

```
  The value of n selects the continued fraction convergent;
begin integer i, ndigt;
  real r, f;
  r := if parm ≦ 3 then − x ↑ 2 else x ↑ 2;
  f := 4 × n + 2;
  for i := n step −1 until 1 do f := 4 × i − 2 + r/f;
  ndigt := if parm ≦ 3 then parm + 1 else parm −3;
  answer := if ndigt = 1 then (f+x)/(f−x)
    else if ndigt = 2 then 2 × x × f/((f ↑ 2) − r)
    else if ndigt = 3 then ((f ↑ 2)+r)/((f ↑ 2)−r)
    else if ndigt = 4 then 2 × x × f/((f ↑ 2)+r)
    else x;
end
```

CERTIFICATION OF ALGORITHM 229 [B1] ELEMENTARY FUNCTIONS BY CONTINUED FRACTIONS [James C. Morelock, *Comm. ACM 7* (May 1964), 296]

T. A. BRAY (Recd. 18 June 1964)
Boeing Scientific Research Laboratories, Seattle, WA 98124

KEY WORDS AND PHRASES: continued factions, Padé table
CR CATEGORIES: 5.19

Algorithm 229 was coded in FORTRAN II and run on the IBM 1620 computer for $x = 0.50$ and 0.75, for $n = 1, 2, 3, 4$, and for *parm* = 1, 2, 3, 4, 5, 6, 7.

For $x = 0.50$ my values agree with the author's up to $\pm 10^{-11}$.

For $x = 0.75$ and $n = 4$, my values of sin x, cos x, tan x, and exp x agree with tabulated values to within $\pm 10^{-11}$. For the same x and n my values of sinh x, and cosh x, and tanh x agree with tabulated values to within $\pm 10^{-10}$; no tables were available to check the 11th decimal.

ALGORITHM 230
MATRIX PERMUTATION

J. Boothroyd (Recd 18 Nov. 1963)
English Electric-Leo Computers, Kidsgrove, Stoke-on-Trent, England

```
procedure matrixperm(a,b,j,k,s,d,n,p); value n; real a,b;
  integer array s,d; integer j,k,n,p;
comment a procedure using Jensen's device which exchanges
  rows or columns of a matrix to achieve a rearrangement specified
  by the permutation vectors s,d[1:n]. Elements of s specify the
  original source locations while elements of d specify the desired
  destination locations. Normally a and b will be called as sub-
  scripted variables of the same array. The parameters j,k nom-
  inate the subscripts of the dimension affected by the permuta-
  tion, p is the Jensen parameter. As an example of the use of this
  procedure, suppose r,c[1:n] to contain the row and column sub-
  scripts of the successive matrix pivots used in a matrix inver-
  sion of an array a[1:n,1:n]; i.e. r[1], c[1] are the relative sub-
  scripts of the first pivot r[2], c[2] those of the second pivot and
  so on. The two calls
              matrixperm (a[j,p], a[k,p], j,k,r,c,n,p)
          and matrixperm (a[p,j], a[p,k], j,k,c,r,n,p)
  will perform the required rearrangement of rows and columns
    respectively;
begin integer array tag, loc[1:n]; integer i,t; real w;
comment set up initial vector tag number and address arrays;
  for i := 1 step 1 until n do tag[i] := loc[i] := i;
comment start permutation;
  for i := 1 step 1 until n do
    begin t := s[i]; j := loc[t]; k := d[i];
      if j≠k then begin for p := 1 step 1 until n do
                    begin w := a; a := b; b := w end;
                      tag[j] := tag[k]; tag[k] := t;
                      loc[t] := loc[tag[j]]; loc[tag[j]] := j
                  end jk conditional
    end i loop
end matrixperm
```

ALGORITHM 231
MATRIX INVERSION

J. Boothroyd (Recd 18 Nov. 1963)
English Electric-Leo Computers, Kidsgrove, Stoke-on-Trent, England

```
procedure matrixinvert (a,n,eps,singular); value n,eps; ar-
  ray a; integer n; real eps; label singular;
comment inverts a matrix in its own space using the Gauss-
  Jordan method with complete matrix pivoting. I.e., at each
  stage the pivot has the largest absolute value of any element in
  the remaining matrix. The coordinates of the successive matrix
  pivots used at each stage of the reduction are recorded in the
  successive element positions of the row and column index
  vectors r and c. These are later called upon by the procedure
  matrixperm which rearranges the rows and columns of the
  matrix. If the matrix is singular the procedure exits to an appro-
  priate label in the main program;
begin integer i,j,k,l,pivi,pivj,p; real pivot; integer array
  r,c[1:n];
comment set row and column index vectors;
  for i := 1 step 1 until n do r[i] := c[i] := i;
comment find initial pivot; pivi := pivj := 1;
  for i := 1 step 1 until n do for j := 1 step 1 until n do
    if abs (a[i,j]) > abs (a[pivi,pivj]) then begin pivi := i;
    pivj := j end;
comment start reduction;
  for i := 1 step 1 until n do
  begin l := r[i]; r[i] := r[pivi]; r[pivi] := l; l := c[i];
    c[i] := c[pivj]; c[pivj] := l;
    if eps > abs (a[r[i],c[i]]) then
    begin comment here include an appropriate output pro-
      cedure to record i and the current values of r[1:n] and
      c[1:n]; go to singular end;
  for j := n step −1 until i+1, i−1 step −1 until 1 do a[r[i],c[j]]
    := a[r[i],c[j]]/a[r[i],c[i]]; a[r[i],c[i]] := 1/a[r[i],c[i]];
    pivot := 0;
  for k := 1 step 1 until i−1, i+1 step 1 until n do
    begin for j := n step −1 until i+1, i−1 step −1 until 1 do
      begin a[r[k],c[j]] := a[r[k],c[j]] − a[r[i],c[j]] × a[r[k],c[i]];
        if k>i ∧ j>i ∧ abs (a[r[k],c[j]]) > abs(pivot) then
          begin pivi := k; pivj := j;
            pivot := a[r[k],c[j]] end conditional
      end jloop;
        a[r[k],c[i]] := −a[r[i],c[i]] × a[r[k],c[i]]
    end kloop
  end iloop and reduction;
comment rearrange rows; matrixperm (a[j,p],a[k,p],j,k,r,c,n,p);
comment rearrange columns;
  matrixperm (a[p,j],a[p,k],j,k,c,r,n,p)
end matrixinvert
```

[Editor's Note. On many compilers *matrixinvert* would run much faster if the subscripted variables *r*[*i*], *c*[*i*], *r*[*k*] were replaced by simple integer variables *ri*, *ci*, *rk*, respectively, inside the *j* loop.—G.E.F.]

REMARK ON ALGORITHM 231 [F1]
MATRIX INVERSION

[J. Boothroyd, *Comm. ACM 6* (June 1964), 347]

Mats Ferring (Recd. 23 Nov. 1964)
Flygmotor Aeroengine Company, Trollhättan, Sweden

The algorithm cannot accept the pivot element = 0 which reduces the detection of singularities. We suggest the correction:

```
if k > i ∧ j > i ∧ abs(a[r[k], c[j]]) > abs(pivot) then
```

should be

```
if k > i ∧ j > i ∧ abs(a[r[k], c[j]]) ≧ abs(pivot) then
```

ALGORITHM 232
HEAPSORT

J. W. J. Williams (Recd 1 Oct. 1963 and, revised, 15 Feb. 1964)

Elliott Bros. (London) Ltd., Borehamwood, Herts, England

```
comment The following procedures are related to TREESORT
  [R. W. Floyd, Alg. 113, Comm. ACM 5 (Aug. 1962), 434, and
  A. F. Kaupe, Jr., Alg. 143 and 144, Comm. ACM 5 (Dec. 1962),
  604] but avoid the use of pointers and so preserve storage space.
  All the procedures operate on single word items, stored as
  elements 1 to n of the array A. The elements are normally so
  arranged that A[i]≦A[j] for 2≦j≦n, i=j÷2. Such an arrange-
  ment will be called a heap. A[1] is always the least element of
  the heap.
    The procedure SETHEAP arranges n elements as a heap,
  INHEAP adds a new element to an existing heap, OUTHEAP
  extracts the least element from a heap, and SWOPHEAP is
  effectively the result of INHEAP followed by OUTHEAP. In
  all cases the array A contains elements arranged as a heap on
  exit.
    SWOPHEAP is essentially the same as the tournament sort
  described by K. E. Iverson—A Programming Language, 1962,
  pp. 223-226—which is a top to bottom method, but it uses an
  improved storage allocation and initialisation. INHEAP re-
  sembles TREESORT in being a bottom to top method. HEAP-
  SORT can thus be considered as a marriage of these two
  methods.
    The procedures may be used for replacement-selection sort-
  ing, for sorting the elements of an array, or for choosing the
  current minimum of any set of items to which new items are
  added from time to time. The procedures are the more useful
  because the active elements of the array are maintained densely
  packed, as elements A[1] to A[n];
procedure SWOPHEAP (A,n,in,out);
  value in,n; integer n; real in,out; real array A;
  comment SWOPHEAP is given an array A, elements A[1]
    to A[n] forming a heap, n≧0. SWOPHEAP effectively adds
    the element in to the heap, extracts and assigns to out
    the value of the least member of the resulting set, and leaves
    the remaining elements in a heap of the original size. In
    this process elements 1 to (n+1) of the array A may be dis-
    turbed. The maximum number of repetitions of the cycle
    labeled scan is log₂n;
  begin integer i,j; real temp, temp 1;
    if in≦A[1] then out := in else
    begin i := 1;
      A[n+1] := in; comment this last statement is only
        necessary in case j=n at some stage, or n=0;
      out := A[1];
    scan: j := i+i;
      if j≦n then
      begin temp := A[j];
        temp 1 := A[j+1];
        if temp 1<temp then
        begin temp := temp 1;
          j := j+1
        end;
        if temp < in then
        begin A[i] := temp;
          i := j;
          go to scan
        end
      end;
      A[i] := in
    end
  end SWOPHEAP;
procedure INHEAP (A, n, in);
  value in; integer n; real in; real array A;
  comment INHEAP is given an array A, elements A[1] to
    A[n] forming a heap and n≧0. INHEAP adds the element in
    to the heap and adjusts n accordingly. The cycle labeled
    scan may be repeated log₂n times, but on average is repeated
    twice only;
  begin integer i,j;
    i := n :=n+1;
  scan: if i>1 then
    begin j := i÷2;
      if in<A[j] then
      begin A[i] := A[j];
        i := j;
        go to scan
      end
    end;
    A[i] := in
  end INHEAP;
procedure OUTHEAP (A,n,out);
  integer n; real out; real array A;
  comment given array A, elements 1 to n of which form a heap,
    n≧1, OUTHEAP assigns to out the value of A[1], the least
    member of the heap, and rearranges the remaining members
    as elements 1 to n−1 of A. Also, n is adjusted accordingly;
  begin SWOPHEAP (A,n−1, A[n],out);
    n := n−1
  end OUTHEAP;
procedure SETHEAP (A,n);
  value n; integer n; real array A;
  comment SETHEAP rearranges the elements A[1] to A[n]
    to form a heap;
  begin integer j;
      j := 1;
  L: INHEAP(A,j,A[j+1]);
      if j<n then go to L
  end SETHEAP
```

ALGORITHM 233
SIMPSON'S RULE FOR MULTIPLE INTEGRATION

FRANK OLYNYK* (Recd 24 Dec. 1963)
Case Institute of Technology, Cleveland, Ohio

*Partially sponsored by the National Science Foundation under Grant GP-642.

```
real procedure Simps (X, x1, x2, delta, f);
  value x1, x2, delta;  real X, x1, x2, delta, f;
comment  This procedure calculates a single integral by Simp-
  son's rule in such a way that it can be called recursively for the
  evaluation of an iterated integral. x1 and x2 are the lower and
  upper limits, respectively, which may be any mathematically
  meaningful expressions. Hence in using Simps for multiple
  integration the region is not limited to rectangular boxes. The
  algorithm terminates when two successive evaluations pass the
  test involving delta. The formal parameter f stands for the
  expression to be integrated.
    As an example of the use of Simps,
```

$$\int_0^1 dx \int_0^{(1-x^2)^{\frac{1}{2}}} g(x, y)\ dy$$

```
  would be evaluated by
Simps(x, 0, 1, delta, Simps(y, 0, sqrt(1 - x ↑ 2), delta2, g(x, y))).
    Simps has been written and run in ALGOL 60 on the Univac
  1107 at Case Institute.
    [EDITOR'S NOTE. Experience of W. McKeeman suggests the
  wisdom of choosing delta2 < delta.—G.E.F.];
begin
  Boolean turing;  real z1, z2, z3, h, k;
  turing := false;
  if x1 = x2 then begin z1 := 0;  go to box2 end;
  if x1 > x2 then begin h := x1;  x1 := x2;  x2 := h;
    turing := true end;
  X := x1;  z1 := f;  X := x2;  z3 := z1 := z1 + f;
  k := x2 - x1;
box:
  z2 := 0;  h := k/2;
  for X := x1 + h step k until x2 do  z2 := z2 + f;
  z1 := z1 + 4 × z2;
  if h × abs((z1 - 2 × z3)/(if z1 = 0 then 1.0 else z1)) < delta
    then go to box2
  else z3 := z1;
  z1 := z1 - 2 × z2;
  k := h;
  go to box;
box2:
  if turing then h := -h;
  Simps := h × z1/3
end Simps
```

REMARK ON ALGORITHM 233 [D1]
SIMPSON'S RULE FOR MULTIPLE INTEGRATION

[Frank Olynyk, *Comm. ACM 7* (June 1964), 348]

L. G. PROLL (Recd. 6 Apr. 1970)
Department of Mathematics, University of Southampton, U.K.

KEY WORDS AND PHRASES: numerical integration, multiple integration, Simpson's rule
CR CATEGORIES: 5.16

Algorithm 233 fails in the case $x1 = x2$ since h and, thus, the value of the function *Simps* are undefined. This situation can be avoided by replacing the line

```
  if x1 = x2 then begin z1 := 0;  go to box2 end;
```

by

```
  if x1 = x2 then begin Simps := 0.0;  go to box3 end;
```

and by replacing the last two lines of the procedure by

```
    Simps := h × z1/3.0;
  box3:
  end Simps
```

The algorithm can be marginally improved by replacing each integer constant by its equivalent decimal number.

ALGORITHM 234
POISSON-CHARLIER POLYNOMIALS [S23]

J. M. S. Simões Pereira (Recd. 6 Jan. 1964)
Gulbenkian Scientific Computing Center, Lisboa, Portugal

```
real procedure PCpolynomial (x, n, a);
  integer n; real x, a;
comment PCpolynomial computes values of the Poisson-
  Charlier polynomial p_n(x) defined by L. Carlitz, Characteriza-
  tion of certain sequences of orthogonal polynomials, Portugaliae
  Mathematica 20 (1961), 43-46:
```

$$p_n(x) = a^{n/2}(n!)^{-1/2}\sum_{r=0}^{n}(-1)^{n-r}\binom{n}{r}r!\,a^{-r}\binom{x}{r}.$$

```
  In this algorithm u stands for the successive terms of the summa-
  tion, s stands for the sum of these terms and all other symbols
  possess evident meanings. Clearly each term of the summation
  is obtained from the preceding one by the indicated multipli-
  cation;
begin
  integer j; real u, s, c;
  u := (−1) ↑ n;
  s := u;
  c := 1;
  for j := 1 step 1 until n do c := c × j;
  for j := 0 step 1 until n − 1 do
    begin u := − u × (n − j) × (x − j)/(a × (j + 1)); s :=
      s + u end;
  PCpolynomial := sqrt(a ↑ n/c) × s
end PCpolynomial
```

CERTIFICATION OF ALGORITHM 234 [S23]
POISSON-CHARLIER POLYNOMIALS [J. M. S. Simões-Pereira. *Comm. ACM 7* (July 1964), 420]

P. A. Samet (Recd. 17 Aug. 1964)
Computation Lab., The University, Southampton, Eng.

PC polynomial was compiled correctly by the Pegasus-Algol compiler and ran without trouble. The procedure was tested for $n = 0(1)4$, values of a in the range 0.2 to 2.0, and x in the range 0 to 1. The values produced were spotchecked by hand.

The procedure could be improved by

(i) putting x, n, a in the **value** part.

(ii) replacing $u := (-1) \uparrow n$ by

$u :=$ **if** $n = n \div 2 \times 2$ **then** 1 **else** -1

(iii) eliminating the separate evaluation of $n!$ by including the evaluation of $a^n \cdot (n!)^{-1}$ in the main loop. This gives a simpler argument for *sqrt* in the final assignment statement.

The revised algorithm then reads

```
real procedure PCpolynomial (x, n, a);
  value x, n, a; real x, a; integer n;
begin integer j; real u, s, c;
  s := u := if n = n ÷ 2 × 2 then 1 else −1;
  c := 1;
  for j := 0 step 1 until n − 1 do
  begin u := −u × (n−j) × (x−j)/(a × (j+1));
    s := s + u;
    c := c × a/(j+1)
  end;
  PCpolynomial := sqrt(c) × s
end PCpolynomial
```

This version gave the same results as the original but was appreciably faster.

ALGORITHM 235
RANDOM PERMUTATION [G6]

RICHARD DURSTENFELD (Recd. 2 Jan. 64)
General Atomic, San Diego 12, Calif.

```
procedure SHUFFLE (a, n, random);
  value n;  integer n;  real procedure random; integer
    array a;
begin
  comment  SHUFFLE applies a random permutation to the
    sequence a[i] where i = 1, 2, ... , n. The procedure random is
    supposed to supply a random element from a large population
    of real numbers uniformly distributed over the open unit
    interval 0 < r < 1. The array a is declared to be integer but
    actually it suffices for its type to agree with that of the vari-
    able b (in the procedure body);
  integer i, j; real b;
  for i := n step − 1 until 2 do
    begin j := entier (i × random + 1);
           b := a[i];  a[i] := a[j];  a[j] := b
    end   loop i
end SHUFFLE
```

Note. Numbers in brackets following Algorithm titles indicate the subject category for the algorithm, based on the Modified SHARE Classification listing given in the March, 1964 issue of the *Communications of the ACM*.

REMARK ON ALGORITHM 235 [G6]
RANDOM PERMUTATION [Richard Durstenfeld, *Comm. ACM 7* (July 1964), 420]

M. C. PIKE (Recd. 11 Feb. 1965 and 5 Apr. 1965)
Statistical Research Unit of the Medical Research Council,
University College Hospital Medical School, London,
England

SHUFFLE applies a random permutation to the complete sequence $a[i]$ where $i = 1, 2, \cdots, n$. *SHUFFLE* does this in such a way that after k calls of the real procedure *random* the elements $a[i]$ for $i = n-k+1, n-k+2, \cdots, n$ are a random permutation of the original n elements $a[i]$ where $i = 1, 2, \cdots, n$ taken k at a time. In many applications this will be all that is required and by coming out of the procedure at this point the remaining $n - k - 1$ calls of *random* and the subsequent transfers will be avoided; this will result in a considerable saving in time if k is much smaller than n. The necessary modifications are:

(1) Amend the procedure heading by adding the variable k:

```
      procedure SHUFFLE (a, n, k, random);
      value n, k;  integer n, k;
```

(2) Amend the line

```
      for i := n step −1 until 2 do
```

to read:

```
      k := n+1−k;
      for i := n step −1 until k do
```

Note that at exit $a[1:n]$ will still contain all the elements of the original $a[1:n]$, and that if $k=n$ that these modifications will make the procedure call *random* one more time than the original *SHUFFLE*.

ALGORITHM 236
BESSEL FUNCTIONS OF THE FIRST KIND [S17]

Walter Gautschi (Recd. 10 Aug. 1963 and 10 Apr. 1964)
Oak Ridge National Laboratory, Oak Ridge, Tenn.*

* Now at Purdue University, Lafayette, Ind.

```
real procedure t(y);   value y;   real y;
comment  This is an auxiliary procedure which evaluates the
  inverse function t = t(y) of y = t ln t  (t≧1) to an accuracy of
  about 1%. For the interval 0 ≦ y ≦ 10 a fifth degree approxi-
  mating polynomial was obtained by truncating a series expan-
  sion in Chebyshev polynomials. For y > 10 the approximation
  t(y)  ≐  (y/ln(y/α))(1+(lnα−lnln(y/α))/(1+ln(y/α)))⁻¹  where
  ln α = .775† is used;

begin real p, z;
  if y ≦ 10 then
    begin
      p := .000057941 × y − .00176148;  p := y × p + .0208645;
      p := y × p − .129013;  p := y × p + .85777;
      t := y × p + 1.0125
    end
  else
    begin
      z := ln (y) − .775;  p := (.775−ln (z))/(1+z);
      p := 1/(1+p);  t := y × p/z
    end
end t;

procedure Japlusn (x, a, nmax, d, J);   value x, a, nmax, d;
  integer nmax, d;  real x, a;  array J;
comment  This procedure evaluates to d significant digits the
  Bessel functions J_{a+n}(x) for fixed a, x and for n = 0, 1, ⋯ , nmax.
  The results are stored in the array J. It is assumed
  that 0 ≦ a < 1, x > 0, and nmax ≧ 0. If any of these variables is
  not in the range specified, control is transferred to a nonlocal
  label called alarm. The procedure makes use of the real procedure
  t. In addition, it calls for a nonlocal real procedure gamma which
  evaluates Γ(z) for 1 ≦ z ≦ 2. (See [2].) The method of computa-
  tion is a variant of the backward recurrence algorithm of J. C.
  P. Miller. (See [1].) The purported accuracy is obtained by a
  judicious selection of the initial value ν of the recursion index,
  together with at least one repetition of the recursion with ν re-
  placed by ν + 5. Near a zero of one of the Bessel functions
  generated, the accuracy of that particular Bessel function may
  deteriorate to less than d significant digits. The algorithm is
  most efficient when x is small or moderately large;
begin integer n, nu, m, limit;  real epsilon, sum, d1, r, s, L,
  lambda;  array Japprox, Rr[0:nmax];
  if a < 0 ∨ a ≧ 1 ∨ x ≦ 0 ∨ nmax < 0 then go to alarm;
  epsilon := .5 × 10 ↑ (−d);
  for n := 0 step 1 until nmax do Japprox[n] := 0;
  sum := (x/2)↑a/gamma (1+a);
  d1 := 2.3026 × d + 1.3863;
  if nmax > 0 then r := nmax × t(.5×d1/nmax) else r := 0;
  s := 1.3591 × x × t(.73576×d1/x);
  nu := 1 + entier (if r≦s then s else r);
L0:  m := 0;  L := 1;  limit := entier (nu/2);
L1:  m := m + 1;
  L := L × (m+a)/(m+1);
  if m < limit then go to L1;
  n := 2 × m;  r := s := 0;
L2:  r := 1/(2×(a+n)/x−r);
  comment  Conceivably, but very unlikely, division by an
    exact zero, or overflow, may take place here. The user may
    wish to test the divisor for zero, and, if necessary, enlarge it
    slightly to avoid overflow, before this statement is carried out.
    As such a test depends on the particular machine used, it was
    not included here;
  if entier (n/2) ≠ n/2 then lambda := 0 else
    begin
      L := L × (n+2)/(n+2×a);
      lambda := L × (n+a)
    end;
  s := r × (lambda+s);  if n ≦ nmax then Rr[n−1] := r;
  n := n − 1;  if n ≧ 1 then go to L2;
  J[0] := sum/(1+s);
  for n := 0 step 1 until nmax − 1 do J[n+1] := Rr[n] × J[n];
  for n := 0 step 1 until nmax do
    if abs((J[n] − Japprox[n])/J[n]) > epsilon then
    begin
      for m := 0 step 1 until nmax do Japprox[m] := J[m];
      nu := nu + 5;  go to L0
    end
end Japlusn;

procedure Iaplusn(x, a, nmax, d, I);   value x, a, nmax, d;
  integer nmax, d;  real x, a;  array I;
comment  This procedure evaluates to d significant digits the
  modified Bessel functions I_{a+n}(x) for fixed a, x, with 0 ≦ a < 1,
  x > 0, and for n = 0, 1, ⋯ , nmax. The results are stored in the
  array I. For the setup of the procedure, and the method of com-
  putation used, see the comment in Japlusn;
begin integer n, nu, m;  real epsilon, sum, d1, r, s, L, lambda;
  array Iapprox, Rr[0:nmax];
  if a < 0 ∨ a ≧ 1 ∨ x ≦ 0 ∨ nmax < 0 then go to alarm;
  epsilon := .5 × 10 ↑ (−d);
  for n := 0 step 1 until nmax do Iapprox[n] := 0;
  sum := exp(x) × (x/2) ↑ a/gamma(1+a);
  d1 := 2.3026 × d + 1.3863;
  if nmax > 0 then r := nmax × t(.5×d1/nmax) else r := 0;
  s := if x < d1 then 1.3591 × x × t(.73576×(d1−x)/x) else
    1.3591 × x;
  nu := 1 + entier (if r≦s then s else r);
L0:  n := 0;  L := 1;
L1:  n := n + 1;
  L := L × (n+2×a)/(n+1);
  if n < nu then go to L1;
  r := s := 0;
```

† In an earlier version of this procedure the author used $\alpha = 1$. The value $\ln \alpha = .775$ was found empirically by H. C. Thacher, Jr. to yield somewhat better approximations.

```
L2: r := 1/(2×(a+n)/x+r);
  L := L × (n+1)/(n+2×a);
  lambda := 2 × (n+a) × L;
  s := r × (lambda+s); if n ≦ nmax then Rr[n−1] := r;
  n := n − 1; if n ≧ 1 then go to L2;
  I[0] := sum/(1+s);
  for n := 0 step 1 until nmax − 1 do I[n+1] := Rr[n] × I[n];
  for n := 0 step 1 until nmax do
    if abs((I[n]−Iapprox[n])/I[n]) > epsilon then
      begin
        for m := 0 step 1 until nmax do Iapprox[m] := I[m];
        nu := nu + 5; go to L0
      end
end Iaplusn;

procedure Jaminusn(x, a, nmax, d, J); value x, a, nmax, d;
  integer nmax, d; real x, a; array J;
comment This procedure evaluates to d significant digits the
  Bessel functions J_{a−n}(x) for fixed a, x, with 0 < a < 1, x > 0,
  and for n = 0, 1, ··· , nmax. The results are stored in the array
  J. The procedure makes use of the real procedure t, and the
  procedure Japlusn. In addition, it calls for a nonlocal real pro-
  cedure gamma which evaluates Γ(z) for 1 ≦ z ≦ 2. (See [2].) The
  accuracy may deteriorate to less than d significant digits if a is
  close to 0 or 1;
begin integer n; array J1[0:1];
  if a = 0 then go to alarm;
  Japlusn(x, a, 1, d, J1);
  J[0] := J1[0];
  J[1] := 2 × a × J[0]/x − J1[1];
  for n := 1 step 1 until nmax − 1 do
    J[n+1] := 2 × (a−n) × J[n]/x − J[n−1]
end Jaminusn;

procedure Iaminusn(x, a, nmax, d, I); value x, a, nmax, d;
  integer nmax, d; real x, a; array I;
comment This procedure evaluates to d significant digits the
  modified Bessel functions I_{a−n}(x) for fixed a, x, with 0 < a < 1,
  x > 0, and for n = 0, 1, ··· , nmax. The results are stored in the
  array I. The procedure makes use of the real procedure t, and
  the procedure Iaplusn. In addition, it calls for a nonlocal real
  procedure gamma which evaluates Γ(z) for 1 ≦ z ≦ 2. (See [2].)
  The accuracy may deteriorate to less than d significant digits if
  a is close to 0 or 1;
begin integer n; array I1[0:1];
  if a = 0 then go to alarm;
  Iaplusn(x, a, 1, d, I1);
  I[0] := I1[0];
  I[1] := 2 × a × I[0]/x + I1[1];
  for n := 1 step 1 until nmax − 1 do
    I[n+1] := 2 × (a−n) × I[n]/x + I[n−1]
end Iaminusn;

procedure Complex Japlusn(x, y, a, nmax, d, u, v); value x, y, a,
  nmax, d;
  integer nmax, d; real x, y, a; array u, v;
comment This procedure evaluates to d significant digits the
  Bessel functions J_{a+n}(z) = u_n + iv_n for fixed real a, fixed complex
  z = x + iy, and for n = 0, 1, ··· , nmax. The real parts u_0,
  u_1, ···, u_nmax of the results are stored in the array u, the imagi-
  nary parts v_0, v_1, ··· , v_nmax in the array v. It is assumed that
  0 ≦ a < 1, nmax ≧ 0, and that z is not on the negative real axis
  x ≦ 0, y = 0. Otherwise, control is transferred to the nonlocal
  label alarm upon entry of the procedure. The procedure makes
  use of the real procedure t. In addition, it calls for a nonlocal
  real procedure gamma which evaluates Γ(z) for 1 ≦ z ≦ 2. (See
  [2].) The method of computation is a complex extension of the
  method used in the procedure Japlusn. The algorithm is most
  efficient when | z | is small or moderately large;
begin integer n, nu, m; real epsilon, y1, r02, r0, phi, c, c1, c2,
  sum1, sum2, d1, r, s, lambda1, lambda2, L, r1, r2, s1, s2; array
  uapprox, vapprox, Rr1, Rr2[0:nmax];
  if a < 0 ∨ a ≧ 1 ∨ (x≦0∧y=0) ∨ nmax < 0 then go to alarm;
  epsilon := .5 × 10 ↑ (−d);
  for n := 0 step 1 until nmax do uapprox[n] := vapprox[n] := 0;
  y1 := abs(y); r02 := x ↑ 2 + y ↑ 2; r0 := sqrt(r02);
  phi := if x = 0 then 1.5707963268 else if x > 0 then arctan(y1/x)
    else 3.1415926536 + arctan(y1/x);
comment The two constants π/2 and π in the preceding state-
  ment are to be supplied with the full accuracy desired in the
  final results;
  c := exp(y1) × (r0/2)↑a/gamma (1+a);
  sum1 := c × cos(a×phi−x); sum2 := c × sin(a×phi−x);
  d1 := 2.3026 × d + 1.3863;
  if nmax > 0 then r := nmax × t(.5×d1/nmax) else r := 0;
  s := if y1 < d1 then 1.3591 × r0 × t(.73576×(d1−y1)/r0) else
    1.3591 × r0;
  nu := 1 + entier (if r≦s then s else r);
L0: n := 0; L := 1; c1 := 1; c2 := 0;
L1: n := n + 1;
  L := L × (n+2×a)/(n+1);
  c := −c1; c1 := c2; c2 := c;
  if n < nu then go to L1;
  r1 := r2 := s1 := s2 := 0;
L2: c := (2×(a+n)−x×r1+y1×r2) ↑ 2 + (x×r2+y1×r1) ↑ 2;
  r1 := (2×(a+n)×x−r02×r1)/c;
  r2 := (2×(a+n)×y1+r02×r2)/c;
  L := L × (n+1)/(n+2×a); c := 2 × (n+a) × L;
  lambda1 := c × c1; lambda2 := c × c2;
  c := c1; c1 := −c2; c2 := c;
  s := r1 × (lambda1+s1) − r2 × (lambda2+s2);
  s2 := r1 × (lambda2+s2) + r2 × (lambda1+s1);
  s1 := s;
  if n ≦ nmax then begin Rr1[n−1] := r1; Rr2[n−1] := r2 end;
  n := n − 1;
  if n ≧ 1 then go to L2;
  c := (1+s1) ↑ 2 + s2 ↑ 2;
  u[0] := (sum1×(1+s1)+sum2×s2)/c;
  v [0] := (sum2×(1+s1)−sum1×s2)/c;
  for n := 0 step 1 until nmax − 1 do
    begin
      u[n+1] := Rr1[n] × u[n] − Rr2[n] × v[n];
      v[n+1] := Rr1[n] × v[n] + Rr2[n] × u[n]
    end;
  if y < 0 then for n := 0 step 1 until nmax do v[n] := − v[n];
  for n := 0 step 1 until nmax do
    if sqrt(((u[n]−uapprox[n]) ↑ 2+(v[n]−vapprox[n]) ↑ 2)
      /(u[n] ↑ 2+v[n] ↑ 2)) > epsilon
    then
    begin
      for m := 0 step 1 until nmax do
        begin uapprox[m] := u[m]; vapprox[m] := v[m] end;
      nu := nu + 5; go to L0
    end
end Complex Japlusn
```

REFERENCES

1. Gautschi, W. Recursive computation of special functions. U. Mich. Engineering Summer Conferences, Numerical Analysis, 1963.
2. ——. Algorithm 221—Gamma function. *Comm. ACM 7* (Mar. 1964), 143.

CERTIFICATION OF ALGORITHM 236 [S17] BESSEL FUNCTIONS OF THE FIRST KIND [Walter Gautschi, *Comm. ACM 7* (Aug. 1964), 479]

WALTER GAUTSCHI (Recd. 24 Aug. 1964 and 2 Nov. 1964)
Purdue University, Lafayette, Ind.

All procedures were tested on the CDC 1604-A computer, using the Oak Ridge ALGOL compiler.

1. The procedure *Japlusn* was submitted to the following tests:

(a) Values of $J_n(2)$ and $J_{n+1/2}(10)$ were produced for $n = 0(1)10$, calling for an accuracy of $d = 6$ significant digits. The values obtained for $J_n(2)$ agreed with those of Table 9.4 in [1] to 10 significant digits (with occasional discrepancies of one unit in the tenth figure). The results for $J_{n+1/2}(10)$ were compared against those of $J_{n+1/2}(10) = 2.523132521 \times j_n(10)$ obtained from Table 10.5 in [1]. The maximum discrepancy was found to be five units in the tenth figure, occurring for $n = 3$.

(b) To observe the performance of the procedure near a zero of a Bessel function, we generated $J_n(x)$, $n = 0(1)10$, for $x = 2.40482556$--the 8D value of the first zero $j_{0,1}$ of J_0—calling for $d = 10$ significant digits. The results are shown in the table below.

n	$J_n(j_{0,1})$	n	$J_n(j_{0,1})$
0	$-1.1936252775_{10}-9$	6	$3.4048184902_{10}-3$
1	$5.1914749680_{10}-1$	7	$6.0068836955_{10}-4$
2	$4.3175480738_{10}-1$	8	$9.2165787385_{10}-5$
3	$1.9899990578_{10}-1$	9	$1.2517271082_{10}-5$
4	$6.4746666371_{10}-2$	10	$1.5253656182_{10}-6$
5	$1.6389243276_{10}-2$		

The entry for $n = 1$ agrees to 9 figures with that of $-J_0'(j_{0,1})$ given in Table 9.5 of reference [1].

(c) We drove the procedure to calculate $J_{x+\nu-1}(x)$ to 6 significant digits, for $x = 4(4)20$, $\nu = 0(.1)1.9$. The results agreed with those tabulated in [2].

2. The procedure *Iaplusn* was called to generate test values to 6 significant figures of $I_n(20)$, $I_{n+1/2}(10)$, $I_{n+1/4}(.1)$, for $n = 0(1)10$. The first two sets of values were compared with those in [3] and in Table 10.10 of [1], respectively, and found to be in error by at most 5 units in the tenth figure. The value for $I_{1/4}(.1)$ agreed to 10 figures with that given in [5].

3. Further checks were made on the procedures *Japlusn, Iaplusn,* as well as the procedures *Jaminusn, Iaminusn,* by having them "verify" the relation

$$f_{2a+2}(2x) = f^2_{a+1}(x) + 2\sum_{n=0}^{\infty} f_{a-n}(x)f_{a+n+2}(x)$$

for $x = 1$, $a = .2(.2).8$, where $f_\nu(x)$ stands for either $J_\nu(x)$ or $I_\nu(x)$ (cf. [4], p. 100, formula (21)). That is, we printed the relative errors incurred when the infinite series is truncated after the $(N+1)$-st term, $N = 0(5)20$. Selected results (rounded to four digits) are shown in the table below.

$a \backslash N$	0	5	10	15	20
.2	$1.165_{10}-2$	$2.519_{10}-4$	$-3.568_{10}-5$	$1.043_{10}--5$	$-4.234_{10}-6$
.8	$-7.945_{10}-2$	$4.968_{10}-5$	$-3.459_{10}-6$	$6.517_{10}--7$	$-1.923_{10}-7$
.4	$-8.091_{10}-2$	$1.245_{10}-4$	$-1.456_{10}-5$	$3.714_{10}-6$	$-1.361_{10}-6$
.6	$-1.023_{10}-1$	$7.590_{10}-5$	$-7.041_{10}-6$	$1.553_{10}--6$	$-5.115_{10}-7$

The first two lines refer to $f = J$, the last two lines to $f = I$. The driver program follows.

```
begin integer n;  real a, sumJ, sumI, sJ, sI, errorJ, errorI;
  array J1, I1[0:3], J2, I2[0:22], J3, I3[0:20];
  for a := .2 step .2 until .9 do
  begin
    if 2 × a < 1 then
      begin
        Japlusn (2.0, 2 × a, 2, 6, J1);  Iaplusn (2.0, 2 × a, 2, 6, I1);
        sumJ := J1[2];  sumI := I1[2]
      end
    else
      begin
        Japlusn (2.0, 2 × a−1, 3, 6, J1);
        Iaplusn (2.0, 2 × a−1, 3, 6, I1);
        sumJ := J1[3];  sumI := I1[3]
      end;
    Japlusn (1.0, a, 22, 6, J2);  Jaminusn (1.0, a, 20, 6, J3);
    Iaplusn (1.0, a, 22, 6, I2);  Iaminusn (1.0, a, 20, 6, I3);
    sJ := sI := 0;
    for n := 0 step 1 until 20 do
    begin
      sJ := sJ + J3[n] × J2[n+2];  sI := sI + I3[n] × I2[n+2];
      if entier (n/5) = n/5 then
        begin
          errorJ := (J2[1] ↑ 2 + 2 × sJ − sumJ)/sumJ;
          errorI := (I2[1] ↑ 2 + 2 × sI − sumI)/sumI;
          outstring (1, 'a=');  outreal (1, a);
          outstring (1, 'N=');  outinteger (1, n);
          outstring (1, 'errorJ=');  outreal (1, errorJ);
          outstring (1, 'errorI=');  outreal (1, errorI)
        end
    end
  end;
  go to skip;
alarm: outstring (1, 'parameters not in range');
skip: end
```

4. The procedure *Complex Japlusn* underwent the following tests:

(a) Values of $J_n(re^{i\phi})$ were produced for $n = 0, 1$, $\phi = (r-2) \times 30°$, $r = 1(1)6$, calling for an accuracy of 6 significant digits. Comparison with [6] showed agreement to 9-10 significant figures.

(b) We asked the procedure to "verify" the identity (cf. [4], p. 99, formula (2))

$$(z/2)^a\, J_0(z) = \sum_{n=0}^{\infty} \frac{\Gamma(1-a)\Gamma(a+n)}{(n!)^2\Gamma(1-a-n)}(a+2n)J_{a+2n}(z),$$

by printing the moduli of the relative errors incurred when truncating the infinite series at $n = 0(1)5$. We let a and z run through values $a = .2(.2).8$, $z = 2\exp(i\phi)$, $\phi = -150°\ (30°)\ 150°$, respectively. Selected results (rounded to three figures) are displayed in the table below.

$\phi°$	$a \backslash$ n	0	1	2	3	4	5
−120	.2	$1.17_{10}-1$	$5.51_{10}-3$	$1.31_{10}-4$	$1.85_{10}-6$	$1.72_{10}-8$	$2.02_{10}-10$
−30	.4	$3.16_{10}-1$	$2.02_{10}-2$	$5.61_{10}-4$	$8.70_{10}-6$	$8.64_{10}-8$	$5.27_{10}-10$
60	.6	$2.60_{10}-1$	$1.65_{10}-2$	$4.67_{10}-4$	$7.41_{10}-6$	$7.51_{10}-8$	$3.93_{10}-10$
150	.8	$4.95_{10}-1$	$4.00_{10}-2$	$1.29_{10}-3$	$2.23_{10}-5$	$2.41_{10}-7$	$1.75_{10}-9$

The same pattern persists throughout the range of the variables. The driver program follows.

```
begin integer m, n;  real a, phi, c, s, x, y, sum1, sum2,
  q, s1, s2, p, error;  array u, v[0:10];
  for a := .2 step .2 until .9 do
  for m := −5 step 1 until 5 do
  begin
    phi := .52359877560 × m;
    c := cos(a×phi);  s := sin(a×phi);
    x := 2 × cos(phi);  y := 2 × sin(phi);
    Complex Japlusn (x, y, 0, 0, 6, u, v);
    sum1 := c × u[0] − s × v[0];  sum2 := c × v[0] + s × u[0];
    Complex Japlusn (x, y, a, 10, 6, u, v);
    q := gamma (1+a);
    s1 := q × u[0];  s2 := q × v[0];  p := q/a;
  n := 0;
L: error := sqrt (((sum1−s1) ↑ 2 + (sum2−s2) ↑ 2)/(sum1 ↑ 2
  + sum2 ↑ 2));
  outstring (1, 'a='); outreal (1, a);
  outstring (1, 'phi='); outinteger (1, 30×m);
  outstring (1, 'n='); outinteger (1, n);
  outstring (1, 'error='); outreal (1, error);
  n := n + 1;
  if n ≦ 5 then
  begin
    p := −p × ((n+a−1)/n) ↑ 2; q := (a+2×n) × p;
    s1 := s1 + q × u[2×n]; s2 := s2 + q × v[2×n];
    go to L
  end
end;
    go to skip;
  alarm: outstring (1, 'parameters not in range');
  skip: end
```

REFERENCES:

1. ABRAMOWITZ, M., AND STEGUN, I. A. (EDS.) *Handbook of Mathematical Functions.* NBS Appl. Math. Ser. 55, U.S. Govt. Printing Off., Washington, D.C., 1964.
2. AIREY, J. R. Bessel functions of nearly equal order and argument. *Philos. Mag.* (7) *19* (1935), 230–235.
3. BAAS. *Bessel functions, part II, Functions of positive integer order.* Mathematical Tables, vol. X, Cambridge U. Press, London, 1952.
4. ERDÉLYI, A. (ED.) *Higher Transcendental Functions, vol. II.* McGraw-Hill, New York, 1953.
5. NATIONAL BUREAU OF STANDARDS. *Tables of Bessel functions of fractional order, vol. II.* , Columbia U. Press, New York, 1949.
6. ——. *Table of the Bessel Functions $J_0(z)$ and $J_1(z)$ for Complex Arguments.* Columbia U. Press, New York, 1943.

ACM Transactions on Mathematical Software, Vol. 1, No. 3, September 1975.

REMARK ON ALGORITHM 236

Bessel Functions of the First Kind [S17]
[W. Gautschi, *Comm. ACM 7*, 8 (Aug. 1964), 479–480]

Ove Skovgaard [Recd 6 Nov. 1973 and 3 Feb. 1975]
Institute of Hydrodynamics and Hydraulic Engineering, Technical University of Denmark, DK-2800 Lyngby, Denmark

The procedures in Algorithm 236 were coded in PL/I and run on the IBM 370/165. The following error was discovered for $a = 0$, *nmax* large, and x small, e.g. $nmax = 50$ and $x = 0.5$. In the last if statement in three of the procedures, *Japlusn*, *Iaplusn*, and *Complex Japlusn*, division by zero took place. Not all compilers and computers would pose problems for the above values of the parameters; whether or not they do depends on the permissible magnitude of the floating-point numbers for the compiler and computer used. For the IBM 370/165 the smallest positive floating-point number which the computer can hold is approximately 5.40×10^{-79} (see [10, p. 163]).

The following corrections should be made in the procedure *Japlusn*.

The last if statement should be replaced by

```
if abs(J[n] − Japprox[n]) > epsilon × abs(J[n]) then
comment Conceivably, but very unlikely, underflow, i.e. the exponent of the
   floating-point number exceeds its lower bound, may take place here. In that case
   the machine representation of "floating-point zero" must be produced if the
   program is to work properly;
```

The same comment should be inserted after the statement

```
J[n + 1] := Rr[n] × J[n];
```

The same corrections should be made in the procedures *Iaplusn* and *Complex Japlusn* at the appropriate places.

The corrections of the defective if clauses proposed above are most elegant, but not the most efficient for all compilers and computers. The following general corrections in the procedure *Japlusn* have only one call instead of two calls of the *abs-function* and are therefore more efficient for some compilers.

Before the last if statement two new lines should be inserted:

```
if J[n] ≠ 0 then
begin
```

and before the last end statement one new line should be inserted:

```
end
```

The two proposed comment statements are still necessary.

The numerical results are identical for the two methods.

The same efficient (in some cases) corrections can be made in the procedures *Iaplusn* and *Complex Japlusn* at the appropriate places.

According to [5], all the material relevant for the construction of Algorithm 236 is included in [4, especially Section 5]. This reference is used in the following comments, since reference [1] in Algorithm 236 is not easily available.

The last for statement (of which the delinquent if statement is a part) is included for checking purposes only, in order to verify that the required accuracy has indeed been attained. According to [5], Gautschi says, "I believe, however, that my initial choice of ν is conservative enough to guarantee this accuracy. For all practical reason, therefore, the whole for statement in question could be deleted." This has not been checked by the present author.

Because a simplified PL/I version of Algorithm 443 [2, 3] had already been implemented in the local university computer library, the call to the real procedure *t* was replaced by an application of Algorithm 443 (version B). The solution of $w\, exp(w) = y, y > 0$ (furnished by Algorithm 443) corresponds to $w(y) = ln(t(y))$ in terms of the procedure *t*, so that $t(y) = exp(w(y))$ or $t(y) = y/w(y)$. Algorithm 443 is less efficient than procedure *t*. The former is more accurate, although this accuracy is not necessary here.

In order to improve the documentation and thereby facilitate modifications and/or translations of the procedures *Japlusn, Iaplusn,* and *Complex Japlusn*, the mathematical constants corresponding to the four decimal constants in the three procedures are given here: 2.3026 is *ln* 10, 1.3863 is *ln* 4, 1.3591 is *e*/2, .73576 is 2/*e*.

The procedure *Japlusn* was coded using double precision floating-point calculations. For implementation on the IBM 370/165 (chopping with 14 hexadecimal digits) this gives approximately 15 significant decimal digits. The procedure was used to calculate the Bessel function of the first kind for integer orders $J_n(x)$, i.e. $a \equiv 0$. The procedure was programmed with $d \equiv 15$ (the values of $J_n(x)$ were wanted with at least 15 significant digits). The values were checked using the tables in [6, 7, 8] and Table III in [9]. It was discovered that the values often had an error of 1 to 2 units in the fifteenth digit, where there was no zero of one of the Bessel functions to deteriorate the accuracy to less than 15 digits. Tests were run to determine whether the results were dependent on the selection of the initial ν; it must be remembered that the estimate of ν is very conservative; see [4, pp. 50–51]. Systematic tests revealed that it was impossible to obtain the wanted accuracy with any ν. To simplify testing, when $a \equiv 0$, all even $\lambda \equiv 2$ were used (according to [4, p. 49, line 1]), rather than the recursively generated even λ (according to [4, p. 48, last 11 lines]). With this simplification the procedure evaluated $J_n(x)$ to 15 significant digits. Near a zero of one of the generated Bessel functions, the accuracy of that particular function still deteriorated to less than 15 significant digits. This deterioration was generally of the same magnitude as occurred when λ was generated recursively.

If the procedures *Japlusn, Iaplusn,* and *Complex Japlusn* are contemplated for use in the calculation of Bessel functions of integer order only, then they might be rewritten directly employing the explicit values of λ, rather than generating them by an upward and downward recursion. This will make the procedures more efficient and slightly more accurate. In this connection it is relevant to refer to two more recent algorithms, due to Sookne [11–14], dealing with Bessel functions of integer order. Procedure *Beslri* in [12] was translated to PL/I, and tests disclosed that the execution time for procedures *Japlusn* and *Iaplusn* is of the order twice the execution time for procedure *Beslri.* Therefore Sookne's procedures, and not the procedures in Algorithms 21 and 236 (see [1] and the editorial comment in [15]), should be used for the calculation of Bessel functions of integer order.

REFERENCES

1. Börsch-Supan, W. Bessel functions for a set of integer orders, Algorithm 21. *Comm. ACM 3*, 11 (Nov. 1960), 600.
2. Einarsson, B. Remark on Algorithm 443, Solution of the transcendental equation $we^w = x$. *Comm. ACM 17*, 4 (April 1974), 225.
3. Fritsch, F.N., Shafer, R.E., and Crowley, W.P. Solution of the transcendental equation $we^w = x$, Algorithm 443. *Comm. ACM 16*, 2 (Feb. 1973), 123–124.
4. Gautschi, W. Computational aspects of three-term recurrence relations. *SIAM Rev. 9* (1967), 24–82.
5. Gautschi, W. Personal communication, Nov. 1973.
6. Gray, A., Mathews, G.B., and MacRobert, T.M. *A Treatise on Bessel Functions and Their Applications to Physics, 2nd ed.* Dover Publications, New York, 1966, pp. xiv and 327.
7. Annals of the Computation Lab., Harvard U. *Tables of the Bessel Functions of the First Kind of Orders Two and Three, Vol. IV.* Harvard U. Press, Cambridge, Mass., 1947, pp. v and 652.
8. Annals of the Computation Lab., Harvard U. *Tables of the Bessel Functions of the First Kind of Orders Zero and One, Vol. III.* Harvard U. Press, Cambridge, Mass., 1947, pp. xxxvii and 652.
9. Hayashi, K. *Tafeln der Besselschen, Theta-, Kugel- und anderer Funktionen.* Springer, Berlin, 1930, pp. v and 125.
10. *IBM System/370 Principles of Operation.* IBM Systems, Order No. GA22-7000-3, IBM, White Plains, N.Y., 1973, pp. xii and 318.
11. Sookne, D.J. Bessel functions *I* and *J* of complex argument and integer order. *J. Res. Nat. Bur. Standards 77B* (1973), 111–114.
12. Sookne, D.J. Bessel functions of real argument and integer order. *J. Res. Nat. Bur. Standards 77B* (1973), 125–132.
13. Sookne, D.J. Certification of an algorithm for Bessel functions of complex argument. *J. Res. Nat. Bur. Standards 77B* (1973), 133–136.
14. Sookne, D.J. Certification of an algorithm for Bessel functions of real argument. *J. Res. Nat. Bur. Standards 77B* (1973), 115–124.
15. Stafford, J. Certification of Algorithm 21, Bessel function for a set of integer orders. *Comm. ACM 8*, 4 (April 1965), 219.

ALGORITHM 237
GREATEST COMMON DIVISOR [A1]

J. E. L. Peck (Recd. 16 Dec. 1963)
University of Alberta, Calgary, Alberta, Canada

```
integer procedure Euclidean (a) dimension : (n) linear coeffi-
  cients : (x); value a; integer array a, x; integer n;
comment This procedure finds the greatest common divisor of
  the n nonnegative elements of the vector a, and produces values
  for $x_i$ in the expression $(a_1, a_2, \cdots, a_n) = a_1x_1 + a_2x_2 + \cdots$
  $+ a_nx_n$;
begin integer array M[1:n, 1:n];
  integer i, j, min, max, imin, imax, q, t;
  comment We set up M as an identity matrix;
INITIALISE:
  for i := 1 step 1 until n do
    for j := 1 step 1 until n do M[i, j] := 0;
  for i := 1 step 1 until n do M[i, i] := 1; max := 0;
  comment We search for the least nonzero integer in the array
    a. Note that this step need not be repeated at every iteration
    (see statement labelled DIVIDES);
MINIMUM:
  for i := 1 step 1 until n do
    begin t := a[i];
    if t ≠ 0 ∧ (max=0∨t<max) then
      begin max := t; imax := i end
    end of minimum search. If the use of the identifier max is
    confusing, observe the two statements following the label
    MAXIMUM, where the confusion is resolved;
    if max = 0 then go to ERROR; comment ERROR is a global
    label;
MAXIMUM: imin := imax; min := max;
  comment We search for the greatest element of a;
  max := a[1]; imax := 1;
  for i := 2 step 1 until n do if a[i] > max then
    begin max := a[i]; imax := i end of maximum search;
  if max ≠ min then
REDUCTION:
  begin comment Note that the identity $a_i = \sum_{j=1}^{n} m_{ij}a_j$
    holds at each stage of the reduction;
  q := max ÷ min; a[imax] := max := max − q × min;
  for j := 1 step 1 until n do
    M[imax, j] := M[imax, j] − q × M[imin, j];
DIVIDES: go to if max = 0 then MINIMUM else MAXIMUM
  end of the reduction. Note that if max ≠ 0 then max now con-
  tains the new nonzero minimum.
If max = min then we are ready with the results;
for j := 1 step 1 until n do x[j] := M[imin, j];
Euclidean := min
end of procedure Euclidean
```

REFERENCE

1. Blankinship, W. A. A new version of the Euclidean algorithm. *Amer. Math. Mon. 70* (1963), 742–745.

CERTIFICATION OF ALGORITHM 237 [A1]
GREATEST COMMON DIVISOR [J. E. L. Peck, *Comm. ACM 7* (Aug. 1964), 481]

T. A. Bray (Recd. 8 Sept. 1964)
Boeing Scientific Research Laboratories, Seattle, Washington

This procedure was translated into the Fortran IV language and tested on the Univac 1107. No corrections were required and the procedure gave correct results for all cases tested.

ALGORITHM 238
CONJUGATE GRADIENT METHOD [F4]

C. M. Reeves (Recd. 18 Nov. 1963)
Electronic Computing Lab., Univ. of Leeds, England

```
procedure conjugate gradients (x, r, n, matmult);
value n; real array x, r; integer n; procedure matmult;
comment The method of conjugate gradients [cf: BECKMAN,
  F. S. Mathematical Methods for Digital Computers. Ch. 4, Ralston,
  A., and Wilf, H. S., (EDS.), Wiley 1960.] is applied to solve the
  equations Ax = b where A is a general nonsingular matrix of
  order n, and x and b are vectors. At entry x contains an initial
  approximation to the solution, and r contains b, the vector of
  constants. Both x and r have bounds [1:n]. Up to n+1 iterations
  are carried out and at exit the solution is in x and the
  corresponding residuals r = b - Ax are in r.
    The procedure matmult has the following heading, with
  semicolons which must now be omitted:
  procedure matmult (transpose, dat, res)
  Boolean transpose real array dat, res
  comment The datum vector dat is premultiplied by the
    matrix B and the result formed in res where, denoting the
    transpose of A by At,

              B = if transpose then At else A

  The body of matmult will depend upon whether A is stored on
  magnetic tape, and whether all or only its nonzero elements
  are stored. The products should be accumulated in double
  precision, if possible.;
begin integer iterations; real alpha, beta, At r sq;
  real array p, temp [1:n];
  real procedure dot (u, v);
  real array u, v;
  comment dot is the scalar product of the vectors u and v;
  begin integer i; real sum; sum := 0;
    for i := 1 step 1 until n do sum := sum + u[i] × v[i];
    dot := sum
  end of dot;
  procedure combine (f) plus: (c) times: (g) to form: (h);
  value c;
  real c; real array f, g, h;
  comment f + cg is formed in h;
  begin integer i;
    for i := 1 step 1 until n do h[i] := f[i] + c × g[i]
  end of combine;
Start:
  for iterations := 0 step 1 until n do
  begin if iterations = 0
    then begin matmult (false, x) in : (temp);
        combine (r, -1, temp) in : (r);
        matmult (true, r) in : (p);
        At r sq := dot (p, p);
      end of forming r = b - Ax, p = At r, and At r sq
    else begin matmult (true, r) giving At r in : (temp);
        beta := dot (temp, temp)/At r sq;
        combine (temp, beta, p) in : (p);
        At r sq := beta × At r sq
      end;
    if At r sq = 0 then go to finish;
    matmult (false, p) giving Ap in : (temp);
    alpha := dot (temp, temp);
    if alpha = 0 then go to finish;
    alpha := dot (r, temp)/alpha;
    combine (x, alpha, p) in : (x);
    combine (r, -alpha, temp) in : (r)
  end of iterative loop;
finish :
end of conjugate gradients;
```

ALGORITHM 239
FREE FIELD READ [I5]

W. M. McKeeman (Recd. 12 Dec. 63 and 1 May 1964)
Computation Center, Stanford University, Stanford, Calif.

```
procedure inreal (channel, destination); value channel;
  integer channel; real destination;
begin comment Each invocation of inreal will read one ⟨number⟩
  [Revised Report ··· ALGOL 60, section 2.5.1] from the input
  medium designated by the parameter channel and convert it
  into the internal machine representation appropriate for real
  numbers. Successive data values within the data string are
  separated by the blank character ␣. Integer values from the
  input medium are converted into values of type real. A nonlocal
  procedure error is invoked whenever a non-⟨number⟩ is encountered
  in the input string. The action of error is left undefined;
  real sig, fp, d;
  integer esig, ep, ip, ch;
  integer procedure CHAR;
  begin comment The value of CHAR is the integer representing
    the next character from the input string. insymbol
    is defined in the "Report on Input-Output Procedure for ALGOL
    60," ALGOL Bull. No. 16 (May 1964), 9-13; Comm. ACM, to appear.
    Characters occurring in the second parameter of insymbol
    are mapped onto the integers corresponding to their
    position, left-to-right, within the string. Other basic symbols
    map onto the integer 0.
      The present procedure inreal differs from the inreal of
    the referenced Report on Input-Output Procedures for
    ALGOL 60 in the following ways:
      (a) The report does not specify what values may be presented
    in its inreal, only that whatever is presented will be
    assigned to the second parameter of inreal. I demand that a
    ⟨number⟩ be presented.
      (b) No separator of values on the foreign medium is specified.
    I demand an ALGOL string blank.;
    real c;
    insymbol (channel, '0123456789.−+10␣'    , c);
    if c ≦ 0 then error; comment an illegal character;
    CHAR := c − 1
  end CHAR;
  integer procedure unsigned integer;
  begin comment ⟨unsigned integer⟩ ::= ⟨digit⟩ | ⟨unsigned
    integer⟩ ⟨digit⟩;
    integer u;
    u := 0;
  K: u := 10 × u + ch;
    ch := CHAR;
    if ch < 10 then go to K;
    unsigned integer := u
  end unsigned integer;
  sig := 1.0; ep := 0; fp := 0;
L: ch := CHAR;
  if ch = 14 then go to L; comment suppress initial blanks;
  comment ⟨number⟩ ::= ⟨unsigned number⟩ | +⟨unsigned
    number⟩ | −⟨unsigned number⟩;
  if ch = 12 then ch := CHAR
  else if ch = 11 then
  begin comment 12 = "+" and 11 = "−";
    sig := −1.0;
    ch := CHAR
  end;
  comment ⟨unsigned number⟩ ::= ⟨decimal number⟩ | ⟨exponent
    part⟩ | ⟨decimal number⟩⟨exponent part⟩;
  if ch ≦ 10 then
  begin comment ⟨decimal number⟩ ::= ⟨unsigned integer⟩ |
    ⟨decimal fraction⟩ | ⟨unsigned integer⟩⟨decimal fraction⟩;
    if ch < 10 then ip := unsigned integer else ip := 0;
    if ch = 10 then
    begin comment ⟨decimal fraction⟩ ::= .⟨unsigned integer⟩;
      ch := CHAR;
      if ch ≧ 10 then error; comment a digit must follow the
        ".";
      fp := 0; d := 0.1;
    M: fp := fp + ch × d;
      d := d × 0.1;
      comment a table ot reciprocal powers of ten is preferable
        to the statement d := d × 0.1;
      ch := CHAR;
      if ch < 10 then go to M
    end
  end else if ch = 13 then ip := 1 else error;
  if ch = 13 then
  begin comment ⟨exponent part⟩ ::= 10⟨integer⟩;
    ch := CHAR; esig := 1;
    comment ⟨integer⟩ ::= ⟨unsigned integer⟩ | +⟨unsigned
      integer⟩ | −⟨unsigned integer⟩;
    if ch = 12 then ch := CHAR
    else if ch = 11 then
    begin comment negative exponent;
      esig := −1;
      ch := CHAR
    end;
    if ch < 10 then ep := unsigned integer × esig else error
  end;
  if ch ≠ 14 then error; comment the required "␣" separator;
  destination := sig × (ip+fp) × 10.0 ↑ ep
end inreal
```

ALGORITHM 240
COORDINATES ON AN ELLIPSOID [Z]

Egon Dorrer (Recd. 8 Jan. 1964 and, rev., 19 May 1964)
Inst. f. Photogrammetrie, Techn. Hochschule, Munich,
Germany

```
procedure GEODH 1 (L, B, AZ, S, EPS, lim, A, F, FAIL);
  value S, EPS, lim, A, F;  real L, B, AZ, S, EPS, A, F;
  integer lim;  label FAIL;
comment  GEODH 1 solves the problem of transferring of geo-
  graphical coordinates on an arbitrary ellipsoid of rotation. A is
  the radius of the equator, F is the flattening of the meridian
  ellipse. Before executing GEODH 1, L and B are longitude and
  latitude of a point P1 on the ellipsoid. AZ is the azimuth at P1 ,
  measured from north, of the geodesic to another point P2 , and
  S is the distance from P1 to P2 , measured in the same unit as A.
  After execution of GEODH 1, L and B represent the longitude
  and latitude of P2 , and AZ is the final azimuth of the geodesic
  at P2 . Here L, B, AZ, and EPS are measured in radians. Arbi-
  trarily long distances S can be used, even more than the circum-
  ference. However, the geodesic must not cross the poles or come
  near to them. The problem has been solved by reiterated use of
  the Runge-Kutta method to solve the system of the three first-
  order differential equations of the geodesic on a rotation ellip-
  soid. EPS is the convergence parameter, e.g. a small number
  indicating the desired accuracy, normally 10^-8 or 10^-9. lim is the
  upper limit on iterations, it depends on EPS, and should not be
  chosen greater than 11 or 12. If lim is reached, computations
  stop, and the FAIL exit is used:
begin
  real EP2, Lo, Bo, AZo, LL, BL, AZL, So, SL, H, DL, DB, DAZ,
    KL, KB, KAZ, BQ, AZQ, W, H1, T, SINBQ;
  integer i, n, j, z;
  array D[1:4];  D[1] := D[4] := 1;  D[2] := D[3] := 2;
  EP2 := F × (2 − F);  Lo := L:  Bo := B;  AZo := AZ;
  n := 1;  z := 0;
ITERATION: if z = lim then go to FAIL;
  So := 0;  LL := Lo;  BL := Bo;  AZL := AZo;
  for i := 1 step 1 until n do
  begin
    SL := S × i/n;  H := (SL − So)/A;
    DL := DB := DAZ := KL := KB := KAZ := 0;
    for j := 1 step 1 until 4 do
    begin
      T := D[j];
      BQ := BL + DB/T;  AZQ := AZL + DAZ/T;  SINBQ :=
        sin(BQ);
      W := 1 − EP2 × SINBQ × SINBQ;  H1 := H × sqrt(W);
      DL := H1 × sin(AZQ)/cos(BQ);
      DB := H1 × W × cos(AZQ)/(1 − EP2);
      DAZ := DL × SINBQ;
      KL := KL + DL × T;  KB := KB + DB × T;  KAZ :=
        KAZ + DAZ × T
    end j;
    So := SL;  LL := LL + KL/6;  BL := BL + KB/6;
      AZL := AZL + KAZ/6
  end i;
  DL := LL − L;  DB := BL − B;  DAZ := AZL − AZ;
  L := LL;  B := BL;  AZ := AZL;
  if abs(DAZ) < EPS/sin(S/A) ∧ (abs(DL) < EPS/cos(B) ∨
    abs(DB) < EPS) then go to END;
  z := 1 + z;  n := 2 × n;
  go to ITERATION;
  END:
end GEODH 1
```

ALGORITHM 241
ARCTANGENT [B1]

K. W. Mills (Recd. 21 Nov. 1963)
Computing Centre, University of Adelaide, So. Australia

```
real procedure arg(x, y) exit: (error);  value x, y;   real x, y;
  label error;
comment  This procedure calculates the argument of a complex
  number x + iy, using a method which is substantially that of
  E. G. Kogbetliantz, IBM J. Research Develop., Jan. 1958, pp.
  43-53. The result lies in the interval [-π, π] and the exit error
  is provided for the case when x = y = 0. The procedure is es-
  sentially an ALGOL program for the calculation of the arctan-
  gent. arctan(y) is obtained most conveniently by calling the pro-
  cedure with x = 1;
begin
  array ct, csc2[2:5], tn[1:4];  integer k;  real w, v, pi, r, z;
  pi := 3.1415926536;  if x = 0 then
  begin
    if y = 0 then go to error;
L1: arg := pi/2 × sign(y);  go to exit
  end;
  w := y/x;  v := abs(w);
  if v > 1.34₁₀8 then go to L1;
  if v < 2.13₁₀-22 then r := w else
  begin
    ct[2] := tn[4] := 2.7474774195;
    ct[3] := tn[3] := 1.1917535926;
    ct[4] := tn[2] := .57735026919;
    ct[5] := tn[1] := .17632698071;
    csc2[2] := 8.548632169;
    csc2[3] := 2.420276626;
    csc2[4] := 1.333333333;
    csc2[5] := 1.031091204;
    if v < tn[1] then
    begin
      k := 1;  z := .16363636364 × v
    end
    else
    begin
      for k := 2 step 1 until 4 do if v < tn[k] then go to L3;
      k := 5;
L3:   z := .16363636364 × (ct[k]-csc2[k]/(v+ct[k]))
    end;
    r := (pi × (k-1)/9 + z/(z × z + .216649136 - .00270998425/
      (z × z + .0511194591))) × sign(w)
  end;
  arg := if x > 0 then r else
         if y = 0 then r + pi else
         r + pi × sign(y);
  exit:
end arg
```

ALGORITHM 242
PERMUTATIONS OF A SET WITH REPETITIONS [G6]

T. W. Sag (Recd. 10 Feb. 1964 and 19 June 1964)
Math. Dept., Manchester U., Manchester, England

```
procedure PERMUTATION (X, K, j, process);
  array X; integer array K; integer j; procedure process;
comment PERMUTATION generates all the distinct permuta-
  tions of an array of numbers consisting of K[1] numbers equal to
  X[1], K[2] numbers equal to X[2], ··· , K[j] numbers equal to
  X[j]. The K[i]'s must be positive integers. Each permutation is
  stored in the array Y and processed according to the user's wish
  by the procedure process before the next permutation is gen-
  erated.
    {The procedure is more efficient if the sequence K[i] is mono-
  tone decreasing.—Ref.};
begin
  real x; integer M, N, i; array B[1:K[j]];
  procedure permutation (x, M, N, j, B, process);
  real x; integer M, N, j; array B; procedure process;
  begin
        real A; integer i, KK, N1, N2, j1;
          integer array J[1:N+1];
        array Y[1:N+M]; N2 := N + M;
        if M = 0 then go to 1;
        for i := N + 1 step 1 until N2 do Y[i] := x;
    1:  for i := 1 step 1 until N do J[i] := i;
        J[N+1] := N2 + 1; j1 := j − 1; KK := N;
    2:  for i := 1 step 1 until KK do Y[J[i]] := B[i];
        if j1 ≦ 1 then begin process(Y); go to 3 end;
        A := X[j1−1]; N1 := K[j1--1];
        permutation (A, N1, N2, j1, Y, process);
    3:  for i := 1 step 1 until N do
        begin
              Y[J[i]] := x; J[i] := J[i] + 1;
              if J[i] − J[i+1] + 1 ≦ 0 then go to 4 else go to 5;
          4:  KK := i; go to 2;
          5:  J[i] := i
        end
  end of permutation;
  if j = 1 then begin x := X[1]; M := 0; go to 1 end;
  x := X[j−1]; M := K[j−1];
1: for i := 1 step 1 until K[j] do B[i] := X[j];
   permutation (x, M, K[j], j, B, process);
end of PERMUTATION
```

ALGORITHM 243
LOGARITHM OF A COMPLEX NUMBER [B3]
REWRITE OF ALGORITHM 48 [*Comm. ACM 4* (Apr. 1961), 179; *5* (Jun. 1962), 347; *5* (Jul. 1962), 391; *7* (Aug. 1964), 485]

David S. Collens [Recd. 24 Jan. 1964 and 1 Jun. 1964]
Computer Laboratory, The University, Liverpool, 3, England

This procedure was tested using the DEUCE ALGOL Compiler and a small sample of the test data and results are given below.

```
procedure LOGC (a, b, c, d, FAIL);  value a, b, FAIL;  real
  a, b, c, d;  label FAIL;
comment  This procedure computes the number c + di which is
  equal to the principal value of the natural logarithm of a + bi,
  i.e. such that −π < d ≦ +π. A nonlocal label must be supplied
  as a parameter of the procedure, to be used as an exit when the
  real part of the result becomes −∞. Where required in the body
  of the procedure the numerical values for π, π/2, and the log-
  arithm of the square root of 8 are provided;
  if a = 0 ∧ b = 0 then go to FAIL
  else
  begin
    real e, f;
    e := 0.5 × a;  f := 0.5 × b;
    if abs(e) < 0.5 ∧ abs(f) < 0.5 then
    begin
      c := abs(2×a) + abs(2×b);
      d := 8 × a/c × a + 8 × b/c × b;
      c := 0.5 × (ln(c)+ln(d)) −1.03972077084
    end
    else
    begin
      c := abs(0.5×e) + abs(0.5×f);
      d := 0.5 × e/c × e + 0.5 × f/c × f;
      c := 0.5 × (ln(c)+ln(d)) + 1.03972077084
    end;
    d := if a ≠ 0 ∧ abs(e) ≧ abs(f) then arctan(b/a) +
      (if sign(a)≠−1 then 0 else if sign(b)≠−1 then
        3.14159265359  else  −3.14159265359)  else -- arctan(a/b)
        + 1.57079632679 × sign(b)
  end LOGC
```

TEST OF LOGC

a	*b*	*c*	*d*
−2	−2	+1.039721	−2.356194
−2	+1	+0.804719	+2.677945
−1	−1	+0.346573	−2.356194
−1	+0	+0.000000	+3.141593
+0	−2	+0.693147	−1.570796
+0	−1	+0.000000	−1.570796
+0	+1	+0.000000	+1.570796
+0	+2	+0.693147	+1.570796
+1	−1	+0.346573	−0.785398
+1	+0	+0.000000	+0.000000
+2	−2	+1.039721	−0.785398
+2	+1	+0.804719	+0.463647

CERTIFICATION OF ALGORITHM 243 [B3]
LOGARITHM OF A COMPLEX NUMBER [David S. Collens *Comm. ACM 7*(Nov. 1964), 660]

J. Boothroyd (Recd. 18 Jan. 1965)
Computing Centre, U. of Tasmania, Hobart, Tasmania

With the label parameter *FAIL* removed from the value list to accommodate a restriction of Elliott 503 Algol, the algorithm was successfully run on an Elliott 503, using the data test cases published with the algorithm. The constants in the algorithm were rounded to nine significant decimal digits, and this probably explains the two differences between the results obtained and those published, namely:

a	*b*	*c*	*d*
−1	−1	0.346574	
2	1		0.463648

ALGORITHM 244
FRESNEL INTEGRALS [S20]

HELMUT LOTSCH* (Recd. 27 May 64 and 11 Jun. 64)
W. W. Hansen Laboratories, Stanford U., Stanford, Calif.
AND
MALCOLM GRAY†
Computation Center, Stanford U., Stanford, Calif.
(* now at Northrup Space Laboratories, Hawthorne, Calif.)
(† now at The Boeing Company, Seattle, Wash.)

procedure *FRESNEL* (*w*, *eps*, *C*, *S*); **value** *w*, *eps*; **real** *w*, *eps*, *C*, *S*;

comment This procedure computes the Fresnel sine and cosine integrals $C(w) = \int_0^w \cos[(\pi/2)t^2]\,dt$ and $S(w) = \int_0^w \sin[(\pi/2)t^2]$ *dt*. It is a modification of Algorithm 213 (*Comm. ACM*, *6* (Oct. 1963), 617) such that the accuracy, expressed by *eps*, is improved. *eps* can arbitrarily be chosen up to $eps = {}_{10} - 6$ for a computer with sufficient word length as, for example, the Burroughs B5000 which has 11–12 significant digits. Referring to the formulas of Algorithm 213: if $|w| < \surd(26.20/\pi)$ the series expansions $C(w)$ and $S(w)$ are terminated when the absolute value of the relative change in two successive terms is $\leqq eps$. If $|w| \geqq \surd(26.20/\pi)$ the series $Q(x)$ and $P(x)$ are terminated when the absolute value of the terms is $\leqq eps/2$. However, this truncation point is not necessarily valid for the range $\surd(26.20/\pi) \leqq |w| < \surd(28.50/\pi)$ when $eps = {}_{10} - 6$, since the asymptotic series must be terminated before arriving at the minimum. In this range the ignored terms of the series expansions are $< 3_{10} - 6$, and for larger arguments $< {}_{10} - 6$. This accuracy may be improved if desired: the switch-over point from the regular to the asymptotic series expansions has to be displaced to larger arguments;

```
begin
  real x, x2, term; integer n;
  if abs(w) ≦ ₁₀ − 12 then
    begin C := S := 0; go to aend end
  else x := w × w/0.636619772368;
  x2 := − x × x; if x ≧ 13.10 then go to asympt;
  begin
    real frs, frsi;
    frs := x/3; n := 5; term := x × x2/6;
    frsi := frs + term/7;
loops: if abs((frs−frsi)/frs) ≦ eps then go to send;
    frs := frsi; term := term × x2/(n×n−n);
    frsi := frs + term/(2×n+1);
    n := n + 2; go to loops;
send: S := frsi × w
  end;
  begin
    real frc, frci;
    frc := 1; n := 4; term := x2/2;
    frci := 1 + term/5;
loopc: if abs((frc−frci)/frc) ≦ eps then go to cend;
    frc := frci; term := term × x2/(n×n−n);
    frci := frc + term/(2×n+1);
    n := n + 2; go to loopc;
cend: C := frci × w
  end;
  go to aend;
asympt:
  begin
    real s1, s2, half, temp; integer i;
    x2 := 4 × x2; term := 3/x2; s1 := 1 + term; n := 8;
    for i := 1 step 1 until 6 do
    begin
      n := n + 4;
      term := term × (n−7) × (n−5)/x2;
      s1 := s1 + term;
      if abs(term) ≦ eps/2 then go to next
    end i;
next: term := s2 := 0.5/x; n := 4;
    for i := 1 step 1 until 6 do
    begin
      n := n + 4;
      term := term × (n−5) × (n−3)/x2;
      s2 := s2 + term;
      if abs(term) ≦ eps/2 then go to final
    end i;
final: half := if w < 0 then −0.5 else 0.5;
    term := cos(x); temp := sin(x); x2 := 3.14159265359 × w;
    C := half + (temp×s1−term×s2)/x2;
    S := half − (term×s1+temp×s2)/x2
  end;
aend:
end FRESNEL
```

ALGORITHM 245
TREESORT 3 [M1]

ROBERT W. FLOYD (Recd. 22 June 1964 and 17 Aug. 1964)
Computer Associates, Inc., Wakefield, Mass.

```
procedure TREESORT 3 (M, n);
  value n;  array M;  integer n;
comment TREESORT 3 is a major revision of TREESORT
  [R. W. Floyd, Alg. 113, Comm. ACM 5 (Aug. 1962), 434] sug-
  gested by HEAPSORT [J. W. J. Williams, Alg. 232, Comm.
  ACM 7 (June 1964), 347] from which it differs in being an in-place
  sort. It is shorter and probably faster, requiring fewer compari-
  sons and only one division. It sorts the array M[1:n], requiring
  no more than 2 × (2↑p−2) × (p−1), or approximately 2 ×
  n × (log₂(n)−1) comparisons and half as many exchanges in
  the worst case to sort n = 2↑p − 1 items. The algorithm is
  most easily followed if M is thought of as a tree, with M[j÷2]
  the father of M[j] for 1 < j ≦ n;
begin
  procedure exchange (x,y);  real x,y;
    begin real t;  t := x;  x := y;  y := t
    end exchange;
  procedure siftup (i,n);  value i, n;  integer i, n;
  comment  M[i] is moved upward in the subtree of M[1:n] of
    which it is the root;
  begin real copy;  integer j;
    copy := M[i];
  loop: j := 2 × i;
    if j ≦ n then
    begin if j < n then
        begin if M[j+1] > M[j] then j := j + 1 end;
      if M[j] > copy then
        begin M[i] := M[j];  i := j;  go to loop end
    end;
    M[i] := copy
  end siftup;
  integer i;
  for i := n÷2 step −1 until 2 do siftup (i,n);
  for i := n step −1 until 2 do
  begin siftup (1,i);
    comment  M[j÷2] ≧ M[j] for 1 < j ≦ i;
    exchange (M[1], M[i]);
    comment  M[i:n] is fully sorted;
  end
end TREESORT 3
```

CERTIFICATION OF ALGORITHM 245 [M1]
TREESORT 3 [Robert W. Floyd, *Comm. ACM 7* (Dec. 1964), 701]

PHILIP S. ABRAMS (Recd. 14 Jan. 1965)
Computation Center, Stanford University, Stanford, California

The procedure *TREESORT* 3 was translated into B5000 Extended ALGOL and tested on the Burroughs B5500. Tests were run on arrays of length 50 to 1000 in steps of 50. For each array size, 50 random arrays were generated, sorted, timed and checked for sequencing. No corrections were required and the procedure gave correct results for all cases tested.

exchange is unnecessary as a separate procedure, since it is used at only one place in *TREESORT* 3. Sorts were found to run significantly faster when the body of *exchange* was inserted in the appropriate place, than when run with the algorithm as published.

CERTIFICATION OF ALGORITHM 245 [M1]
TREESORT 3 [Robert W. Floyd, *Comm. ACM 7* (Dec. 1964), 701]: PROOF OF ALGORITHMS—A NEW KIND OF CERTIFICATION

RALPH L. LONDON* (Recd. 27 Feb. 1969 and 8 Jan. 1970)
Computer Sciences Department and Mathematics Research Center, University of Wisconsin, Madison, WI 53706

* This work was supported by NSF Grant GP-7069 and the Mathematics Research Center, US Army under Contract Number DA-31-124-ARO-D-462.

ABSTRACT: The certification of an algorithm can take the form of a proof that the algorithm is correct. As an illustrative but practical example, Algorithm 245, *TREESORT 3* for sorting an array, is proved correct.

KEY WORDS AND PHRASES: proof of algorithms, debugging, certification, metatheory, sorting, in-place sorting
CR CATEGORIES: 4.42, 4.49, 5.24, 5.31

Certification of algorithms by proof. Since suitable techniques now exist for proving the correctness of many algorithms [for example, 3–7], it is possible and appropriate to certify algorithms with a proof of correctness. This certification would be in addition to, or in many cases instead of, the usual certification. Certification by testing still is useful because it is easier and because it also provides, for example, timing data. Nevertheless the existence of a proof should be welcome additional certification of an algorithm. The proof shows that an algorithm is debuggged by showing conclusively that no bugs exist.

It does not matter whether all users of an algorithm will wish to, or be able to, verify a sometimes lengthy proof. One is not required to accept a proof before using the algorithm any more than one is expected to rerun the certification tests. In both cases one could depend, in part at least, upon the author and the referee.

As an example of a certification by proof, the algorithm *TREESORT 3* [2] is proved to perform properly its claimed task of sorting an array $M[1:n]$ into ascending order. This algorithm has been previously certified [1], but in that certification, for example, no arrays of odd length were tested. Since *TREESORT 3* is a fast practical algorithm for in-place sorting and one with sufficient complexity so that its correctness is not immediately apparent, its use as the example is more than an abstract exercise. It is an example of considerable practical importance.

Outline of TREESORT 3 and method of proof. The algorithm is most easily followed if the array is viewed as a binary tree. $M[k \div 2]$ is the parent of $M[k]$, $2 \leq k \leq n$. In other words the children of $M[j]$ are $M[2j]$ and $M[2j+1]$ provided one or both of the children exist.

The first part of the algorithm permutes the M array so that for a segment of the array, each parent is larger than both of the children (one child if the second does not exist). Each call of the auxiliary procedure *siftup* enlarges the segment by causing one more parent to dominate its children. The second part of the algorithm uses *siftup* to make the parents larger over the whole array, exchanges $M[1]$ with the last element and repeats on an array one element shorter. The above statements are motivation and not part of the formal proof.

That *TREESORT 3* is correct is proved in three parts. First the procedure *siftup* is shown to perform as it is formally defined below. Then the body of *TREESORT 3*, which uses *siftup* in two ways, is shown to sort the array into ascending order. (The proof of the procedure *exchange* is omitted.) The proofs are by a method described in [3, 4, 7]: assertions concerning the progress of the computation are made between lines of code, and the proof consists of demonstrating that each assertion is true each time control reaches that assertion, under the assumption that the previously encountered assertions are true. Finally termination of the algorithm is shown separately.

The lines of the original algorithm have been numbered and the assertions, in the form of program comments, are numbered correspondingly. The numbers are used only to refer to code and to assertions and have no other significance. One extra begin-end pair has been inserted into the body of *TREESORT 3* in order that the control points of two assertions (3.1 and 4.1) could be distinguished. In *siftup* the assertions 10.1 and 10.2 express the correct result; in the body of *TREESORT 3* the assertions 9.3 and 9.4 do likewise.

Definition of siftup and notation. We now define formally the procedure *siftup(i,n)*, where n is a formal parameter and not the length of the array M. Let $A(s)$ denote the set of inequalities $M[k \div 2] \geq M[k]$ for $2s \leq k \leq n$. (If $s > n \div 2$, then $A(s)$ is a vacuous statement.) If $A(i+1)$ holds before the call of *siftup(i,n)* and if $1 \leq i \leq n \leq$ *array size*, then after *siftup(i,n)*:

(1) $A(i)$ holds;

(2) the segment of the array $M[i]$ through $M[n]$ is permuted; and

(3) the segment outside $M[i]$ through $M[n]$ is unaltered.

In order to prove these properties of *siftup*, some notation is required. The formal parameter i will be changed inside *siftup*. Since i is called by value, that change will be invisible outside *siftup*. Nevertheless it is necessary to use the initial value of i as well as the current value of i in the proof of *siftup*. Let i_0 denote the value of i upon entry to *siftup*.

Similarly let M_0 denote the M array upon entry to *siftup*. The notation "$M = p(M_0)$ with $M := copy$" means "if $M[i] := copy$ were done, M is some permutation of M_0 as described in (2) and (3) of the definition of *siftup*." "$M = p(M_0)$" means the same without the reference to $M[i] := copy$ being done.

Code and assertions for siftup.

```
0    procedure siftup(i, n);  value i, n;  integer i, n;
1    begin real copy;  integer j;
       comment
         1.1: $1 \leq i_0 = i \leq n \leq$ array size
         1.2: $A(i_0+1)$
         1.3: $M = p(M_0)$;
2      copy := M[i];
3      loop: j := 2 × i;
       comment
         3.1: $i \leq n$
         3.2: $2i = j$
         3.3: $i = i_0$ or $i \geq 2i_0$
         3.4: $M = p(M_0)$ with $M[i] := copy$
         3.5: $A(i_0)$ or ($i = i_0$ and $A(i_0+1)$)
         3.6: $M[i \div 2] > copy$ or $i = i_0$
         3.7: $M[i \div 2] \geq M[i]$ or $i = i_0$;
4      if j ≤ n then
5      begin if j < n then
6a       begin if M[j+1] > M[j] then
6b         j := j + 1 end;
         comment
           6.1: $i = j \div 2$
           6.2: $2i \leq j \leq n$
           6.3: $i = i_0$ or $i \geq 2i_0$
           6.4: $M = p(M_0)$ with $M[i] := copy$
           6.5: $A(i_0)$ or ($i = i_0$ and $A(i_0+1)$)
           6.6: $M[i \div 2] > copy$ or $i = i_0$
           6.7: $M[i \div 2] \geq M[i]$ or $i = i_0$
           6.8: ($2i < n$ and $M[j] = \max(M[2i], M[2i+1])$) or
                ($2i = n$ and $M[j] = M[n]$)
           6.9: $M[i] \geq M[j]$ or $i = i_0$;
7        if M[j] > copy then
8a       begin M[i] := M[j];
           comment
             8.1: $i = i_0$ or $i \geq 2i_0$
             8.2: $2i \leq j \leq n$
             8.3: $M[j \div 2] = M[i] = M[j] > copy$
             8.4: $M[i \div 2] \geq M[j]$ or $i = i_0$
             8.5: $M = p(M_0)$ with $M[j] := copy$
             8.6: $A(i_0)$;
8b         i := j;
           comment
             8.7: $i \geq 2i_0$
             8.8: $i = j \leq n$
             8.9: $M[i \div 2] > copy$
             8.10: $M[i \div 2] \geq M[i]$
             8.11: $M = p(M_0)$ with $M[i] := copy$
             8.12: $A(i_0)$;
8c       go to loop end
9      end;
       comment
         9.1: $M[j] \leq copy$ if reached from 7 or
              $2i = j > n$ if reached from 4;
10     M[i] := copy;
       comment
         10.1: $M = p(M_0)$
         10.2: $A(i_0)$;
11   end siftup;
```

Verification of the assertions of siftup. Reasons for the truth of each assertion follow:

1.1–1.2: Assumptions for using *siftup*.

1.3: p is the identity permutation.

3.1–3.7: If reached from 2,
    3.1: 1.1.
    3.2: 3.
    3.3, 3.5–3.7: $i = i_0$ by 1.1. 3.5 also requires 1.2.
    3.4: 1.3 and 2.
  If reached from 8, respectively, 8.8, 3, 8.7, 8.11, 8.12, 8.9 and 8.10.

6.1: At 3.2 $j = 2i$ and by 6b, j might be $2i + 1$. $i = j \div 2$ in either case.

6.2: After 4, $j \leq n$. j is altered from 3.1 to 6.2 only at 6b. Before 6b, $j < n$ by 5. Hence $j \leq n$ at 6.2. $2i \leq j$ by 6.1.

6.3–6.7: 3.3–3.7, respectively.

6.8: If 4 is **true** and 5 is **false**, $j = 2i = n$ (using 3.2) so the second clause of 6.8 holds. If 4 is **true** and 5 is **true**, then at 6a, $2i = j < n$ (using 3.2) so $M[j+1] = M[2i+1]$ is defined. Now at 6.8, $j = 2i$ or $j = 2i+1$. In either case, by 6a and 6b, the first clause of 6.8 holds.

6.9: By 6.5 $i \neq i_0$ gives $A(i_0)$. $2i_0 \leq 2i \leq j \leq n$ by 6.3 and 6.2. Hence $A(i_0)$ and 6.1 give $M[i] = M[j \div 2] \geq M[j]$.

8.1: 6.3.

8.2: 6.2.

8.3: $i = j \div 2$ by 6.1, $M[i] = M[j]$ by 8a and $M[j] > copy$ by 7.

8.4: 6.7 and 6.9.

8.5: 6.4 requires that $M[i]$ be replaced by *copy*. Since $M[i] = M[j]$ by 8a, $M[j]$ may equally well be replaced with *copy*. 8.1 and 8.2 give $i_0 \leq i \leq n$ so that the change to M at 8a is in the segment $M[i_0]$ through $M[n]$.

8.6: By 8a and if 6.8 (first clause) holds, $M[i] \geq M[2i]$ and $M[i] \geq M[2i+1]$. By 8a and if 6.8 (second clause) holds, $M[i] = M[j] = M[n] = M[2i]$ and $M[2i+1]$ does not exist for this call of *siftup*. $A(i_0+1)$ holds at 6.5 since $A(i_0)$ implies $A(i_0+1)$. If $i = i_0$, $A(i_0+1)$ and the relations above on $M[i]$ give $A(i_0)$. If $i \neq i_0$, then 8a, 8.4, $A(i_0)$ at 6.5 and the relations above on $M[i]$ give $A(i_0)$ at 8.6.

8.7: 8b, 8.1 and 8.2.

8.8: 8b and 8.2.

8.9: 8b and 8.3.

8.10: At 8.6, $2i_0 \leq j \leq n$ by 8.1 and 8.2. Hence by 8.6, $M[j \div 2] \geq M[j]$. Use 8b on $M[j \div 2] \geq M[j]$.

8.11: 8b and 8.5.

8.12: 8.6.

9.1: 9.1 is reached only if 7 is **false** or if 4 is **false.** $2i = j$ by 3.2.

10.1–10.2: If reached from 7,

10.1: 6.4 and 10. (6.2 and 6.3 give $i_0 \leq i \leq n$ ensuring the change to M at 10 is in the segment $M[i_0]$ through $M[n]$.)

10.2: By 10, 9.1, 6.2 and 6.8, $M[i] = copy \geq M[j] \geq M[2i]$ and, if $M[2i+1]$ exists, $M[j] \geq M[2i+1]$. If $i = i_0$, 10.2 follows as in 8.6. If $i \neq i_0$, 6.6 and 10 give $M[i \div 2] > copy = M[i]$. $A(i_0)$ at 6.5 now gives $A(i_0)$ at 10.2.

If reached from 4,

10.1: 3.4 and 10. (3.1 and 3.3 give $i_0 \leq i \leq n$.)

10.2: $2i > n$ means no relations in $A(i_0)$ of the form $M[i] \geq \cdots$. If $i = i_0$, 3.5 gives 10.2. If $i \neq i_0$, 3.6 and 10 give $M[i \div 2] > copy = M[i]$. $A(i_0)$ at 3.5 now gives 10.2.

Code and assertions for the body of TREESORT 3.

```
0   integer i;
    comment
      0.1: $A(n \div 2+1)$;
1   for i := n÷2 step −1 until 2 do
2   begin
      comment
        2.1: $A(i+1)$
        2.2: Assumptions of siftup satisfied;
3     siftup(i,n);
      comment
        3.1: $A(i)$;
4   end;
    comment
      4.1: $M[p] \leq M[p+1]$ for $n + 1 \leq p \leq n - 1$
      4.2: $A(2)$, i.e. $M[k \div 2] \geq M[k]$ for $4 \leq k \leq n$;
5   for i := n step −1 until 2 do
6   begin
      comment
        6.1: $M[p] \leq M[p+1]$ for $i + 1 \leq p \leq n - 1$
        6.2: $M[k \div 2] \geq M[k]$ for $4 \leq k \leq$ i
        6.3: $M[i+1] \geq M[r]$ for $1 \leq r \leq i$
        6.4: Assumptions of siftup satisfied;
7     siftup (1,i);
      comment
        7.1: $M[p] \leq M[p+1]$ for $i + 1 \leq p \leq n - 1$
        7.2: $M[k \div 2] \geq M[k]$ for $2 \leq k \leq i$
        7.3: $M[1] \geq M[r]$ for $2 \leq r \leq i$
        7.4: $M[i+1] \geq M[1]$;
8     exchange (M[1], M[i]);
      comment
        8.1: $M[i] \geq M[r]$ for $1 \leq r \leq i - 1$
        8.2: $M[p] \leq M[p+1]$ for $i \leq p \leq n - 1$
        8.3: $M[k \div 2] \geq M[k]$ for $4 \leq k \leq i - 1$;
9   end;
    comment
      9.1: $M[p] \leq M[p+1]$ for $2 \leq p \leq n - 1$
      9.2: $M[2] \geq M[1]$
      9.3: $M[p] \leq M[p+1]$ for $1 \leq p \leq n - 1$, i.e. M is fully
           ordered
      9.4: M is a permutation of $M_0$;
```

Verification of the assertions for the body of TREESORT 3. Reasons for the truth of each assertion follow:

0.1: Vacuous statement since $2(n \div 2+1) > n$.

2.1: If reached from 0.1, by 1 substitute $i = n \div 2$ in 0.1.
If reached from 3.1, by 1 substitute $i = i + 1$ in 3.1 to account for the change in i from 3.1 to 2.1.

2.2: 2.1, the bound on i implied by 1 and the array size being n.

3.1: 2.1 and the definition of *siftup*(i, n).

4.1: Vacuous statement.

4.2: If $n \geq 4$, 3 is executed; hence 3.1 with $i = 2$. If $n \leq 3$, vacuous statement.

6.1–6.3: If reached from 4.1,

6.1–6.2: By 5 substitute $i = n$ in 4.1 and 4.2.

6.3: Vacuous statement for $i = n$.

If reached from 8.1, by 5 substitute $i = i + 1$ in 8.2, 8.3 and 8.1, respectively.

6.4: 5 and 6.2, i.e. $A(2)$ for the subarray $M[1{:}i]$.

7.1: 6.1 and (3) of *siftup.*

7.2: 6.2 and (1) of *siftup.*

7.3: 7.2 noting that $M[1] = M[k \div 2]$ if $k = 2$ and using the transitivity of $\geq$.

7.4: Vacuous for $i = n$. Otherwise 6.3 for the appropriate r since by (2) of *siftup*, $M[1]$ at 7.3 is one of the $M[r]$, $1 \leq r \leq i$, at 6.3.

8.1: 7.3 with the changes caused by 8 (only $M[1]$ and $M[i]$ are altered by 8).

8.2: By 8 substitute $M[i]$ for $M[1]$ in 7.4; then 7.1 also holds for $p = i$.

8.3: 7.2 excluding only the one or two relations $M[1] \geq \cdots$, and the one relation $\cdots \geq M[i]$.

9.1–9.3: If $n \geq 2$, 8 is executed;

9.1: 8.2 with $i = 2$.

9.2: 8.1 with $i = 2$.

9.3: 9.1 and 9.2.

If $n \leq 1$, 9.1–9.3 are vacuous statements.

9.4: The only operations done to M are *siftup* and *exchange* all of which leave M as a permutation of M_0.

Proof of termination of TREESORT 3. Provided *siftup* and *exchange* terminate, it is clear that *TREESORT 3* terminates. Note that each parameter of *siftup* is called by value so that i is not changed in the body of the for loops.

The procedure *exchange* certainly terminates. In *siftup* the only possibility for an unending loop is from 3 to 8b and back to 3. Note that all changes to i (only at 8b) and to j (only at 3 and 6b) occur in this loop and that on each cycle of this loop both i and j are changed. By the test at 4, it is sufficient to show that j strictly increases in value. $i \geq 1$ means $2i > i$. At 8b, $j = i < 2i$ while at 3, $j = 2i$, i.e. $j(\text{at } 3) = 2i > i = j(\text{at 8b})$. Hence each setting to j

at 3 strictly increases the value of j. The only other setting to j (at 6b), if made, similarly increases the value of j.

References:

1. Abrams, P. S. Certification of Algorithm 245. *Comm. ACM 8* (July 1965), 445.
2. Floyd, R. W. Algorithm 245, TREESORT 3. *Comm. ACM 7* (Dec. 1964), 701.
3. Floyd, R. W. Assigning meanings to programs. Proc. of a Symposium in Applied Mathematics, Vol. 19—Mathematical Aspects of Computer Science, J. T. Schwartz (Ed.), American Math. Society, Providence, R. I., 1967, pp. 19–32.
4. Knuth, D. E. *The Art of Computer Programming, Vol. 1—Fundamental Algorithms*. Addison-Wesley, Reading, Mass., 1968, Sec. 1.2.1.
5. McCarthy, J. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (Eds.), North Holland, Amsterdam, 1963, pp. 33–70.
6. McCarthy, J., and Painter, J. A. Correctness of a compiler for arithmetic expressions. Proc. of a Symposium in Applied Mathematics, Vol. 19—Mathematical Aspects of Computer Science, J. T. Schwartz (Ed.), American Math. Society, Providence, R. I., 1967, pp. 33–41.
7. Naur, P. Proof of algorithms by general snapshots. *BIT 6* (1966), 310–316.

ALGORITHM 246
GRAYCODE [Z]

J. Boothroyd* (Recd. 18 Nov. 1963)
English Electric-Leo Computers, Kidsgrove, Stoke-on-Trent, England
* Now at University of Tasmania, Hobart, Tasmania, Aust.

```
procedure graycode (a) dimension: (n) parity: (s); value n,s;
  Boolean array a; integer n; Boolean s;
comment elements of the Boolean array a[1:n] may together be
  considered as representing a logical vector value in the Gray
  cyclic binary-code. [See e.g. Phister, M., Jr., Logical Design of
  Digital Computers, Wiley, New York, 1958. pp. 232, 399.] This
  procedure changes one element of the array to form the next
  code value in ascending sequence if the parity parameter s
  = true or in descending sequence if s = false. The procedure
  may also be applied to the classic "rings-o-seven" puzzle [see
  K. E. Iverson, A Programming Language, p. 63, Ex. 1.5];
begin integer i,j;  j := n + 1;
  for i := n step −1 until 1 do if a[i] then begin s := ¬ s;
    j := i end;
  if s then a[1] := ¬ a[1] else if j < n then a[j+1] := ¬ a[j+1]
    else a[n] := ¬ a[n]
end graycode
```

CERTIFICATION OF ALGORITHM 246 [Z]
GRAYCODE [J. Boothroyd, *Comm. ACM 7* (Dec. 1964), 701]

William D. Allen (Recd. 8 Feb. 1965 and 23 Feb. 1965)
Computing Ctr., U. of Kentucky, Lexington, Ky.

graycode was coded in Fortran IV and tested on the IBM 7040. *graycode* code was generated from 0 to 10,000 in both ascending and descending sequence. The procedure required no corrections and gave correct results for all cases tested.

ACM Transactions on Mathematical Software, Vol. 1, No. 3, September 1975.

REMARK ON ALGORITHM 246

Graycode [Z]
[J. Boothroyd, *Comm. ACM 7*, 12 (Dec. 1964), 701]

Jayadev Misra [Recd 13 May 1974 and 28 April 1975]
Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712

The following modifications to Algorithm 246 will generate Gray code for any N, with each code word being generated in a bounded amount of time. Let A be a vector of zeros and ones of length N which will be the successive code words. New code words are successively generated by reversing a single bit in A each time. Routine OUTPUT, to be supplied by the user, is called on generation of every new code word.

Initially A contains all zeros. At every odd-numbered step, $A[N]$ is reversed. At every even-numbered step, $A[J - 1]$ is reversed, where $A[J]$ is the rightmost one-bit in A. (In case $J = 1$, the algorithm terminates.) The positions of all the one-bits are stored in an increasing order in a stack S, from bottom to top. This helps in quickly locating J, the rightmost one-bit.

REFERENCES

1. Ehrlich, G. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *J. ACM 20*, 3 (July 1973), 500–513.

ALGORITHM 247
RADICAL-INVERSE QUASI-RANDOM POINT SEQUENCE [G5]

J. H. Halton and G. B. Smith (Recd. 24 Jan. 1964 and 21 July 1964)
Brookhaven National Laboratory, Upton, N. Y., and University of Colorado, Boulder, Colo.

```
procedure QRPSH (K, N, P, Q, R, E);
integer K, N;  real array P, Q;  integer array R;  real E;
```

comment This procedure computes a sequence of N quasi-random points lying in the K-dimensional unit hypercube given by $0 < x_i < 1$, $i = 1, 2, \cdots, K$. The ith component of the mth point is stored in $Q[m,i]$. The sequence is initiated by a "zero-th point" stored in P, and each component sequence is iteratively generated with parameter $R[i]$. E is a positive error-parameter. K, N, E, and the $P[i]$ and $R[i]$ for $i = 1, 2, \cdots, K$, are to be given.

The sequence is discussed by J. H. Halton in *Num. Math. 2* (1960), 84–90. If any integer n is written in radix-R notation as

$$n = n_m \cdots n_2 n_1 n_0 \,.\, 0 = n_0 + n_1 R + n_2 R^2 + \cdots + n_m R^m,$$

and reflected in the radical point, we obtain the R-inverse function of n, lying between 0 and 1,

$$\phi_R(n) = 0 \,.\, n_0 n_1 n_2 \cdots n_m = n_0 R^{-1} + n_1 R^{-2} + n_2 R^{-3} + \cdots + n_m R^{-m-1}.$$

The problem solved by this algorithm is that of giving a compact procedure for the addition of R^{-1}, in any radix R, to a fraction, with downward "carry".

If $P[i] = \phi_{R[i]}(s)$, as will almost always be the case in practice, with s a known integer, then $Q[m,i] = \phi_{R[i]}(s+m)$. For quasi-randomness (uniform limiting density), the integers $R[i]$ must be mutually prime.

For exact numbers, E would be infinitesimal positive. In practice, round-off errors would then cause the "carry" to be incorrectly placed, in two circumstances. Suppose that the stored number representing $\phi_R(n)$ is actually $\phi_R(n) + \Delta$. (a) If $|\Delta| \geqq R^{-m-1}$, we see that the results of the algorithm become unpredictable. It is necessary to stop before this event occurs. It may be delayed by working in multiple-length arithmetic. (b) If $n = R^{m+1} - 1$, so that $\phi_R(n) = 1 - R^{-m-1}$, and$\Delta < 0$, the computed successor of the stored value can be seen to be about R^{-m}, instead of $R^{-m-2} = \phi_R(n+1)$. This error can be avoided, without disturbing the rest of the computation, by adopting a value of E greater than any $|\Delta|$ which may occur, but smaller than the least $(nR)^{-1}$ (which is smaller than the least R^{-m-1}) to be encountered.

Small errors in the $P[i]$ will not affect the sequence. Any set of $P[i]$ in the computer may be considered as a set of $\phi_{R[i]}(s_i)$, for generally large and unequal integers s_i, with small round-off errors. The arguments used in J. H. Halton's paper to establish the uniformity of the sequence of points

$$[\phi_{R_1}(n), \phi_{R_2}(n), \cdots, \phi_{R_K}(n)], \quad n = 1, 2, \cdots, N$$

can be applied identically to the more general sequence

$$[\phi_{R_1}(s_1+n), \phi_{R_2}(s_2+n), \cdots, \phi_{R_K}(s_K+n)], \quad n = 1, 2, \cdots, N.$$

Thus, theoretically, any "zero-th point" P will do. However, the difficulty described in (a) above limits us to the use of $P[i]$ corresponding to relatively small integers s_i .;

```
begin integer i, m;  real r, f, g, h;
  for i := 1 step 1 until K do
  begin r := 1.0/R[i];
    for m := 1 step 1 until N do
    begin if m > 1 then f := 1.0 − Q[m−1,i] else
      f := 1.0−P[i];
      g := 1.0;  h := r;
    repeat: if f − h < E then
        begin g := h;  h := h × r;  go to repeat end;
      Q[m,i] := g + h − f
    end
  end
end QRPSH
```

ALGORITHM 248
NETFLOW [H]

WILLIAM A. BRIGGS (Recd. 18 Jan. 1964 and 17 Aug. 1964)
Marathon Oil Company, Findlay, Ohio

```
procedure NETFLOW (nodes, arcs, I, J, cost, hi, lo, flow, pi,
    INFEAS);
  value nodes, arcs;  integer nodes, arcs;
  integer array I, J, cost, hi, lo, flow, pi;  label INFEAS;
comment This procedure determines the least-cost flow pattern
  over an upper and lower bound capacitated flow network.
    Each directed network arc a is defined by nodes I[a] and
  J[a], has upper and lower flow bounds hi[a] and lo[a], and cost
  per unit of flow cost[a]. Costs and flow bounds may be any
  positive or negative integers. An upper flow bound must be
  greater than or equal to its corresponding lower flow bound for
  a feasible solution to exist. There may be any number of parallel
  arcs connecting any two nodes.
    A multi-source, multi-demand, capacitated transportation or
  transshipment problem may be stated as a network flow problem
  as follows:
    Append to the network (1)  bounded arcs from the demand
  node(s) to a "super sink," (2)  bounded arcs from a "super
  source" to the supply node(s), (3)  an arc directed from the
  "super sink" to the "super source" with zero lower bound, a
  large positive upper bound, and a negatively large cost.
    NETFLOW will maximize flow through the low-cost arc from
  "supper sink" to "super source"—subject to the capacity
  constraints of the network—fulfilling all demands optimally.
    The procedure returns vectors flow and pi. Flow[a] is the
  computed optimal flow over network arc a. Pi[n] is a number
  —the dual variable—which represents the relative value of
  injecting one unit of flow into the network at node n. NETFLOW
  may be entered with any values in vectors flow and pi (such as
  those from a previous or a guessed solution) feasible or not. If
  the initial contents of flow do not conserve flow at any node,
  the solution values will also not conserve flow at that node, by
  the same amount. This fact can be frequently used to advantage
  in transportation problem definition. The closer initial values
  of flow and pi are to solution values, the shorter the computation.
    Procedure NETFLOW is a mechanization of the out-of-kilter
  network flow algorithm described by D. R. FULKERSON in J.
  Soc. Indust. Appl. Math. 9 (1961), 18-27, and elsewhere. Many
  thanks are due the referee for noting some erroneous comments
  and for suggesting ways to increase the efficiency and utility of
  the procedure;
begin integer a, aok, c, cok, del, e, eps, inf, lab, n, ni, nj, src, snk;
  integer array na, nb[1: nodes];
  integer procedure min (x, y);  value x, y;  integer x, y;
    begin if x < y then min := x else min := y end min;
  comment  check feasibility of formulation;
  for a := 1 step 1 until arcs do if lo[a] > hi[a] then go to INFEAS;
  inf := 99999999;  comment set inf to max available integer;
  aok := 0;
  comment  find an out-of-kilter arc;
Seek:  for a := 1 step 1 until arcs do
  begin c := cost [a] + pi [I[a]] − pi [J[a]];
    if flow [a] < lo [a] ∨ (c<0∧ flow[a]<hi[a]) then
      begin src := J [a];  snk := I [a];   e := +1;   go to LABL
      end;
    if flow [a] > hi [a] ∨ (c>0∧flow[a]>lo[a]) then
      begin src := I[a];  snk := J[a];  e := −1;  go to LABL
      end;
  end;
  comment  no remaining out-of-kilter arcs;
  go to FINI;
  comment  attempt to bring found out-of-kilter arc into kilter;
LABL:  if a = aok ∧ na[src] ≠ 0 then go to SKIP;
  aok := a;
  for n := 1 step 1 until nodes do na[n] := nb[n] := 0;
  na[src] := abs (snk) × e;  nb[src] := abs (aok) × e;
SKIP:  cok := c;
LOOP:  lab := 0;
  for a := 1 step 1 until arcs do
  begin  if  (na[I[a]]=0∧na[J[a]]=0)  ∨
       (na[I[a]]≠0∧na[J[a]]≠0)  then  go  to  XC;
     c := cost[a] + pi[I[a]] − pi[J[a]];
     if na[I[a]] = 0 then go to XA;
     if flow[a] ≧ hi[a] ∨ (flow[a]≧lo[a]∧c>0) then go to XC;
     na[J[a]] := I[a];  nb[J[a]] := a;  go to XB;
XA: if flow[a] ≦ lo[a] ∨ (flow[a]≦hi[a]∧c<0) then go to XC;
     na[I[a]] := −J[a];  nb[I[a]] := −a;
XB: lab := +1;
     comment  node labeled, test for breakthru;
     if na[snk] ≠ 0 then go to INCR;
XC:  end;
  comment  no breakthru;
  if lab ≠ 0 then go to LOOP;
  comment  determine change to pi vector;
  del := inf;
  for a := 1 step 1 until arcs do
  begin  if  (na[I[a]]=0∧na[J[a]]=0)  ∨
       (na[I[a]]≠0∧na[J[a]]≠0)  then  go  to  XD;
    c := cost[a] + pi[I[a]] − pi[J[a]];
    if na[J[a]] = 0 ∧ flow[a] < hi[a] then del := min (del, c);
    if na[J[a]] ≠ 0 ∧ flow[a] > lo[a] then del := min (del, −c);
XD:  end;
  if del = inf ∧ (flow[aok]=hi[aok]∨flow[aok]=lo[aok]) then
    del := abs (cok);
  if del = inf then go to INFEAS;  comment exit, no feasible
  flow pattern;
  comment  change pi vector by computed del;
  for n := 1 step 1 until nodes do if na[n] = 0 then pi[n] :=
    pi[n] + del;
  comment  find another out-of-kilter arc;
  go to SEEK;
  comment  breakthru, compute incremental flow;
INCR:  eps := inf;
  ni := src;
BACK:  nj := abs (na[ni]);  a := abs (nb[ni]);
  c := cost[a] − abs (pi[ni]−pi[nj]) × sign (nb[ni]);
  if nb[ni] < 0 then go to XE;
  if c > 0 ∧ flow[a] < lo[a] then eps := min (eps, lo[a]−flow[a]);
  if c ≦ 0 ∧ flow[a] < hi[a] then eps := min (eps, hi[a]−flow[a]);
  go to XF;
XE: if c < 0 ∧ flow[a] > hi[a] then eps := min (eps,flow[a]
    −hi[a]);
  if c ≧ 0 ∧ flow[a] > lo[a] then eps := min (eps, flow[a]−lo[a]);
XF: ni := nj; if ni ≠ src then go to BACK;
  comment change flow vector by computed eps;
BACK2: nj := abs (na[ni]); a := abs (nb[ni]);
  flow[a] := flow[a] + eps × sign (nb[ni]);
  ni := nj; if ni ≠ src then go to BACK2;
  comment find another out-of-kilter arc;
  aok := 0; go to SEEK;
FINI: end NETFLOW with a feasible, optimal flow pattern
```

REMARK ON ALGORITHM 248 [H]
NETFLOW [William A. Briggs, *Comm. ACM 8* (Feb. 1965), 103]

J. H. Henderson, R. M. Knapp, and M. E. Volberding

(Recd. 7 Apr. 1966)

Northern Natural Gas Company, Omaha, Neb.

KEY WORDS AND PHRASES: capacitated network, linear programming, minimum-cost flow, network flow, out-of-kilter
CR CATEGORIES: 5.32, 5.41

Algorithm 248 was transcribed into Burroughs Extended Algol for the Burroughs B5500. After modification it has been used successfully. Before modification it was found to give erroneous values of *pi* for transportation problems and nonoptimal solutions for networks representing multitime level trans-shipment problems. This was caused by the method utilized within the procedure for exiting with the best solution. The difficulty was circumvented by inserting a statement just before label *SKIP* reading:

```
if nb [src] = arcs then go to FINI;
```

This statement enables the user to exit the procedure without a pass through the *pi* incrementation block and a final pass through the out-of-kilter arc-finding block, saving a significant amount of time on sizeable problems. With the arcs arranged so that the arc directed from the "super sink" to the "super source" is the last one in the arc array, it must be the last arc remaining out-of-kilter. Therefore, by the time the search block discovers it as an out-of-kilter arc, an optimal solution has already been found.

[Algorithm 336 [*Comm. ACM 11* (Sept. 1968), 631–632] is an improved version of Algorithm 248, which by its very construction bypasses this error.—J.G.H.]

REMARK ON ALGORITHM 248 [H]
NETFLOW [William A. Briggs, *Comm. ACM 8* (Feb. 1965), 103]

T. A. Bray and C. Witzgall

(Recd. 2 Oct. 1967 and 20 May 1968)

Boeing Scientific Research Laboratories, Seattle, WA 98124

KEY WORDS AND PHRASES: capacitated network, linear programming, minimum-cost flow, network flow, out-of-kilter
CR CATEGORIES: 5.32, 5.41

We found that

1. in the statement
   `c := cost[a] − abs(pi[ni]−pi[nj]) × sign(nb[ni]);`
   on page 103, column 2, line 3 from below,
   the "*abs*" should be deleted.
2. in the statement
   `LABL: if a = aok ∧ na[src] ≠ 0 then go to SKIP;`
   on page 103, column 2, line 13 from above,
   the value of *na*[*src*] may be undefined.

The algorithm worked satisfactorily after the corresponding changes had been made. We acknowledge a correspondence with R. M. Van Slyke and R. D. Sanderson of the University of California, Berkeley, on the subject.

Algorithm 336 [*Comm. ACM 11* (Sept. 1968), 631–632] is an improved version of Algorithm 248 incorporating these changes.

ALGORITHM 249
OUTREAL N [I5]

NIKLAUS E. WIRTH (Recd. 28 Aug. 1964 and 2 Nov. 1964)
Computer Science Div., Stanford U., Stanford, Calif.

```
procedure outreal n (ch,x,n);
  value ch, x, n;  real x;  integer ch, n;
comment  outreal n outputs to channel ch the real number x as
  a sequence of characters with n significant decimal digits in the
  form ± d.d ··· d10 ± d ·· d, where d stands for a digit. Like the
  procedures outboolean, outstring, ininteger (cf. Report on Input-
  Output Procedures for ALGOL 60, [Comm. ACM 7, (Oct. 1964),
  628-629]) and inreal [Alg. 239, Comm.ACM 7 (Aug.1964), 481] it
  constitutes an example of the use of the primitive procedure
  pair insymbol-outsymbol defined in the referenced Report;
begin integer i, j, k, s;  real f;  integer array a[1:n];
  procedure outchar(x)  value x;  integer x;
    outsymbol  (ch,'0123456789+−.10', x+1);
  s := k := 0;  f :=; 1;
  outchar (if x ≧ 0 then 10 else 11);  x := abs(x);
  if x = 0 then begin outchar(0);  go to L4 end;
  if  x  ≧  1  then
    begin L1:  f := f × 10;  s := s + 1;  if x ≧ f then go to L1;
      f := f × 0.1;  s := s − 1
    end
else
    begin L2:  f := f × 0.1;  s := s − 1;
      if x < f then go to L2
    end;
  x := x/f;  comment now 1 ≦ x < 10;
  for j := 1 step 1 until n − 1 do
  begin  i := entier(x);  a[j] := i;  x := (x−i) × 10 end;
  a[n] := x;
  for j := n − 1 step −1 until 1 do
  begin if a[j+1] < 10 then go to L6;  a[j+1] := 0;
    a[j] := a[j] + 1
  end;
  if a[1] = 10 then begin a[1] := 1;  s := s + 1 end;
L6:  outchar(a[1]);  outchar(12);
  for j := 2 step 1 until n do outchar(a[j]);
  comment now process the scale factor s;
  if s = 0 then go to L4;
  outchar(13);
  outchar (if s ≧ 0 then 10 else 11);  s := abs(s);
  j := 10;
L3:  if s ≧ j then begin j := j × 10;  k := k + 1;  go to L3 end;
L5:  if k > 0 then
  begin j := j ÷ 10;  i := s ÷ j;  outchar (i);  s := s − i × j;
    k := k − 1;  go to L5
  end;
  outchar(s);
L4:
end
```

ALGORITHM 250
INVERSE PERMUTATION [G6]

B. H. Boonstra (Recd. 12 Oct. 1964)
Nationaal Kasregisters, NCR Holland, Amsterdam.

```
procedure inversepermutation (P) of natural numbers up to: (n);
  value n; integer n; integer array P;
comment given a permutation P(i) of the numbers i = 1(1)n,
  the inverse permutation is computed in situ. The process is
  based on the lemma that any permutation can be written as a
  product of mutually exclusive cycles. Procedure inversepermuta-
  tion has been tested for several permutations including n = 1;
begin integer i, j, k, first;
  switch sss := tag, cycle, next, endcycle, finish;
tag: for i := 1 step 1 until n do P[i] := -P[i];
  comment now P[i] contains a negative number if original and
    a positive number if inverse;
  first := 1;
cycle: k := first; i := -P[k];
next: j := -P[i]; P[i] := k;
  if i = first then go to endcycle;
  k := i; i := j; go to next;
endcycle: if first = n then go to finish;
  first := first + 1;
  if P[first] < 0 then go to cycle else go to endcycle;
finish: end inversepermutation
```

REMARK ON ALGORITHM 250 [G6]
INVERSE PERMUTATION

[B. H. Boonstra, *Comm. ACM 8* (Feb. 1965), 104]

C. W. Medlock (Recd. 12 Apr. 1965 and 14 July 1965)
IBM Corp., Programming Systems, Poughkeepsie, N.Y.

Several simplifications may be made to the subject algorithm to permit more efficient operation.

1. On many compilers, the procedure would be more efficient if the outer loop were written as a **for** loop.

2. The initialization of the vector P to negative values may be omitted by reversing the interpretation of positive and negative values. As revised, $P[i]$ contains a negative number if it contains the inverse value and i is less than the current value of the parameter n. $P[i]$ contains a positive value in all other cases. This allows the **for** loop labeled *tag* to be eliminated.

3. The variable *first* may be eliminated by declaring the parameter n as a value parameter, and utilizing it as the controlled variable of the outer loop.

The author wishes to thank the referee for valuable suggestions.

The revised algorithm then reads:

```
procedure inversepermutation (P) of natural numbers up to: (n);
  value n; integer n; integer array P;
comment Given a permutation P(i) of the numbers i = 1(1)n,
  the inverse permutation is computed in situ;
begin integer i, j, k;
  for n := n step -1 until 1 do
  begin i := P[n];
    if i < 0 then P[n] := -i
    else if i ≠ n then
    begin k := n;
loop: j := P[i]; P[i] := -k;
      if j = n then P[n] := i
      else
      begin k := i; i := j; go to loop
      end
    end
  end
end inversepermutation
```

ALGORITHM 251
FUNCTION MINIMISATION [E4]

M. Wells (Recd. 13 July 1964 and 5 Oct. 1964)
Electronic Computing Lab., U. of Leeds, England

```
procedure FLEPOMIN (n, x, f, est, eps, funct, conv, limit, h,
        loadh);
  value n, est, eps, loadh, limit;
  real f, est, eps; integer n, limit; Boolean conv, loadh;
  array x, h; procedure funct;
comment function minimisation by the method of Fletcher
  and Powell [Comput. J. 6, 163-168 (1963)]. On entry x[1:n] is an
  estimate of the position of the minimum, est an estimate of the
  minimum value, eps a tolerance used in terminating the procedure
  when the first derivative of f nearly vanishes, and loadh
  indicates whether or not an approximation to the inverse of the
  matrix of second derivatives of f is available. If loadh is true
  the procedure supplies the unit matrix as this estimate, other-
  wise it is assumed that the upper triangle of a symmetric posi-
  tive definite matrix is stored by rows in h[1:n × (n+1)/2].
  The statement funct (n, x, f, g) assigns to f the function value
  and to g[1:n] the gradient vector.
    A successful exit from FLEPOMIN, with conv true, occurs
  if two successive values of f are equal, or if a new value of f
  is larger than the previous value (due to rounding errors), or
  if after n or more iterations the lengths of the vectors s and
  sigma are less than eps. If the number of iterations exceeds
  limit, then an exit occurs with conv false. In either case, the
  final function value, estimated position of the minimum and
  inverse matrix of second derivatives are in f, x and h;
begin
  real oldf, sg, ghg;
  integer i, j, k, count;
  array g, s, gamma, sigma [1:n];
  real procedure dot (a, b);
    array a, b;
    comment inner product of a and b [In this procedure ana
      in up dot greater accuracy would be obtained by accumulat-
      ing the inner products in double precision.—Ref.];
    begin integer i; real s; s := 0;
      for i := 1 step 1 until n do s := s + a[i] × b[i];
      dot := s
    end of dot;
  real procedure up dot (a, b, i);
    value i;
    array a, b; integer i;
    comment multiply b by the ith row of the symmetric
      matrix a, whose upper triangle is stored by rows;
    begin integer j, k; real s; k := i; s := 0;
      for j := 1 step 1 until i - 1 do
        begin s := s + a[k] × b[j]; k := k + n - j end steps
          to diagonal. Now go along row;
      for j := i step 1 until n do s := s + a[k+j-i] × b[j];
        up dot := s
    end of up dot;
set initial h:
  if loadh then
  begin k := 1;
    for i := 1 step 1 until n do
      begin h[k] := 1;
        for j := 1 step 1 until n - i do h[k+j] := 0;
        k := k + n - i + 1
      end
  end formation of unit matrix in h;
start of minimisation:
  conv := true;
  funct (n, x, f, g);
  for count := 1, count + 1 while oldf > f do
  begin oldf := f;
    for i := 1 step 1 until n do
    begin sigma[i] := x[i]; gamma[i] := g[i];
      s[i] := -up dot(h, g, i)
    end preservation of x, g and formation of s;
search along s:
  begin real ya, yb, va, vb, vc, h, k, w, z, t, ss;
    yb := f; vb := dot(g, s); ss := dot(s, s);
    if vb ≧ 0 then go to skip;
    k := 2 × (est-f)/vb;
scale: h := if k > 0 and k ↑ 2 × ss < 1 then k else 1/sqrt(ss);
    k := 0;
extrapolate: ya := yb; va := vb;
    for i := 1 step 1 until n do x[i] := x[i] + h × s[i];
    funct(n, x, f, g);
    yb := f; vb := dot(g, s);
    if vb < 0 and yb < ya then
    begin h := k := h + k; go to extrapolate end;
    t := 0;
interpolate: z := 3 × (ya-yb)/h + va + vb;
    w := sqrt(z↑2-va×vb);
    k := h × (vb+w-z)/(vb-va+2×w);
    for i := 1 step 1 until n do x[i] := x[i] + (t-k) × s[i];
    funct(n, x, f, g);
    if f > ya or f > yb then
    begin vc := dot(g, s);
      if vc < 0 then
      begin ya := f; va := vc; t := h := k end
        else
      begin yb := f; vb := vc; t := 0; h := h - k end;
      go to interpolate
    end;
skip: end of search along s;
  for i := 1 step 1 until n do
  begin sigma[i] := x[i] - sigma[i];
    gamma[i] := g[i] - gamma[i]
  end;
  sg := dot(sigma, gamma);
  if count ≧ n then
    begin if sqrt(dot(s, s)) < eps and sqrt(dot(sigma, sigma)) <
      eps then go to finish
    end test for vanishing derivative;
  for i := 1 step 1 until n do s[i] := up dot(h, gamma, i);
  ghg := dot(s, gamma);
  k := 1;
  for i := 1 step 1 until n do for j := i step 1 until n do
  begin h[k] := h[k] + sigma[i] × sigma[j]/sg - s[i] × s[j]/ghg;
    k := k + 1
  end updating of h;
  if count > limit then go to exit;
  end of loop controlled by count; go to finish;
exit: conv := false;
finish: end of FLEPOMIN
```

CERTIFICATION OF ALGORITHM 251 [E4] FUNCTION MINIMISATION [M. Wells, *Comm. ACM 8* (Mar. 1965), 169]

R. Fletcher (Recd. 9 Aug. 1965 and 24 Mar. 1966)
Electronic Computing Lab., U. of Leeds, England

Two points need correcting concerning the procedure *FLEPOMIN*.

(i) When the method has converged, either or both of the vectors **s** and **g** can become zero, hence also the scalars *sg* and *ghg*, causing division by zero when updating the matrix *h*.

(ii) The part of the procedure connected with the linear search along **s** does not make use of the fact that the identifier *h* (η in the Appendix to the source paper Fletcher and Powell [1]) tends to 1 as the process converges. This knowledge must be included to achieve the rapid convergence obtained by Fletcher and Powell. However, the particular choice of η given there can also be insufficient when its optimum value would be much greater than 1 (as happens for example in the minimization of $f(\mathbf{x}) = [\mathbf{H}(\mathbf{x}-\mathbf{1})]^2$ where **1** is the vector (1, 1, ···, 1) and **H** is a segment of the Hilbert matrix, from an initial approximation $\mathbf{x} = (0, 0, \cdots, 0)$).

An alternative approach is to estimate η by using its value at the previous iteration, increasing or decreasing its value by some constant factor when appropriate (I have arbitrarily used 4). This approach removes the need for the estimate *est* of the minimum value of $f(x)$.

The appropriate changes to be made are thus:

(i) omit *est* as a formal parameter,

(ii) include amongst the **real** identifiers at the head of the procedure body the following:

step, ita, fa, fb, ga, gb, w, z, lambda

(iii) replace the statements from the label

start of minimisation

to the end of the program by the following:

```
start of minimisation:
  conv := true;  step := 1;
  funct(n,x,f,g);
  for count := 1, count +1 while oldf > f do
  begin
     for i := 1 step 1 until n do
     begin sigma[i] := x[i];  gamma[i] := g[i];
        s[i] := −up dot(h,g,i)
end preservation of x,g and
     formation of s;
search along s:
     fb := f;  gb := dot (g,s);
     if gb ≥ 0 then go to exit;
     oldf := f;  ita := step;
     comment a change of ita × s is made in x and the function
        is examined. ita is determined from its value at the previous
        iteration (step) and is increased or decreased by 4 where
        necessary. It should tend to 1 at the minimum;
extrapolate:  fa := fb;  ga := gb;
     for i := 1 step 1 until n do x[i] := x[i] +ita × s[i];
     funct (n,x,f,g);
     fb := f;  gb := dot(g,s);
     if gb <0 ∧ fb < fa then
     begin ita := 4 × ita;  step := 4 × step;  go to extrapolate
end;
interpolate:  z := 3 × (fa−fb)/ita + ga + gb;
     w := sqrt (z↑2−ga×gb);
     lambda := ita × (gb+w−z)/(gb−ga+2×w);
     for i := 1 step 1 until n do x[i] := x[i] − lambda × s[i];
     funct (n,x,f,g);
     if f > fa ∨ f > fb then
     begin step := step/4;
        if fb < fa then
        begin for i := 1 step 1 until n do x[i] := x[i] + lambda ×
           s[i];  f := fb
        end else
        begin gb := dot(g,s);
           if gb < 0 ∧ count > n ∧ step <10−6 then go to exit;
           fb := f;  ita := ita − lambda;
           go to interpolate
        end;
skip: end of search along s;
     for i := 1 step 1 until n do
     begin sigma [i] := x [i] − sigma [i];
        gamma[i] := g[i] − gamma[i]
     end;
     sg := dot(sigma,gamma);
     if count ≥ n then
     begin if sqrt (dot(s,s)) < eps ∧ sqrt(dot(sigma,sigma)) <eps
        then go to finish
     end;
     for i := 1 step 1 until n do s[i] := up dot (h,gamma,i);
     ghg := dot(s, gamma);
     k := 1;
     if sg = 0 ∨ ghg = 0 then go to test;
     for i := 1 step 1 until n do for j := i step 1 until n do
     begin h[k] := h[k] + sigma[i] × sigma[j]/sg − s[i] × s[j]/ghg;
        k := k + 1
     end updating of h;
test: if count > limit then go to exit;
   end of loop controlled by count;  go to finish;
exit:conv := false;
finish:
end of FLEPOMIN
```

With these changes the procedure was run successfully on a KDF 9 computer on the first of the test functions used by Fletcher and Powell, and the appropriate rate of convergence was achieved. (The corresponding values in [1, Table 1, col. 4] being 24.200, 3.507, 2.466, 1.223, 0.043, 0.008, 4 × 10^{-5}). It could well be, however, that these changes may still not prove satisfactory on some functions. In such cases it will most likely be the search for the linear minimum along **s** which will be at fault, and not the method of generating **s**. It should not be necessary to evaluate the function and gradient more than 5 or 6 times per iteration in order to estimate the minimum along **s**, except possibly at the first few iterations.

I am indebted to William N. Nawatani of Dynalectron Corporation, Calif., for pointing out the discrepancies in the rates of convergence, and to the referee for his calculations and comments with regard to the Hilbert Matrix function.

REFERENCE

1. Fletcher, R., and Powell M. J. D. A rapidly convergent descent method for minimization. *Comput. J. 6* (July 1963), 163.

REMARK ON ALGORITHM 251 [E4]
FUNCTION MINIMIZATION [M. Wells, *Comm. ACM 8* (Mar. 1965), 169]

P. A. Hamilton and J. Boothroyd (Recd. 17 Dec. 1968)
University of Tasmania, Hobart, Tasmania, Australia

KEY WORDS AND PHRASES: function minimization
CR CATEGORIES: 5.19

The changes proposed by Fletcher in his "Certification of Algorithm 251," *Comm. ACM 9* (Sept. 1966), 686, contain one mistake and one unprotected possible source of error. On page 687, line 2, the assignment statement $f := fb$ should be replaced by the procedure statement *funct(n,x,f,g)* in order to reset the gradients in $g[1:n]$.

In theory, the conditions on z,ga,gb valid for interpolation imply $z\uparrow 2-ga\times gb\geq 0$. The statement $w := sqrt(z\uparrow 2-ga\times gb)$ should therefore be safe. In practice, round-off errors may give rise to small negative values of the argument, resulting in an error condition which may be avoided with:

```
w := z↑2−ga×gb;
w := if w<0 then 0 else sqrt(w);
```

Numerous tests of this procedure indicate that two other changes are beneficial in reducing the number of function calls required to yield a minimum to some prescribed accuracy. These concern the method of calculating the minimum of the interpolating cubic and a modification to the extrapolation strategy.

In the notation of Fletcher's identifiers, the position of the minimum along **s** over the interval $(a=0,\ b=ita)$ is $a+r$ where r is the root of a quadratic equation given by:

$$r = ita\times(ga+z+w)/(ga+gb+2\times z) \qquad (1)$$

and, for $ga+z\geq 0$, it may be shown that r is the root of larger magnitude; otherwise, it is the root of smaller magnitude. The distance of the minimum from b is $lambda=ita-r$ and Davidon[1] seems to have originated the proposal that *lambda* should be evaluated by:

$$lambda := ita\times(gb+w-z)/(gb-ga+2\times w)$$

in order to avoid cancellation. In this respect it is only partly successful, and our experience shows that to avoid cancellation completely *lambda* should be calculated in the more orthodox manner:

```
lambda := ita×(1−(if ga+z≥0 then (ga+z+w)/(ga+gb+2×z)
   else ga/(ga+z−w)));
```

Once the minimum along **s** has been bounded, the use of cubic interpolation is rewardingly accurate and it is natural to inquire whether cubic extrapolation can provide a better farther bound than is afforded by an arbitrary search. It may be shown that, provided $z\uparrow 2-ga\times gb\geq 0$ and $r>0$ where r is given by eq. (1), cubic extrapolation will yield the position of the predicted minimum along **s** as $a+r$, using a value for *ita* given by the step length of the previous iteration. To bound the minimum we take the interval $(a,a+2\times r)$ if the above conditions are satisfied; otherwise, we adopt Fletcher's strategy of using the interval of the previous iteration scaled by a factor of 4.

Reference:

1. Davidon, W. C. Variable metric method for minimization. ANL-5990. US Atomic Energy Commission Res. Develop. Rep., 1959.

Remark on Algorithm 251 [E4]
Function Minimization [M. Wells, *Comm. ACM 8* (Mar. 1965), 169.]

F. R. House [Recd. 25 Aug. 1970 and 1 Dec. 1970]
Department of Pharmacology, Guy's Hospital Medical School, London, S.E.1. England

The above procedure, as modified by Fletcher [1], and Hamilton and Boothroyd [2], may appear to fail if the process converges after fewer than *n* iterations. In particular, if the starting point coincides with the minimum, failure is certain. The trouble arises from the statement

```
if gb ≥0 then go to exit;
```

which appears two lines after the label *search along s*.

The following modifications are proposed.

(i) After the first call of *funct* insert the statement

```
if sqrt(dot(g, g)) < eps then
  begin
  for i := 1 step 1 until n do x[i] := x[i] + 1;  funct (n, x, f, g)
  end;
```

(ii) Replace the statement

```
  if gb ≥ 0 then go to exit;
```

by

```
if gb = 0 then go to skip;
if gb > 0 then go to exit;
```

The apparently perverse move away from the minimum implied by modification (i) ensures that *h* is updated at least once.

(iii) The text from

if *count* ≥ *n* **then**

to

end;

should occur after the label *test*. The relevant portion of the program reads

```
test:  if count ≥ n then
  begin
    if sqrt(dot(s, s)) < eps ∧ sqrt(dot(sigma, sigma)) < eps
    then go to finish
  end;
  if count > limit then go to exit;
```

Experience with the algorithm has shown that when the process converges from a poor starting point on a nonquadratic surface the final estimate of *h* is inclined to be somewhat erratic.

This modification causes *h* to be updated once more using the very latest information, and will often effect a substantial improvement in accuracy. The estimated position of the minimum is, of course, unaffected.

References:

1. Fletcher, R. *Comm. ACM 9* (Sept. 1966), 686–687.
2. Hamilton, P. A., and Boothroyd, J. *Comm. ACM 12* (Sept. 1969), 512–513.

ALGORITHM 252 [Z]
VECTOR COUPLING OR CLEBSCH-GORDAN COEFFICIENTS

J. H. Gunn

(Recd. 17 Aug. 1964, 13 Nov. 1964 and 21 Dec. 1964)

Nordisk Institut for Teoretisk Atomfysik, Copenhagen, Denmark

real procedure *VCC*(*J*1, *J*2, *J*, *M*1, *M*2, *M*, *factorial*);
value *J*1, *J*2, *J*, *M*1, *M*2, *M*;
integer *J*1, *J*2, *J*, *M*1, *M*2, *M*; **array** *factorial*;
comment *VCC* calculates the vector coupling or Clebsch-Gordan coefficients defined by the following formula

$$(j_1\, m_1\, j_2\, m_2 \mid j_1\, j_2\, j\, m)$$
$$= \delta(m_1 + m_2, m)\left[\frac{(2j+1)(j_1+j_2-j)!(j_1-j_2+j)!(-j_1+j_2+j)!}{(j_1+j_2+j+1)!}\right]^{\frac{1}{2}}$$
$$\times\, [(j_1+m_1)!(j_1-m_1)!(j_2+m_2)!(j_2-m_2)!(j+m)!(j-m)!]^{\frac{1}{2}}$$
$$\times \sum_z (-1)^z/[z!(j_1+j_2-j-z)!(j_1-m_1-z)!$$
$$(j_2+m_2-z)!(j-j_2+m_1+z)!(j-j_1-m_2+z)!]$$

where $j1 = J1/2$, $j2 = J2/2$, $j = J/2$, $m1 = M1/2$, $m2 = M2/2$, $m = M/2$. [Reference formula 3.6.11, p. 45 of Edmonds, Alan R. Angular momentum in quantum mechanics. In *Investigations in Physics*, *4*, Princeton U. Press, 1957.]. The parameters of the procedure, *J*1, *J*2, *J*, *M*1, *M*2 and *M*, are interpreted as being twice their physical value, so that actual parameters may be integers. Thus to call the procedure to calculate $(\frac{1}{2}\ 0\ \frac{1}{2}\ 0 \mid \frac{1}{2}\ \frac{1}{2}\ 0\ 0)$ the call would be *VCC*(1, 1, 0, 0, 0, 0, *factorial*). The procedure checks that the triangle conditions for the existence of a coefficient are satisfied and that $j1 + j2 + j$ is integral. If the conditions are not satisfied the value of the procedure is zero. The parameter *factorial* is an array containing the factorials from 0 up to $j1 + j2 + j + 1$. Since in actual calculations the procedure *VCC* will be called many times it is more economical to have the factorials in a global array rather than compute them on every entry to the procedure;

```
begin integer z, zmin, zmax; real cc;
if M1 + M2 ≠ M ∨ abs(M1) > abs(J1) ∨ abs(M2) > abs(J2) ∨
   abs(M) > abs(J) ∨ J > J1 + J2 ∨ J < abs(J1−J2) ∨ J1
   + J2 + J ≠ 2 × ((J1+J2+J)÷2) then VCC := 0 else
  begin zmin := 0;
     if J − J2 + M1 < 0 then zmin := −J + J2 − M1;
     if J − J1 − M2 + zmin < 0 then zmin := −J + J1 + M2;
     zmax: = J1 + J2 − J;
     if J2 + M2 − zmax < 0 then zmax := J2 + M2;
     if J1 − M1 − zmax < 0 then zmax := J1 − M1;
     cc := 0;
     for z := zmin step 2 until zmax do
     cc := cc + (if z=4×(z÷4) then 1 else −1)/(factorial[z÷2]
         × factorial[(J1+J2−J−z)÷2]
         × factorial[(J1−M1−z)÷2]
         × factorial[(J2+M2−z)÷2]
         × factorial[(J−J2+M1+z)÷2]
         × factorial[(J−J1−M2+z)÷2]);
     VCC := sqrt((J+1) × factorial[(J1+J2−J)÷2]
         × factorial[(J1−J2+J)÷2]
         × factorial[(−J1+J2+J)÷2] × factorial[(J1+M1)÷2]
         × factorial[(J1−M1)÷2] × factorial[(J2+M2)÷2]
         × factorial[(J2−M2)÷2] × factorial[(J+M)÷2]
         × factorial[(J−M)÷2]/factorial[(J1+J2+J+2)÷2])
         × cc
  end
end VCC
```

ALGORITHM 253 [F2]
EIGENVALUES OF A REAL SYMMETRIC MATRIX BY THE QR METHOD

P. A. Businger*

(Recd. 17 Aug. 1964, 3 Nov. 1964 and 8 Dec. 1964)

University of Texas, Austin, Texas

* This work was supported in part by the National Science Foundation through grant NSF GP-217 and the Army Research Office through grant DA-ARO(D) 31-124-G388. Thanks are due the referee for suggesting several improvements.

```
procedure symmetric QR 1 (n, g); value n; integer n;
  array g;
comment uses Householder's method and the QR algorithm to
  find all n eigenvalues of the real symmetric matrix whose lower
  triangular part is given in the array g[1:n, 1:n]. The computed
  eigenvalues are stored as the diagonal elements g[i, i]. The
  original contents of the lower triangular part of g are lost during
  the computation whereas the strictly upper triangular part of g
  is left untouched.
```

References:

Francis, J. G. F. The QR transformation—Part 2. *Comput. J. 4* (1961), 332-345.

Ortega, J. M., and Kaiser, H. F. The LL^T and QR methods for symmetric tridiagonal matrices. *Comput. J. 6* (1963), 99-101.

Parlett, B. The development and use of methods of LR type. New York U., 1963.

Wilkinson, J. H. Householder's method for symmetric matrices. *Numer. Math. 4*, (1962), 354-361.

Test Results:

A version of this procedure acceptable to the Oak Ridge Algol compiler was tested on a CDC 1604 computer (relative machine precision $1.5_{10}-11$). For a number of testmatrices of order up to 64 the dominant eigenvalue was found to at least 8 digits and it was always among the most accurate values computed. In some cases the accuracy of the nondominant eigenvalues varied greatly, in one case the least accurate value had only 4 good digits.

Example:

For the 5×5 symmetric matrix whose lower triangular part is

```
5
4 6
3 0 7
2 4 6 8
1 3 5 7 9
```

this prodecure computed the eigenvalues 22.406875305, 7.5137241530, 4.8489501197, −1.0965951813, 1.3270455994;

```
begin
  real procedure sum (i, m, n, a); value m, n;
    integer i, m, n; real a;
  begin real s; s := 0;
    for i := m  step 1 until n do s := s+a; sum := s
  end sum;
  real procedure max (a, b); value a, b; real a, b;
    max := if a > b then a else b;
  procedure Householder tridiagonalization 1 (n, g, a, bq, norm);
    value n; integer n; array g, a, bq; real norm;
    comment nonlocal real procedure sum, max;
  comment reduces the given real symmetric n by n matrix g
    to tridiagonal form using n−2 elementary orthogonal trans-
    formations (I−2ww') = (I-gamma uu'). Only the lower tri-
    angular part of g need be given. The diagonal elements and
    the squares of the subdiagonal elements of the reduced matrix
    are stored in a[1:n] and bq[1:n−1] respectively. norm is set
    equal to the infinity norm of the reduced matrix. The columns
    of the strictly lower triangular part of g are replaced by the
    nonzero portions of the vectors u;
  begin integer i, j, k; real t, absb, alpha, beta, gamma, sigma;
    array p[2:n];
    norm := absb := 0;
    for k := 1 step 1 until n−2 do
    begin a[k] := g[k, k];
      sigma := bq[k] := sum(i, k+1, n, g[i, k]↑2);
      t := absb+abs(a[k]); absb := sqrt(sigma);
      norm := max(norm, t+absb);
      if sigma ≠ 0 then
      begin alpha := g[k+1, k];
        beta := if alpha < 0 then absb else−absb;
        gamma := 1/(sigma−alpha×beta); g[k+1, k] := alpha−
          beta;
        for i := k+1 step 1 until n do
          p[i] := gamma×(sum(j, k+1, i, g[i, j]×g[j, k])+
            sum(j, i+1, n, g[j, i]×g[j, k]));
        t := 0.5×gamma×sum(i, k+1, n, g[i, k]×p[i]);
        for i := k+1 step 1 until n do p[i] := p[i]−t×g[i, k];
        for i := k+1 step 1 until n do
          for j := k+1 step 1 until i do
            g[i, j] := g[i, j]−g[i, k]×p[j]−p[i]×g[j, k]
      end
    end k;
    a[n−1] := g[n−1, n−1]; bq[n−1] := g[n, n−1]↑2;
    a[n] := g[n, n]; t := abs(g[n, n−1]);
    norm := max(norm, absb+abs(a[n−1])+t);
    norm := max(norm, t+abs(a[n]))
  end Householder tridiagonalization 1;
  integer i, k, m, m1; real norm, epsq, lambda, mu, sq1, sq2, u,
    pq, gamma, t; array a[1:n], bq[0:n−1];
  Householder tridiagonalization 1(n, g, a, bq, norm);
  epsq := 2.25₁₀−22×norm↑2; comment The tolerance used in
    the QR iteration depends on the square of the relative ma-
    chine precision. Here 2.25₁₀−22 is used which is appropriate
    for a machine with a 36-bit mantissa;
  mu := 0; m := n;
inspect: if m=0 then go to return else i := k := m1 := m−1;
  bq[0] := 0;
  if bq[k] ≦ epsq then
  begin g[m, m] := a[m]; mu := 0; m := k;
    go to inspect
  end;
  for i := i−1 while bq[i] > epsq do k := i;
  if k = m1 then
  begin comment treat 2 × 2 block separately;
    mu := a[m1]×a[m]−bq[m1]; sq1 := a[m1]+a[m];
    sq2 := sqrt((a[m1]−a[m])↑2+4×bq[m1]);
    lambda := 0.5×(if sq1 ≧ 0 then sq1+sq2 else sq1−sq2);
    g[m1, m1] := lambda; g[m, m] := mu/lambda;
    mu := 0; m := m−2; go to inspect
  end;
  lambda := if abs(a[m]−mu) < 0.5×abs(a[m]) then a[m]+0.5×
    sqrt(bq[m1]) else 0.0;
  mu := a[m];  sq1 := sq2 := u := 0;
  for i := k step 1 until m1 do
  begin comment  shortcut single QR iteration;
    gamma := a[i]−lambda−u;
    pq := if sq1≠1 then gamma↑2/(1−sq1) else (1−sq2)×
      bq[i−1];
    t := pq+bq[i];  bq[i−1] := sq1×t;  sq2 := sq1;
    sq1 := bq[i]/t;  u := sq1×(gamma+a[i+1]−lambda);
    a[i] := gamma+u+lambda
  end  i;
  gamma := a[m]−lambda−u;
  bq[m1]  :=  sq1×(if  sq1≠1  then  gamma↑2/(1−sq1)  else
    (1−sq2)×bq[m1]);
  a[m] := gamma+lambda;  go to inspect;
return: end symmetric QR 1
```

CERTIFICATION OF ALGORITHM 253 [F2] EIGENVALUES OF A REAL SYMMETRIC MATRIX BY THE QR METHOD [P. A. Businger, *Comm. ACM 8* (April 1965), 217]

John H. Welsch (Recd. 3 June 1965, 1 Aug. 1966 and 1 Mar. 1967)
Stanford Linear Accelerator Center, Stanford, California

The procedure *symmetric QR* 1 was transcribed into Algol for the Burroughs B5500 (39-bit mantissa) and tested with no syntax or logic changes (except to change the tolerance from $2.25_{10}-22$ to $3.35_{10}-24$). The eigenvalues of the matrix in the example given in the procedure declaration were found to 15 units in the 11th significant place and in the order given.

Two defects of this algorithm have been found (personal communication from Prof. W. Kahan); one concerning the convergence, the other concerning the numerical stability.

The procedure *symmetric QR* 1 was slow to converge on matrices of large order with the form

$$\begin{bmatrix} 0 & 1 & & & & \\ 1 & 0 & 1 & & & \\ & 1 & 0 & 1 & & \\ & & \cdot & & & \\ & & & \cdot & & \\ & & & & \cdot & 1 \\ & & & & 1 & 0 \end{bmatrix}$$

The trouble is caused by a poor choice of the shift, *lambda*, for accelerating convergence. The fault was corrected as described in the Certification of Algorithm 254.

The second defect is not as easy to detect or correct. On matrices of large order with pairs of eigenvalues of opposite sign, members of the pairs were found to varying accuracy. Another indication of an instability was a distinct jump in the computed values of the eigenvalues of the matrix

$$\begin{bmatrix} x & 1 & & \\ 1 & 1 & 1 & \\ & 1 & -x & 1 \\ & & 1 & -1 \end{bmatrix}$$

at $x = 10^{-5}$, as x was given the values $10^{-3}, 10^{-4}, \cdots, 10^{-11}$.

It appears that the square-root-free *QR* Algorithm described by Ortega and Kaiser ("The LL^T and *QR* methods for symmetric tridiagonal matrices," *Comput. J. 6* (1963), 99–101) is numerically unstable; therefore Algorithm 253 should be avoided. [Rutishauser (Letter to the Editor, *Comput. J. 6* (1963), 133) suggested a modification which is also mentioned by Wilkinson (*The Algebraic Eigenvalue Problem*, Clarendon Press, Oxford, 1965, p. 567). However, even with this modification the Algorithm is numerically unstable as was pointed out in a private communication from Wilkinson to Kahan (1966)—Ref.]

ALGORITHM 254 [F2]
EIGENVALUES AND EIGENVECTORS OF A REAL SYMMETRIC MATRIX BY THE QR METHOD

P. A. Businger*

(Recd. 17 Aug. 1964, 17 Nov. 1964 and 8 Dec. 1964)

University of Texas, Austin, Texas

```
procedure symmetric QR 2 (n, g, x);  value n;  integer n;
  array g, x;
comment uses Householder's method and the QR algorithm to
  find all n eigenvalues and eigenvectors of the real symmetric
  matrix whose lower triangular part is given in the array g. The
  computed eigenvalues are stored as the diagonal elements
  g[i, i] and the eigenvectors as the corresponding columns of the
  array x. The original contents of the lower triangular part of g
  are lost during the computation whereas the strictly upper
  triangular part of g is left untouched.
```

References:

Francis, J. G. F. The QR transformation—Part 2. *Comput. J.* *4* (1961), 332–345.

Parlett, B. The development and use of methods of LR type. New York U., 1963.

Wilkinson, J. H. Householder's method for symmetric matrices. *Numer. Math.* *4* (1962), 354–361.

Test Results:

A version of this procedure acceptable to the Oak Ridge Algol compiler was tested on a CDC 1604 computer (relative machine precision 1.5_{10}-11). For a number of testmatrices of order up to 64 the dominant eigenvalue was found to at least 9 digits. Eigenvalues much smaller in magnitude than the dominant eigenvalue have fewer accurate digits. In some cases the components of the eigenvectors were slightly less accurate than the eigenvalues.

Example:

For the 5 × 5 symmetric matrix whose lower triangular part is

$$\begin{array}{ccccc} 5 \\ 4 & 6 \\ 3 & 0 & 7 \\ 2 & 4 & 6 & 8 \\ 1 & 3 & 5 & 7 & 9 \end{array}$$

this procedure computed the eigenvalues $\lambda_1=22.406875306$, $\lambda_2=7.5137241547$, $\lambda_3=4.8489501203$, $\lambda_4=-1.0965951820$, $\lambda_5=1.3270455995$, and the corresponding eigenvectors

x_1 = (0.24587793851, 0.30239603954, 0.45321452335, 0.57717715229, 0.55638458400),

x_2 = (0.55096195546, 0.70944033954, −0.34017913315, −0.083410953290, −0.26543567685),

x_3 = (0.54717279573, −0.31256992008, 0.61811207635, −0.11560659356, −0.45549374666),

x_4 = (−0.46935807220, 0.54221219466, 0.54445240360, −0.42586566248, −0.088988503134),

x_5 = (−0.34101304185, 0.11643462042, 0.019590672072, 0.68204303436, −0.63607121400);

```
begin
  real procedure sum (i, m, n, a);  value m, n;
    integer i, m, n;  real a;
  begin real s;  s := 0;
    for i := m step 1 until n do s := s+a;  sum := s
  end sum;
  real procedure max (a, b);  value a, b;  real a, b;
    max := if a > b then a else b;
  procedure Householder tridiagonalization 2 (n, g, a, b, x, norm);
    value n;  integer n;  array g, a, b, x;  real norm;
    comment nonlocal real procedure sum, max;
  comment reduces the given real symmetric n by n matrix g
    to tridiagonal form using n−2 elementary orthogonal trans-
    formations (I−2ww') = (I−gamma uu'). Only the lower
    triangular part of g need be given. The computed diagonal
    and subdiagonal elements of the reduced matrix are stored in
    a[1:n] and b[1:n−1] respectively. The transformations on the
    right are also applied to the n by n matrix x. The columns of
    the strictly lower triangular part of g are replaced by the
    nonzero portion of the vectors u. norm is set equal to the in-
    finity norm of the reduced matrix;
  begin integer i, j, k;  real t, sigma, alpha, beta, gamma, absb;
    array p[2:n];
    norm := absb := 0;
    for k := 1 step 1 until n−2 do
    begin a[k] := g[k, k];
      sigma := sum(i, k+1, n, g[i, k]↑2);
      t := absb+abs(a[k]);  absb := sqrt(sigma);
      norm := max(norm, t+absb);  alpha := g[k+1, k];
      b[k] := beta := if alpha < 0 then absb else −absb;
      if sigma ≠ 0 then
      begin gamma := 1/(sigma−alpha×beta);
        g[k+1, k] := alpha−beta;
        for i := k+1 step 1 until n do
          p[i] := gamma×(sum(j, k+1, i, g[i, j]×g[j, k])
            +sum(j, i+1, n, g[j, i]×g[j, k]));
        t := 0.5×gamma×sum(i, k+1, n, g[i, k]×p[i]);
        for i := k+1 step 1 until n do p[i] := p[i]−t×g[i, k];
        for i := k+1 step 1 until n do
          for j := k+1 step 1 until i do
            g[i, j] := g[i, j]−g[i, k]×p[j]−p[i]×g[j, k];
        for i := 2 step 1 until n do
          p[i] := gamma×sum(j, k+1, n, x[i, j]×g[j, k]);
        for i := 2 step 1 until n do
          for j := k+1 step 1 until n do
            x[i, j] := x[i, j]−p[i]×g[j, k]
      end
    end k;
    a[n−1] :=g[n−1,n−1];  a[n] :=g[n,n];  b[n−1] := g[n, n−1];
    t := abs(b[n−1]);
    norm := max(norm, absb+abs(a[n−1])+t);
    norm := max(norm, t+abs(a[n]))
  end Householder tridiagonalization 2;
  integer i, j, k, m, m1;  real t, norm, eps, sine, cosine, lambda,
    mu, a0, a1, b0, beta, x0, x1;
  array a[1:n], b[0:n], c[0:n−1], cs, sn[1:n−1];
  for i := 1 step 1 until n do
  begin comment set x equal to the identity matrix;
    x[i, i] := 1;
    for j := i+1 step 1 until n do x[i, j] := x[j, i] := 0
  end i;
```

* This work was supported in part by the National Science Foundation through grant NSF GP-217 and the Army Research Office through grant DA-ARO(D) 31-124-G388. Thanks are due the referee for suggesting several improvements.

```
  Householder tridiagonalization 2 (n, g, a, b, x, norm);
  eps := norm×1.5_10-11; comment the tolerance used in the
    QR iteration is set equal to the product of the infinity norm
    of the reduced matrix and the relative machine precision
    (here assumed to be 1.5_10-11 which is appropriate for a machine
    with a 36-bit mantissa);
  b[0] := mu := 0;  m := n;
inspect: if m=0 then go to return else i := k := m1 := m-1;
  if abs(b[k]) ≦ eps then
  begin
    g[m, m] := a[m];  mu := 0;  m := k;  go to inspect
  end;
  for i := i-1 while abs(b[i]) > eps do k := i;
  lambda := if abs(a[m]-mu) < 0.5×abs(a[m]) ∨ m1=k then
    a[m]+0.5×b[m1] else 0.0;
  mu := a[m];  a[k] := a[k]-lambda;  beta := b[k];
  for j := k step 1 until m1 do
  begin comment transformation on the left;
    a0 := a[j];  a1 := a[j+1]-lambda;  b0 := b[j];
    t := sqrt(a0↑2+beta↑2);
    cosine := cs[j] := a0/t;  sine := sn[j] := beta/t;
    a[j] := cosine×a0+sine×beta;  a[j+1] := -sine×b0+
      cosine×a1;
    b[j] := cosine×b0+sine×a1;  beta := b[j+1];
    b[j+1] := cosine×beta;  c[j] := sine×beta
  end j;
  b[k-1] := c[k-1] := 0;
  for j := k step 1 until m1 do
  begin comment transformation on the right;
    sine := sn[j];  cosine := cs[j];
    a0 := a[j];  b0 := b[j];
    b[j-1] := b[j-1]×cosine+c[j-1]×sine;
    a[j] := a0×cosine+b0×sine+lambda;
    b[j] := -a0×sine+b0×cosine;  a[j+1] := a[j+1]×cosine;
    for i := 1 step 1 until n do
    begin x0 := x[i, j];  x1 := x[i, j+1];
      x[i, j] := x0×cosine+x1×sine;  x[i, j+1] := -x0×sine+
        x1×cosine
    end i
  end j;
  a[m] := a[m]+lambda;  go to inspect;
return: end symmetric QR 2
```

CERTIFICATION OF ALGORITHM 254 [F2] EIGENVALUES AND EIGENVECTORS OF A REAL SYMMETRIC MATRIX BY THE QR METHOD

[P. A. Businger, *Comm. ACM 8* (April 1965), 218]

John H. Welsch (Recd. 3 June 1965, 1 Aug. 1966 and 1 Mar. 1967)

Stanford Linear Accelerator Center, Stanford, California

The procedure *symmetric QR* 2 was transcribed into Algol for the Burroughs B5500 (39-bit mantissa) and tested with no syntax or logic changes (except to change the tolerance from $1.5_{10} - 11$ to $1.83_{10} - 12$). The eigenvalues of the matrix given in the initial comment of the procedure declaration were found to 8 units in the 11th significant place and in the order given. The components of the eigenvectors found by the procedure differed from those given by at most 7 units in the 10th significant place and that occurred in the smallest component of X_2. The computed vectors X_3 and X_4 were the negative of those given.

It was found (personal communication from Prof. W. Kahan, University of Toronto) that *symmetric QR* 2 was slow to converge on matrices of large order with the form

$$\begin{bmatrix} 0 & 1 & & & & \\ 1 & 0 & 1 & & & \\ & 1 & 0 & 1 & & \\ & & \cdot & & & \\ & & & \cdot & & \\ & & & & \cdot & \\ & & & & 1 & 0 \end{bmatrix}$$

The trouble observed seems to be caused by a poor choice of the shift, *lambda*, for accelerating convergence. The following change corrects this fault and did not change the results of these tests except that the eigenvalues are found in a different order. Replace the 8 lines following the line labeled *inspect* by:

```
if abs(b[k]) ≦ eps then
  begin g[m, m] := a[m];  m := k;  go to inspect end;
for i := i - 1 while abs(b[i]) > eps do k := i;
comment find eigenvalues of lower 2 × 2;
b0 := b[m1] ↑ 2;  a1 := sqrt((a[m1]-a[m])↑2+4×b0);
t := a[m1] × a[m] - b0;  a0 := a[m1] + a[m];
lambda := 0.5 × (if a0≧0 then a0+a1 else a0-a1);
t := t/lambda;  comment compute the shift;
if abs(t-mu) < 0.5 × abs(t) then mu := lambda := t
else if abs(lambda-mu) < 0.5 × abs(lambda) then mu := lambda
else begin mu := t;  lambda := 0 end;
a[k] := a[k] - lambda;  beta := b[k];
```

The modified procedure (called *QR* 2 below) was compared with the procedures given by J. H. Wilkinson [*Numer. Math. 4* (1962), 354–376] of the Householder tridiagonalization, Sturm sequence bisection, and inverse iteration algorithms. Evaluation of the Sturm sequence caused exponent underflows and overflows, so the procedures were modified (referred to as *HSI* below) by scaling and overflow detection.

To measure the effectiveness of the procedures, two quantities, E_1 and E_2, were evaluated for each of eleven matrices used as test data. These quantities are suggested by Prof. W. Kahan (in "Inclusion Theorems for Clusters of Eigenvalues of Hermitian Matrices," University of Toronto, Feb. 1967) and are defined as follows. Let A be a Hermitian matrix, Λ a diagonal matrix of its approximate eigenvalues and V a matrix whose columns are approximate eigenvectors ordered to correspond with Λ. Define $W = V^*V - I$ and $R = AV - V\Lambda$, then

$$E_1 = \| W \|_2 \text{ and } E_2 = \| R \|_2 / \| \Lambda \|_2 ,$$

where $\| X \|_2^2$ = maximum eigenvalue of X^*X. Then it is shown that the maximum absolute error in an eigenvalue is less than or equal to

$$\frac{E_2 \| \Lambda \|_2}{\sqrt{1 - E_1}} \quad \text{if } E_1 < 1.$$

The computation of W and R was done with double-precision inner products.

The results of the tests are summarized as follows:

(a) Both *QR* 2 and *HSI* found the dominant eigenvalues to better relative accuracy, but the same or worse absolute accuracy than the other eigenvalues.

(b) *QR* 2 was on the average 1.8 times faster than *HSI* (*QR* 2 required 2.5 seconds on a Hilbert segment of order 15).

(c) *QR* 2 always found orthogonal eigenvectors ($E_1 \sim 10^{-11}$);

(d) in most cases $E_1 \sim 10^{-11}$ for *HSI* also, but several times *HSI* found two eigenvectors almost parallel ($E_1 \sim 1.0$).

(e) $E_2 \sim 10^{-11}$ for both *QR* 2 and *HSI* with neither being consistently better than the other.

Conclusions. The orthonormalized eigenvectors, speed, and comparable accuracy would recommend *symmetric QR* 2 over the Wilkinson procedures for finding all of the eigenvalues and eigenvectors of a real symmetric matrix. The latter procedures are good for finding selected eigenvalues and eigenvectors.

ALGORITHM 255
COMPUTATION OF FOURIER COEFFICIENTS [C6]

LINDA TEIJELO (Recd. 18 Nov. 1964 and 25 Nov. 1964)
Stanford Computation Ctr., Stanford U., Calif.

```
procedure FOURIER(F, eps, subdivmax, m, cosine, sine, cint,
  sint);
  value eps, subdivmax, m, cosine, sine;  real eps, cint, sint;
  Boolean cosine, sine;  integer subdivmax, m;
    real procedure F;
comment FOURIER computes the Fourier coefficients cint =
  $\int_0^1 F(x)cos(m\pi x)\,dx$ (if cosine is true) and/or sint = $\int_0^1 F(x)$
  $sin(m\pi x)\,dx$ (if sine is true), where m > 0. The method is that
  of Filon (for a brief exposition see [1] and for Filon's original work
  see [2] or [3]). Computation is terminated when the number of
  times the interval [0,1] has been halved (n) has exceeded sub-
  divmax (10 is suggested), or when n > 5 and two successive ap-
  proximations of the integral agree to within eps ($10^{-7}$ is sug-
  gested) times the value of the last approximation. In the former
  case, cint or sint is assigned the value of the last approximation.
  The condition n > 5 is imposed because of substantial cancella-
  tions which may take place during the early stages of sub-
  dividing;
begin real sumcos, sumsine, oddcos, oddsine, pi, a, b, g, t, h, p, k,
  c0, c1, s0, s1, int1, int2, prevint1, prevint2, tn1, t3, temp;
  integer n, i;  Boolean bool;
  bool := false;  pi := 3.14159265359;  k := m × pi;
  sumcos := (F(1.0) × cos(k)+F(0)) × .5;
  sumsine := F(1.0) × sin(k) × .5;
L0:  n := 1;  h := 0.5;  t := .5 × k;  tn1 := 1;
L1:  c0 := cos(2.0×t);  c1 := cos(t);
  s0 := sin(2.0×t);  s1 := sin(t);
  t3 := t↑3;  p := c1 × s1;
  a := (t↑2−s1↑2×2.0+t×p)/t3;
  b := (2.0×(t×(c1↑2+1.0)−2.0×p))/t3;
  g := 4.0 × (−t×c1 + s1)/t3;
  if bool then go to L2;
  if sine then
    begin
      oddsine := F(h) × s1;
      for i := 2 step 1 until tn1 do
      begin temp := c1 × c0 − s1 × s0;
        s1 := s1 × c0 + c1 × s0;
        c1 := temp;
        oddsine := F((2×i−1)×h)×s1 + oddsine
      end;
      if n = 1 then
        begin n := 2;  h := .25;  t := .25×k;  tn1 := 2;
          prevint2 := (a×(F(0)−F(1.0)×cos(k))+
            b×sumsine+g×oddsine) × .5;
          sumsine := sumsine + oddsine;  go to L1
        end
      else
        begin int2 := h × (a×(F(0)−F(1.0)×cos(k))+
          b×sumsine+g×oddsine);
          if abs(prevint2−int2)<eps×int2∧n>5 then
            begin sint := int2;  bool := true;  go to L0 end
          else
            begin n := n + 1;
              if n > subdivmax then
                begin bool := true;
                  sint := int2;  go to L0
                end;
              sumsine := sumsine + oddsine;  h := .5 × h;
                t := .5 × t;  tn1 := 2 × tn1;
              prevint2 := int2;  go to L1
            end
        end
    end of sine computations;
L2:  if cosine then
    begin
      oddcos := F(h) × c1;
      for i := 2 step 1 until tn1 do
      begin temp := c1 × c0 − s1 × s0;
        s1 := s1 × c0 + c1 × s0;
        c1 := temp;
        oddcos := F((2×i−1)×h) × c1 + oddcos
      end;
      if n = 1 then
        begin n := 2;  h := .25;  t := .25 × k;  tn1 := 2;
          prevint1 := (a×F(1.0)×sin(k)+b×sumcos+g×oddcos)
            × .5;
          sumcos := sumcos + oddcos;  bool := true;  go to L1
        end
      else
        begin int1 := h ×(a×F(1.0)×sin(k)+b×sumcos+g×
          oddcos);
          if abs(prevint1−int1) < eps × int1 ∧ n > 5 then
            begin cint := int1;  go to exit end
          else
            begin n := n + 1;
              if n > subdivmax then begin cint := int1;
                go to exit end;
              sumcos := sumcos + oddcos;  h := .5 × h;
                t := .5 × t;  tn1 := 2 × tn1;
              prevint1 := int1;  go to L1
            end
        end
    end of cosine computations;
exit:  end FOURIER
```

REFERENCES:

1. HAMMING, R. W. *Numerical Methods for Scientists and Engineers.* McGraw-Hill, 1962, pp. 319–321.
2. TRANTER, C. J. *Integral Transforms in Mathematical Physics.* Methuen & Co., Ltd., 1951, pp. 67–72.
3. FILON, L. N. G. On a quadrature formula for trigonometric integrals. Proc. Roy. Soc. Edinburgh *49*, 1928–29, 38–47.

CERTIFICATION OF ALGORITHM 255 [C6] COMPUTATION OF FOURIER COEFFICIENTS

[Linda Teijelo, *Comm. ACM 8* (May 1965), 279]

GILLIAN HALL* AND VALERIE A. RAY† (Recd. 31 Mar. 1969 and 1 July 1969)

National Physical Laboratory, Teddington, Middlesex, England

* M.R.C. team, Division of Computer Science (formerly of Division of Numerical and Applied Mathematics).

† Division of Numerical and Applied Mathematics.

KEY WORDS AND PHRASES: numerical integration, Fourier coefficients, Filon's method

CR CATEGORIES: 5.16

The algorithm was translated using the KDF9 Kidsgrove ALGOL compiler, and needed the following correction.

The tests for convergence on lines 51 and 83 should read respectively:

```
if abs(prevint2−int2) < eps × abs(int2) ∧ n > 5 then
if abs(prevint1−int1) < eps × abs(int1) ∧ n > 5 then
```

With this alteration, the program was tested successfully on a series of functions $F(x)$ using a range of values of *m* and *eps* for each function. The parameter *subdivmax* was set at the recommended value, 10. For $F(x) = x^2$, for which the method is exact, results were obtained correct to machine accuracy, i.e. 10½ decimal places.

Remarks. (i) It would be better to declare the identifier *tn*1 as type **integer**, i.e. to replace lines 20 and 21 of the text by:

```
c0, c1, s0, s1, int1, int2, prevint1, prevint2, t3, temp;
integer n, i, tn1; Boolean bool;
```

(ii) There is no indication, after execution of the algorithm, whether the computation was terminated because of apparent convergence or because the number of times, *n*, that the interval was halved became greater than *subdivmax*. The following modification provides such an indication; it has the effect that *cosine* and *sine* will retain their entry values except in the case where *cosine* or *sine* has the value *true* on entry and *n* becomes greater than *subdivmax* in the course of computation. In this case the value on exit will be *false*.

Line 3 becomes:

```
value eps, subdivmax, m; real eps, cint, sint;
```

Line 57 becomes:

```
sint := int2; sine := false; go to L0
```

Line 88 becomes:

```
cosine := false; go to exit end;
```

(iii) To avoid the repeated evaluation of $F(0)$, $F(1.0)$ the following modification is suggested:

Declare a new variable *term*1 of type **real** on line 20.

Replace lines 23 and 24 by:

```
term1 := F(1.0) × cos(k);
sumcos := (F(0)+term1) × 0.5;
sumsine := 0;
term1 := 2 × (sumcos−term1);
```

Replace lines 44, 45 and 49, 50 by:

```
prevint2 := (a×term1+b×sumsine+g×oddsine) × 0.5;
begin  int2 := h × (a×term1+b×sumsine+g×oddsine);
```

Replace lines 76, 77 and 81, 82 by:

```
prevint1 := (b×sumcos+g×oddcos) × 0.5;
begin int1 := h × (b×sumcos+g×oddcos);
```

The work described above has been carried out at the National Physical Laboratory.

ALGORITHM 256
MODIFIED GRAEFFE METHOD [C2]

A. A. Grau (Recd. 29 July 1964, 23 Oct. 1964 and 18 Jan. 1965)
Northwestern University, Evanston, Illinois

The algorithm given here mechanizes a modified form of the Graeffe process designed to avoid an expanding number range. This was discussed in [1]; the notation used below is the same as in that article.

Let the given polynomial be

$$a_0x^n + \cdots + a_n ;$$

the degree n and the array of coefficients a are input parameters of the procedure. An additional input parameter w is used to determine the number of stages needed to obtain a desired order of resolution; this may be considered to be roughly the number of significant decimal places expected in the zeros of the polynomial.

The algorithm finds the moduli, d_s $(s = 1, \cdots, n)$, of the zeros of the polynomial and the number of stages used for this, p. If the algorithm succeeds, the output parameter q is set equal to 0; otherwise, the value of q serves as the indicator for the reason of failure: $q = 1$ if the polynomial has a zero-valued coefficient, and $q = 2$ if a zero-valued divisor is encountered somewhere in the process. In either case, the moduli of the zeros are not found. Apart from these two cases, the algorithm applies generally; this includes the cases where some zeros have equal moduli or are imaginary.

The algorithm has been tested with polynomials of degree up to 10, including ill-conditioned cases such as polynomials with one or more sets of multiple or imaginary zeros. The algorithm has been compiled as it stands using both the Oak Ridge Algol Translator for the Control Data 1604 and the Share Algol Translator for the IBM 709/7090. In the case of the latter, one change as noted in a comment had to be made; this is presumably no longer necessary in a revision of the translator.

Garwick's device [2] is used as convergence criterion in both root extraction and the basic process. From w and the number of stages determined from it, it is possible to conclude whether some zeros may be considered to be of equal moduli; in such cases an adjustment of their values is possible and is made.

The quantities used in the modified Graeffe process are related to those occurring in the ordinary root-squaring process. This implies that in general the limitations of the Graeffe process (see, for example [3, pp. 67–69]) hold also in the modified process; the most serious of these is that initially the condition of successive polynomials may deteriorate.

An expanding number range is avoided by introducing at each step arithmetic divisions. It follows that if c_i is near zero, over- and underflow can occur in computing subsequent quantities. In the usual machine system, such a condition results in the automatic termination of computation; in this case this is not serious. In an Algol system where this is not true, a very unsatisfactory arrangement generally, machine-dependent facilities must be added to the algorithm to obtain the same effect; the Algol language contains no way of doing this. Theoretically a bridging mechanism is possible to work around near-zero divisors, but this has not been attempted here.

The modified process can be expected to perform somewhat better than the standard process in the case of equal moduli.

```
procedure Modified Graeffe (w, n, a, d, p, q);
  value w, n;  integer w, n, p, q;  array a, d;
begin
  real aa, eps, eps2, h, h1, h2, hh2, m, nh2;
  integer i, k, k0, k00, s, s3;
  array c[0:n], d1, hh[1:n], e[1:n, 1:n/2];  comment  Using the
     SHARE processor, the last subscript bound n/2 was replaced by
     entier(n/2);
  eps := eps2 := 10−5;
  k00 := 40;  comment  This is the maximum number of stages
     needed on the CDC 1604 where about 10 significant decimal
     figures may be obtained. On the IBM machines it is less, but
     the figure was not changed for such use;
  for s := 0 step 1 until n do
  begin if a[s] = 0 then begin q := 1;  go to out end end;
Determine the number of stages:
  k0 := entier (3.56 × w + 3.21);
  if k0 > k00 then k0 := k00;
Initialization:
  for s := 1 step 1 until n do
  begin
    if s + s > n then s3 := n−s else s3 := s;
    for i := 1 step 1 until s3 do
      e[s, i] := a[s+i] × a[s−i]/(a[s+i−1] × a[s−i+1]);
    d1[s] := abs (a[s]/a[s−1])
  end;
  c[0] := c[n] := 1;
  m := 1;
Main loop:
  for k := 1 step 1 until k0 do
  begin
    m := m/2;
    for s := 1 step 1 until n−1 do
    begin
      if s + s > n then s3 := n−s else s3 := s;
      h := 0;
      for i := s3 step −1 until 1 do
        h := (1−h) × e[s, i];
      c[s] := 1 − 2 × h;
      if c[s] = 0 then
      begin q := 2;  go to out end
    end;
    for s := 1 step 1 until n do
    begin
      if s + s > n then s3 := n−s else s3 := s;
      for i := 1 step 1 until s3 do
      begin
        h := (c[s+i]/c[s+i−1]) × e[s, i];
        e[s, i] := (c[s−i]/c[s−i+1]) × e[s, i] × h;
      end;
      comment  In the paper [1] on which the algorithm is based,
        there is an error in equation (13) and results derived from
        it. The equation should be
```

$$e_{si}^{(k+1)} = [e_{si}^{(k)}]^2 \frac{c_{s+i}^{(k+1)} c_{s-i}^{(k+1)}}{c_{s+i-1}^{(k+1)} c_{s-i+1}^{(k+1)}} ;$$

```
 Root extraction:
       aa := abs (c[s]/c[s-1]);
       comment If the ↑ operation is suitably implemented for
         fractional exponent, the following 12 lines may be replaced
         by
                     hh[s] := h1 := aa ↑ (1/2↑k);
       h1 := h := 1 + (aa-1) × 10-2 × m;
       nh2 := 1;
 AB:   for i := 1 step 1 until k do h := h × h;
       h2 := (aa/h-1) × m;
       h := h1 := h1 + h1 × h2;
       hh2 := abs (h2);
       if hh2 > eps then go to AB;
       if hh2 < nh2 ∧ hh2 ≠ 0 then
       begin
         nh2 := hh2;  go to AB
       end;
       hh[s] := h1;
       d1[s] := d1[s] × h1
     end;
     h := 0;
     for s := 1 step 1 until n do
     begin
       h1 := abs (hh[s]-1);
       if h1 > eps then go to AC;
       if h1 > h then h := h1
     end;
     if h < eps2 ∧ h ≠ 0 then
     begin eps2 := h;  go to AC end;
     go to Root determination;
 AC:  end Main loop;
   k := k0;
 Root determination:
   q := 0;  p := k;  s := 1;
 BA:  for i := s step 1 until n do
   begin
     if abs (c[i]-1) < eps2 then
     begin k := i;  go to AE end
   end;
   k := n;
 AE:  if k = s then
   begin
     d[s] := d1[s];  go to AG
   end
   else
   begin
     aa := 1;
     for i := s step 1 until k do
       aa := aa × d1[i];
     comment If the ↑ operation is suitably implemented for
       fractional exponents, the following 13 lines may be replaced
       by
                     h := aa ↑ (1/(k-s+1));
     h1 := d1[s];
     nh2 := 1;
 AF: h := 1;
     for i := s step 1 until k do
     h := h × h1;
     h2 := (aa/h-1)/(k-s+1);
     h1 := h1 + h1 × h2;
     hh2 := abs (h2);
     if hh2 > eps then go to AF;
     if hh2 < nh2 ∧ hh2 ≠ 0 then
     begin
       nh2 := hh2;  go to AF
     end;
     for i := s step 1 until k do d[i] := h1
   end;
 AG: if k = n then go to out;
   s := k + 1;
   go to BA;
 out:
 end Modified Graeffe
```

TABLE 1

n	*Coefficients* (a_s)	*Actual Zeros*	p	*Computed Moduli* (d_s)
		Results from CDC 1604		
4	1 −1 −5 −1 −6	±*i* −2 3	29	3.000000000 2.000000000 . 1.000000000 1.000000000
4	1 2 3 2 2	±*i* −1±*i*	35	1.414213563 1.414213563 1.000001068 .9999989324
4	1 4 6 4 1	−1 (four-fold)	35	1.002108373 .9999999404 .9999999400 .9978961853
5	1 −5 −15 125 −226 120	1 2 3 4 −5	9	5.000000000 4.000000001 2.999999999 2.000000000 1.000000000
5	1 5 10 10 5 1	−1 (five-fold)	21	1.003023179 1.000737942 1.000737942 .9977555433 .9977555433
6	1 1 −45 35 524 −1236 720	1 2 3 4 −5 −6	9	5.999999999 5.000000004 3.999999998 3.000000001 2.000000000 1.000000000
6	1 6 15 20 15 6 1	−1 (six-fold)	22	1.009739721 1.009739721 1.000072156 1.000072156 .9902827716 .9902827715
		Results from IBM 709		
6	1 6 15 20 15 6 1	−1 (six-fold)	22	1.0442011 1.0219216 1.0219216 .97855264 .97855264 .95767000
10	1 10 45 120 210 252 210 120 45 10 1	−1 (ten-fold)	23	1.1896983 1.1896983 1.0977241 1.0977241 1.0001204 1.0001204 .91099190 .91099190 .84044056 .84044056
10	1 −55 1320 −18150 157773 −902055 3416930 −8409500 12753576 −10628640 3628800	1 2 3 4 5 6 7 8 9 10	10	10.001153 8.9947183 8.0090868 6.9926022 6.0027695 4.9996995 3.9999811 2.9999883 2.0000007 1.0000000

Tests. Some of the tests (Table 1) were run on the CDC 1604 using an earlier version of the algorithm; minor improvements incorporated afterwards should not affect the results substantially. The results obtained using the Share Algol translator and the IBM 709 suffer in comparison to those obtained on the 1604 for two main reasons: (1) significance of floating-point numbers is 27 bits vs. 35, and (2) input conversion routines introduce greater perturbations into input numbers. The last cases given are very poorly conditioned, so that the rather poor results should not be especially surprising.

Thanks and acknowledgements are due to several members of the Mathematics Division of Oak Ridge National Laboratory for running tests on the Control Data 1604, and to Mrs. Virginia Klema for running tests on the IBM 709 computer at Northwestern University.

References:

1. Grau, A. A. On the reduction of number range in this use of the Graeffe process. *J. ACM 10* (1963), 538–544.
2. Garwick, J. V. The limit of a converging sequence. *Nord Tidskr. Informationsbehandlung (BIT) 1* (1961), 64.
3. Wilkinson, J. H. *Rounding Errors in Algebraic Procesess.* Prentice-Hall, New York, 1964.

REMARK ON ALGORITHM 256 [C2] MODIFIED GRAEFFE METHOD [A. A. Grau, *Comm. ACM 8* (June 1965), 379]

G. Stern (Recd. 8 Mar. 1965 and 24 Mar. 1965)
University of Bristol Computer Unit, Bristol 8, England

This procedure was tested on an Elliott 503 using the two simplifications noted in the comments on page 380. When the 16th line from the bottom of page 380, first column, was changed to read

$$h1 := aa \uparrow (1/(k-s+1));$$

(as suggested in a private communication from the author) correct results were obtained.

ALGORITHM 257
HAVIE INTEGRATOR [D1]

Robert N. Kubik (Recd. 9 June 1964 and 21 Dec. 1964)
The Babcock & Wilcox Co. Lynchburg, Viriginia

```
real procedure havieintegrator (x, a, b, eps, integrand, m);
  value a, b, eps, m;  integer m;
  real integrand, x, a, b, eps;
comment This algorithm performs numerical integration of definite integrals using an equidistant sampling of the function and repeated halving of the sampling interval. Each halving allows the calculation of a trapezium and a tangent formula on a finer grid, but also the calculation of several higher order formulas which are defined implicitly. The two families of approximate solutions will normally bracket the value of the integral and from these convergence is tested on each of the several orders of approximation. The algorithm is based on a private communication from F. Håvie of the Institutt for Atomenergi Kjeller Research Establishment, Norway. A FORTRAN version of the algorithm is in use on the Philco-2000. A few test cases have been run on the Burroughs B5000. In particular, a and b are the lower and upper limits of integration, respectively, eps is the convergence criterion, integrand is the value of the function to be integrated (sampled), and m is the maximum order approximation to be considered in attempting to satisfy the eps convergence criterion. If convergence is not gained, then the value returned is that of the nonlocal variable, mask. The parameter integrand must be an expression involving the variable of integration x. See the driver program of this algorithm for examples of the procedure call;
begin real h, endpts, sumt, sumu, d;
  integer i, j, k, n;
  real array t, u, tprev, uprev[1:m];
  x := a;  endpts := integrand;  x := b;  endpts := 0.5 ×
    (integrand+endpts);
  sumt := 0.0;  i := n := 1;  h := b − a;
estimate:  t[1] := h × (endpts+sumt);  sumu := 0.0;
  comment t[1] = h × (0.5×f[0]+f[1]+f[2]+···+0.5×f[2^(i−1)]);
  x := a − h/2.0;
  for j := 1 step 1 until n do
  begin
    x := x + h;  sumu := sumu + integrand
  end;
  u[1] := h × sumu;  k := 1;
  comment u[1] = h × (f[1/2]+f[3/2]+···+f[(2^i−1)/2]), k
    corresponds to approximate solution with truncation error
    term of order 2k;
test:  if abs(t[k]−u[k]) ≤ eps then
  begin
    havieintegrator := 0.5 × (t[k]+u[k]);  go to exit
  end;
  if k ≠ i then
  begin
    d := 2 ↑ (2×k);
    t[k+1] := (d×t[k]−tprev[k])/(d−1.0);
    tprev[k] := t[k];
    u[k+1] := (d×u[k]−uprev[k])/(d−1.0);
    uprev[k] := u[k];
    comment This implicit formulation of the higher order integration formulas is given in [ROMBERG, W. Vereinfachte Numerische Integration. Det Kong. Norske Videnskabers Selskabs Forhandl. 28, 7 (1955), Trondheim; and in STIEFEL, E. Einführung in der Numerische Mathematik. Teubner Verlagsges., Stuttgart, 1961, pp. 131–136. (English translation: An Introduction to Numerical Mathematics, Academic Press, New York, 1963, pp. 149–155)]. See also Algorithm 60 where the same implicit relationship is used to calculate t[k+1] only;
    k := k + 1;
    if k = m then
    begin
      havieintegrator := mask;  go to exit
    end;
    go to test
  end;
  h := h/2.0;  sumt := sumt + sumu;
  tprev[k] := t[k];  uprev[k] := u[k];
  i := i + 1;  n := 2 × n;
  go to estimate;
exit:  end havieintegrator
```

Following is a driver program to test havieintegrator.

```
begin comment First test case, y = ∫^π/2_0 cos x dx = 1.0
    (0.9999999981 as executed on the B5000), is an example of the
    higher order approximations yielding fast convergence as in
    Algorithm 60; second test case, y = ∫^4.3_0 e^(−x²) dx = .8862269255
    (.8862269739 as executed on the B5000), is an example where
    this algorithm is superior to Algorithm 60 because the higher
    order approximations converge more slowly than the linear
    approximations; see also [THACHER, H. C., JR., Remark on
    Algorithm 60. Comm. A.C.M. 7 (July 1964), 420];
  real a, b, eps, mask, y, answer;
  a := 0.0;  b := 1.5707963;  eps := 0.000001;  mask := 9.99;
  answer := havieintegrator (y, a, b, eps, cos(y), 12);
  outreal (1, answer);
  a := 0.0;  b := 4.3;
  answer := havieintegrator (y, a, b, eps, exp(−y×y), 12);
  outreal (1, answer);
end
```

CERTIFICATION OF ALGORITHM 257 [D1]
HAVIE INTEGRATOR [Robert N. Kubik, *Comm. ACM* *8* (June 1965), 381]

Kenneth Hillstrom (Recd. 28 Feb. 1966, 29 Apr. 1966 and 15 July 1966)
Applied Mathematics Division, Argonne National Laboratory, Argonne, Illinois

Work performed under the auspices of the U.S. Atomic Energy Commission.

Havie Integrator was coded in CDC 3600 FORTRAN. This routine and a FORTRAN-coded Romberg integration routine based upon Algorithm 60, Romberg Integration [*Comm. ACM* *4* (June 1961), 255] were tested with five and four integrands, respectively.

The results of these tests are tabulated below. (The ALGOL-coded Havie routine was transcribed and tested for the two integrands used by Kubik, with identical results in both cases.)

In the following table, A is the lower limit of the interval of integration, B is the upper limit, EPS the convergence criterion, VI the value of the integral and VA the value of the approximation.

Integrand	*A*	*B*	*EPS*	*VI*	*Routine*	*VA*	*Number of Function Evaluations*
$\cos x$	0	$\pi/2$	10^{-6}	1.0	Havie	0.9999999981	17
					Romberg	1.000000000	17
e^{-x^2}	0	4.3	10^{-6}	0.886226924	Havie	0.886226924	17
					Romberg	0.886336925	65
$\ln x$	1	10	10^{-6}	14.0258509	Havie	14.02585084	65
					Romberg	14.02585085	65
$\left(\frac{(x)^{1/2}}{e^{x-4}+1}\right)$	0	20	10^{-6}	5.7707276	Havie	5.770724810	32,769
					Romberg	5.770724810	16,385
$\cos(4x)$	0	π	10^{-6}	0.0	Havie	3.1415926536	3[a]

[a] Since in the Havie procedure, the sample points of the interval, chosen for function evaluation, are determined by halving the interval and are, therefore, function-independent, there are functions for which the convergence criterion is satisfied before the requisite accuracy is obtained. An example is the integrand $f(x) = \cos(4x)$ integrated over the interval $[0, \pi]$. The value obtained from the routine is $= \pi$. The true value of the integral is 0.

This inherent limitation applies to all integration algorithms that obtain sample points in a fixed manner.

CERTIFICATION OF ALGORITHM 257 [D1] HAVIE INTEGRATOR [Robert N. Kubik, *Comm. ACM 8* (June 1965), 381]

I. Farkas (Recd. 29 Apr. 1966 and 18 Aug. 1966)
Institute of Computer Science, University of Toronto, Toronto 5, Ont., Canada

Havieintegrator was translated with some modifications into Fortran IV and was run on the IBM 7094 II at the Institute of Computer Science, University of Toronto. To reduce the effect of roundoff, the calculations were carried through in double precision internally and the result was rounded to single precision. The main change made was that the parameters x and *integrand* in *havieintegrator* were replaced by a single parameter of type FUNCTION in Fortran IV. The other change was that *mask* was removed. The maximum order of approximation was kept less than or equal to 25, and convergence was obtained in every case.

The results obtained for the two test cases were in agreement with the author's result. Besides, 14 other successful tests were made and those shown in Table I are typical.

TABLE I

Integrand	*A*	*B*	*True value*	*eps*	*Error × 10⁸*	*Order required*
e^x	0.0	1.0	1.7182818	10^{-6}	0	3
				10^{-4}	240	2
				10^{-2}	3700	2
x^{12}	0.01	1.1	.26555932	10^{-6}	−2	4
				10^{-4}	59	3
				10^{-2}	36041	2
$\sqrt{x}$	0.0	1.0	.66666667	10^{-6}	−27	3
				10^{-4}	−1982	2
				10^{-2}	−126848	2
$1/\sqrt{x}$	0.01	1.0	1.8000000	10^{-6}	0	3
				10^{-4}	140	2
				10^{-2}	790	2

Like other integration algorithms that determine sample points in the interval in a deterministic manner, *havieintegrator* may fail in certain instances. For example, any integrand with the property that $f(a) = f(b) = f[(a + b)/2)]$ will lead to the value $(b - a)f(a)$ which will in general not be an acceptable approximation to $\int_a^b f(x)\,dx$. Thus $\int_0^{2\pi} \sin^2 x\,dx$ leads to 0. Moreover, $\int_0^{90} xe^{-x}\,dx$ leads to "almost zero" (in fact, 5.7966×10^{-17}).

ALGORITHM 258
TRANSPORT [H]

G. BAYER (Recd. 4 May 1964 and 4 Mar. 1965);
Technische Hochschule, Braunschweig, Germany

```
procedure transport (c, x, a, b, m, n, inf, cost);
  value m, n, inf;  integer m, n, inf, cost;
    integer array c, x, a, b;
comment The parameters are c[i, j] array of costs, the quantities
  available a[i], the quantities required b[j],  i = 1, ··· , m, j =
  1, ··· , n. Sum of a[i] = sum of b[j].  inf has to be the greatest
  positive integer within machine capacity, all quantities have to
  be integer. The flows x[i, j] are computed by the "primal-dual-
  algorithm," cited in [HADLEY, G.  Linear Programming. Read-
  ing, London, 1962, pp. 351–367]. The procedure follows the de-
  scription given on p. 357. Multiple solutions are left out of
  account;
begin integer i, j, p, h, k, y, t, l;
  integer array v, xsj, s, r, listv[1:n], u, xis, d, g, listu[1:m];
  Boolean array xb[1:m, 1:n];
  integer procedure sum(i, a, b, x);  value a, b;
    integer i, a, b, x;
    begin integer s;
      s := 0;
      for i := a step 1 until b do s := s + x;
      sum := s
    end;
  comment  Array xb for notation of "circled cells," listu and
    listv lists of labeled rows and columns. Other notations follow
    Hadley;
  for i := 1 step 1 until m do xis[i] := a[i];
  for j := 1 step 1 until n do xsj[j] := b[j];
  for i := 1 step 1 until m do
  begin h := inf;  for j := 1 step 1 until n do
    begin x[i, j] := 0;  p := c[i, j];  if p < h then h := p end;
    u[i] := h;
    for j := 1 step 1 until n do
      xb[i, j] := if c[i, j] = h then true else false
  end u[i];
  for j := 1 step 1 until n do
  begin h := inf;
    for i := 1 step 1 until m do
    begin if xb[i, j] then
      begin v[j] := 0;  go to aa end;
      d[i] := p := c[i, j] − u[i];
      if p < h then h := p
    end;
    v[j] := h;
    for i := 1 step 1 until m do
    begin if d[i] = h then xb[i, j] := true end;
aa:
  end v[j];
  for j := 1 step 1 until n do listv[j] := 0;
  for i := 1 step 1 until m do listu[i] := 0;
s2:  for i := 1 step 1 until m do
  begin for j := 1 step 1 until n do
    begin if xb[i, j] then
      begin h := x[i, j] := if xsj[j] ≤ xis[i]
        then xsj[j] else xis[i];
        xsj[j] := xsj[j] − h;
        xis[i] := xis[i] − h
      end
    end
  end;
s03:  if sum(j, 1, n, xsj[j]) = 0 then go to s6;
  for j := 1 step 1 until n do s[j] := r[j] := 0;
  h := 0;  k := 1;
s3:  for i := 1 step 1 until m do
  begin if xis[i] > 0 then
    begin d[i] := xis[i];  g[i] := 2 × n;
      for j := 1 step 1 until n do
      begin if xb[i, j] ∧ r[j] = 0 then
        begin s[j] := d[i];  r[j] := i;  listv[k] := j;  k := k + 1;
          if xsj[j] > h then
          begin h := xsj[j];  p := j end
        end
      end
    end
    else d[i] := g[i] := 0
  end;
s53:  if k = 1 then go to s13;
  l := 1;
  for k := 1 step 1 until n do
  begin j := listv[k];  listv[k] := 0;  if j = 0 then go to s33;
    for i := 1 step 1 until m do
    begin if xb[i, j] ∧ x[i, j] > 0 ∧ g[i] = 0 then
      begin d[i] := if x[i, j] ≤ s[j]
        then x[i, j] else s[j];
        g[i] := j;  listu[l] := i;  l := l + 1
      end
    end
  end;
s33:  if l = 1 then go to s13;
  k := 1;
  for l := 1 step 1 until m do
  begin i := listu[l];  listu[l] := 0;  if i = 0 then go to s43;
    for j := 1 step 1 until n do
    begin if xb[i, j] ∧ r[j] = 0 then
      begin s[j] := d[i];  r[j] := i;  listv[k] := j;  k := k + 1;
        if xsj[j] > h then
        begin h := xsj[j];  p := j end
      end
    end
  end;
s43:  go to s53;
s13:;  comment  end of labeling process;
  if h > 0 then go to s4 else
    if sum(j, 1, n, xsj[j]) = 0 then go to s6 else go to s5;
s4:  k := p;
  h := if s[k] < xsj[k] then s[k] else xsj[k];
s41:  y := r[k];  x[y, k] := x[y, k] + h;
  xis[y] := xis[y] − h;  xsj[k] := xsj[k] − h;
  t := g[y];  if t = 2 × n then go to s03;  x[y, t] := x[y, t] − h;
  xis[y] := xis[y] + h;  xsj[t] := xsj[t] + h;  k := t;  go to s41;
s5:  h := inf;
  for i := 1 step 1 until m do
  for j := 1 step 1 until n do
  begin if g[i] ≠ 0 ∧ r[j] = 0 then
    begin p := c[i, j] − u[i] − v[j];
      if p < h then h := p
    end
  end;
  for i := 1 step 1 until m do
  begin if g[i] ≠ 0 then u[i] := u[i] + h end;
  for j := 1 step 1 until n do
  begin if r[j] ≠ 0 then v[j] := v[j] − h end;
  for i := 1 step 1 until m do
  for j := 1 step 1 until n do
  begin if c[i, j] = u[i] + v[j] then xb[i, j] := true end;
  go to s03;
s6: cost := sum(i, 1, m, a[i]×u[i]) + sum(j, 1, n, b[j]×v[j])
end;
```

REMARK ON ALGORITHM 258 [H]
TRANSPORT [G. Bayer, *Comm. ACM 8* (June 1965), 381]

G. Bayer (Recd. 11 June 1965)
Technische Hochschule, Braunschweig, Germany

The following correction should be made in the procedure. Change the second line above the label *s*6 from

```
    begin if c[i,j] = w[i]+v[j] then xb[i,j] := true end;
```

to

```
    xb[i,j] := c[i,j] = u[i] + v[j];
```

CERTIFICATION OF:
ALGORITHM 258 [H]
TRANSPORT
[G. Bayer, *Comm. ACM 8* (June 1965), 381]
ALGORITHM 293 [H]
TRANSPORTATION PROBLEM
[G. Bayer, *Comm. ACM 9* (Dec. 1966), 869]

Lee S. Sims (Recd. 21 Feb. 1967 and 17 Mar. 1967)
Kates, Peat, Marwick & Co., Toronto, Ont., Canada

Both of these algorithms were coded in Extended Algol 60 and tested on a Burroughs B5500. Three problems were solved correctly, one of them being of medium size (55 × 167). On this larger problem *transp*1 was found to be about twice as fast as *transport*.

In coding and debugging *transp*1 three apparent errors were found. In the right-hand column on page 870, after line 27 which is

```
  i := listu[u];  nlvi := nlv[i];
```

a line is missing. This line should read

```
  for s := (i−1) × n + 1 step 1 until nlvi do
```

Also in the right-hand column, the line

```
  s4: ;
```

should be inserted ahead of line −12, which begins

```
  comment  Step 4. A column j with b[j] has been labeled, b[j]
```

On page 871, in the left-hand column, line −22 which reads

```
  for s := 1 step 1 until n do
```

should read

```
  for s := l step 1 until n do
```

ALGORITHM 259
LEGENDRE FUNCTIONS FOR ARGUMENTS LARGER THAN ONE* [S16]

Walter Gautschi (Recd. 5 Mar. 1965)
Purdue University, Lafayette, Ind. and Argonne National Laboratory, Argonne, Ill.

```
begin
comment Control is transferred to a nonlocal label, called
  alarm, whenever the input variables are not in the intended
  range;
procedure integer Legendre 1 (x, a, nmax, P);
  value x, a, nmax;  integer a, nmax;  real x;  array P;
comment This procedure generates the associated Legendre
  functions of the first kind,
```

$$P_a^n(x) = \frac{(x^2-1)^{n/2}}{2^a\ a!}\frac{d^{a+n}}{dx^{a+n}}(x^2-1)^a,$$

for $n = 0(1)nmax$, assuming $a \geq 0$ an integer, and $x > 1$. The results are stored in the array P. The method of computation is derived from the (finite) continued fraction

$$(n+a)F_n/F_{n-1} = \frac{(n+a)\ (a+1-n)}{nx_1+}\ \frac{(n+a+1)\ (a-n)}{(n+1)x_1+}\ \frac{(n+a+2)\ (a-n-1)}{(n+2)x_1+}\cdots\frac{2a\cdot 1}{ax_1}\qquad (1 \leqq n \leqq a),$$

where $F_n = P_a^n(x)/(n+a)!$, $x_1 = 2x(x^2-1)^{-\frac{1}{2}}$, and the identity

$$F_0 + 2\sum_{n=1}^{a} F_n = [x+(x^2-1)^{\frac{1}{2}}]^a/a!.$$

If x is very close to 1, the computation of x_1 is subject to cancellation of significant digits. In such cases it would be better to use $y = x-1$ as input variable, and to compute $(x^2-1)^{\frac{1}{2}}$ by $[y(2+y)]^{\frac{1}{2}}$ everywhere in the procedure body;

```
begin integer n;  real x1, c, sum, r, s;
  array Rr[0:nmax-1];
  if x < 1 ∨ a < 0 ∨ nmax < 0 then go to alarm;
  if x = 1 ∨ a = 0 then
  begin
    P[0] := 1;  for n := 1 step 1 until nmax do P[n] := 0;
    go to L
  end;
  for n := a+1 step 1 until nmax do P[n] := 0;
  x1 := sqrt (x↑2-1);
  c := 1;  for n := 2 step 1 until a do c := n × c;
  sum := (x+x1)↑a/c;  x1 := 2 × x/x1;
  r := s := 0;
  for n := a step - 1 until 1 do
  begin
    r := (a+1-n)/(n×x1+(n+a+1)×r);  s := r × (2+s);
    if n ≤ nmax then Rr[n-1] := r
  end;
  P[0] := c × sum/(1+s);
  for n := 0 step 1 until if nmax ≤ a then nmax-1 else a-1 do
    P[n+1] := (n+a+1) × Rr[n] × P[n];
L: end integer Legendre 1;
procedure integer Legendre 2(x, m, nmax, d, Q);
  value x, m, nmax, d;  integer m, nmax, d;  real x;  array Q;
comment This procedure generates to d significant digits the
  associated Legendre functions of the second kind, Qnm(x), for
```

$n = 0(1)nmax$, assuming $m \geq 0$ an integer, and $x > 1$. The results are stored in the array Q. The procedure first generates $Q_0{}^m(x)$ from the recurrence relation

$$Q_n^{r+1} + \frac{2rx}{(x^2-1)^{\frac{1}{2}}}Q_n{}^r + (r+n)(r-n-1)Q_n^{r-1} = 0 \qquad (r = 1,2,\cdots,m-1) \tag{1}$$

with $n = 0$, and the initial values

$$Q_0{}^0(x) = \frac{1}{2}\ln\frac{x+1}{x-1},\qquad Q_0{}^1(x) = -(x^2-1)^{-\frac{1}{2}}.$$

Then a variant of the backward recurrence algorithm of J. C. P. Miller is applied to the recursion

$$(n-m+1)Q_{n+1}^m - (2n+1)xQ_n{}^m + (n+m)Q_{n-1}^m = 0 \qquad (n=1,\ 2,\ 3,\cdots). \tag{2}$$

(For more details see [2]. See also [4] for a very similar algorithm.) If $m > 1$, the leading coefficient in (2) vanishes for $n = m - 1$, which invalidates the theoretical justification for the backward recurrence procedure. Nevertheless, it appears that the procedure produces valid results for arbitrary $m \geq 0$. Convergence of the backward recurrence algorithm is slow for x near 1, but improves rapidly as x increases;

```
begin integer n, nu, p;  real x1, Q0, Q1, Q2, epsilon, r;
  array Qapprox, Rr[0: nmax];
  if x ≤ 1 ∨ nmax < 0 ∨ m < 0 then go to alarm;
  x1 := sqrt (x↑2-1);
  Q1 := .5 × ln ((x+1)/(x-1));
  if m = 0 then Q[0] := Q1 else
  begin
    Q2 := -1/x1;  x1 := 2 × x/x1;
    for n := 1 step 1 until m - 1 do
    begin
      Q0 := Q1;  Q1 := Q2;
      Q2 := -n × x1 × Q1 - n × (n-1) × Q0
    end;
    Q[0] := Q2
  end;
  for n := 0 step 1 until nmax do Qapprox[n] := 0;
  epsilon := .5 × 10↑(-d);
  nu := 20 + entier (1.25 × nmax);
L0: r := 0;
  for n := nu step - 1 until 1 do
  begin
    r := (n+m)/((2×n+1)×x-(n-m+1)×r);
    if n ≤ nmax then Rr[n-1] := r
  end;
  for n := 0 step 1 until nmax-1 do Q[n+1] := Rr[n] × Q[n];
  for n := 0 step 1 until nmax do
  if abs(Q[n]-Qapprox[n]) > epsilon × abs(Q[n]) then
  begin
```

* Work performed in part under the auspices of the U.S. Atomic Energy Commission.

```
    for p := 0 step 1 until nmax do Qapprox[p] := Q[p];
    nu := nu + 10;  go to L0
  end
end integer Legendre 2;

procedure integer Legendre 3(x, n, mmax, d, Q);
  value x, n, mmax, d;  integer n, mmax, d;  real x;  array Q;
comment
```

This procedure generates to d significant digits, and stores in the array Q, the Legendre functions of the second kind, $Q_n{}^m(x)$, for $m = 0(1)mmax$, assuming $n \geq 0$ an integer, and $x > 1$. The procedure *integer Legendre* 2 is used to obtain initial values $Q_n{}^0$, $Q_n{}^1$, and subsequent values are obtained from the recursion (1) of the preceding comment;

```
begin integer m;  real x1;  array Q1[0:n];
  if n < 0 ∨ mmax < 0 then go to alarm;
  integer Legendre 2(x, 0, n, d, Q1);  Q[0] := Q1[n];
  x1 := 2 × x/sqrt(x↑2−1);
  if mmax > 0 then
  begin
    integer Legendre 2(x, 1, n, d, Q1);  Q[1] := Q1[n]
  end;
  for m := 1 step 1 until mmax−1 do
    Q[m+1] := −m × x1 × Q[m] − (m+n) × (m−n−1) × Q[m−1]
end integer Legendre 3;
procedure Legendre 1(x, alpha, nmax, d, P1);
  value x, alpha, nmax, d;  integer nmax, d;
  real x, alpha;  array P1;
comment
```

This procedure evaluates to d significant digits the Legendre functions

$$P_\alpha{}^n(x) = \frac{\Gamma(\alpha+n+1)}{\pi\Gamma(\alpha+1)} \int_0^\pi [x + (x^2-1)^{\frac{1}{2}} \cos t]^\alpha \cos nt \, dt$$

for $n = 0(1)nmax$, where $x > 1$ and α is real. The results are stored in the array $P1$. It is assumed that a nonlocal procedure *gamma* be available which evaluates $\Gamma(z)$ for $0 < z \leq 2$. (See [3].) The procedure first generates the quantities $f_n = P_\alpha{}^n(x)/\Gamma(\alpha+n+1)$ from the recurrence relation

$$f_{n+1} + \frac{2nx}{(n+\alpha+1)\,(x^2-1)^{\frac{1}{2}}} f_n + \frac{n-\alpha-1}{n+\alpha+1} f_{n-1} = 0,$$

and the identity

$$f_0 + 2\sum_{n=1}^{\infty} f_n = \frac{[x + (x^2-1)^{\frac{1}{2}}]^\alpha}{\Gamma(\alpha+1)},$$

applying a variant of the backward recurrence algorithm of J. C. P. Miller. (See [2] for more details.) Then $P_\alpha{}^n(x) = \Gamma(\alpha+n+1)f_n$ is obtained recursively. If $\alpha < -\frac{1}{2}$, we let $a = -\alpha-1$ and compute $P_\alpha{}^n(x) = P_a{}^n(x)$. The substitution is made to avoid loss of accuracy when x is large. The rate of convergence of this procedure decreases as x increases. A general idea of the speed of convergence may be obtained from the graphs in [2, §6]. If x is very close to 1, the same changes as mentioned in the first procedure are recommended;

```
begin integer n, nu, m;  real a, epsilon, x1, sum, c, r, s;
  array Papprox, Rr[0:nmax];
  if x < 1 ∨ nmax < 0 ∨ entier(alpha) − alpha = 0 then
  go to alarm;  if x = 1 then
  begin
    P1[0] := 1;  for n := 1 step 1 until nmax do P1[n] := 0;
    go to L1
  end;
  a := if alpha < −.5 then − alpha − 1 else alpha;
  for n := 0 step 1 until nmax do Papprox[n] := 0;
  epsilon := .5 × 10↑(−d);
  if a ≤ 1 then c := gamma(1+a) else
  begin
    m := entier(a) − 1;  c := gamma(a−m);
    for n := 0 step 1 until m do c := (a−n) × c
  end;
  x1 := sqrt (x↑2−1);  sum := (x+x1)↑a/c;  x1 := 2 × x/x1;
 nu := 20 + entier ((37.26+.1283×(a+38.26)×x)×nmax/
    (37.26+.1283×(a+1)×x));
L0:  r := s := 0;
  for n := nu step − 1 until 1 do
  begin
    r := (a+1−n)/(n×x1+(n+a+1)×r);  s := r × (2+s);
    if n ≤ nmax then Rr[n−1] := r
 end;
  P1[0] := sum/(1+s);
  for n := 0 step 1 until nmax − 1 do
    P1[n+1] := Rr[n] × P1[n];
  for n := 0 step 1 until nmax do
    if abs (P1[n]−Papprox[n]) > epsilon × abs (P1[n]) then
    begin
      for m := 0 step 1 until nmax do Papprox [m] := P1[m];
      nu := nu + 10; go to L0
    end;
  P1[0] := c × P1[0];
  for n := 1 step 1 until nmax do
  begin
    c := (a+n) × c;  P1[n] := c × P1[n]
  end;
L1:  end Legendre 1;

procedure Legendre 2(x, a, m, nmax, d, P2);
  value x, a, m, nmax, d;  integer m, nmax, d;  real x, a;
  array P2;
comment
```

This procedure evaluates to d significant digits the Legendre functions $P^m_{a+n}(x)$ for fixed $x > 1$, a, $m \geq 0$, and for $n = 0(1)nmax$. The results are stored in the array $P2$. They are obtained recursively from

$$P^m_{a+n+1}(x) = \frac{2n+2a+1}{n+a-m+1} xP^m_{a+n}(x) - \frac{n+a+m}{n+a-m+1} P^m_{a+n-1}(x),$$

the initial values being calculated with the help of the procedure *Legendre* 1;

```
begin integer n;  array P1[0: m];
  if m < 0 then go to alarm;
  Legendre 1(x, a, m, d, P1);  P2[0] := P1[m];
  if nmax > 0 then
  begin
  Legendre 1(x, a+1, m, d, P1);  P2[1] := P1[m]
  end;
  for n := 1 step 1 until nmax−1 do
    P2[n+1] := ((2×n+2×a+1)×x×P2[n]
      −(n+a+m)×P2[n−1])/(n+a−m+1)
end Legendre 2;

procedure conical (x, tau, nmax, d, P);
  value x, tau, nmax, d;  integer nmax, d;  real x, tau;  array P;
comment
```

This is an adaption of the procedure *Legendre* 1 to the case $\alpha = -\frac{1}{2} + i\tau$, where τ is real. The procedure thus generates Mehler's conical functions $P^n_{-\frac{1}{2}+i\tau}(x)$ to d significant digits for $n = 0(1)nmax$ and $x > 1$. The results are stored in the array P. To avoid excessively large and excessively small numbers, we let $f_n = P^n_{-\frac{1}{2}+i\tau}(x)/n!$ and first compute f_n from the recurrence relation

$$f_{n+1} + \frac{2nx}{(n+1)\,(x^2-1)^{\frac{1}{2}}} f_n + \frac{(n-\frac{1}{2})^2+\tau^2}{n(n+1)} f_{n-1} = 0,$$

and the identity

$$f_0 + \sum_{n=1}^{\infty} \lambda_n f_n = [x+(x^2-1)^{\frac{1}{2}}]^{-\frac{1}{2}} \cos\,(\tau \ln\,[x+(x^2-1)^{\frac{1}{2}}]),$$

where

$$\lambda_n = n!\left[\frac{\Gamma(\frac{1}{2}+i\tau)}{\Gamma(\frac{1}{2}+i\tau+n)} + \frac{\Gamma(\frac{1}{2}-i\tau)}{\Gamma(\frac{1}{2}-i\tau+n)}\right]$$

The λ's are obtained recursively by

$$\lambda_1 = \frac{1}{\frac{1}{4} + \tau^2}, \qquad \lambda_2 = \frac{3 - 4\tau^2}{(\frac{1}{4} + \tau^2)\,(\frac{9}{4} + \tau^2)},$$

$$\lambda_{n+1} = \frac{1 + \frac{1}{n}}{\left(1 + \frac{1}{2n}\right)^2 + \left(\frac{\tau}{n}\right)^2}\,(2\lambda_n - \lambda_{n-1}) \qquad (n = 2, 3, \cdots).$$

The procedure converges rather slowly if x and τ are both large (see the graphs in §6 of [2]). If the accuracy requirement as specified by d is too stringent the procedure may not converge at all due to the accumulation of rounding errors;

```
begin integer n, nu, m;  real epsilon, t, x1, x2, sum, lambda 1,
  lambda 2, lambda, r, s;  array Papprox, Rr[0:nmax];
  if x < 1 ∨ nmax < 0 then go to alarm;
  if x = 1 then
  begin
    P[0] := 1;  for n := 1 step 1 until nmax do P[n] := 0;
  go to L3
  end;
  t := tau↑2;
  for n := 0 step 1 until nmax do Papprox[n] := 0;
  epsilon := .5 × 10↑(−d);
  x1 := sqrt(x↑2−1);  x2 := x + x1;
  sum := cos(tau×ln(x2))/sqrt(x2);  x1 := 2 × x/x1;
  nu := 30 + entier ((1+(.140+.0246×tau) × (x−1))×nmax);
L0:  n := 2;
  lambda 1 := 1/(.25+t);
  lambda 2 := (3-4×t)/((.25+t)×(2.25+t));
L1:  lambda := (1+1/n) × (2×lambda 2−lambda 1)/
  ((1+.5/n)↑2 + (tau/n)↑2);
  if n < nu then
  begin
  lambda 1 := lambda 2,  lambda 2 := lambda;
    n := n + 1;  go to L1
  end;
  r := s := 0;
L2:  r := −((1−.5/n)↑2+(tau/n)↑2)/(x1+(1+1/n)×r);
  s := r × (lambda 2+s);
  if n ≤ nmax then Rr[n−1] := r;
  lambda 1 := lambda 2;
  lambda 2 := 2 × lambda 2 − ((1+.5/n)↑2+(tau/n)↑2)
    × lambda/(1+1/n);
  lambda := lambda 1;
  n := n − 1;  if n ≥ 1 then go to L2;
  P[0] := sum/(1+s);
  for n := 0 step 1 until nmax − 1 do P[n+1] := Rr[n] × P[n];
  for n := 0 step 1 until nmax do
    if abs (P[n]−Papprox[n]) > epsilon × abs(P[n]) then
    begin
      for m := 0 step 1 until nmax do Papprox[m] := P[m];
      nu := nu + 60;  comment  To avoid an infinite loop in
        case of divergence the user should provide for an upper
        bound on nu, say 1000, and exit from the procedure when
        nu exceeds this bound, printing an appropriate error
        message;
      go to L0
    end;
  t := 1;
  for n := 1 step 1 until nmax do
  begin
    t := n × t;  P[n] := t × P[n]
  end;
L3:  end conical;
procedure toroidal (x, m, nmax, d, Q);
  value x, m, nmax, d;  integer m, nmax, d;  real x;  array Q;
comment  This procedure generates to d significant digits the
  toroidal functions of the second kind, Q^m_{−½+n}(x), for n = 0(1)
  nmax, where x > 1, and m is an integer, positive, negative or
  zero. The method of computation is based on the recurrence
  relation
```

$$(n-m+\tfrac{1}{2})Q^m_{-\frac{1}{2}+n+1}(x) - 2nxQ^m_{-\frac{1}{2}+n}(x) + (n+m-\tfrac{1}{2})Q^m_{-\frac{1}{2}+n-1}(x) = 0,$$

and the identity

$$Q^m_{-\frac{1}{2}}(x) + 2\sum_{n=1}^{\infty} Q^m_{-\frac{1}{2}+n}(x) = (-1)^m \sqrt{\frac{\pi}{2}}\,\Gamma(m + \tfrac{1}{2})(x - 1)^{-\frac{1}{2}}\left(\frac{x+1}{x-1}\right)^{m/2},$$

```
  to which a variant of J. C. P. Miller's backward recurrence
  algorithm is applied. (See [2] for more details.) The convergence
  of this procedure is slow for x near 1, and improves rapidly as x
  increases;
begin integer n, nu, p;  real epsilon, x1, c, sum, r, s;
  array Qapprox, Rr[0:nmax];
  if x ≤ 1 ∨ nmax < 0 then go to alarm;
  for n := 0 step 1 until nmax do Qapprox[n] := 0;
  epsilon := .5 × 10↑(−d);
  c := 2.2214414691;
  if m ≥ 0 then
    for n := 0 step 1 until m−1 do c := −(n+.5) × c
  else
    for n := 0 step −1 until m+1 do c := −c/(n−.5);
  sum := c × ((x+1)/(x−1))↑(m/2)/sqrt(x−1);  x1 := 2 × x;
  nu := 20 + entier ((1.15+(.0146+.00122×m)/(x−1))×nmax);
L0:  r := s := 0;
  for n := nu step −1 until 1 do
  begin
    r := (n+m−.5)/(n×x1−(n−m+.5)×r);  s := r × (2+s);
    if n ≤ nmax then Rr[n−1] := r
  end;
  Q[0] := sum/(1+s);
  for n := 0 step 1 until nmax − 1 do Q[n+1] := Rr[n] × Q[n];
  for n := 0 step 1 until nmax do
    if abs(Q[n]−Qapprox[n]) > epsilon × abs(Q[n]) then
    begin
      for p := 0 step 1 until nmax do Qapprox[p] := Q[p];
      nu := nu + 10;  go to L0
    end
end toroidal;
```

comment All procedures were tested on the CDC 3600 computer. Some of the tests that were run are described below;

comment The procedures *integer Legendre* 1–3 were driven to print test values to 6 significant digits of $P_n{}^m(x), Q_m{}^n(x), Q_n{}^m(x)$, $m = 0(1)10$, for $x = 1.5, 3, 10$, and $n = 0(1)5$. As far as possible, the results were compared with values tabulated in [5], and found to be in complete agreement. Similarly, test values of $P^m_{-\frac{1}{2}+n}(x)$, $m = 0(1)4$, were obtained from the procedure *Legendre* 1, for $x = 1.5, 3, 10$, and $n = 0(1)5$. All agreed with values tabulated in [5]. More extensive tests could be run by having the procedure "verify" the addition theorem

$$P_\alpha(xy - \sqrt{(x^2-1)}\sqrt{(y^2-1)}) = P_\alpha(x)P_\alpha(y) + 2\sum_{m=1}^{\infty} (-1)^m \frac{\Gamma(\alpha - m + 1)}{\Gamma(\alpha + m + 1)} P_\alpha{}^m(x)P_\alpha{}^m(y), \quad x > 1, y > 1;$$

comment The procedure *conical* (with $d=6$) was run to produce test values of $P^m_{-\frac{1}{2}+i\tau}(x)$, $m = 0, 1$, for $x = 1.5, 5, 10, 20$, and $\tau = 0(10)30$. The results agreed to 6 significant digits with those in [10], [11];

comment The procedure *toroidal* was driven to generate test values to 6 significant digits of $Q^m_{-\frac{1}{2}+n}(x)$, $Q^{-m}_{-\frac{1}{2}+n}(x)$, $n = 0(1)5$, for $x = 1.5, 3, 10$, and $m = 0(1)4$. All values of $Q^m_{-\frac{1}{2}+n}(x)$ were checked against those in [5]. There were no discrepancies. The values of $Q^{-m}_{-\frac{1}{2}+n}(x)$ were compared with those of $[\Gamma(n-m+\frac{1}{2})/\Gamma(n+m+\frac{1}{2})]Q^m_{-\frac{1}{2}+n}(x)$. The largest relative error observed was $1.5_{10}-9$, occurring at $m = 4$, $n = 5$, $x = 1.5$;

comment Integrals of the form

$$f_n(k^2, \alpha) = (-1)^n \int_0^{\pi/2} [1 - k^2 \sin^2 \psi]^\alpha \cos 2n\psi \, d\psi, \qquad 0 < k < 1,$$

are repeatedly encountered in applied mathematics (see, e.g., [6]-[9], where $\alpha = -\frac{3}{2}$, or $\alpha = -\frac{5}{2}$). It is readily seen that

$$f_n(k^2, \alpha) = (-1)^n \frac{\pi}{2} \frac{\Gamma(\alpha + 1)}{\Gamma(\alpha+n+1)} (1 - k^2)^{\alpha/2} P_\alpha{}^n \left(\frac{2 - k^2}{2\sqrt{(1 - k^2)}}\right).$$

The program that follows generates $(1-k^2)f_n(k^2, \alpha)$, $n = 0(1)10$, for $\alpha = -\frac{3}{2}, -\frac{5}{2}$, and $k^2 = .1, .5, .9$, calling for an accuracy of 6 significant digits. Selected results are shown below.

α	k^2	n	$(1 - k^2) f_n(k^2, \alpha)$
−1.5	.1	0	1.5307576371
		5	$5.2456440472_{10}-8$
		10	$9.0801648667_{10}-16$
	.5	1	$3.4378228849_{10}-1$
		4	$2.8295844423_{10}-3$
		7	$1.8215954880_{10}-5$
	.9	2	$4.8615561237_{10}-1$
		6	$5.2878408708_{10}-2$
		9	$8.8107743954_{10}-3$
−2.5	.1	0	1.6169191877
		5	$2.3969022984_{10}-7$
		10	$7.3394117106_{10}-15$
	.5	1	$8.4721308463_{10}-1$
		4	$1.4940149605_{10}-2$
		7	$1.4764302684_{10}-4$
	.9	2	4.9389962376
		6	$9.7073200383_{10}-1$
		9	$2.1695170317_{10}-1$

Those for $\alpha = -\frac{3}{2}$ were compared with values tabulated in [6]. There was agreement in all four decimal places given;

```
begin integer n;  real alpha, k2, c;  array P1[0:10];
  for alpha := -1.5, -2.5 do
  for k2 := .1, .5, .9 do
    begin
      c := 1.570796327 × (1-k2)↑(1+alpha/2);
      Legendre 1 (.5×(2-k2)/sqrt(1-k2), alpha, 10, 6, P1);
      for n := 0 step 1 until 10 do
      begin
        P1[n] := c × P1[n];  c := -c/(n+alpha+1);
        outreal (1, P1[n])
      end
    end;
  go to skip;
alarm:  outstring (1, 'parameters not in range');
skip: end;
```

comment The integrals

$$\Omega_j(k) = \int_0^\pi (1 - k^2 \cos \phi)^{-\frac{1}{2}-j} \, d\phi, \qquad 0 \leqq k < 1, j = 0, 1, 2, \cdots$$

arose in recent radiation field studies ([1]). One has

$$\Omega_j(k) = \pi(1-k^4)^{-(\frac{1}{2}+j)/2} P_{-\frac{1}{2}+j}((1-k^4)^{-\frac{1}{2}}).$$

The program below calculates $\Omega_j(k)$ to 8 significant digits for $k^2 = .2(.2).8$, $j = 0(1)9$. The results agree to 8 figures with the values tabulated in [1];

```
begin integer j;  real k2, x, x1;  array P2, omega [0:9];
  for k2 := .2 step .2 until .9 do
  begin
    x := 1/sqrt(1-k2↑2);
    Legendre 2(x, -.5, 0, 9, 8, P2);
    x1 := 3.1415926536 × sqrt(x);
    omega [0] := x1 × P2[0];
    for j := 1 step 1 until 9 do
    begin
      x1 := x × x1; omega[j] := x1 × P2[j]
    end;
    for j := 0 step 1 until 9 do outreal (1, omega[j])
  end;
  go to skip;
alarm: outstring (1, 'parameters not in range');
skip: end
end
```

References:

1. Epstein, L. F., and Hubbell, J. H. Evaluation of a generalized elliptic-type integral. *J. Research NBS 67B* (1963), 1-17.
2. Gautschi, W. Computational aspects of three-term recurrence relations. Unpublished.
3. ——. Algorithm 221—Gamma function. *Comm. ACM 7* (Mar. 1964), 143.
4. Herndon, J. R. Algorithm 62—A set of associate Legendre polynomials of the second kind. *Comm. ACM 4* (July 1961), 320-321; Remark on Algorithm 62. *Comm. ACM 4* (Dec. 1961), 544.
5. *NBS Tables of Associated Legendre Functions.* Columbia University Press, New York, 1945.
6. Riegels, F. Formeln und Tabellen für ein in der räumlichen Potentialtheorie auftretendes elliptisches Integral. *Archiv der Mathematik 2* (1949/50), 117-125.
7. Siekmann, J. Concerning an integral occurring in airfoil theory. *SIAM Review 3* (1961), 243-246.
8. ——. Analysis of ring aerofoils of elliptic cross section, Part I: General theory. *J. SIAM 11* (1963), 941-963.
9. ——. Note on a Riegels-type integral. *Z. Angew. Math. Phys. 15* (1964), 79-83.
10. Žurina, M. I., and Karmazina, L. N. Tablicy funkciĭ Ležandra $P_{-\frac{1}{2}+i\tau}(x)$, Vol. I. Akad. Nauk SSSR, Moscow, 1962.
11. ——, and ——. Tablicy funkciĭ Ležandra $P^1_{-\frac{1}{2}+i\tau}(x)$. Akad. Nauk SSSR, Moscow, 1963.

ACM Transactions on Mathematical Software, Vol. 3, No. 2, June 1977, Pages 204-205.

REMARK ON ALGORITHM 259

Legendre Functions for Arguments Larger than One [S16]
[W. Gautschi, *Comm. ACM 8*, 8 (Aug. 1965), 488–492]

J.K.M. Jansen [Recd 24 May 1976 and 12 August 1976]
Technological University, Eindhoven, The Netherlands

The purpose of the changes presented here is to simplify the procedures developed by Gautschi, in particular to remove the necessity of calling the gamma-function routine in procedure *Legendre* 1.

(i) **procedure** *integer Legendre* 1

(1) Replace $F_n = P_a^n(x)/(n+a)!$
by $F_n = P_a^n(x) \times a!/(n+a)!$

(2) Replace $F_0 + 2\sum_{n=1}^{a} F_n = [x + (x^2 - 1)^{1/2}]^a/a!$
by $F_0 + 2\sum_{n=1}^{a} F_n = [x + (x^2 - 1)^{1/2}]^a$

(3) Replace **real** *x*1, *c*, *sum*, *r*, *s*;
by **real** *x*1, *sum*, *r*, *s*;

(4) Omit the statements
c := 1; **for** *n* := 2 **step** 1 **until** a **do** *c* := *n*×*c*;

(5) Replace *sum* := (*x* + *x*1) ↑ *a*/*c*;
by *sum* := (*x* + *x*1) ↑ *a*;

(6) Replace *P*[0] := *c*×*sum*/(1 + *s*);
by *P*[0] := *sum*/(1 + *s*);

(ii) **procedure** *Legendre* 1

(1) Omit the sentence of the **comment**
It is assumed that a nonlocal procedure *gamma* be available which evaluates $\Gamma(z)$ for $0 < z \leq 2$. (See [3].)

(2) Replace $f_n = P_\alpha^n(x)/\Gamma(\alpha + n + 1)$
by $f_n = P_\alpha^n(x) \times \Gamma(\alpha + 1)/\Gamma(\alpha + n + 1)$

(3) Replace $f_0 + 2\sum_{n=1}^{\infty} f_n = [x + (x^2 - 1)^{1/2}]^\alpha/\Gamma(\alpha + 1)$
by $f_0 + 2\sum_{n=1}^{\infty} f_n = [x + (x^2 - 1)^{1/2}]^\alpha$

(4) Replace $P_\alpha^n(x) = \Gamma(\alpha + n + 1)f_n$
by $P_\alpha^n(x) = [\Gamma(\alpha + n + 1)/\Gamma(\alpha + 1)]f_n$

(5) Omit the statements

```
if a ≤ 1 then c := gamma(1 + a) else
begin
  m := entier(a) − 1; c := gamma(a − m);
  for n := 0 step 1 until m do c := (a − n)×c
end;
```

(6) Replace *sum* := (*x* + *x*1) ↑ *a*/*c*;
by *sum* := (*x* + *x*1) ↑ *a*;

(7) Replace *P*1[0] := *c*×P1[0];
by *c* := 1;

(8) During computations it sometimes happens that *entier*(*alpha*) − *alpha* = 0 and consequently the process stops. We remark that if *entier*(*alpha*) − *alpha* = 0 this algorithm accomplishes the same as the procedure *integer Legendre* 1, although in an inefficient manner. To continue the computations we propose to replace
if *x* < 1 ∨ *nmax* <0 ∨ *entier*(*alpha*) − *alpha* = 0 **then**
by
if *x* < 1 ∨ *nmax* < 0 **then**

The same tests as described by Gautschi were run on the Burroughs B6700 and Philips P9200 digital computers of the Computer Center of the Technological University at Eindhoven and were found to be in complete agreement.

ALGORITHM 260
6-J SYMBOLS [Z]

J. H. Gunn (Recd. 13 Nov. 1964)
Nordisk Institut for Teoretisk Atomfysik, Copenhagen, Denmark

```
real procedure SJS (J1, J2, J3, L1, L2, L3, factorial);
  value J1, J2, J3, L1, L2, L3;
  integer J1, J2, J3, L1, L2, L3;
  array factorial;
```

comment *SJS* calculates the 6-*j* symbols defined by the following formula

$$\begin{Bmatrix} j1 & j2 & j3 \\ l1 & l2 & l3 \end{Bmatrix} = \Delta(j1, j2, j3)\Delta(j1, l2, l3)\Delta(l1, j2, l3)\Delta(l1, l2, j3) \times \textstyle\sum_z (-1)^z(z+1)!/((z-j1-j2-j3)!(z-j1-l2-l3)! (z-l1-j2-l3)!(z-l1-l2-j3)!(j1+j2+l1+l2-z)! (j2+j3+l2+l3-z)!(j3+j1+l3+l1-z)!)$$

where

$$\Delta(a, b, c) = \left[\frac{(a+b-c)!\,(a-b+c)!\,(-a+b+c)!}{(a+b+c+1)!}\right]^{\frac{1}{2}}$$

and where $j1 = J1/2, j2 = J2/2, j3 = J3/2, l1 = L1/2, l2 = L2/2$' $l3 = L3/2$. [Reference formula 6.3.7 page 99 of Edmonds, A. R. Angular momentum in quantum mechanics. In *Investigations in Physics*, *4*, Princeton U. Press, 1957]. The parameters of the procedure *J*1, *J*2, *J*3, *L*1, *L*2, *L*3 are interpreted as being twice their physical value, so that actual parameters may be inserted as integers. Thus to calculate the 6-*j* symbol

$$\begin{Bmatrix} 2 & 2 & 0 \\ 2 & 2 & 0 \end{Bmatrix}$$

the call would be *SJS* (4, 4, 0, 4, 4, 0, *factorial*). The procedure checks that the triangle conditions for the existence of a coefficient are satisfied and that $j1 + j2 + j3$, $j1 + l2 + l3$, $l1 + j2 + l3$ and $l1 + l2 + j3$ are integral. If the conditions are not satisfied the value of the procedure is zero. The parameter *factorial* is an array containing the factorials from 0 up to at least 1 + largest of $j1 + j2 + j3, j1 + l2 + l3, l1 + j2 + l3$ and $l1 + l2 + j3$. Since in actual calculations the procedure *SJS* will be called many times it is more economical to have the factorials in a global arrav rather than compute them on every entry to the procedure. The notation is consistent with that used in the procedure for calculating Vector-coupling coefficients. See Algorithm 252, Vector Coupling or Clebsch-Gordan Coefficients [*Comm. ACM 8* (Apr. 1965), 217];

```
begin integer w, wmin, wmax;
real omega;
real procedure delta (a, b, c);
  value a, b, c;
  integer a, b, c;
begin delta := sqrt (factorial [(a+b−c)÷2]
  × factorial [(a−b+c)÷2]
  × factorial [(−a+b+c)÷2]/factorial [(a+b+c+2)÷2])
end delta;
if J1 + J2 < J3 ∨ abs(J1 − J2) > J3 ∨ J1 + J2 + J3 ≠
  2 × ((J1+J2+J3)÷2)
∨ J1 + L2 < L3 ∨ abs(J1−L2) > L3 ∨ J1 + L2 + L3 ≠ 2 ×
  ((J1+L2+L3)÷2)
∨ L1 + J2 < L3 ∨ abs(L1−J2) > L3 ∨ L1 + J2 + L3 ≠ 2 ×
  ((L1+J2+L3)÷2)
∨ L1 + L2 < J3 ∨ abs(L1−L2) > J3 ∨ L1 + L2 + J3 ≠ 2 ×
  ((L1+L2+J3)÷2)
then SJS := 0 else
begin
  omega := 0;
  wmin := J1 + J2 + J3;
  if wmin < J1 + L2 + L3 then wmin := J1 + L2 + L3;
  if wmin < L1 + J2 + L3 then wmin := L1 + J2 + L3;
  if wmin < L1 + L2 + J3 then wmin := L1 + L2 + J3;
  wmax := J1 + J2 + L1 + L2;
  if wmax > J2 + J3 + L2 + L3 then wmax := J2 + J3 +
    L2 + L3;
  if wmax > J3 + J1 + L3 + L1 then wmax := J3 + J1 +
    L3 + L1;
  for w := wmin step 2 until wmax do
  omega := omega + (if w=4×(w÷4) then 1 else −1)
    × factorial [w÷2+1]/(factorial · [(w−J1−J2−J3)÷2]
    × factorial [(w−J1−L2−L3)÷2]
    × factorial [(w−L1−J2−L3)÷2]
    × factorial [(w−L1−L2−J3)÷2]
    × factorial [(J1+J2+L1+L2−w)÷2]
    × factorial [(J2+J3+L2+L3−w)÷2]
    × factorial [(J3+J1+L3+L1−w)÷2]);
  SJS := delta (J1, J2, J3) × delta (J1, L2, L3)
    × delta (L1, J2, L3) × delta (L1, L2, J3) × omega;
end
end SJS
```

ALGORITHM 261
9-J SYMBOLS [Z]

J. H. Gunn (Recd. 13 Nov. 1964)
Nordisk Institut for Teoretisk Atomfysik, Copenhagen, Denmark

```
real procedure NJS(J11, J12, J13, J21, J22, J23, J31, J32, J33,
    factorial);
  value J11, J12, J13, J21, J22, J23, J31, J32, J33;
  integer J11, J12, J13, J21, J22, J23, J31, J32, J33;
  array factorial;
```

comment *NJS* calculates the 9-*j* symbols defined by the following formula

$$\begin{Bmatrix} j11 & j12 & j13 \\ j21 & j22 & j23 \\ j31 & j32 & j33 \end{Bmatrix} = \sum\nolimits_k (-1)^{2k}(2k+1)\begin{Bmatrix} j11 & j21 & j31 \\ j32 & j33 & k \end{Bmatrix} \begin{Bmatrix} j12 & j22 & j32 \\ j21 & k & j23 \end{Bmatrix}\begin{Bmatrix} j13 & j23 & j33 \\ k & j11 & j12 \end{Bmatrix}.$$

where $j11 = J11/2$, $j12 = J12/2$, $j13 = J13/2$, $j21 = J21/2$, $j22 = J22/2$, $j23 = J23/2$, $j31 = J31/2$, $j32 = J32/2$, $j33 = J33/2$ [Reference formula 6.4.3 page 101 of Edmonds, A. R. Angular momentum in quantum mechanics. In *Investigations in Physics*, *4*, Princeton U. Press, 1957]. The parameters of the procedure *J*11, *J*12, *J*13, *J*21, *J*22, *J*23, *J*31, *J*32, *J*33 are interpreted as being twice their physical value, so that actual parameters may be inserted as integers. Thus to calculate the 9-*j* symbol

$$\begin{Bmatrix} 2 & 2 & 0 \\ 2 & 2 & 0 \\ 0 & 0 & 0 \end{Bmatrix}$$

the call would be *NJS* (4, 4, 0, 4, 4, 0, 0, 0, 0, *factorial*). The procedure checks that the triangle conditions for the existence of a coefficient are satisfied and that $j11 + j21 + j31$, $j21 + j22 + j23$, $j31 + j32 + j33$, $j11 + j12 + j13$, $j12 + j22 + j32$, $j13 + j23 + j33$ are integral. If the conditions are not satisfied the value of the procedure is zero. The parameter *factorial* is an array containing the factorials from 0 up to at least 1 + largest of $j11 + j21 + j31$, $j21 + j22 + j23$, $j31 + j32 + j33$, $j11 + j12 + j13$, $j12 + j22 + j32$, $j13 + j23 + j33$. The procedure makes use of the procedure *SJS* [Algorithm 260, 6-*j* symbols, *Comm. ACM 8* (Aug. 1965), 492], for calculating 6-*j* symbols;

```
begin integer k, kmin, kmax;
  real NJ;
  if J11 + J21 < J31 ∨ abs(J11−J21) > J31 ∨ J11 + J21 +
    J31 ≠ 2 × ((J11+J21+J31)÷2)
  ∨ J21 + J22 < J23 ∨ abs(J21−J22) > J23 ∨ J21 + J22 +
    J23 ≠ 2 × ((J21+J22+J23)÷2)
  ∨ J31 + J32 < J33 ∨ abs(J31−J32) > J33 ∨ J31 + J32 +
    J33 ≠ 2 × ((J31+J32+J33)÷2)
  ∨ J11 + J12 < J13 ∨ abs(J11−J12) > J13 ∨ J11 + J12 +
    J13 ≠ 2 × ((J11+J12+J13)÷2)
  ∨ J12 + J22 < J32 ∨ abs(J12−J22) > J32 ∨ J12 + J22 +
    J32 ≠ 2 × ((J12+J22+J32)÷2)
  ∨ J13 + J23 < J33 ∨ abs(J13−J23) > J33 ∨ J13 + J23 +
    J33 ≠ 2 × ((J13+J23+J33)÷2)
  then NJS := 0 else
  begin NJ := 0;
    kmin := abs(J21−J32);
    if kmin < abs(J11−J33) then kmin := abs(J11−J33);
    if kmin < abs(J12−J23) then kmin := abs(J12−J23);
      kmax := J21 + J32;
    if kmax > J11 + J33 then kmax := J11 + J33;
    if kmax > J12 + J23 then kmax := J12 + J23;
      for k := kmin step 2 until kmax do
    NJ := NJ + (if k=2×(k÷2) then 1 else −1) × (k+1) ×
      SJS(J11, J21, J31, J32, J33, k, factorial) ×
      SJS(J12, J22, J32, J21, k, J23, factorial) ×
      SJS(J13, J23, J33, k, J11, J12, factorial);
    NJS := NJ
  end
end NJS
```

ALGORITHM 262
NUMBER OF RESTRICTED PARTITIONS OF N [A1]

J. K. S. McKay (Recd. 7 Dec. 1964 and 9 Mar. 1965)
Computer Unit, University of Edinburgh, Scotland

```
procedure set (p, N);  integer N;  integer array p;
comment  The number of partitions of n with parts less than
  or equal to m is set in p[n, m] for all n, m such that N ≥ n ≥
  m ≥ 0.
    REFERENCES:
  1. GUPTA, H., GWYTHER, C. E., AND MILLER, J. C. P. Tables of
       partitions. In Royal Society Mathematical Tables, vol. 4,
       Cambridge U. Press, 1958.
  2. HARDY, G. H., AND WRIGHT, E. M.  The Theory of Numbers.
       Ch. 19, 4th ed., Clarendon Press, Oxford, 1960;
begin integer m, n;
  p[0, 0] := 1;
  for n := 1 step 1 until N do
  begin p[n, 0] := 0;
    for m := 1 step 1 until n do
      p[n, m] := p[n, m−1] +
        p[n−m, if n−m<m then n−m else m]
  end
end set
```

ALGORITHM 263
PARTITION GENERATOR [A1]

J. K. S. McKay (Recd. 7 Dec. 1964 and 9 Mar. 1965)
Computer Unit, University of Edinburgh, Scotland.

```
procedure generate (p, N, position, ptn, length);
  integer array p, ptn; integer N, length, position;
comment The partitions of N may be mapped in their natural
  order, 1 — 1, onto the consecutive integers from 0 to P(N)−1
  where P(N)(=p[N, N]) is the number of unrestricted partitions
  of N. The array p is set by the procedure set [Algorithm 262,
  Number of Restricted Partitions of N, Comm. ACM 8 (Aug.
  1965), 493]. On entry position contains the integer into which
  the partition is mapped. On exit length contains the number of
  parts and ptn[1: length] contains the parts of the partition in
  descending order.
    REFERENCE:
  1. LITTLEWOOD, D. E. The Theory of Group Characters. Ch. 5,
      2nd ed., Clarendon Press, Oxford, 1958;
begin integer m, n, psn;
  n := N; psn := position; length := 0;
A:  length := length + 1; m := 1;
B:  if p[n, m] < psn then begin m := m + 1; go to B end else
    if p[n, m] > psn then
C:  begin
      ptn[length] := m; psn := psn − p[n, m−1]; n := n − m;
      if n ≠ 0 then go to A; go to D
    end
    else m := m + 1; go to C;
D:  end generate
```

ALGORITHM 263 A
GOMORY 1 [H]

H. Langmaack (Recd. 17 June 1964 and 13 May 1965)
Mathematisches Institut der Technischen Hochschule, München, Germany

When testing Algorithm 153 GOMORY [F. L. Bauer, *Comm. ACM 6* (Feb. 1963), 68] in Algol on the SIEMENS 2002 and TELEFUNKEN TR4 computers and in PROSA (assembler code) on the SIEMENS 2002 computer I found that some corrections were necessary. After discussions with Prof. Dr. Bauer I wish to submit the following remarks on Algorithm 153 GOMORY and on Certification of Algorithm 153 GOMORY [B. Lefkowitz and D. A. D'Esopo, *Comm. ACM 6* (Aug. 1963), 449]. The improved algorithm GOMORY 1 is presented below.

1. The evaluation of the integer number $t[j]$ in the algorithm GOMORY or t in the revised form of the algorithm GOMORY is not correct, since $t[j]$ (or t) must be the largest integer number such that column j of the matrix a is not lexicographically less than column l multiplied by $t[j]$ (or t), provided such a $t[j]$ (or t) exists. A suitable change is incorporated in the algorithm GOMORY 1 given below.

2. The second remark deals with the fact that a theoretically correct Algol program may not necessarily run correctly when translated into a particular machine language and run on that machine. In general real numbers are represented only approximately and the mathematical division indicated by the Algol operator / is transformed into the approximate operation of machine division. There are two possibilities that the algorithm GOMORY might fail:

A. The *lambda* calculated by

$$\mathit{abs}\ (a[r,\ j]/t[j])$$

in the algorithm GOMORY (or by

$$-a[r,\ j]/t$$

in the revised form of the algorithm GOMORY) may be less than the exact theoretical value of *lambda*. This may lead to columns which are lexicographically negative, but this situation is not allowed.

B. The quantities $c[j]$ (or c) calculated by

$$\mathit{entier}\ (a[r,\ j]/\mathit{lambda})$$

may be different from the exact values, a situation which may lead to incorrect matrix transformations.

To avoid these unwanted effects the author suggests remedying the problem in the following way:

a. Since *lambda* is only an intermediate result, it is proposed to keep the numerators and denominators of the candidates for *lambda* separate and to compare them by cross multiplication.

b. It is preferable to compute

$$A/\mathit{lambda}$$

by

$$(A \times \mathit{denominator\ of\ lambda})/\mathit{numerator\ of\ lambda}$$

where A is an integer type expression.

c. In the algorithm GOMORY there are statements of the form

$$C := \mathit{entier}\ (A/B)$$

where C is an integer variable, and A and B are integer type expressions. In order to prevent roundoff errors the result C should be checked to make sure that

$$C \times B \leq A < C \times B + B$$

and corrected if these inequalities are not satisfied.

The corrections, a, b, c, lead to a program which cannot fail unless the products developed should overflow. However, anyone who wishes to use the algorithm may prefer to do some analysis of the particular division his computer performs and seek an alternative which is not as time-consuming. Many machines have a built-in Euclidean division instruction for integer numbers which would be very useful for Gomory's algorithm. Unfortunately Algol translators are not likely to produce this instruction in their object programs since an arithmetical expression A/B is a real type expression by definition.

```
procedure Gomory 1 (m, n) transient: (a) exit: (no solution);
  value m, n;
  integer m, n;
  integer array a;
  label no solution;
comment Gomory 1 algorithm for all-integer programming.
  The objective of this procedure is to determine the integer solu-
  tion x[1], ··· , x[n−1] of a linear programming problem with inte-
  ger coefficients only. In other words: The problem is to find
  integer numbers
      x[1], ··· , x[n−1]
  minimizing the objective function
      a[0, 1] × x[1] + ··· + a[0, n−1] × x[n−1]
  under the constraints
      x[1] ≥ 0, ··· , x[n−1] ≥ 0
  and
      a[i, 1] × x[1] + ··· + a[i, n−1] × x[n−1] ≤ a[i, n]
  for i = 1, ··· , m−n+1   (2 ≤ n ≤ m).
    The tableau matrix a used by the procedure consists of m+1
  rows and n columns. The components are a[i, j] for i = 0, 1,
  ··· , m, j = 1, ··· , n.
    The input values for the components are given partly by the
  problem itself (see above). The remaining components must
  have been previously assigned in the following manner:
      a[0, n] := 0
  and
      a[i, j] := if i = j + m − n + 1 then −1 else 0
  for i = m−n+2, ··· , m, j = 1, ··· , n. The tableau columns,
  with the exception of the last column, have to be lexicographically
  positive.
    The algorithm is finished if all entries in the last column,
  except the topmost entry, are non-negative. Then −a[0, n] is
  the value of the objective function. The optimal solution
  x[1], ··· , x[n−1] is given by the n−1 components a[m−n+2, n],
  ··· , a[m, n] of the last column of a.
    The exit no solution is used if a row is found which has a
  negative entry in the last column, but otherwise only non-
  negative entries;
```

```
begin integer i, k, j, l, r, c, t, s, lambda num, lambda denom;
  integer procedure Euclid (u, v);
    value u, v;
    integer u, v;
  begin integer w;
    w := entier (u/v);
L8:  if w × v > u then
      begin w := w−1;  go to L8 end;
L9:  if (w+1) × v ≤ u then
      begin w := w+1;  go to L9 end;
      Euclid := w
  end Euclid;
L1:  for i := 1 step 1 until m do if a [i, n] < 0 then
      begin r := i;  go to L2 end;
  go to end;
L2:  for k := 1 step 1 until n−1 do if a[r, k] < 0 then go to L4;
  go to no solution;
L4:  l := k;
  for j := k+1 step 1 until n−1 do if a[r, j] < 0 then
     begin i := 0;
L3:  if a[i, j] < a[i, l] then l := j else
      if a[i, j] = a[i, l] then
      begin i := i+1;  go to L3 end
     end;
  s := 0;
L5: if a[s, l] = 0 then
   begin s := s+1;  go to L5 end;
   lambda num := −a[r, l];
   lambda denom := 1;
   for j := 1 step 1 until l−1, l+1 step 1 until n−1 do
     if a[r, j] < 0 then
      begin
       for i := 0 step 1 until s−1 do if a[i, j] ≠ 0 then go to L7;
       t := Euclid (a[s, j], a[s, l]);
       if (t×a[s, l] = a[s, j]) ∧ (t>1) then
       begin i := s;
L6:     i := i+1;
        if t×a[i, l] = a[i, j] then go to L6 else
        if t×a[i, l] > a[i, j] then t := t−1
       end;
       if −a[r, j] × lambda denom > t × lambda num then
       begin lambda num := −a[r, j];  lambda denom := t.end;
L7:  end;
  for j := 1 step 1 until l−1, l+1 step 1 until n do
    begin c := Euclid (a[r, j] × lambda denom, lambda num);
     if c ≠ 0 then
     for i := 0 step 1 until m do
       a[i, j] := a[i, j] + c × a[i, l]
    end;
  go to L1;
end:
end
```

CERTIFICATION OF ALGORITHM 263A [H] GOMORY 1 [H. Langmaack, *Comm. ACM 8* (Oct. 1965), 601–602]

L. G. Proll (Recd. 15 Sept. 1969)
Department of Mathematics, University of Southampton, U.K.

KEY WORDS AND PHRASES: linear programming, integer variables, dual method
CR CATEGORIES: 5.41

Algorithm 263A was coded in Algol for an ICL 1907 computer and ran successfully without alteration. Execution times and pivot counts for a sample of 12 published examples are given in Table I.

Problem 1 is taken from Haley [1, p. 127]. Problems 2, 3, and 4 are Balas [2, ex. 2, 3, 4] in which the variables were not restricted to be 0 or 1. Problems 5–10 are IBM [3, test problems 1–5 and 9]. Problems 11 and 12 are Pierce [4, ex. 1, 2].

Wilson [5] has shown that it is possible to derive potentially stronger cuts than those of Gomory with little extra computation.

TABLE I

Problem	*m*	*n*	*No. of pivots*	*Time (sec.)*
1	13	10	13	1
2	17	11	8	1
3	13	10	35	1
4	18	13		600†
5	14	8	9	1
6	14	8	16	1
7	10	8	16	1
8	30	16	17	2
9	30	16	2569	248
10	65	16		600†
11	41	32	5	2
12	31	27	5	2

† termination not reached

Wilson's cuts can be incorporated into GOMORY 1 by means of the following alterations:

(a) in the declarations at the head of the procedure body, insert **Boolean** *null, nflag*;

(b) in the line commencing *L4*: *l* := *k*; insert the stat ment *null* := **true**;

(c) replace *L7*: **end**; by

```
L7:  end
     else null := false;
     c := Euclid (a[r, n]×lambda denom, lambda num);
     s := −(c+1);  t := −a[r, n];  nflag := true;
     if null then go to L10;
     for j := 1 step 1 until n − 1 do if a[r, j] > 0 then
     begin c := Euclid (a[r, j]×lambda denom, lambda num);
       if s × a[r, j] < t × c then
       begin t := a[r, j];  s := c;  nflag := false end
     end;
L10: if s × lambda num < t × lambda denom then
     begin lambda num := if nflag then 100 × t − 1 else t;
       lambda denom := if nflag then 100 × s else s
     end;
```

(d) replace the line commencing

```
    begin c := Euclid(a[r, j]×lambda denom, lambda num);
  by
    begin c := if lambda denom ≠ 0 then Euclid(a[r, j]×
      lambda denom, lambda num)
      else if a[r, j] < 0 then −1 else 0;
```

TABLE II

Problem	*No. of pivots*	*Time (sec.)*
7	7	1
9	2238	235

With these alterations some reduction in the number of pivots needed to solve problems 7, 9 was observed. New pivot counts and execution times for these problems are given in Table II. Execution times for the problems not listed in Table II were unaltered.

REFERENCES:

1. HALEY, K. B. *Mathematical Programming for Business and Industry*. Macmillan, New York, 1968.
2. BALAS, E. An additive algorithm for solving linear programs with zero-one variables. *Oper. Res. 13* (1965), 517-545.
3. HALDI, J. 25 integer programming test problems. Working Paper No. 43, Grad. Sch. of Bus., Stanford U., Stanford, Calif., 1964.
4. PIERCE, J. F. Application of combinatorial programming to a class of all zero-one integer programming problems. *Man. Sci. 15* (1968), 191-209.
5. WILSON, R. B. Stronger cuts in Gomory's all-integer integer programming algorithm. *Oper. Res. 15* (1967), 155-156.

ALGORITHM 264

MAP OF PARTITIONS INTO INTEGERS [A1]

J. K. S. McKay (Recd. 7 Dec. 1964 and 9 Mar. 1965)
Computer Unit, University of Edinburgh, Scotland

```
integer procedure place(p, n, ptn);  value n;
  integer array p, ptn;  integer n;
comment place is the inverse of the procedure generate [Al-
  gorithm 263, Partition Generator, Comm. ACM 8 (Aug. 1965),
  493]. The array p is set by the procedure set [Algorithm 262,
  Number of Restricted Partitions of N, Comm. ACM 8 (Aug.
  1965), 493]. The procedure produces the integer into which
  the partition of n, stored in descending order of parts in ptn[1]
  onwards, is mapped;
begin integer j, d;
  d := 0;
  if n = 0 then go to B;
  j : = 0;
A: j := j + 1;  d := p[n, ptn[j]−1] + d;  n := n − ptn[j];
  if n ≠ 0 then go to A;
B: place := d
end place
```

ALGORITHM 264 A
INTERPOLATION IN A TABLE [E1]

J. STAFFORD (Recd. 16 Nov. 1964 and 7 June 1965)
Westland Aircraft Ltd., Saunders-Roe Division, East
Cowes, Isle of Wight, England

```
real procedure INPOL(T, X, I, N, OUT, XOUT, EXPOL);
  value X, N;  array T, X;  integer I;  integer array N;
  real XOUT, EXPOL;  Boolean OUT;
comment  Evaluation of a function by polynomial interpola-
  tion in a table of values.
    The values may be specified at arbitrary intervals, at nodes
  of a multidimensional rectangular grid. The interpolation is
  by Neville's process, repeated in each dimension.
    The given values are arranged in a one-dimensional real array
  T, as follows. The first value in the table, T[0], is D, the number
  of independent variables (or dimensions). It will normally be
  integral (although of type real), but if not then its integral part
  is taken. T[1], T[2], ··· , T[D] are the numbers of values of
  X1, X2, ··· , XD, and must be integral. These are followed by
  T[1] values of X1, T[2] values of X2, ··· T[D] values of XD.
  The values of each of these independent variables must all be
  distinct and must be arranged in monotonic order. Finally
  come the T[1] × T[2] × ··· × T[D] values of the dependent
  variable F(X1, X2, ··· , XD), arranged as T[D] sets of T[D−1]
  sets of ··· of T[2] sets of T[1] values of F.
    The table is represented by a one-dimensional array because
  it is not feasible to use a general D-dimensional array.
    The given values of the independent variables are X[I]
  (I=1, 2, ··· , D). N[I] of the tabulated values X[I] are used to
  interpolate in the Ith dimension. INPOL is the required value
  of the function. The actual parameter corresponding to the
  formal parameter EXPOL should be an expression which pro-
  vides the value of INPOL if any of the X[I] is outside the range
  covered by the array T. If this occurs XOUT is the particular
  value of X[I] concerned. The variables I, OUT and XOUT are
  declared as formal parameters of INPOL so that they may be
  used in the actual parameter corresponding to the formal pa-
  rameter EXPOL.
    An example of a call of INPOL is Z := INPOL(A, X, K, N,
  OUT, Y, if K=1 then EXTRAPOLATE (A, 1, N, OUT, Y)
  else if K=2 then LIMTAB (A, 2, OUT, Y) else Y−2). If
  X[1] is outside the range covered by the array A this statement
  will use the extrapolatory procedure EXTRAPOLATE (given
  below) to provide a value for INPOL. If X[2] is out of range
  the procedure LIMTAB (also given below) will be used to
  replace the value of X[2] by its value at the nearer edge of the
  table, before returning to INPOL to continue the interpolation.
  If some other variable (X[3], say) is out of range the value of
  INPOL is taken as X[3] − 2.
    The procedures INPOL, EXTRAPOLATE and LIMTAB
  were tested on an ICT Atlas computer. They were also tested
  on a National-Elliott 803 computer, after being altered to
  conform to the restrictions of the 803 ALGOL compiler. The
  tests were for D = 0, 1, 2 and 3, and included all special cases;
begin integer D, J, K, L, M, Q, XI;

  procedure FORS3(N, P, V, UB);
    value N;  integer N;  procedure P;
    integer array V, UB;
  comment  Nesting of for statements, adapted from procedure
    Fors 1 [Algorithm 137, Comm. ACM 5 (Nov. 1962), 555];
  begin integer J;
    if N = 0 then P else for J := 1 step 1 until UB[N] do
      begin V[N] := J;  FORS3(N−1, P, V, UB) end
  end FORS3;

  real procedure NEV(X, AX, SAX, AY, SAY, N);
    value X, SAX, SAY, N;  real X;  integer SAX, SAY, N;
    array AX, AY;
  comment  One-dimensional interpolation by Neville's process.
    N values of the independent variable are used in the inter-
    polation, namely, N consecutive elements of array AX start-
    ing at subscript SAX. The corresponding values of the de-
    pendent variable are the N consecutive elements of array
    AY starting at subscript SAY. X is the value of the inde-
    pendent variable for which the value of the dependent vari-
    able (namely, NEV) is to be interpolated;
  begin integer I, J, NJ, KI;  array F[0: N−1];
    for J := 0 step 1 until N − 1 do F[J] := AY[SAY + J];
    for J := 1 step 1 until N − 1 do
    begin
      NJ := N − J − 1;
      for I := 0 step 1 until NJ do
      begin
        KI := SAX + I;
        F[I] := (F[I+1]−F[I]) × (X−AX[KI])/
          (AX[KI+J]−AX[KI])+F[I]
    end;
    NEV := F[0]
  end NEV;

D := entier (T[0]);
comment  D = number of dimensions. The special case D = 0
  implies that the tabulated function F is a constant, the value
  of which is T[1]. The same value is taken if D < 0;
if D < 1 then INPOL := T[1] else
begin XI := 1;
  for I := 1 step 1 until D do
  begin
    if N[I] < 2 then N[I] := 2;
    if N[I] > T[I] then N[I] := T[I];
    comment  Adjustment of number of points used for inter-
      polation. Normally N[I] must be at least 2, and if N[I] < 2
      it is set equal to 2. N[I] also may not exceed the number
      of values of the independent variable in the correspond-
      ing dimension (namely, T[I]), and if it does so it is re-
      duced accordingly.
        The combination of these two tests, in this order,
      permits as a special case one-point interpolation in any
      particular dimension (I, say), if T[I] = 1. This implies
      that the dependent variable is independent of X[I]. If
      this is intended then the actual parameter corresponding
      to the formal parameter EXPOL must be a function
      designator which (if called for) replaces the value of
      XOUT by the single value of the Ith variable from the
      array T. (Procedure LIMTAB may be used for this
      purpose.)
        Since array N is called by value none of these adjust-
      ments affects the values of N[I] in the nonlocal array N;
    XI := XI + N[I]
```

```
end I;
begin array F[1: XI−N[1]]; integer array V, XINIT,
  YINC [1: D];

 procedure ONEWAY;
   comment Performs an interpolation in the first dimen-
     sion. If this is the last of a set of N[2] such interpola-
     tions, a further interpolation is performed in the second
     dimension, and so on to as many higher levels as neces-
     sary;
 begin F[V[1]] := NEV (X[1], T, XINIT[1], T, Q, L);
   I := 1;  M := 0;
   for K := 1 step 1 until D − 1 do
   begin Q := Q + YINC[K];
     if V[K] ≠ N[K] then go to CONTINUE else
     begin M := M + N[K];
       F[M+V[K+1]] := NEV(X[K+1], T, XINIT[K+1],
         F, I, N[K]);
       I := I + N[K]
     end
   end;
 CONTINUE:
 end ONEWAY;

 Q := XI := D + 1;  M := 1;
 for I := 1 step 1 until D do
 begin K := XI + T[I] − 1;
   OUT := (X[I] − T[XI]) × (X[I] − T[K]) > 0;
   if OUT then
   begin
     XOUT := X[I];  INPOL := EXPOL;  X[I] := XOUT;
     if T[0] ≤ 0
       then begin K := K + T[0];  T[0] := D end
   end;
    comment If X[I] is outside the range covered by the
      table, the extrapolatory expression EXPOL is evalu-
      ated. It is expected that it will often be or contain one
      or more function designators, together with criteria
      for choosing between them, as in the example above.
        EXPOL may incorporate, e.g., any of the following
      alternatives. In the first and third of these the side
      effects are the important ones, the value assigned to
      EXPOL being merely a dummy to conform with Section
      5.4.4 of the Revised Report on Algol 60 [Comm. ACM 6
      (Jan. 1963), 1–17].
      1. EXPOL may be a function designator which uses
         the interpolatory formula to extrapolate by executing
         the statement OUT := false and returning to INPOL.
         The last N[I] values of X[I] are used in the formula,
         but EXPOL may arrange to use the first N[I] values
         instead (which will usually be preferable if X[I] lies
         beyond the lower limit of the table) by executing the
         statement T[0] := N[I] − T[I] (in which the value of
         the local N[I] is to be used if it differs from that of
         the nonlocal N[I]). The procedure EXTRAPOLATE
         (given below) may be used for this purpose.
      2. EXPOL may use some other formula to extrapolate,
         after which it must return to INPOL without altering
         the value of the Boolean variable OUT. If this is all
         that is required the actual parameter corresponding
         to EXPOL may be an ordinary arithmetic expression
         containing no function designators.
      3. EXPOL may be a function designator which con-
         strains X[I] to lie within range by replacing it by the
         value of the Ith variable at the nearer limit of the
         table (or by some other value). In doing this it must
         operate on the value of XOUT and not directly on
         X[I]. The nonlocal array X will not be affected.
         EXPOL must also execute the statement OUT :=
         false before returning to INPOL. The procedure
         LIMTAB (given below) may be used for this purpose.
      4. EXPOL may do something else and continue the
         program without returning to INPOL (e.g., by a go
         to statement referring to a nonlocal label). This
         should be considered an error exit as the value of
         INPOL will be undefined (see Section 5.4.4 of the
         Revised Report on Algol 60);
   if OUT then go to B;
   comment If OUT = true on exit from INPOL then
     extrapolation has occurred. The converse is not neces-
     sarily true, as it depends on the nature of the actual
     parameter corresponding to the formal parameter
     EXPOL;
   J := XI;
A: L := (J+K) ÷ 2;
   if (X[I]−T[J]) × (X[I]−T[L]) > 0 then J := L else
     K := L;
   if K − J > 1 then go to A;  comment Find X[I] in
     table;
   L := K − N[I] ÷ 2;
   if L ≤ XI then L := XI else
   begin
     K := XI + T[I] − N[I];  if L > K then L := K
   end Adjustment near edge of table;
   Q := Q + T[I] + (L − XI) × M;  XINIT[I] := L;
   XI := XI + T[I];
   YINC[I] := M × (T[I] − (if I=1 then 0 else N[I]));
   M := M × T[I]
 end I;
 V[D] := 1;  L := N[1];
 for I := 1 step 1 until D − 1 do N[I] := N[I+1];
 FORS3(D−1, ONEWAY, V, N);  INPOL := F[M+1]
 end scope of F
 end D ≥ 1;
B:
end INPOL;

real procedure EXTRAPOLATE(T, I, N, OUT, XOUT);
  array T;  integer I;  integer array N;  Boolean OUT;
  real XOUT;
comment This function designator is intended for use in the
  actual parameter corresponding to the formal parameter
  EXPOL in a call of procedure INPOL. The parameters have
  the same significance as in INPOL.
    EXTRAPOLATE arranges for the interpolatory formula to
  be used to extrapolate for the Ith variable, and for the first
  N[I] values of this variable to be used in the formula instead
  of its last N[I] values if it lies beyond the lower limit of the
  table;
begin integer J, K;
  OUT := false; EXTRAPOLATE := 0;
  comment This statement assigns a dummy value to EXTRAP-
    OLATE to conform with Section 5.4.4 of the Revised Report
    on Algol 60;
  J := 1;  for K := 0 step 1 until I − 1 do J := J + T[K];
  if T[I] = 1 then XOUT := T[J] else
    if abs(XOUT − T[J]) < abs(XOUT−T[J+T[I]−1]) then
    begin K := N[I];
      if K < 2 then K := 2;
      if K > T[I] then K := T[I];
      T[0] := K − T[I]
    end
end EXTRAPOLATE;
```

```
real procedure LIMTAB(T, I, OUT, XOUT);
  array T;  integer I;  Boolean OUT;  real XOUT;
comment This function designator is intended for use in the
  actual parameter corresponding to the formal parameter
  EXPOL in a call of procedure INPOL. The parameters have
  the same significance as in INPOL.
    LIMTAB replaces the value of XOUT, which is outside the
  range of the table, by the value of the Ith variable at the nearer
  edge of the table;
begin integer J, K;
  J := 1;  for K := 0 step 1 until I - 1 do J := J + T[K];
  K := J + T[I] - 1;
  LIMTAB := XOUT := if abs(XOUT-T[J]) >
    abs(XOUT-T[K]) then T[K] else T[J];
  comment This statement assigns a dummy value to LIMTAB
    to conform with Section 5.4.4 of the Revised Report on
    Algol 60;
  OUT := false
end LIMTAB
```

ALGORITHM 265
FIND PRECEDENCE FUNCTIONS [L2]

NIKLAUS WIRTH (Recd. 14 Dec. 1964 and 22 Dec. 1964)
Computer Science Dept., Stanford U., Stanford, Calif.

```
procedure Precedence (M, f, g, n, fail);
value n;  integer n;  integer array M, f, g;  label fail;
comment  M is a given n × n matrix of integers designating one
  of the four relations <, =, >, ∘. The identifiers ls, eq, gr des-
  ignate variables declared outside the procedure to which distinct
  integers representing the relations <, =, > have been assigned.
  This procedure then determines integers f[1] . . f[n] and g[1]
  . . g[n] such that for all i, j, f[i] M[i, j] g[j] is true and so that the
  smallest of these integers is +1. ∘ designates the empty relation,
  so that x ∘ y is true for arbitrary x,y. If M is such that no f and g
  exist which satisfy all n² relations, then control is transferred to
  the label parameter fail. This procedure has been used to deter-
  mine the precedence functions of symbols in a given precedence
  grammar (see [FLOYD, R. Syntactic analysis and operator
  precedence. J.ACM 10 (1963), 316–333]);
begin integer i, j, k, k1, fmin, gmin;
  procedure fixrow (i, l, x);  value i, l, x;  integer i, l, x;
  begin integer j; f[i] := g[l] :+ x;
    if k = k1 then
    begin if M[i, k] = ls ∧ f[i] ≥ g[k] then go to fail else
      if M[i, k] = eq ∧ f[i]  ≠ g[k] then go to fail
    end;
    for j := k1 step −1 until 1 do
    if M[i, j] = ls ∧ f[i] ≥ g[j] then fixcol (i, j, 1) else
    if M[i, j] = eq ∧ f[i] ≠ g[j] then fixcol (i, j, 0)
  end fixrow;
  procedure fixcol (l, j, x); value l, j, x; integer l, j, x;
  begin integer i; g[j] := f[l] + x;
    if k ≠ k1 then
    begin if M[k,j] = gr ∧ f[k] ≤ g[j] then go to fail else
      if M[k,j] = eq ∧ f[k] ≠ g[j] then go to fail
    end;
    for i := k step −1 until 1 do
    if M[i, j] = gr ∧ f[i] ≤ g[j] then fixrow (i, j, 1) else
    if M[i, j] = eq ∧ f[i] ≠ g[j] then fixrow (i, j, 0)
  end fixcol;
  k1 := 0;
  for k := 1 step 1 until n do
  begin fmin := 1;
    for j := 1 step 1 until k1 do
      if M[k, j] = gr ∧ fmin ≤ g[j] then fmin := g[j]+1 else
      if M[k, j] = eq ∧ fmin < g[j] then fmin := g[j];
    f[k] := fmin;
    for j := k1 step −1 until 1 do
      if M[k, j] = ls ∧ fmin ≥ g[j] then fixcol (k, j, 1) else
      if M[k, j] = eq ∧ fmin > g[j] then fixcol (k, j, 0);
    k1 := k1+1; gmin := 1;
    for i := 1 step 1 until k do
      if M[i, k] = ls ∧ f[i] ≥ gmin then gmin := f[i]+1 else
      if M[i, k] = eq ∧ f[i] > gmin then gmin := f[i];
    g[k] := gmin;
    for i := k step −1 until 1 do
      if M[i, k] = gr ∧ f[i] ≤ gmin then fixrow (i, k, 1) else
      if M[i, k] = eq ∧ f[i] < gmin then fixrow (i, k, 0)
  end k
end Precedence
```

ALGORITHM 266
PSEUDO-RANDOM NUMBERS [G5]

M. C. Pike and I. D. Hill

(Recd. 15 Feb. 1965 and 6 July 1965)

Medical Research Council, London, England

```
real procedure random (a, b, y);
  real a, b;  integer y;
comment random generates a pseudo-random number in the
  open interval (a, b) where a < b. The procedure assumes that
  integer arithmetic up to 3125 × 67108863 = 209715196875 is
  available. The actual parameter corresponding to y must be an
  integer identifier, and at the first call of the procedure its value
  must be an odd integer within the limits 1 to 67108863 inclusive.
  If a correct sequence is to be generated, the value of this inte-
  ger identifier must not be changed between successive calls of
  the procedure;
begin
  y := 3125 × y;  y := y − (y÷67108864) × 67108864;
  random := y/67108864.0 × (b−a) + a
end random
```

Coveyou [2] showed that for multiplicative congruential methods of generating pseudorandom numbers, the correlation between successive numbers will be approximately the reciprocal of the multiplying factor. Greenberger [3] showed further that the factor should be considerably less than the square root of the modulus.

The method of Algorithm 133 [1] satisfies Greenberger's condition, but since the reciprocal of its multiplying factor is as high as 0.2, Coveyou's result shows that it is very unsatisfactory for purposes requiring statistically independent consecutive random numbers.

Algorithms 133 and 266 have both been tested by computing a number of sets of 2000 successive random integers between 0 and 9, dividing each set into 400 groups of 5, and performing the poker test [4]. The results were classified in the following seven categories:

(*i*)	all different
(*ii*)	1 pair
(*iii*)	2 pairs
(*iv*)	3 of a kind
(*v*)	3 of a kind and 1 pair
(*vi*)	4 of a kind
(*vii*)	5 of a kind.

The following tables resulted:

ALGORITHM 133

Run	*Starting Value*	(*i*)	(*ii*)	(*iii*)	(*iv*)	(*v*)	(*vi*)	(*vii*)
1	13421773	114	193	42	37	7	7	0
2	22369621	111	181	46	40	14	8	0
3	33554433	130	178	48	28	7	6	3
4	6871947673	118	179	51	35	10	5	2
5	11453246123	128	189	44	28	6	4	1
6	17179869185	135	155	45	52	6	5	2
Expected for each Run		120.96	201.60	43.20	28.80	3.60	1.80	0.04
Total for 6 Runs		736	1075	276	220	50	35	8
Expected for Total		725.76	1209.60	259.20	172.80	21.60	10.80	0.24

ALGORITHM 266

Run	*Starting Value*	(*i*)	(*ii*)	(*iii*)	(*iv*)	(*v*)	(*vi*)	(*vii*)
1	13421773	132	191	35	38	2	2	0
2	22369621	140	187	45	27	0	1	0
3	33554433	129	198	44	25	4	0	0
4	8426219	107	202	50	37	2	2	0
5	42758321	101	207	60	25	5	2	0
6	56237485	118	203	42	34	1	2	0
7	62104023	119	206	41	27	6	1	0
Expected for each Run		120.96	201.60	43.20	28.80	3.60	1.80	0.04
Total for 7 Runs		846	1394	317	213	20	10	0
Expected for Total		846.72	1411.20	302.40	201.60	25.20	12.60	0.28

Combining categories (*vi*) and (*vii*) in each case, the observed totals give χ^2 values (on 5 degrees of freedom) of 159.0 for Algorithm 133, and of 3.28 for Algorithm 266.

References:

1. Behrenz, P. G. Algorithm 133, Random. *Comm. ACM 5* (Nov. 1962), 553.
2. Coveyou, R. R. Serial correlation in the generation of pseudo-random numbers. *J. ACM 7*(1960), 72–74.
3. Greenberger, M. An a priori determination of serial correlation in computer generated random numbers. *Math. Comput. 15*(1961), 383–389. Correction in *Math. Comput.16*(1962), 126.
4. Kendall, M. G., and Babington-Smith, B. Randomness and random sampling numbers. *J. Royal Statist. Soc. 101* (1938), 147–166.

REMARK ON ALGORITHM 266 [G5] PSEUDO-RANDOM NUMBERS [M. C. Pike and I. D. Hill, *Comm. ACM 8* (Oct. 1965), 605]

M. C. Pike and I. D. Hill (Recd. 9 Sept. 1965)
Medical Research Council, London, England

Algorithm 266 assumes that integer arithmetic up to 3125 × 67108863 = 209715196875 is available, which is not so on many computers. The difficulty arises in the statements

```
y := 3125 × y;  y := y − (y÷67108864) × 67108864;
```

They may be replaced by

```
integer k;
for k := ⟨for list⟩ do
begin
  y := k × y;
  y := y − (y÷67108864) × 67108864
end;
```

where the ⟨for list⟩ may be

125, 25 (requiring integer arithmetic up to less than 2^{33})
25, 25, 5 (requiring integer arithmetic up to less than 2^{31})
or
5, 5, 5, 5, 5 (requiring integer arithmetic up to less than 2^{29})

according to the maximum integer allowable. The first is appropriate for the ICT Atlas. [And also for the IBM 7090, the second for the IBM System/360 . . . Ref.]

Note. There are frequently machine-dependent instructions available which will give the same values as the above statements much more quickly, if speed is of much importance.

REMARK ON ALGORITHM 266 [G5] PSEUDO-RANDOM NUMBERS [M. C. Pike and I. D. Hill, *Comm. ACM 8* (Oct. 1965), 605]

L. Hansson (Recd. 25 Jan. 1966)
DAEC, Riso, Denmark

As stated in Algorithm 266, that algorithm assumes that integer arithmetic up to 3125 × 67108863 = 209715196875 is available. Since this is frequently not the case, the same algorithm with the constants 125 and 2796203 may be useful. In this case the procedure should read

```
real procedure random (a, b, y);
  real a, b;  integer y;
begin
  y = 125 × y; y := y − (y÷2796203) × 2796203;
  random := y/2796203 × (b−a) + a
end
```

The necessary available integer arithmetic is 125 × 2796203 = 348525375 < 2 ↑ 29. With this procedure body, any start value within the limits 1 to 2796202 inclusive will do.

Seven typical runs of the poker-test gave the results:

start value	*all different*	*1 pair*	*2 pairs*	*3*	*3 + pair*	*4*	*5*
100001	129	199	39	31	2	0	0
1082857	115	206	45	31	2	1	0
724768	120	195	49	32	3	1	0
78363	130	198	36	31	5	0	0
1074985	127	189	44	34	4	2	0
2567517	124	193	50	28	3	2	0
2245723	119	202	49	24	4	1	1
Totals for 7 runs:							
	864	1382	312	211	23	7	1
Totals for 100 consecutive runs with first start value 100001:							
	12023	20297	4301	2837	358	181	3

Certification of Algorithm 266 [G5] Pseudo-Random Numbers [M.C. Pike and I.D. Hill, *Comm. ACM 8* (Oct. 1965), 605]

Walter L. Sullins [Recd. 12 Feb. 1971]
School of Education, Indiana State University
Terre Haute, IN 47809

Key Words and Phrases: pseudo-random numbers, testing random number generators
CR Categories: 5.5

The Pike and Hill Algorithm 266 [2] generates pseudo-random numbers in a prescribed open interval. Pike and Hill presented favorable evidence for the serial and poker tests [1] but omitted discussion of frequency tests.

The purpose of the present certification was to test the hypothesis that the numbers generated by the algorithm are rectangularly distributed. Nine sequences of numbers in the interval (0, 1) were generated, and each was divided into 500 blocks of various lengths. In each case the distribution of numbers was tested against a uniform distribution, with .1 interval width, by computing χ^2 on nine degrees of freedom for each of the 500 blocks within the sequence. The results are given in the table below.

Run	Starting value	Block length	Sequence length	Proportion
1	32347753	400	200,000	.012
2	52142147	600	300,000	.018
3	52142123	640	320,000	.014
4	53214215	960	480,000	.008
5	23521425	1000	500,000	.006
6	42321479	1040	520,000	.006
7	20302541	1560	780,000	.006
8	32524125	1600	800,000	.010
9	42152159	2600	1,300,000	.004

The proportions reported are the proportions of the 500 blocks which produced significant chi-square values when the probability of incorrectly rejecting the hypothesis of uniformity was set at .01. Thus there is considerable assurance that the numbers generated by the algorithm are rectangularly distributed. These findings also support the algorithm with respect to Yule's [3] recommendation that block sums be compared with expectation.

References

1. Kendall, M.G., and Babington-Smith, B. Randomness and random sampling numbers. *J. Royal Statist. Soc. 101* (1938), 147–166.
2. Pike, M.C., and Hill, I.D. Algorithm 266: Pseudo-random numbers. *Comm. ACM 8* (Oct. 1965), 605.
3. Yule, G. Udny. A test of Tippett's random sampling numbers. *J. Royal Statist. Soc. 101* (1938), 167–172.

ALGORITHM 267
RANDOM NORMAL DEVIATE [G5]

M. C. Pike (Recd. 3 May 1965 and 6 July 1965)
Medical Research Council, London, England

```
procedure RND(x1, x2, Random);
  real procedure Random;  real x1, x2;
comment RND uses two calls of the real procedure Random
  which is any pseudo-random number generator which will
  produce at each call a random number lying strictly between 0
  and 1. A suitable procedure is given by Algorithm 266, Pseudo-
  Random Numbers [Comm. ACM 8(Oct. 1965), 605] if one chooses
  a = 0, b = 1 and initializes y to some large odd number, such as
  13421773. RND produces two independent random variables x1
  and x2 each from the normal distribution with mean 0 and
  variance 1. The method used is given by Box, G.E.P., AND
  MULLER, M.E., A note on the generation of random normal
  deviates. [Ann. Math. Stat. 29 (1958), 610-611];
begin real t;
  x1 := sqrt(-2.0 × ln(Random));
  t := 6.2831853072 × Random;
  comment 6.2831853072 = 2 × pi;
  x2 := x1 × sin(t);  x1 := x1 × cos(t)
end RND
```

Algorithm 121, NormDev [*Comm. ACM 5* (Sept. 1962), 482; *8* (Sept. 1965), 556] also produces random normal deviates and Algorithm 200, NORMAL RANDOM [*Comm. ACM 6* (Aug. 1963), 444; *8* (Sept. 1965), 556] produces random deviates with an approximate normal distribution, but the procedure *RND* seems preferable to both of them.

We may compare *NORMAL RANDOM* to *RND* (which is exact) by noting that at recommended minimum *n NORMAL RANDOM* requires 10 calls of *Random* while *RND* gets two independent normal deviates from 2 calls of *Random* and one call each of *sqrt*, *ln*, *sin* and *cos*. Under the stated test conditions a single call of *NORMAL RANDOM* (with $n = 10$) took 20 percent more computing time than a single call of *RND* when the real procedure *Random* was given by Algorithm 266.

To compare *NormDev* to *RND* in the same way, we have first to calculate the expected number of calls of *ln*, *sqrt*, *exp* and *Random* for each call of *NormDev*. This may be done by noting that there is (1) an initial single call of *Random*, then (2) with probability 0.68 a random normal deviate restricted to (0, 1) has to be found and this requires on average 1.36 calls of *Random* and 1.18 calls of *exp*, and (3) with probability 0.32 a random normal deviate restricted to $(1, \infty)$ has to be found and this requires on average 2.04 calls of *Random* and 1.52 calls of each of *ln* and *sqrt*. *NormDev* thus requires on average 2.58 calls of *Random*, 0.80 calls of *exp*, 0.49 calls of *ln* and 0.49 calls of *sqrt*. (Note: *NormDev* requires one further call of *Random* if a signed normal deviate is required.) Under the stated test conditions a single call of *NormDev* took virtually the same amount of computing time as a single call of *RND* when the real procedure *Random* was as above.

(Note: In testing *NormDev* the procedure was speeded up by replacing *A* by 0.6826894 wherever it occurred and removing it from the parameter list. In testing *NORMAL RANDOM Mean*, *Sigma*, *n* were replaced by 0, 1.0 and 10 respectively and removed from the parameter list.)

ALGORITHM 268
ALGOL 60 REFERENCE LANGUAGE EDITOR [R2]

W. M. McKeeman* (Recd. 9 Dec. 1964, 23 Feb. 1965 and 17 May 1965)
Computer Science Department, Stanford University, Stanford, California

* Supported in part by the Office of Naval Research under Contract Nonr 225(37), NR 044-211.

The author expresses his thanks to the referee for several valuable suggestions.

```
procedure Algoledit(characterset, linelimit);
string characterset;
integer linelimit;
comment If this procedure is presented an ALGOL 60 program
or procedure in the form of a sequence of basic symbols, it will
transmit to the output medium a copy of the text with indenta-
tions between each begin-end pair and some rearrangement of the
blank spaces within the text. This procedure is an example of its
own output. It is used to edit ALGOL 60 text that is difficult to read
because, for example, the ALGOL has been transcribed from
printed documents, or written by inexperienced programmers, or
stored in compressed form (i.e., with all redundant blank spaces
removed). The integer "—1" will represent the nonbasic symbol
"carriage return", "—2" will represent an end-of-file mark, other
symbols will have the integer value corresponding to their position
in the parametric string "characterset". The string must contain
exactly the 116 basic symbols of ALGOL 60. The parameter "line-
limit" sets an upper bound on the number of basic symbols that
the user wishes to appear on a line of output. The identifiers
"lsq" and "rsq" will be used in place of strings of length one whose
only elements are " ‘ " and " ’ ", respectively;
begin integer array spacesbefore, spacesafter[1 : 116],
    buffer[1 : linelimit];
    integer tabstop, symbol, i, symbolcount, level;
    Boolean newline;
    integer procedure val(s);
    string s;
    comment The value of this procedure is the integer
    corresponding to the position in the string "characterset"
    of the symbol in the string "s". The body of the
    procedure must be expressed in code;
    procedure get(symbol);
    integer symbol;
    begin insymbol(2, characterset, symbol);
        if symbol = − 2 then go to eof
    end get;
    procedure send(symbol);
    integer symbol;
    begin comment "send" must not break identifiers
        across lines or insert spurious characters into
        strings;
        integer i, u, v;
        if symbol = − 1 ∨ symbolcount ≥ linelimit
        then
        begin v := tabstop;
            if newline then go to E;
            if level ≠ 0 then
            begin comment Inside a string;
                for i := 1 step 1 until
                symbolcount do outsymbol(1,
                characterset, buffer[i]);
                outsymbol(1, characterset, − 1);
                v := 0
            end else
            begin u := symbolcount;
                newline := true;
                if symbol = − 1 then go to D;
                comment Find a convenient place to
                break the line;
                for u := symbolcount − 1 step −
                1 until 1 do if buffer[u + 1] =
                val('␣') ∨ buffer[u] = val(rsq) then
                go to D;
                u := symbolcount;
                comment Send the line;
                D : for i := 1 step 1 until u do
                outsymbol(1, characterset, buffer[i]);
                outsymbol(1, characterset, − 1);
                comment Find a non-blank character
                to start the next line;
                for i := u + 1 step 1 until
                symbolcount do if buffer[i] ≠ val('␣')
                then go to F;
                go to G;
                comment Move a new line to the
                head of the buffer area;
                F : for i := i step 1 until
                symbolcount do
                begin v := v + 1;
                    newline := false;
                    buffer[v] := buffer[i]
                end;
                comment Insert blanks for tab stops;
                G : for i := 1 step 1 until
                tabstop do buffer[i] := val('␣')
            end;
            E : symbolcount := v
        end;
        comment Now we can put the new symbol in the
        buffer array;
        if symbol ≠ − 1 ∧ ¬ (newline ∧ symbol
        = val('␣')) then
        begin symbolcount := symbolcount + 1;
            newline := false;
            buffer[symbolcount] := symbol
        end
    end send;
    for symbol := 1 step 1 until 116 do
    spacesbefore[symbol] := spacesafter[symbol] := 0;
    for symbol := val('+'), val('−'), val('¬'), val(':'),
    val(':='), val('<'), val('≤'), val('='), val('≠'),
    val('≥'), val('>') do spacesbefore[symbol] :=
    spacesafter[symbol] := 1;
```

```
for symbol := val('∧'), val('∨'), val('⊃'), val('≡'),
val('then'), val('else'), val('step'), val('until'),
val('while'), val('do') do spacesbefore[symbol] :=
spacesafter[symbol] := 2;
for symbol := val('go to'), val('begin'), val('if'),
val('for'), val('procedure'), val('value'), val('own'),
val('real'), val('Boolean'), val('integer'), val('array'),
val('switch'), val('label'), val('string'), val(',') do
spacesafter[symbol] := 2;
level := symbolcount := tabstop := 0;
newline := true;
nextsymbol : deblank : get(symbol);
scanned : if symbol = val('␣') ∨ symbol = − 1
then go to deblank;
if symbol = val('begin') then send( − 1) else
if symbol = val('end') then
begin tabstop := tabstop − 5;
      send( − 1)
end;
for i := 1 step 1 until spacesbefore[symbol] do
send(val('␣'));
send(symbol);
for i := 1 step 1 until spacesafter[symbol] do
send(val('␣'));
if symbol = val('comment') then
begin comment Pass comments on unchanged;
      for i := 1 while symbol ≠ val(';') do
      begin get(symbol);
            send(symbol)
      end
end else if symbol = val('end') then
begin comment "end" comments;
      for i := 1 while symbol ≠ val(';') do
      begin get(symbol);
            if symbol = val('else') ∨ symbol =
            val('end') then go to scanned;
            send(symbol)
      end
end else if symbol = val(lsq) then
begin comment Pass strings on unchanged;
      level := 1;
      for i := 1 while level ≠ 0 do
      begin get(symbol);
            send(symbol);
            if symbol = val(lsq) then level := level
            + 1 else if symbol = val(rsq)
            then level := level − 1
      end
end;
if symbol = val('begin') then tabstop := tabstop + 5
else if symbol = val(';') then send( − 1);
go to nextsymbol;
eof : send( − 1);
outsymbol(1, characterset, − 2)
end Algoledit
```

REMARK ON ALGORITHM 268 [R2]
ALGOL 60 REFERENCE LANGUAGE EDITOR

[W. M. McKeeman, *Comm. ACM 8* (Nov. 1965), 667]

G. Sauer (Recd. 23 Dec. 1968)
Institut für Theoretische Physik der Justus-Liebig-Universität, 63 Giessen, West Germany

KEY WORDS AND PHRASES: symbol manipulation
CR CATEGORIES: 4.49

In the **procedure** *send*, replace the line

1 until 1 **do if** *buffer*[*u*+1] =

with the line

1 until *tabstop* **do if** *buffer*[*u*+1] = [1]

The published version fails to clear the buffer when a line to be printed contains no blanks and *tabstop* > 0, causing an array bounds violation. Knowing *buffer*[*tabstop*+1] never to contain a blank character, the search for blanks may be stopped at *u* = *tabstop* + 1.

[1] The author is indebted to the referee for suggesting this brief form.

ALGORITHM 269
DETERMINANT EVALUATION [F3]

Jaroslav Pfann and Josef Straka
(Recd. 10 Sept. 1964 and 29 Dec. 1964)
Institute of Nuclear Research, Řež by Prague, Czechoslovakia

```
real procedure determinant (A, n); array A; integer n;
comment This procedure evaluates a determinant by triangu-
  larization with searching for pivot in row and with scaling of
  the rows of the matrix before the triangularization. This was
  done as in procedure EQUILIBRATE of the Algorithm 135
  [Comm. ACM 5 (Nov. 1962), 553];
begin real product, temp; integer i, j, r, s;
  array mult[1:n];
  procedure EQUILIBRATE(A, n, mult);
    integer n; array A, mult;
  begin integer i, j; real mx;
    for i := 1 step 1 until n do
    begin mx := 0.0;
      for j := 1 step 1 until n do
        if abs(A[i, j]) > mx then mx := abs(A[i, j]);
      if mx = 0.0 then
      begin determinant := 0; go to RETURN end;
      mult[i] := mx; comment := base ↑ ex for exact scaling;
      if mx ≠ 1.0 then
      for j := 1 step 1 until n do A[i, j] := A[i, j]/mx;
    end
  end EQUILIBRATE;
  EQUILIBRATE(A, n, mult);
  product := 1;
  for r := 1 step 1 until n−1 do
  begin s := r; temp := abs(A[r, r]);
    for j := r + 1 step 1 until n do
      if temp < abs(A[r, j]) then
      begin temp := abs(A[r, j]); s := j end;
    if temp = 0 then begin determinant := 0; go to RETURN
      end;
    if s ≠ r then
    begin product := − product;
      for i := r step 1 until n do
      begin temp := A[i, r]; A[i, r] := A[i, s];
        A[i, s] := temp
      end
    end;
    product := product × A[r, r];
    comment Be on guard against overflow or underflow here;
    for i := r+1 step 1 until n do
    begin temp := A[i, r]/A[r, r];
      for j := r+1 step 1 until n do
        A[i, j] := A[i, j] − A[r, j] × temp
    end
  end;
  temp := product × A[n, n];
  for r := 1 step 1 until n do temp := temp × mult [r];
  comment Again danger of overflow or underflow;
  determinant := temp;
RETURN:
end determinant
```

Reference:

McKeeman, W. M. Algorithm 135—Crout with equilibration and iteration. *Comm. ACM 5* (Nov. 1962), 553.

CERTIFICATION OF:
ALGORITHM 41 [F3]
EVALUATION OF DETERMINANT

[Josef G. Solomon, *Comm. ACM 4* (Apr. 1961), 171]

ALGORITHM 269 [F3]
DETERMINANT EVALUATION

[Jaroslav Pfann and Josef Straka, *Comm. ACM 8* (Nov. 1965), 668]

A. Bergson (Recd. 4 Jan. 1966 and 4 Apr. 1966)
Computing Lab., Sunderland Technical College,
Sunderland, Co. Durham, England

Algorithms 41 and 269 were coded in 803 ALGOL and run on a National-Elliott 803 (with automatic floating-point unit).

The following changes were made:
(i) **value** *n*; was added to both Algorithms;
(ii) In Algorithm 269, since procedure *EQUILIBRATE* is only called once, it was not written as a procedure, but actually written into the **procedure** *determinant* body.

The following times were recorded for determinants of order N (excluding input and output), using the same driver program and data.

N	T_1 *Algorithm 41*	T_2 *Algorithm 269*
	(minutes)	
10	0.87	0.78
15	2.77	2.18
20	6.47	4.78
25	12.47	8.99
30	21.37	14.98

From a plot of $\ln(T_1)$ against $\ln(N)$ it was found that

$$T_1 = 0.00104N^{2.92}.$$

Similarly,

$$T_2 = 0.00153N^{2.70}$$

From a plot of T_1 against T_2, it was found that Algorithm 269 was 30.8 percent faster than Algorithm 41, but Algorithm 41 required less storage.

ALGORITHM 270
FINDING EIGENVECTORS BY GAUSSIAN ELIMINATION [F2]

ALBERT NEWHOUSE (Recd. 3 May 1965 and 16 July 1965)
University of Houston, Houston, Texas

procedure *NULLSPACE* (*n*, *a*, *ec*, *eps*); **value** *n*, *eps*; **integer** *n*, *ec*; **real** *eps*; **array** *a*;

comment *NULLSPACE* computes the vectors x of order n such that $xa = z$, where a is an $n \times n$ matrix, z is the zero-vector of order n, *eps* is a small positive number such that if the maximum pivot element is numerically less than *eps* the procedure considers it zero. The *ec* vectors x are to be found in the first *ec* rows of the matrix a upon exit from this procedure;

comment In finding the eigenvectors x of an $n \times n$ matrix B after having found the eigenvalues λ of B by any of the many available methods, it is often desirable to start from the original matrix B and not from its transform from which the λ's were obtained. Whereas the resulting eigenvectors will still be influenced by errors in the λ's, the eigenvectors would not be influenced by errors in the transformed matrix.

Since $\lambda I - B = A$ is a singular matrix of rank r the problem is to find $ec = n - r$ vectors x which form a basis of the left null space of A.

Note: If the right null space is desired the matrix A should be transposed.

The following algorithm finds these $n-r$ linearly independent vectors by the Gauss-Jordan elimination in place using the maximal available element for the pivot. The process will terminate after r steps, since the maximal available elements for pivoting are then equal to zero.

Now, replacing these zero pivot elements by unity, the rows of the matrix, from which no nonzero element has been chosen, are the basis of the null space of A, that is, if x is such a row then $xA = z$, the zero vector of order n.

The proof for this is established by the fact that the elimination amounts to premultiplying B by a matrix A', a product of elementary matrices, such that $A'A$ is a matrix with ones on r of the diagonal positions and zeros everywhere else.

Test results. A version of this procedure acceptable to the IBM 7094 (ALCOR-ILLINOIS 7090 ALGOL Compiler) was tested.

With $eps = 10^{-6}$ the results for the 5×5 matrix

$$\begin{array}{rrrrr} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{array}$$

showed the dimension of the null space as 3 having as a basis

$$x_1 = (-.75,\ 1.00,\ 0.00,\ 0.00,\ -.25)$$

$$x_2 = (-.50,\ 0.00,\ 1.00,\ 0.00,\ -.50)$$

$$x_3 = (-.25,\ 0.00,\ 0.00,\ 1.00,\ -.75)$$

exact to 6 decimal places;

```
begin integer array r, c[1:n];  integer i, j, k, m, jj, kk, t;
  real max, temp;
  for i := 1 step 1 until n do r[i] := c[i] := 0;
  for m := 1 step 1 until n do
  begin max := 0;
    for k := 1 step 1 until n do
    begin if r[k] ≠ 0 then go to L else
      for j := 1 step 1 until n do
        if c[j] = 0∧ abs(a[k, j]) > max then
        begin kk := k; jj := j;  max := abs(a[k, j])
        end j loop;
L:  end k loop;
    if max < eps then go to SORT;
    c[jj] := kk;  r[kk] := jj;  temp := 1/a[kk, jj];  a[kk, jj] := 1;
    for j := 1 step 1 until n do a[kk, j] := a[kk, j] × temp;
    for k := 1 step 1 until kk − 1, kk + 1 step 1 until n do
    begin temp := a[k, jj];  a[k, jj] := 0;
      for j := 1 step 1 until n do
      begin
        a[k, j] := a[k, j] − temp × a[kk, j];
        if abs(a[k, j]) < eps then a[k, j] := 0
      end;
    end k loop;
  end m loop;
SORT:  for j := 1 step 1 until n do
  begin
REPEAT:  if c[j] ≠ 0∧j ≠ c[j] then
    begin
      for k := 1 step 1 until n do
        if r[k] = 0 then
        begin temp := a[k, j];
          a[k, j] := a[k, c[j]];  a[k, c[j]] := temp
        end k loop;
      t := c[j];  c[j] := c[t];  c[t] := t;  go to REPEAT
    end;
  end conditional and j loop;
  ec := 0;
  for k := 1 step 1 until n do
    if r[k] = 0 then
    begin ec := ec + 1;  a[k, k] := 1;
      if ec ≠ k then
      begin
        for j := 1 step 1 until n do a[ec, j] := a[k, j]
      end;
    end conditional and k loop;
  comment The first ec rows of the matrix a are the vectors
    which are orthogonal to the columns of the matrix a;
end NULLSPACE
```

ALGORITHM 271
QUICKERSORT [M1]

R. S. Scowen* (Recd. 22 Mar. 1965 and 30 June 1965)
National Physical Laboratory, Teddington, England

* The work described below was started while the author was at English Electric Co. Ltd, completed as part of the research programme of the National Physical Laboratory and is published by permission of the Director of the Laboratory.

```
procedure quickersort(a, j);
  value j; integer j; array a;
begin integer i, k, q, m, p; real t, x; integer array ut,
  lt[1:ln(abs(j)+2)/ln(2)+0.01];
comment The procedure sorts the elements of the array a[1:j]
  into ascending order. It uses a method similar to that of QUICK-
  SORT by C. A. R. Hoare [1], i.e., by continually splitting the
  array into parts such that all elements of one part are less than
  all elements of the other, with a third part in the middle con-
  sisting of a single element. I am grateful to the referee for point-
  ing out that QUICKERSORT also bears a marked resemblance
  to sorting algorithms proposed by T. N. Hibbard [2, 3]. In par-
  ticular, the elimination of explicit recursion by choosing the
  shortest sub-sequence for the secondary sort was introduced by
  Hibbard in [2].
    An element with value t is chosen arbitrarily (in QUICKER-
  SORT the middle element is chosen, in QUICKSORT a random
  element is chosen). i and j give the lower and upper limits of
  the segment being split. After the split has taken place a value
  q will have been found such that a[q] = t and a[I] ≤ t ≤ a[J]
  for all I, J such that i ≤ I < q < J ≤ j. The program then
  performs operations on the two segments a[i:q−1] and a[q+1:j]
  as follows. The smaller segment is split and the position of the
  larger segment is stored in the lt and ut arrays (lt and ut are
  mnemonics for lower temporary and upper temporary). If the
  segment to be split has two or fewer elements it is sorted and
  another segment obtained from the lt and ut arrays. When no
  more segments remain, the array is completely sorted.
    REFERENCES:
1. HOARE, C. A. R.  Algorithms 63 and 64. Comm. ACM 4 (July
     1961), 321.
2. HIBBARD, THOMAS N. Some combinatorial properties of
     certain trees with applications to searching and sorting.
     J. ACM 9 (Jan. 1962), 13.
3. ——. An empirical study of minimal storage sorting. Comm.
     ACM 6 (May 1963), 206–213;
  i := m := 1;
N: if j−i > 1 then
   begin comment This segment has more than two elements,
     so split it;
     p := (j+i) ÷ 2;
     comment p is the position of an arbitrary element in the
       segment a[i:j]. The best possible value of p would be one
       which splits the segment into two halves of equal size, thus
       if the array (segment) is roughly sorted, the middle ele-
       ment is an excellent choice. If the array is completely
       random the middle element is as good as any other.
         If however the array a[1:j] is such that the parts a[1:j÷2]
       and a[j÷2+1:j] are both sorted the middle element could
       be very bad. Accordingly in some circumstances
       p := (i+j) ÷ 2 should be replaced by p := (i+3×j) ÷ 4
       or p := RANDOM(i, j) as in QUICKSORT;
     t := a[p];
     a[p] := a[i];
     q := j;
     for k := i + 1 step 1 until q do
     begin comment Search for an element a[k] > t starting
         from the beginning of the segment;
       if a[k] > t then
       begin comment Such an a[k] has been found;
         for q := q step −1 until k do
         begin comment Now search for a[q] < t starting from
             the end of the segment;
           if a[q] < t then
           begin comment a[q] has been found, so exchange
             a[q] and a[k];
             x := a[k];
             a[k] := a[q];
             a[q] := x;
             q := q−1;
             comment Search for another pair to exchange;
             go to L
           end
         end for q;
         q := k − 1;
         comment q was undefined according to Para. 4.6.5 of
           the Revised ALGOL 60 Report [Comm. ACM 6 (Jan.
           1963), 1–17];
         go to M
       end;
L:   end for k;
     comment We reach the label M when the search going up-
       wards meets the search coming down;
M:   a[i] := a[q];
     a[q] := t;
     comment The segment has been split into the three parts
       (the middle part has only one element), now store the
       position of the largest segment in the lt and ut arrays and
       reset i and j to give the position of the next largest segment;
     if 2 × q > i + j then
     begin
       lt[m] := i;
       ut[m] := q−1;
       i := q+1
     end
     else
     begin
       lt[m] := q+1;
       ut[m] := j;
       j := q−1
     end;
     comment Update m and split this new smaller segment;
     m := m+1;
     go to N
   end
   else if i ≥ j then
   begin comment This segment has less than two elements;
     go to P
   end
   else
   begin comment This is the case when the segment has just
       two elements, so sort a[i] and a[j] where j = i + 1;
     if a[i] > a[j] then
    begin
      x := a[i];
      a[i] := a[j];
      a[j] := x
    end;
    comment If the lt and ut arrays contain more segments
      to be sorted then repeat the process by splitting the smallest
      of these. If no more segments remain the array has been
      completely sorted;
P:  m := m-1;
    if m > 0 then
    begin
      i := lt[m];
      j := ut[m];
      go to N
    end;
  end
end quickersort
```

CERTIFICATION OF ALGORITHM 271 (M1) QUICKERSORT [R. S. Scowen, *Comm. ACM 8* (Nov. 1965), 669]

Charles R. Blair (Recd. 11 Jan. 1966)
Department of Defense, Washington, D.C.

QUICKERSORT compiled and ran without correction through the ALDAP translator for the CDC 1604A. Comparison of average sorting times, shown in Table I, with other recently published algorithms demonstrates *QUICKERSORT*'s superior performance.

TABLE I. Average Sorting Times in Seconds

Number of items	Algorithm 201 Shellsort		Algorithm 207 Stringsort		Algorithm 245 Treesort 3		Algorithm 271 Quickersort	
	Integers	Reals	Integers	Reals	Integers	Reals	Integers	Reals
10	0.01	0.01	0.03	0.03	0.02	0.02	0.01	0.01
20	0.02	0.02	0.05	0.05	0.04	0.04	0.02	0.02
50	0.08	0.08	0.20	0.20	0.11	0.12	0.06	0.06
100	0.19	0.22	0.39	0.40	0.26	0.27	0.13	0.13
200	0.48	0.53	1.0	1.1	0.59	0.62	0.28	0.30
500	1.5	1.7	2.8	2.9	1.7	1.8	0.80	0.85
1000	3.7	4.2	6.6	6.9	3.7	4.0	1.8	1.9
2000	9.1	10.	13.	14.	8.2	8.7	3.9	4.1
5000	27.	30.	40.	41.	23.	24.	11.	12.
10000	65.	72.	93.	97.	49.	52.	23.	25.

ALGORITHM 272
PROCEDURE FOR THE NORMAL DISTRIBUTION FUNCTIONS* [S15]

M. D. MacLaren

(Recd. 28 July 1964, 17 Nov. 1964 and 26 July 1965)

Argonne National Laboratory, Argonne, Ill., and Boeing Scientific Research Laboratories, Seattle, Wash.

* Work performed in part under the auspices of the US Atomic Energy Commission.

```
real procedure phi(a, k);  value a, k;  real a;  integer k;
comment  Before use, this procedure must be called once with
  k = 3 to initialize own variables. Thereafter for k = 1 the
  procedure gives
```

$$\Phi(a) = \frac{1}{(2\pi)^{\frac{1}{2}}} \int_{-}^{a} \exp\,(-t^2/2)\;dt,$$

and for $k = 2$ it gives

$$\Phi^*(a) = 2(\Phi(|\,a\,|) - .5)$$
$$= \left(\frac{2}{\pi}\right)^{\frac{1}{2}} \int_0^{|a|} \exp\,(-t^2/2)\;dt;$$

```
begin own integer N;
  own real B, EPS, EPS2, EPS3, ONE, DELTA, DELTA2, PI2;
```

comment $\Phi^*(a)$ is computed by Taylor's series expansion in the interval $0 \leq a \leq B$, and by asymptotic series in the interval $B < a$. The Taylor's series expanson is made about one of the points $0, B/N, 2B/N, \cdots, B$, and the coefficients in the series are computed using the recursion formula for Hermite polynomials. The number of terms to take in the series is determined by an error estimate based on a majorizing series. This procedure, which is essentially the familiar one of interpolating in a stored table of values, gives a fast program and can be used effectively for many functions. In this case another significant increase in speed could be obtained by also storing a table of values of the first derivative cf Φ^*. The **own** variables B, EPS and N might be called program parameters. By suitably choosing their values the programmer may make the procedure as accurate as desired and may increase the speed of the procedure at the cost of extra storage space. This is the advantage of this procedure over others previously published in this journal (see [1–4]).

The values of these program parameters are determined when the procedure is coded, not when it is called. They are set by means of an initializing call with $k = 3$. The other **own** variables are computed from B, EPS and N when the initializing call is made. If Fortran IV were used, all the **own** variables could be set by use of a DATA statement. An alternative to either method is to replace all occurrences of the parameters by the appropriate constants.

The choice of the parameter N depends mainly on speed versus storage considerations. The larger N is, the faster the procedure will be and the more storage will be needed. Note, however, that N must be chosen large enough so that $B^2(1/(2N) + 1/(4N^2)) \leq 1$, for otherwise the method of estimating the error in the Taylor's series may fail. The choice of B may also affect the speed, because for smaller values of a the asymptotic series for $\Phi^*(a)$ will take longer than the Taylor's series. The choice of B depends, however, mainly on the error desired. Neglecting roundoff, the maximum error in the computed value of $\Phi^*(a)$ will be EPS if $a \leq B$ or max $(EPS, \delta(a)/2)$ if $B < a$, where $\delta(a)$ is the absolute value of the smallest term in the asymptotic series for $\Phi^*(a)$. Some values of $\delta(a)$ are: $\delta(4) = 3.0 \times 10^{-8}$, $\delta(5) = 3.0 \times 10^{-12}$, $\delta(5.5) = 1.4 \times 10^{-14}$, and $\delta(6) = 4.4 \times 10^{-17}$. If N is large enough, roundoff will be no problem. (The referee has pointed out that the computation for $B < a$ could be made by continued fractions, as in Algorithm 180. The advantage of this would be that the continued fraction expansion converges for all $a > 0$, but roundoff errors may be significant for smaller values of a.)

With the program parameters having the values given below, the procedure was compiled as a Fortran II subroutine on the IBM 1620, using eight-digit floating point arithmetic, and tested for many values of a. The error never exceeded 2×10^{-8}. The program was also compiled with $B = 6.0$, $EPS = 2 \times 10^{-15}$ and $N = 60$, using 15 digit arithmetic. Spot checks turned up no errors greater than 2×10^{-15};

```
own real array C[0:16];
comment  The array C must give the value of Φ*(a) at the
  point of expansion, i.e., C[m] must equal Φ*(mB/N). Tables of
  Φ*(a) to fifteen decimal places are published by the National
  Bureau of Standards [5]. The upper bound for the array must
  equal the value of the program parameter N;
real f, f1, f2, x, y, z, t, t2, xt;
integer m;
real procedure max(x, y);  value x, y;  real x, y;
begin max := if x ≤ y then y else x;
end max;
if k = 3 then
begin comment  initialize own variables;
  EPS := .00000002;  B := 4.0;  N := 16;  C[0] := 0.0;
    C[1] := .19741265;
  C[2] := .38292492;   C[3] := .5467530;
    C[4] := .68268949;
  C[5] := .78870045;   C[6] := .86638560;
    C[7] := .91988169;
  C[8] := .95449974;   C[9] := .97555105;
    C[10] := .98758067;
  C[11] := .99404047;  C[12] := .99730020;
    C[13] := .99884595;
  C[14] := .99953474;  C[15] := .99982317;
    C[16] := .99993666;
  ONE := .99999999;
  comment  ONE is the largest number less than 1 which may
    be represented in the machine. This prevents loss of ac-
    curacy in some implementations of floating point sub-
    traction;
  PI2 := .797884560802865;
  comment  PI2 = (2/π)^(1/2);
  DELTA := B/N;
  DELTA2 := .5 × DELTA;
  EPS3 := 2.0 × EPS;
  t2 := max(B×DELTA, sqrt(2.0)×DELTA2);
  t := DELTA2 × (B+DELTA2);
  x := (t+sqrt(t)) × exp(.5×t);
  y := t2 × (1.0+t2) × exp(.5×t2↑2);
  if t2 ≤ 1 ∧ y ≤ x then EPS2 := EPS/y else EPS2 := EPS/x;
```

```
   phi := 0
 end initialization
 else
 begin comment  compute Φ(a);
   y := abs(a);
   if y > B then
   begin comment  computation by asymptotic series;
     x := y ↑ 2;  f := PI2 × exp(−.5×x)/y;
     x := 1.0/x;  z := f; f1 := −f × x;
       for m := 3, m + 2 while abs(f1) < abs(f) do
       begin z := z + f1;  f := f1;  f1 := −f1 × m × x;
            if abs(f) ≤ EPS3 then go to L1
       end;
L1:          z := ONE − z + .5 × f
     end asymptotic computation
     else
     begin comment  Taylor's series computation;
       m := entier (y/DELTA);
       x := m × DELTA;  t := y − x;
       if DELTA2 < t then
       begin m := m + 1;  x := x + DELTA;  t := y − x end;
       xt := x × t;  t2 := t ↑ 2;
       f1 := t × PI2 × exp(−.5×x ↑ 2);
       f2 := −.5 × xt × f1;
       z := C[m] + f1 + f2;
       for m := 3, m + 1 while (m−1) × EPS2 < max(abs(f1),
         abs(f2)) do
       begin
         f := (−xt×f2−t2×(m−2)×f1/(m−1))/m;
         z := z + f;  f1 := f2;  f2 := f;
       end
     end Taylor's series computation;
     if k = 1 then
     begin
       z := if 0 ≤ a then .5 + .5 × z else .5 − .5 × z
       end;
     phi := z
   end computation
end phi
```

References:

1. Crawford, M., and Techo, R. Algorithm 123, Real error function, *ERF*(x). *Comm. ACM 5* (Sept. 1962), 482.
2. Thacher, H. C., Jr. Algorithm 180, Error function—large X. *Comm. ACM 6* (June 1963), 314.
3. Ibbetson, D. Algorithm 209, Gauss. *Comm. ACM 6* (Oct. 1963), 616.
4. Cyvin, S. J. Algorithm 226, Normal distribution function. *Comm. ACM 7* (May 1964), 295.
5. National Bureau of Standards. *Tables of Normal Probability Functions.* Applied Math. Series, No. 23, US Government Printing Off., Washington, D.C., 1953.

REMARKS ON:
ALGORITHM 123 [S15]
REAL ERROR FUNCTION, ERF(x)

[Martin Crawford and Robert Techo *Comm. ACM 5* (Sept. 1962), 483]

ALGORITHM 180 [S15]
ERROR FUNCTION—LARGE X

[Henry C. Thacher Jr. *Comm. ACM 6* (June 1963), 314]

ALGORITHM 181 [S15]
COMPLEMENTARY ERROR FUNCTION—LARGE X

[Henry C. Thacher Jr. *Comm. ACM 6* (June 1963), 315]

ALGORITHM 209 [S15]
GAUSS

[D. Ibbetson. *Comm. ACM 6* (Oct. 1963), 616]

ALGORITHM 226 [S15]
NORMAL DISTRIBUTION FUNCTION

[S. J. Cyvin. *Comm. ACM 7* (May 1964), 295]

ALGORITHM 272 [S15]
PROCEDURE FOR THE NORMAL DISTRIBUTION FUNCTIONS

[M. D. MacLaren. *Comm. ACM 8* (Dec. 1965), 789]

ALGORITHM 304 [S15]
NORMAL CURVE INTEGRAL

[I. D. Hill and S. A. Joyce. *Comm. ACM 10* (June 1967), 374]

I. D. Hill and S. A. Joyce (Recd. 21 Nov. 1966)
Medical Research Council,
Statistical Research Unit, 115 Gower Street, London
W.C.1., England

These algorithms were tested on the ICT Atlas computer using the Atlas Algol compiler. The following amendments were made and results found:

ALGORITHM 123

(i) **value** x; was inserted.
(ii) $abs(T) \leqslant {}_{10}{-10}$ was changed to $Y - T = Y$
both these amendments being as suggested in [1].
(iii) The labels 1 and 2 were changed to $L1$ and $L2$, the **go to** statements being similarly amended.
(iv) The constant was lengthened to 1.12837916710.
(v) The extra statement $x := 0.707106781187 \times x$ was made the first statement of the algorithm, so as to derive the normal integral instead of the error function.

The results were accurate to 10 decimal places at all points tested except $x = 1.0$ where only 2 decimal accuracy was found, as noted in [2]. There seems to be no simple way of overcoming the difficulty [3], and any search for a method of doing so would hardly be worthwhile, as the algorithm is slower than Algorithm 304 without being any more accurate.

ALGORITHM 180

(i) $T := -0.56418958/x/exp(v)$ was changed to
$T := -0.564189583548 \times exp(-v)/x$. This is faster and also has the advantage, when v is very large, of merely giving 0 as the answer instead of causing overflow.
(ii) The extra statement $x := 0.707106781187 \times x$ was made as in (v) of Algorithm 123.
(iii) **for** $m := m + 1$ was changed to **for** $m := m + 2$. $m+1$ is a misprint, and gives incorrect answers.

The greatest error observed was 2 in the 11th decimal place.

ALGORITHM 181

(i) Similar to (i) of Algorithm 180 (except for the minus sign).
(ii) Similar to (ii) of Algorithm 180.
(iii) m was declared as **real** instead of **integer**, as an alternative to the amendment suggested in [4].

The results were accurate to 9 significant figures for $x \leqslant 8$, but to only 8 significant figures for $x = 10$ and $x = 20$.

ALGORITHM 209
No modification was made. The results were accurate to 7 decimal places.

ALGORITHM 226

(i) $10 \uparrow m/(480\times sqrt(2\times 3.14159265))$ was changed to $10 \uparrow m \times 0.000831129750836$.

(ii) **for** i := 1 **step** 1 **until** $2 \times n$ **do** was changed to $m := 2 \times n$; **for** i := 1 **step** 1 **until** m **do**.

(iii) $-(i\times b/n) \uparrow 2/8$ was changed to $-(i\times b/n) \uparrow 2 \times 0.125$.

(iv) **if** $i = 2 \times n - 1$ was changed to **if** $i = m - 1$

(v) $b/(6\times n\times sqrt(2\times 3.14159265))$ was changed to $b/(15.0397696478\times n)$.

Tests were made with $m = 7$ and $m = 11$ with the following results:

x	*Number of significant figures correct*		*Number of decimal places correct*	
	$m = 7$	$m = 11$	$m = 7$	$m = 11$
−0.5	7	11	7	11
−1.0	7	10	7	10
−1.5	7	10	8	10
−2.0	7	9	8	10
−2.5	6	9	8	11
−3.0	6	7	8	9
−4.0	5	7	10	11
−6.0	2	1	12	10
−8.0	0	0	11	9

Perhaps the comment with this algorithm should have referred to decimal places and not significant figures. To ask for 11 significant figures is stretching the machine's ability to the limit, and where 10 significant figures are correct, this may be regarded as acceptable.

ALGORITHM 272
The constant .99999999 was lengthened to .9999999999.

The accuracy was 8 decimal places at most of the points tested, but was only 5 decimal places at $x = 0.8$.

ALGORITHM 304
No modification was made. The errors in the 11th significant figure were:

$abs(x)$	$x > 0 \equiv upper$	$x > 0 \not\equiv upper$
0.5	1	1
1.0	1	2
1.5	21[a](5)	2
2.0	25[a](0)	4
3.0	0	0
4.0	2	3
6.0	6	0
8.0	14	0
10.0	23	0
20.0	35	0

[a] Due to the subtraction error mentioned in the comment section of the algorithm. Changing the constant 2.32 to 1.28 resulted in the figures shown in brackets.

To test the claim that the algorithm works virtually to the accuracy of the machine, it was translated into double-length instructions of Mercury Autocode and run on the Atlas using the EXCHLF compiler (the constant being lengthened to 0.398942280401432677939946). The results were compared with hand calculations using Table II of [5]. The errors in the 22nd significant figure were:

$abs(x)$	$x > 0 \equiv upper$	$x > 0 \not\equiv upper$
1.0	2	3
2.0	7	1
4.0	2	0
8.0	8	0

Timings. Timings of these algorithms were made in terms of the Atlas "Instruction Count," while evaluating the function 100 times. The figures are not directly applicable to any other computer, but the relative times are likely to be much the same on other machines.

Instruction Count for 100 Evaluations

$abs(x)$	Algorithm number							
	123	180	181	209	226 $m = 7$	272	304[a]	304[b]
0.5	58			8	97	24	25	24
1.0	65[c]			8	176	24	29	29
1.5	164	128	127	9	273	25	35	35
2.0	194	78	90	8	387	24	39	39
2.5	252	54	68	10	515	24	131	44
3.0		42	51	9	628	25	97	50
4.0		27	39	9	900[d]	25	67	44
6.0		15	30	6	1400[d]	16	49	23
8.0		9	28	7	2100[d]	18	44	11
10.0		10	25	5	2700[d]	16	38	11
20.0		9	22	5	6500[d]	16	32	11
30.0		9	9	5	10900[d]	16	11	11

[a] Readings refer to $x > 0 \equiv upper$.

[b] Readings refer to $x > 0 \not\equiv upper$.

[c] Time to produce incorrect answer. A count of 120 would fit a smooth curve with surrounding values.

[d] 100 times Instruction Count for 1 evaluation.

Opinion. There are advantages in having two algorithms available for normal curve tail areas. One should be very fast and reasonably accurate, the other very accurate and reasonably fast. We conclude that Algorithm 209 is the best for the first requirement, and Algorithm 304 for the second.

Algorithms 180 and 181 are faster than Algorithm 304 and may be preferred for this reason, but the method used shows itself in

Algorithm 181 to be not quite as accurate, and the introduction of this method solely for the circumstances in which Algorithm 180 is applicable hardly seems worth while.

Acknowledgment. Thanks are due to Miss I. Allen for her help with the double-length hand calculations.

REFERENCES:

1. THACHER, HENRY C. JR. Certification of Algorithm 123. *Comm. ACM 6* (June 1963), 316.
2. IBBETSON, D. Remark on Algorithm 123. *Comm. ACM 6* (Oct. 1963), 618.
3. BARTON, STEPHEN P., AND WAGNER, JOHN F. Remark on Algorithm 123. *Comm. ACM 7* (Mar. 1964), 145.
4. CLAUSEN, I., AND HANSSON, L. Certification of Algorithm 181. *Comm. ACM 7* (Dec. 1964), 702.
5. SHEPPARD, W. F. *The Probability Integral.* British Association Mathematical Tables VII, Cambridge U. Press, Cambridge, England, 1939.

REMARK ON ALGORITHM 272
PROCEDURE FOR THE NORMAL DISTRIBUTION FUNCTIONS [S15] [M. D. MacLaren, *Comm. ACM 8* (Dec. 1965), 789]

M. D. MACLAREN (Recd. 26 Dec. 1967)
Argonne National Laboratory, Argonne, Ill. 60439

KEY WORDS AND PHRASES: normal distribution function, error function, normal function, normal curve integral
CR CATEGORIES: 5.5, 5.12

In [1] Hill and Joyce report that the value produced by Algorithm 272 for the argument $a = 0.8$ is correct only to 5 decimal places, although the algorithm specifies an accuracy of 2×10^{-8}. Upon checking we have found that the source of this inaccuracy is a typographical error in the section beginning "**begin comment** initialize **own** variables;" The statement initializing $C[3]$ should be changed to "$C[3] = .54674530$." With this change the published algorithm is, as far as we know, accurate within the specified error limit of 2×10^{-8}.

In the first comment of the algorithm the lower limit of the first integral should be minus infinity and not merely a minus sign.

REFERENCE:

1. HILL, I. D., AND JOYCE, S. A. Remark on algorithm 123. *Comm. ACM 10* (June 1967), 377.

ALGORITHM 273
SERREV [C1]

HENRY C. THACHER, JR. (Recd. 2 Apr. 1965)
Argonne National Laboratory, Argonne, Illinois

(Work supported by the US Atomic Energy Commission.)

```
procedure SERREV (A, B, C, N);
  value N;  integer N;  array A, B, C;
comment  This procedure produces in the array C the coefficients
  of the power series $y^j = \sum_{i=j}^N C_{ij}x^i$, where y is the solution of

          $f(y) = \sum_{i=1}^N A_i y^i = g(x) = \sum_{i=1}^N B_i x^i$

  and $A_1 = 1$. The arrays A and B are linear, with bounds 1 and
  $M \geqq N$. The array C is square, with bounds 1:M, 1:M. Ele-
  ments above the diagonal are not used. The derivation of the
  method is given in [1];
begin integer I, J, K, LIM;  real T;
  for I := 1 step 1 until N do
    begin for J := I-1 step -1 until 1 do
      begin T := 0;  LIM := I-J;
        for K := 1 step 1 until LIM do T := C[K,1] × C[I-K,J]
          + T;  C[I, J+1] := T
      end for J;
      T := B[I];
      for J := 2 step 1 until I do T := T-A[J] × C[I,J];
      C[I,1] := T
    end for I
end
```

REFERENCE:

1. THACHER, H. C., Jr. Solution of transcendental equations by series reversion. *Comm. ACM 9* (Jan. 1966), 10-11.

ALGORITHM 274
GENERATION OF HILBERT DERIVED TEST MATRIX [F1]

J. Boothroyd (Recd. 19 May 1965 and 27 Aug. 1965)
University of Tasmania, Hobart, Tas., Australia

procedure *testmx*(*a*,*n*); **value** *n*; **integer** *n*; **array** *a*;

comment T. J. Dekker, "Evaluation of Determinants, Solution of Systems of Linear Equations and Matrix Inversion" [Rep. No. MR63, Mathematical Centre, Amsterdam] describes a test matrix $M[1:n,\ 1:n]$ with the following properties:

(a) elements $M[i,j]$ are positive integers,
(b) the inverse has elements $(-1) \uparrow (i+j) \times M[i,j]$,
(c) the degree of ill-condition increases rapidly with increasing n.

Such matrices may be formed by $M = FG^{-1}HG$ where F is a diagonal matrix *diag*(*fi*) with $fi = \mathit{factorial}\ (n+i-1)/(\mathit{factorial}\ (i-1)\uparrow 2)/\mathit{factorial}(n-i)$, H is the order n segment of a Hilbert matrix and G is diagonal, *diag*(*gi*), with *gi* derived from the prime decomposition of *fi* by:

$$fi = p1^{m1}p2^{m2} \cdots pk^{mk}, \qquad gi = p1^{m \div 2}p2^{m \div 2} \cdots pk^{mk \div 2}.$$

This procedure forms matrices $a[1:n,\ 1:n]$ of this type and follows Dekker in principle but not in detail. Factorials are avoided by evaluating the *fi* with a recursion sequence

$$f[1] := n, \qquad f[i+1] := f[i] \times (n\uparrow 2 - i\uparrow 2) \div i\uparrow 2 \qquad (i=1,\ 2,\ \cdots,\ n-1),$$

permitting the exact computation of *fi* for much larger n than would otherwise be possible. In the evaluation of expressions of the form $(a\times b) \div c$, where the result is integral but c is not a factor of either a or b, numerator integer overflow is avoided by the simple device

$$\text{expression} := q \times b + (r\times b) \div c \quad \text{where} \quad a = q \times c + r.$$

Test matrices for $2 \leqq n \leqq 15$ have been computed on a machine with a 39-bit integer register. During tests of the procedure the specification of the array parameter was changed from **real** to **integer** and the results checked by matrix multiplication using an exact double precision integer inner-product routine. The unit matrix was obtained in all cases. As **real** arrays these matrices will find use only for values of n such that all integer elements have an exact floating point representation. For $10 \leqq n \leqq 15$ the values of the elements of largest modulus are:

n	$M[i,j]max$
10	1616615
11	49884120
12	108636528
13	490804314
14	1859890032
15	22096817600;

```
begin integer i, j, k, fi, gi, d, q, r;  Boolean even;
  integer array f, g[1:n];
  comment  First we compute F = diag(fi);
  fi := f[1] := n;  j := n × n;
  for i := 1 step 1 until n−1 do
  begin d := i × i;  k := j−d;
    q := fi ÷ d;  r := fi−q × d;
    f[i+1] := fi := q × k + (r×k) ÷ d
  end;
  comment  And now, using a modified prime factors algorithm
    to obtain G = diag(gi), we compute FG⁻¹, whose elements re-
    place those of F;
  for i := 1 step 1 until n do
  begin d := gi := 1;  q := fi := f[i];  j := 2;
newj:  even := false;
next:  if q ≧ j then
    begin q := fi ÷ j;
      if fi ≠ q × j then
      begin j := j+d;  d := 2;  go to newj end;
      if even then gi := gi × j;  even := ¬ even;
      fi := q;  go to next
    end;
    g[i] := gi;  f[i] := f[i] ÷ gi
  end;
  comment  Finally, in one operation (FG⁻¹)HG where H is a
    nonexistent Hilbert matrix whose reciprocal elements,
    i+j−1, are computed as we go;
  for i := 1 step 1 until n do
  begin fi := f[i];
    for j := 1 step 1 until n do
    begin gi := g[j];  k := i+j−1;
      q := fi ÷ k;  r := fi − q × k;
      a[i, j] := q × gi + (r×gi) ÷ k
    end
  end

end testmx
```

REMARK ON ALGORITHM 274 [F1]
GENERATION OF HILBERT DERIVED TEST MATRIX [J. Boothroyd, *Comm. ACM 9* (Jan. 1966), 11]

J. Boothroyd (Recd. 7 Jan. 1969)
University of Tasmania, Hobart, Tasmania, Australia

KEY WORDS AND PHRASES: test matrix, Hilbert matrix
CR CATEGORIES: 5.14

An alternative, simpler, and more efficient procedure for generating test matrices having the same properties as those generated by Algorithm 274 is given below. The method, like that of Algorithm 274, is due to T. J. Dekker and may be described as follows.

The elements of the inverse of a segment of a Hilbert matrix are given by

$$(H^{-1}) = (-1)^{i+j} \times f_i \times f_j/(i+j-1)$$

where

$$f_i = \mathit{factorial}\ (n+i-1)/(\mathit{factorial}\ (i-1)) \uparrow 2/\mathit{factorial}\ (n-i).$$

The f_i may be factored as $f_i = f_{i1} \times f_{i2}$, in which

$$f_{i1} = \binom{n+i-1}{i-1} \times n, \qquad f_{i2} = \binom{n-1}{n-i}.$$

Test matrices T are constructed by $T = D_1 H D_2$ where $D_1 = diag\ (f_{i1})$, $D_2 = diag\ (f_{i2})$, and H is the Hilbert matrix segment $H_{i,j} = 1/(i+j-1)$. It may be seen that this is equivalent to defining the T matrices by:

$$T_{i,j} = (fi)(fj)/(i+j-1),$$

$$fi = \binom{n+i-1}{i-1} \times n, \qquad fj = \binom{n-1}{n-j},$$

with fi, fj given by the recurrence relations:

$$(fi)_1 = n, \qquad (fi)_{i+1} = (fi)_i \times (n+i)/i,$$

$$(fj)_1 = 1, \qquad (fj)_{j+1} = (fj)_j \times (n-j)/j.$$

That the condition $K(T)$ of these matrices is severe may be seen from an observation of the referee, who notes that

$$K(T) = \| T \| \times \| T^{-1} \| ,$$

$$\geq \max_{1 \leq i,j \leq n} (t_{i,j})\uparrow 2 = (t_{n,(n+1)} \div 2)\uparrow 2 \sim (2\uparrow 3n/13n)\uparrow 2,$$

where $\| \cdot\cdot \|$ is the L_1 , L_2 , L_∞ , or the Euclidean matrix norm.

Other properties of these matrices shared by those of Algorithm 274 are:

(a) Each matrix has unit determinant;

(b) The eigenvalues form a set λ_1 , λ_2 , $\cdots$, $1/\lambda_2$, $1/\lambda_1$, so that odd order matrices have one eigenvalue of unity.

The procedure *testmx*1 below has been tested on an Elliott 503 (positive integer word length of 38 bits) and matrices of all orders up to 13 were generated before integer overflow occurred with $n = 14$.

```
procedure testmx1 (a, n);  value n;  integer n;  array a;
comment  generates in a[1 : n, 1 : n] test matrices with integer
  elements given by
```

$$t_{i,j} = \binom{n+i-1}{i-1} \times n \times \binom{n-1}{n-j} / (i+j-1)$$

and such that the elements of T inverse are $(-1)^{i+j} \times t_{i,j}$.

To determine for a particular computer that limit on n which permits the exact machine representation of all elements of these matrices, the following maximum values are listed:

n	$t_{i,j}$ (max)
8	163800
9	1178100
10	8314020
11	61108047
12	440936496;

```
begin
  integer i, j, fi, fj, iless1;
  fi := n;  iless1 := 0;
  for i := 1 step 1 until n do
  begin
    fj := 1;
    for j := 1 step 1 until n do
    begin
      a[i, j] := (fi×fj) ÷ (iless1+j);
      fj := ((n−j)×fj) ÷ j
    end;
    fi := ((n+i)×fi) ÷ i;  iless1 := i
  end
end testmx1
```

Proofs that the test matrices described above have integer elements and checkerboard inverses follow the lines of similar proofs given in [1].

Acknowledgments: Thanks are due to T. J. Dekker for communicating details of this method and to the referee for the contribution mentioned.

REFERENCE:

1. DEKKER, T. J. Evaluation of determinants, solution of systems of linear equations and matrix inversion. Rep. No. MR63, Mathematical Centre, Amsterdam, June 1963, pp. 8 and 9.

ALGORITHM 275
EXPONENTIAL CURVE FIT [E2]

GERARD R. DEILY (Recd. 27 July 1964 and 16 Apr. 1965)
US Department of Defense, Washington, D. C.
(Now with HRB-Singer, Inc., State College, Pa.)

```
procedure EXPCRVFT (a, b, c, E squared, n, x, y, epsilon, l max,
  flag);
  integer n, l max, flag;
  real a, b, c, E squared, epsilon;
  real array x, y;
comment This algorithm will fit a curve defined by the equation
  y = a × exp(b × x) + c to a set {x_i , y_i} of n data points. The
  Taylor series modification of the classical least squares method
  is utilized to approximate a solution to the system of nonlinear
  equations of condition. After every iteration, the statistic E
  squared is computed as a measure of the goodness of fit. Com-
  mencing with the second iteration, the successive values of E
  squared are differenced, and when the difference in absolute
  value becomes less than epsilon, the calculations cease. If the
  number of iterations necessary to achieve this result exceeds
  l max, a flag is set to a nonzero value and the procedure is termi-
  nated;
begin
  integer i, l, m;
  comment Computation of initial estimates follows;
  b  := 2 × ln(abs(((y[n] − y[n−1]) × (x[2] − x[1]))/
                    ((y[2] − y[1]) × (x[n] − x[n−1]))))/
                    (x[n] + x[n−1] − x[2] − x[1]);
  a  := (y[n] − y[n−1])/((x[n] − x[n−1])
    × exp((b × (x[n] + x[n−1]))/2) × b);
  m := (n+1) ÷ 2;
  c  := y[m] − a × exp(b × x[m]);
  E squared := 0;
  for i := 1 step 1 until n do
    E squared := E squared + (y[i] − c − a × exp(b × x[i]))↑2;
  comment Computation of corrections follows;
  for l := 1 step 1 until l max do
  begin
    real sumex1, sumex2, sumxiex1, sumxiex2, sumxi2ex2, sumyi,
      sumyiex1, sumxyiex1, d11, d12, d13, d22, d23, d33, e1, e2, e3,
      delta11, delta12, delta13, delta22, delta23, delta33, delta, u, v, w,
      save;
    sumex1 := sumex2 := sumxiex1 := sumxiex2 := sumxi2ex2 :=
      sumyi := sumyiex1 := sumxyiex1 := 0;
    for i := 1 step 1 until n do
    begin
      real ex1, ex2, xiex1, xiex2, xi2ex2;
      ex1 := exp(b × x[i]);
      ex2 := ex1↑2;
      xiex1    := x[i] × ex1;
      xiex2    := x[i] × ex2;
      xi2ex2   := x[i] × xiex2;
      sumex1 := sumex1 + ex1;
      sumex2 := sumex2 + ex2;
      sumxiex1   := sumxiex1 + xiex1;
      sumxiex2   := sumxiex2 + xiex2;
      sumxi2ex2 := sumxi2ex2 + xi2ex2;
      sumyi       := sumyi + y[i];
      sumyiex1   := sumyiex1 + y[i] × ex1;
      sumxyiex1 := sumxyiex1 + y[i] × xiex1;
    end computation of sum terms in normal equations;
    d11 := sumex2;
    d12 := sumxiex2 × a;
    d13 := sumex1;
    d22 := sumxi2ex2 × a ↑ 2;
    d23 := sumxiex1 × a;
    d33 := n;
    e1  := − sumex2 × a − sumex1 × c + sumyiex1;
    e2  := − sumxiex2 × a ↑ 2 − sumxiex1 × c × a +
      sumxyiex1 × a;
    e3  := − sumex1 × a − n × c + sumyi;
    delta11 := d22 × d33 − d23 ↑ 2;
    delta12 := d13 × d23 − d12 × d33;
    delta13 := d12 × d23 − d13 × d22;
    delta22 := d11 × d33 − d13 ↑ 2;
    delta23 := d12 × d13 − d11 × d23;
    delta33 := d11 × d22 − d12 ↑ 2;
    delta   := d11 × delta11 + d12 × delta12 + d13 × delta13;
    u := (e1 × delta11 + e2 × delta12 + e3 × delta13)/delta;
    v := (e1 × delta12 + e2 × delta22 + e3 × delta23)/delta;
    w := (e1 × delta13 + e2 × delta23 + e3 × delta33)/delta;
    a  := a + u;
    b  := b + v;
    c  := c + w;
    E squared := 0;
    for i := 1 step 1 until n do
      E squared := E squared + (y[i] − c − a × exp(b × x[i])) ↑ 2;
    if l = 1 then go to retry;
    if abs(save − E squared) < epsilon
    then go to 73
    else if l < l max
      then go to retry
      else go to unfurl;
retry: save := E squared;
  end computation of corrected values of a, b, and c;
unfurl: flag := 1;
73: end least squares curve fit to y = a × exp(b × x) + c
```

ALGORITHM 276
CONSTRAINED EXPONENTIAL CURVE FIT [E2]

GERARD R. DEILY (Recd. 27 July 1964 and 16 Apr. 1965)
US Department of Defense, Washington, D. C.
(Now with HRB-Singer, Inc., State College, Pa.)

```
procedure CSXPCVFT (a, b, c, E squared, n, x, y, k, z, epsilon,
  l max, flag, jump);
  integer n, k, l max, flag, jump;
  real a, b, c, E squared, z, epsilon;
  real array x, y;
comment This algorithm will fit a curve defined by the equation
  y = a × exp(b × x) + c to a set {x_i , y_i} of n data points, and
  constrain the curve so it contains the point (x_k , z). The Taylor
  series modification of the classical least squares method is
  utilized to approximate a solution to the system of nonlinear
  equations of condition. After every iteration, the statistic E
  squared is computed as a measure of the goodness of fit. Com-
  mencing with the second iteration, the successive values of E
  squared are differenced, and when the difference in absolute
  value becomes less than epsilon, the calculations cease. If the
  number of iterations necessary to achieve this result exceeds
  l max, a flag is set to a nonzero value and the procedure is termi-
  nated. In normal usage, the jump parameter is brought in as a
  ZERO.
    With certain data sets, convergence difficulties will be ex-
  perienced. In these cases it is sometimes helpful to first utilize
  the procedure EXPCRVFT [Algorithm 275, Comm. ACM 9
  (Feb. 1966), 85] to obtain initial values for b and c, and then
  bring the jump parameter in as a ONE in order to bypass the
  following starting value computations for b and c.;
begin
  integer i, l, m;
  real exp factor;
  if jump = 1 then go to entry;
  comment Computation of initial estimates follows;
  b := 2 × ln(abs(((y[n] - y[n-1]) × (x[2] - x[1]))/
                  ((y[2] - y[1]) × (x[n] - x[n-1]))))/
                  (x[n] + x[n-1] - x[2] - x[1]);
  m := (n+1) ÷ 2;
  exp factor := exp(b × (x[m] - x[k]));
  c := (y[m] - z × exp factor)/(1 - exp factor);
  a := (z - c) × exp(-b × x[k]);
  E squared := 0;
  for i := 1 step 1 until n do
    E squared := E squared + (y[i] - c - a × exp(b × x[i])) ↑ 2;
  comment Computation of corrections follows;
entry: for l := 1 step 1 until l max do
  begin
    real sumex1, sumex2, sumqex1, sumqex2, sumqex1lsex2,
      sumq2ex2, sumyi, sumyiex1, sumqyiex1, zlsc, d11, d12, d22,
      e1, e2, delta, v, w, save;
    sumex1 := sumex2 := sumqex1 := sumqex2 := sumqex1lsex2 :=
      sumq2ex2 := sumyi := sumyiex1 := sumqyiex1 := 0;
    for i := 1 step 1 until n do
    begin
      real q, ex1, ex2, qex1, qex2, qex1lsex2, q2ex2;
      q         := x[i] - x[k];
      ex1       := exp(b × q);
      ex2       := ex1 ↑ 2;
      qex1      := q × ex1;
      qex2      := q × ex2;
      qex1lsex2    := qex1 - qex2;
      q2ex2        := qex2 × q;
      sumex1       := sumex1 + ex1;
      sumex2       := sumex2 + ex2;
      sumqex1      := sumqex1 + qex1;
      sumqex2      := sumqex2 + qex2;
      sumqex1lsex2 := sumqex1lsex2 + qex1lsex2;
      sumq2ex2     := sumq2ex2 + q2ex2;
      sumyi        := sumyi + y[i];
      sumyiex1     := sumyiex1 + ex1 × y[i];
      sumqyiex1    := sumqyiex1 + qex1 × y[i];
    end computation of sum terms in normal equations;
    zlsc := z - c;
    d11  := sumq2ex2 × zlsc ↑ 2;
    d12  := sumqex1lsex2 × zlsc;
    d22  := n - 2 × sumex1 + sumex2;
    e1   := sumqyiex1 × zlsc - sumqex2 × zlsc ↑ 2 -
      sumqex1 × zlsc × c;
    e2 := sumyi - sumyiex1 + sumex1 × (2 × c - z) +
      sumex2 × zlsc - n × c;
    delta := d11 × d22 - d12 ↑ 2;
    v  := (e1 × d22 - e2 × d12)/delta;
    w  := (e2 × d11 - e1 × d12)/delta;
    b  := b + v;
    c  := c + w;
    a  := (z - c) × exp(-b × x[k]);
    E squared := 0;
    for i := 1 step 1 until n do
      E squared := E squared +
        (y[i] - c - a × exp(b × x[i])) ↑ 2;
    if l = 1 then go to retry;
    if abs(save - E squared) < epsilon
    then go to 73
    else if l < l max
      then go to retry
      else go to unfurl;
retry: save := E squared;
  end computation of corrected values of a, b, and c;
unfurl: flag := 1;
73: end constrained least squares fit to y = a × exp(b × x) +
```

ALGORITHM 277
COMPUTATION OF CHEBYSHEV SERIES COEFFICIENTS [C6]

LYLE B. SMITH (Recd. 15 July 1965, 23 July 1965 and 20 Sept. 1965)
Stanford University, Stanford, California

```
procedure CHEBCOEFF (F, N, ODD, EVEN, A);
  value N;
  Boolean ODD, EVEN;
  integer N;
  real procedure F;
  array A;
```

comment This procedure approximates the first $N+1$ coefficients, a_n, of the infinite Chebyshev series expansion of a function $F(x)$ defined on $[-1, 1]$.

$$F(x) = \sum_{n=0}^{\infty}{}' a_n T_n(x), \qquad (1)$$

where $\sum'$ denotes a sum whose first term is halved, and $T_n(x)$ denotes the Chebyshev polynomial of the first kind of degree n, defined by

$$T_n(x) = cos\ n\theta, \qquad x = cos\ \theta \qquad (n = 0, 1, 2, \cdots).$$

The truncated series $\sum_{n=0}^{\prime N} a_n T_n(x)$, gives an approximation to $F(x)$ which has maximum error almost as small as that of the "best" polynomial approximation of degree N, see [1]. In this procedure the coefficients, a_n, are closely approximated by $B_{n,N}$, $n = 0(1)N$, which are the coefficients of a "Lagrangian" interpolation polynomial coincident with $F(x)$ at the points x_i, $i = 0(1)N$ where $x_i = cos(\pi i/N)$, see [2]. The $B_{n,N}$ are given by

$$B_{n,N} = \frac{2}{N} \sum_{i=0}^{N}{}'' F(x_i) T_n(x_i) = \frac{2}{N} \sum_{i=0}^{N}{}'' \ F(x_i) T_i(x_n),$$

where $\sum''$ denotes a sum whose first and last terms are halved. The $B_{n,N}$ are evaluated by a recurrence relation described by Clenshaw in [1] and improved by John Rice [5]. This recurrence relation can also be used to evaluate the truncated series, $\sum_{n=0}^{\prime N} a_n T_n(x)$, once *CHEBCOEFF* has found values for the coefficients. For even N a relation between $B_{n,N/2}$ and $B_{n,N}$ (pointed out by Clenshaw [3, p. 27]) is used in computing $B_{n,N}$. For large N, $B_{n,N}$ is very close to a_n. In [2] the relation is given as

$$B_{n,N} = a_n + \sum_{p=1}^{\infty} (a_{2pN-n} + a_{2pN+n}). \qquad (2)$$

This shows that $\frac{1}{2}B_{N,N}$ approximates a_N quite well for large N since from (2) we see that

$$\tfrac{1}{2}B_{N,N} = a_N + a_{3N} + \cdots . \qquad (3)$$

For even N a simple check on the accuracy is available. Since the relation

$$B_{n,N} = B_{n,N/2} - B_{N-n,N}, \qquad n = 0(1)N/2-1 \qquad (4)$$

is used in the computation, the difference

$$B_{n,N/2} - B_{n,N} = B_{N-n,N}, \qquad (5)$$

which measures in some sense the accuracy of the approximation, is available to the user. For instance, in the example below with $N = 8$ the number $A[7]$ is the difference between $A[1]$ for $N = 4$ and $A[1]$ for $N = 8$.

PARAMETER EXPLANATION. If the function F is odd or even then the Boolean parameters *ODD* or *EVEN* should be true respectively in which case every other coefficient in the array A will be zero. The array A will contain the coefficients of the truncated series with $N+1$ terms.

EXAMPLE. For the function $F(x) = e^x$ the following values were computed for $A[n]$ with $N = 4$ and $N = 8$. The computations were done using this procedure written in Extended ALGOL for the Burroughs B5500 computer. Also shown are computed values for the coefficients of the "best" polynomial of degree 8 from [4] (digits differing from the correct result are in italics).

n	$A[n]$ with $N = 4$	$A[n]$ with $N = 8$	"Best" a_n from [4]	Correct a_n from [1]
0	2.53213 *21539*	2.53213 17555	2.53213 17555	2.53213 17555
1	1.1303*2* *14175*	1.13031 82080	1.13031 82080	1.13031 82080
2	0.271*54* *03174*	0.27149 53395	0.27149 53395	0.27149 53395
3	0.044*87* *97762*	0.04433 68498	0.04433 68498	0.04433 68498
4	0.00547 42404	0.00547 42404	0.00547 42404	0.00547 42404
5		0.00054 29263	0.00054 29263	0.00054 29263
6		0.00004 4977*9*	0.00004 49773	0.00004 49773
7		0.00000 3*2095*	0.00000 31984	0.00000 31984
8		0.00000 01992	0.00000 0199*8*	0.00000 01992;

```
begin
  integer i, m, N2, S1, S2, T1;
  real b0, b1, b2, pi, TWOX, FXN2;
  array FX, X[0:N];
  Boolean TEST;
  pi := 3.14159265359;
  N2 := N ÷ 2;
  comment If N is even TEST is set to true;
  if 2 × N2 = N then TEST := true
  else TEST := false;
  comment Compute the necessary function values;
  for i := 0 step 1 until N do
  begin
    X[i] := cos(pi × i/N);
    FX[i] := F(X[i]);
  end;
  S2 := 1;  S1 := 0;
  comment If F(x) is odd or even initialize accordingly;
  if ODD then
  begin
    for m := 0 step 2 until N do
      A[m] := 0;
    S2 := 2;  S1 := 1;
  end else
  if EVEN then
  begin
    for m := 1 step 2 until N do
      A[m] := 0;
    S2 := 2;  S1 := 0;
  end;
  comment If TEST is true the coefficients are computed in
    two steps;
  FXN2 := FX[N]/2.0;
```

```
 if TEST then
 begin
   for m := S1 step S2 until N2 do
   begin
     b1 := 0;
     b0 := FXN2;
     TWOX := 2.0 × X[2 × m];
     for i := N−2 step −2 until 2 do
     begin
       b2 := b1;  b1 := b0;
       b0 := TWOX × b1−b2 + FX[i];
     end;
     A[m] := 2.0 × (X[2×m]×b0−b1+FX[0]/2.0)/N2;
   end;
   A[N2] := A[N2]/2.0;
   T1 := S1;
   if ODD ∨ EVEN then
   begin
     if 2 × (N2 ÷ 2) = N2
     then S1 := N2 + 2 − S1
     else S1 := N2 + 1 + S1;
   end
   else S1 := N2 + 1;
 end;
 comment Compute the desired coefficients;
 for m := S1 step S2 until N do
 begin
   b1 := 0;
   b0 := FXN2;
   TWOX := 2.0 × X[m];
   for i := N−1 step − 1 until 1 do
   begin
     b2 := b1;  b1 := b0;
     b0 := TWOX × b1 − b2 + FX[i];
   end;
   A[m] := 2.0 × (X[m]×b0−b1+FX[0]/2.0)/N;
 end;
 if TEST then
 begin
   for i := T1 step S2 until N2−1 do
   A[i] := A[i] − A[N−i];
 end;
 A[N] := A[N]/2.0;
end CHEBCOEFF
```

References:

1. Clenshaw, C. W. *Chebyshev Series for Mathematical Functions.* MR 26 #362, Nat. Phys. Lab. Math. Tables, Vol. 5, Dep. Sci. Ind. Res., Her Majesty's Stationery Off., London, 1962.
2. Elliott, D. Truncation errors in two Chebyshev series approximations. *Math. Comp. 19* (1965), 234–248.
3. Clenshaw, C. W. A comparison of "best" polynomial approximations with truncated Chebyshev series expansions. *J. SIAM* {B}, *1* (1964), 26–37.
4. Computed values by Dr. C. L. Lawson. (private communication)
5. Rice, John. On the conditioning of polynomials and rational forms. (submitted for publication).

ALGORITHM 278
GRAPH PLOTTER [J6]

P. Lloyd (Recd. 4 June 1965)
Queen Mary College, London, England

```
procedure graphplotter (N, x, y, m, n, xerror, yerror, g, L, S, EM,
  C0, C1, C2, C3, C4, label);
  value N, m, n, xerror, yerror, g, L, S;
  array x, y;
  integer N, g, m, n, L, S;
  real xerror, yerror;
  string EM, C0, C1, C2, C3, C4;
  label label;
comment This procedure is intended to be used to give an approximate graphical display of a multivalued function, y[i, j] of
  x[i], on a line printer. Output channel N is selected for all output from graphplotter. The display is confined to points for which
  1 ≤ i ≤ m and 1 ≤ j ≤ n where 2 ≤ n ≤ 4. If n = 1, then y is
  considered to be a one-dimensional array y[i] and the display is
  again given for 1 ≤ i ≤ m. The format of the print out is arranged so that a margin of g spaces separates the display from
  the left-hand side of the page. L and S denote the number of
  lines down the page and the number of spaces across the page
  which the display will occupy. The graph is plotted so that lines
  1 and L correspond to the minimum and maximum values of x,
  and the spaces 1 and S correspond to the minimum and maximum values of y, that is, y is plotted across the page and x down
  the page. After the graph has been plotted, the ranges of x and
  y for which the display is given are printed out in the order as
  above, separated from the display by a blank line. The strings
  EM ··· C4 must be such that they occupy only one character
  position when printed out. The characters of C1 C2 C3 C4 represent y[i,1] y[i,2] y[i,3] y[i,4]. EM is the character printed out
  round the perimeter of the display. C0 is printed at empty
  positions. At coincident points the order of precedence of the
  characters is C1 C2 C3 C4 EM C0. For the special case n=1 the
  character C1 represents y[i]. Control is passed from the procedure to the point labeled label if the interval between the
  maximum value and minimum values of x[i] is less than xerror,
  or if the range of y is less than yerror. If the values of x[i] occur
  at equal intervals, choosing L=m will make one line equivalent
  to one interval of x;
begin
  real p, q, xmax, xmin, ymax, ymin;
  integer i, j;
  integer array plot[1:L,1:S];
  xmax := xmin := x[1];
  for i := 2 step 1 until m do
  begin
    if x[i] > xmax then xmax := x[i];
    if x[i] < xmin then xmin := x[i]
  end of hunt for maximum and minimum values of x;
  if n=1 then go to N1A;
  ymax := ymin := y[1,1];
  for i := 1 step 1 until m do
    for j := 1 step 1 until n do
    begin
      if y[i,j] > ymax then ymax := y[i,j];
      if y[i,j] < ymin then ymin := y[i,j]
    end of hunt for maximum and minimum values of y;
escape: if abs(xmax−xmin) < xerror ∨ abs(ymax−ymin) <
  yerror then go to label;
  p := (L−1)/(xmax−xmin); q := (S−1)/(ymax−ymin);
  for i := 1 step 1 until L do
    for j := 1 step 1 until S do plot[i,j] := 2;
  for i := 1, L do
      for j := 1 step 1 until S do plot[i,j] := 1;
   for i := 2 step 1 until L−1 do
      for j := 1, S do plot[i,j] := 1;
   if n = 1 then go to N1B;
   for i := 1 step 1 until m do
      for j := n step −1 until 1 do
        plot[1+entier(0.5+p×(x[i]−xmin)),
          1+entier(0.5+q×(y[i,j]−ymin))] := j+2;
 plotter:
   for i := 1 step 1 until L do
   begin
     NEWLINE(N,1); SPACE(N,g);
     comment NEWLINE and SPACE must be declared
       globally to graphplotter, NEWLINE(N,p) outputs p carriage returns and p line feeds on channel N, SPACE(N,p)
       outputs p blank character positions on channel N;
     for j := 1 step 1 until S do
     begin
       switch SW := SW1, SW2, SW3, SW4, SW5, SW6;
       go to SW[plot[i,j]];
SW1: outstring(N,EM);   go to fin;
SW2: outstring(N,C0);   go to fin;
SW3: outstring(N,C1);   go to fin;
SW4: outstring(N,C2);   go to fin;
SW5: outstring(N,C3);   go to fin;
SW6: outstring(N,C4);
fin:
     end
   end of display output;
   NEWLINE(N,2); SPACE(N,g); outreal(N,xmin);
     outreal(N,xmax);
   outreal(N,ymin); outreal(N,ymax);
   go to end;
N1A:
   ymax := ymin := y[1];
   for i := 2 step 1 until m do
   begin
     if y[i] > ymax then ymax := y[i];
     if y[i] < ymin then ymin := y[i]
   end of hunt for maximum and minimum values of y when
     n = 1;
   go to escape;
N1B:
   for i := 1 step 1 until m do
     plot[1+entier(0.5+p×(x[i]−xmin)),
       1+entier(0.5+q×(y[i]−ymin))] := 3;
   go to plotter;
end:
end of graphplotter
```

ALGORITHM 279
CHEBYSHEV QUADRATURE [D1]

F. R. A. Hopgood and C. Litherland (Recd. 31 July 1964, 1 Dec. 1964, 16 Aug. 1965 and 29 Nov. 1965)
Atlas Computer Laboratory, S.R.C., Chilton, Berks, England

```
real procedure cheb(a, b, error, nmax, f);
  value a, b, error, nmax;  real a, b, error;  integer nmax;  real
    procedure f;
comment This routine evaluates the integral of f(x) with lower
  and upper limits set to a and b respectively. The method is
  that suggested by Curtis and Clenshaw [Num. Math. 2 197–205
  (1960),]. The method consists of fitting 2 ↑ n + 1 point Chebyshev
  polynomial to integrand and thus finding integral.  n is
  tried equal to 2 and increased by 1 if error, the relative error,
  is too large. If n reaches maximum nmax without required accuracy
  obtained a message is printed. Accuracy is determined by
  assuming that error is less than the contribution to the integral
  of the last term in the integrated Chebyshev polynomial. After
  n = 2 has been tried, an estimate of the integral is available
  and subsequently the last term in the Chebyshev polynomial is
  found first and this saves evaluating whole polynomial if accuracy
  not obtained. An extra check is that the next two terms
  are also tested allowing up to 8 times error on previous term in
  each case. A reasonable value for nmax is probably 7. Integrals
  requiring many more points than this would probably be better
  tackled using some method which subdivides the range. Also
  the temporary storage required increases considerably for larger
  values of nmax. For example nmax = 10 requires 2048 words;
begin
  real arminl, araddl, bmina, badda, br, bsum, cs, csaddl, csadd2,
    esterr, x, estint, intdv2, twodvn, twotr, verror;
  integer j, k, m, r, s, mmax, mmaxd2, rk;
  k := 2 ↑ (nmax − 2);
  mmaxd2 := 2 × k;
  mmax := 2 × mmaxd2;
  begin
    real array func, cosine [0:mmax];
    bmina := .5 × (b−a);
    badda := .5 × (b×a);
    twodvn := 1;  m := 4;
    comment m+1 is number of points used in Chebyshev fit;
  start:  twodvn := .5 × twodvn;
    bsum := araddl := 0;
    k := k÷2;
    j := if m = 4 then 0 else k;
  fnretn:  if j ≦ mmaxd2 then
      begin
        cosine [j] := if m = 4 then cos (3.14159265 × j/mmax)
          else if j = k then sqrt ((1 + cosine[2 × j])/2)
            else (cosine[j − k] + cosine[j + k])/(2 × cosine[k]);
        cosine [mmax − j] := −cosine[j]
      end;
    x := bmina × cosine [j] + badda;
    func [j] := if j = mmax then .5 × f(x) else f(x);
    j := 2 × k + j;
    comment Evaluates remaining values of integrand required
      storing .5 × lower bound for easier use in Cr recurrence
      formula;
    if mmax ≧ j then go to fnretn;
    if m = 4 then k := 2 × k;
    verror := error;
    r := m;
    rk := mmax;
    comment verror is the error allowed in Chebyshev coefficient
      compared with estimate of integral;
  brretn: twotr := 2 × cosine[rk];
    csadd2 := 0;
    csaddl := 0;
    s := mmax;
  cretn: cs := twotr × csaddl − csadd2 + func[s];
    if s ≠ 0 then
      begin
        csadd2 := csaddl;
        csaddl := cs;
        s := s − k;
        go to cretn
      end recurrence to evaluate next Chebyshev coefficient of
        original function;
    arminl := .5 × twodvn × (cs − csadd2) × (if r = m then
      .5 else 1.0);
    br := .5 × (arminl − araddl)/(r + 1);
    comment br is Chebyshev coefficient of integrated function;
    bsum := bsum + br;
    araddl := arminl;
    comment integral = (b − a) × (b1 + b3 + ··· + .5 × bn);
    if r = m then esterr := br;
    comment error assumed less than last term added in br sum;
    if m ≠ 4 7433 m ≠ mmax 7433 r ≧ m − 4 then
      begin
        if abs(br) ≧ verror × estint then
          begin
    newm:  m : = 2 × m;
            go to start
          end;
        verror := 8 × verror
      end Checks last 3 coefficients to ensure within allowed
        error bounds;
    if r ≠ 0 then
      begin
        r := r − 2;
        rk := rk − 2 × k;
        go to brretn
      end;
    intdv2 := bsum × bmina;
    estint := abs(bsum);
    if error × estint < abs(esterr) then
      begin
        if m ≠ mmax then go to newm;
        outstring (1, 'Accuracy not obtained');
      end;
    cheb := 2 × intdv2
  end
end cheb
```

REMARK ON ALGORITHM 279
CHEBYSHEV QUADRATURE [D1]

F. R. A. Hopgood and C. Litherland

[*Comm. ACM 9* (Apr. 1966), 270]

The 33rd line of the second column on page 270 should read:

if $m \neq 4 \wedge m \neq mmax \wedge r \geq m - 4$ **then**

A printing error showed $\wedge$ as 7433.

CERTIFICATION OF ALGORITHM 279 [D1]
CHEBYSHEV QUADRATURE [F. R. A. Hopgood and C. Litherland, *Comm. ACM 9*, 4 (Apr. 1966), 270]

KENNETH HILLSTROM (Recd. 16 Dec. 1966 and 30 Jan. 1967)

Applied Mathematics Division, Argonne National Laboratory, Argonne, Illinois

Work performed under the auspices of the US Atomic Energy Commission

The 40th line of the first column on page 270 should read: *badda* := .5 × (*b*+*a*);

So corrected, Chebyshev quadrature was coded in CDC 3600 ALGOL. A modified version of this quadrature scheme was coded in 3600 Compass language. In this modification the cosine values are program constants, with 3600 single-precision accuracy, as opposed to program generated values, which tests show have maximum absolute errors of 2^{-35}. These errors are carried into the integrand argument evaluation, resulting in large relative errors in the integrand evaluation, for functions bounded by unity over the interval of integration, for example, e^{-x^2} over (0, 4.3) and sin(x) over (0, 2π), which in turn delays convergence.

Since 3600 Compass does not permit dynamic allocation of storage, the dimension of the cosine array must be fixed. The choice of $129 = 2^7 + 1$ terms is based on the recommendation in the comments of Algorithm 279, "A reasonable value for *n*max is probably 7."

The Chebyshev quadrature 3600 ALGOL program, the modified 3600 Compass routine, and 3600 FORTRAN-coded Romberg and Havie integration routines were tested with six integrands. The Romberg and Havie routines are based upon Algorithm 60, Romberg Integration [*Comm. ACM 4*, (June 1961), 225], and Algorithm 257, Havie Integration [*Comm. ACM 8* (June 1965), 381].

TABLE I

Integrand	*A*	*B*	*EPS*	*VI*	*Routine*	*VA*	*Number of function evaluations*
e^{-x^2}	0	4.3	10^{-6}	0.886226924	Havie	0.886226924	17
					Romberg	0.886226925	65
					Chebyshev	0.886095576	129
					Chebyshev (Rev.)	0.886226926	17
$\sin(x) + 1$	0	2π	10^{-6}	6.283185308	Havie	6.268233308	129
					Romberg	6.268233309	129
					Chebyshev	6.282993876	129
					Chebyshev (Rev.)	6.283185309	5
$(x)^{-(1/2)} \ln\left(\frac{e}{x}\right)$	0	1	10^{-6}	6.0	Havie	5.034254231	129
					Romberg	5.034254231	129
					Chebyshev	5.829597734	129
					Chebyshev (Rev.)	5.701177427	129
$\ln(x)$	1	10	10^{-6}	14.02585088	Havie	14.02585084	65
					Romberg	14.02585085	65
					Chebyshev	14.02585096	17
					Chebyshev (Rev.)	14.02585097	17
$\ln\left(\frac{e}{x}\right)$	0	1	10^{-6}	2.0	Havie	1.979745104	129
					Romberg	1.979745104	129
					Chebyshev	1.999599461	129
					Chebyshev (Rev.)	1.997983436	129
$\frac{1}{(x^4+x^2+0.9)}$	−1	1	10^{-6}	1.5822329[a]	Havie	1.582238946	17
					Romberg	1.582238946	17
					Chebyshev	1.582232967	17
					Chebyshev (Rev.)	1.582232967	17

[a] The value $\int_{-1}^{+1} \frac{dx}{(x^4 + x^2 + 0.9)} = 1.5822329$ is obtained from C. W. Clenshaw and A. R. Curtis, "A method for numerical integration on an automatic computer," *Numer. Math. 2* (1960), 203.

The results of these tests are tabulated in Table I. In the table, *A* is the lower limit of the interval of integration, *B* is the upper limit, *EPS* the convergence criterion, *VI* the value of the integral, and *VA* the value of the approximation.

Due to storage requirements, Chebyshev quadrature is restricted to a maximum of 129 function evaluations. For reasons of comparison, this limit is also imposed on Romberg and Havie quadratures. Thus, in some cases the accuracy called for was not obtained.

REMARK ON CORRECTION TO CERTIFICATION OF ALGORITHM 279 [D1]
CHEBYSHEV QUADRATURE [F.R.A. Hopgood and C. Litherland, *Comm. ACM 9* (Apr. 1966), 270 and *10* (May 1967), 294]

KENNETH HILLSTROM (Recd. 26 June 1967)

Applied Mathematics Division, Argonne National Laboratory, Argonne, Illinois

There are two corrections that should be appended to the certification of Algorithm 279.

Due to programming error, the integrand function routines for e^{-x^2} and $sin(x)+1$, used by the Chebyshev routine, incorrectly evaluated the functions at $x = 0$, thus delaying convergence.

The revised Chebyshev routine still converges more rapidly than the original scheme in the first two examples, but the advantage is much less pronounced than previously indicated.

The amended Table I should read as follows, with the numerical corrections italicized.

TABLE I

Integrand	*A*	*B*	*EPS*	*VI*	*Routine*	*VA*	*Number of function evaluations*
e^{-x2}	0	4.3	10^{-6}	0.886226924	Havie	0.886226924	17
					Romberg	0.886226925	65
					Chebyshev	*0.8862269261*	*33*
					Chebyshev (Rev.)	0.8862269258	17
$sin(x)+1$	0	2π	10^{-6}	6.283185308	Havie	*6.283185307*	*3*
					Romberg	*6.283185307*	*3*
					Chebyshev	*6.2831853086*	*9*
					Chebyshev (Rev.)	6.2831853089	5

ALGORITHM 280
ABSCISSAS AND WEIGHTS FOR GREGORY QUADRATURE [D1]

JOHN H. WELSCH (Recd. 27 Apr. 1965, 14 May 1965, 14 Sept. 1965 and 9 Dec. 1965)
Computation Center, Stanford University, Stanford, California

procedure *gregoryrule* (n, r, t, w);
  value n, r; **integer** n, r; **real array** t, w;
comment Computes the abscissas and weights of the Gregory quadrature rule with r differences:

$$\int_{t_0}^{t_n} f(t)\ dt \approx h\left(\frac{1}{2}f_0 + f_1 + \cdots + f_{n-1} + \frac{1}{2}f_n\right) - \frac{h}{12}(\nabla f_n - \triangle f_0) - \frac{h}{24}(\nabla^2 f_n + \triangle^2 f_0) - \cdots - hc_{r+1}^*(\nabla^r f_n + \triangle^r f_0) = \sum_{j=0}^{n} w_j f(t_j),$$

where $h = (t_n - t_0)/n$, and the $c_j{}^*$ are given in Henrici [1964]. The number r must be an integer from 0 to n, the number of subdivisions. The left and right endpoints must be in $t[0]$ and $t[n]$ respectively. The abscissas are returned in $t[0]$ to $t[n]$ and the corresponding weights in $w[0]$ to $w[n]$.

If $r = 0$ the Gregory rule is the same as the repeated trapezoid rule, and if $r = n$ the same as the Newton-Cotes rule (closed type). The order p of the quadrature rule is $p = r + 1$ for r odd and $p = r + 2$ for r even. For $n \geqq 9$ and large r some of the weights can be negative.

For $n \leqq 32$ and $r \leqq 24$, the numerical integration of powers (less than r) of x on the interval [0, 1] gave 9 significant digits correct in an 11-digit mantissa. Since the binomial coefficients are generated in the local integer array b, integer overflow may occur for large values of r. The type of b can be changed to real to prevent this with no change in the results stated above.

REFERENCES:

HILDEBRAND, F. B. *Introduction to Numerical Analysis.* McGraw-Hill, New York, 1956, p. 155.

HENRICI, PETER. *Elements of Numerical Analysis.* Wiley, New York, 1964, p. 252.;

```
begin integer i, j;  real h, cj;
  integer array b[0: n];  real array c[0: n + 1];
  b[0] := 1;  c[0] := 1.0;  c|1] := −0.5;  b[n] := 0;
  h := (t[n] − t[0])/n;  w[0] := w[n] := 0.5;
  for i := n−1 step −1 until 1 do
    begin w[i] := 1.0;  t[i] := i × h + t[0];  b[i] := 0 end;
  if r > n then r := n;
  for j := 1 step 1 until r do
  begin cf := 0.5 × c[j];
    for i := j step −1 until 1 do b[i] := b[i| − b[i−1];
    for i := 3 step 1 until j + 2 do cj := cj + c[j+2−i]/i;
    c[j+1] := −cj;
    for i := 0 step 1 until n do
      w[i] := w[i] − cj × (b[n − i] + b[i]);
  end;
  for i := 0 step 1 until n do w[i] := w[i] × h
  end gregoryrule
```

ALGORITHM 281
ABSCISSAS AND WEIGHTS FOR ROMBERG QUADRATURE [D1]

JOHN H. WELSCH (Recd. 27 Apr. 1965, 14 May 1965, 14 Sept. 1965 and 9 Dec. 1965)
Computation Center, Stanford University, Stanford, California

```
procedure rombergrule (n, p, t, w);
  value n, p;  integer n, p;  real array t, w;
comment Computes the abscissas and weights of the pth order
  Romberg quadrature rule which features equally spaced ab-
  scissas and positive weights lying between 0.484 × h and 1.4524
  × h (h = subdivision length). The number of subdivisions n
  must be a power of 2 (say 2 ↑ q) and p an even number from 2 to
  2q+2. Romberg integration is normally given as the extrapola-
  tion to the limit of the trapezoid rule. Let
```

$$T_0^{(k)} = h\left(\frac{1}{2}f_0 + f_1 + \cdots + f_{2^k-1} + \frac{1}{2}f_{2^k}\right), \text{ and } T_m^{(k)} = \frac{4^m T_{m-1}^{(k+1)} - T_{m-1}^{(k)}}{4^m - 1},$$

then the Romberg quadrature rule gives

$$\int_{t_0}^{t_n} f(t)\,dt = T_m^{(k)} \approx \sum_{j=0}^{n} w_j f(t_j),$$

where $n = 2^q$, $m = (p - 2)/2$, and $k = q-m$. The left and right endpoints must be in $t[0]$ and $t[n]$ respectively. The abscissas are returned in $t[0]$ to $t[n]$ and the corresponding weights in $w[0]$ to $w[n]$.

If $p = 2$ the Romberg rule is the same as the repeated trapezoid rule, and if $p = 4$, the same as the repeated Simpson rule.

For $n \leqq 128$ and $p \leqq 16$, the numerical integration of powers (less than p) of x on the interval [0, 1] gave answers correct to one round off error in an 11-digit mantissa.

REFERENCE: Bauer, F. L., Rutishauser, H., and Stiefel, E. New aspects in numerical quadrature. *Proc. of Symp. in Appl. Math.*, Vol. 15: High speed computing and experimental arithmetic. Amer. Math. Soc., Providence, R. I., 1963, pp. 199–218;

```
begin integer i, j, m, m1, m4, s;
  real h, ci;  real array c[0: (p − 2)/2];
  h := (t[n] − t[0])/n;  w[0] := w[n] := 0;
  for i := n−1 step −1 until 1 do
    begin w[i] := c[i] := 0;  t[i] := i × h + t[0] end;
  m := (p − 2)/2;  c[0] := 1.0;  s := m4 := 1;  c[n] := 0;
  if m > ln(n)/ln(2) then m := ln(n)/ln(2);
  for j := 1 step 1 until m do
  begin m4 := 4 × m4;  m1 := m4 − 1;
    for i := j step −1 until 1 do
      c[i] := (m4 × c[i] − c[i − 1])/m1;
    c[0] := c[0] × (m4/m1);
  end;
  for i := 0 step 1 until m do
  begin ci := c[i] × s;
    for j := 0 step s until n do w[j] := w[j]+ci;
    s := 2 × s
  end;
  w[0] := w[n] := 0.5×w[0];
  for j := 0 step 1 until n do w[j] := w[j] × h;
end rombergrule
```

REMARK ON ALGORITHM 281 [D1]
ABSCISSAS AND WEIGHTS FOR ROMBERG QUADRATURE [John H. Welsch, *Comm. ACM 9* (Apr. 1966), 273]

J. BOOTHROYD (Recd. 13 Sept. 1966 and 14 Nov. 1966)
University of Tasmania, Hobart, Tasmania, Australia

The following changes which effect two minor improvements and correct two errors are recommended:

1. The expression $(p - 2)/2$, which occurs twice, should preferably be written $(p - 2)\div 2$

2. Delete $c[i] :=$ from the left part list of the statement $w[i] := c[i] := 0$ which occurs within the scope of the first **for** statement

3. Delete the statement $c[n] := 0$;

4. Add, immediately following $m1 := m\ 4 - 1$, the statement $c[j] := 0$;

These changes have been tested by the author of Algorithm 281 using B5500 ALGOL.

ALGORITHM 282
DERIVATIVES OF e^x/x, $cos(x)/x$, AND $sin(x)/x$* [S22]

WALTER GAUTSCHI (Recd. 19 Aug. 1965)
Argonne National Laboratory, Argonne, Ill.

* Work performed under the auspices of the U.S. Atomic Energy Commission. Author's present address is Purdue University.

```
procedure  dsubn(x, nmax, d);
  value x, nmax;  integer nmax;  real x;  array d;
comment  This procedure generates the derivatives
```

$$d_n(x) = \frac{d^n}{dx^n}\left(\frac{e^x}{x}\right) \quad (n = 0, 1, 2, \cdots, nmax)$$

using the recurrence relation

$$d_n(x) = (e^x - nd_{n-1}(x))/x \qquad (n = 1, 2, 3, \cdots).$$

The results are stored in the array d. If $x = 0$, there is an error exit to a global label called *alarm*;

```
begin integer n;  real e;
  if x = 0 then go to alarm;
  e := exp(x);  d[0] := e/x;
  for n := 1 step 1 until nmax do
    d[n] := (e − n × d[n − 1])/x
end dsubn;
procedure csubn(x, nmax, c);
  value x, nmax;  integer nmax;  real x;  array c;
comment  This procedure obtains the derivatives
```

$$c_n(x) = \frac{d^n}{dx^n}\left(\frac{\cos x}{x}\right) (n = 0,1,2,\cdots, nmax)$$

from the recurrence relation

$$c_n(x) = (\tau_n(x) - nc_{n-1}(x))/x \ (n = 1, 2, 3, \cdots),$$

where $\{\tau_n(x)\}_{n=1}^{\infty} = \{-sin\ x,\ -cos\ x,\ sin\ x,\ cos\ x,\ -sin\ x,\ \cdots\}$. The results are stored in the array c. If $x = 0$, there is an error exit to a global label called *alarm*;

```
begin integer n;  array  tau[1: 4];
  if x = 0 then go to alarm;
  tau[1] := −sin(x);  tau[2] := −cos(x);
  tau[3] := −tau[1];  tau[4] := −tau[2];
  c[0] := tau[4]/x;
  for n := 1 step 1 until nmax do
    c[n] := (tau[n−4×((n−1) ÷ 4)] − n×c[n−1])/x
end csubn;
procedure  ssubn(x, nmax, d, s);
  value  x, nmax, d;  integer nmax, d;  real x;  array s;
comment  This procedure generates to d significant digits the
  derivatives
```

$$s_n(x) = \frac{d^n}{dx^n}\left(\frac{sin\ x}{x}\right) \quad (n = 0,1,2,\cdots,nmax),$$

and stores the results in the array s. The method of computation is based on the recurrence relation

$$s_n(x) = (\sigma_n(x) - ns_{n-1}(x))/x \qquad (n = 1, 2, 3, \cdots),$$

where $\{\sigma_n(x)\}_{n=1}^{\infty} = \{cos\ x,\ -sin\ x,\ -cos\ x,\ sin\ x,\ cos\ x, \cdots\}$. The recurrence relation is applied in forward direction as long as $n \leqq |x|$, and in backward direction for the remaining values of n, starting with an appropriately large $n = \nu$. A detailed discussion of the method will be published elsewhere. It is assumed that a global real procedure $t(y)$ is available, which evaluates the inverse function $t = t(y)$ of $y = t\ ln\ t$ to low accuracy for $y \geqq 0$. (See W. Gautschi, Algorithm 236, Bessel functions of the first kind, *Comm. ACM 7* (Aug. 1964), 479 Gautschi, W. Computation of successive derivatives of $f(z)/z$, in press;

```
begin integer n, n0, nu;  real x1, d1, s1;  array sigma [1: 4];
  x1 := abs(x);
  sigma [1] := cos(x);  sigma [2] := −sin(x);
  sigma [3] := −sigma [1];  sigma [4] := −sigma [2];
  n0 := entier (x1);  s[0] := if x ≠ 0 then sigma [4]/x else 1;
  for n := 1 step 1 until if n0 ≦ nmax then n0 else nmax do
    s[n] := (sigma[n − 4 × ((n − 1) ÷ 4)] − n × s[n − 1])/x;
  if n0 < nmax then
  begin
    s1 := 0;  d1 := 2.3026 × d + .6931;
    nu := if nmax ≦ 2.7183 × x1 0 then
      1 + entier  (2.7183 × x1 × t(.36788 × d1/x1))  else
        1 + entier  (nmax × t(d1/nmax));
    for n := nu step −1 until n0+2 do
    begin
      s1 := (sigma[n − 4 × ((n − 1) ÷ 4)] − x × s1)/n;
      if n ≦ nmax + 1 then s[n−1] := s1
    end
  end
end ssubn
```

REMARK ON ALGORITHM 282* [S22]
DERIVATIVES OF e^x/x, cos $(x)/x$, AND sin $(x)/x$

[Walter Gautschi, *Comm. ACM 9* (April 1966), 272]

WALTER GAUTSCHI AND BRUCE J. KLEIN (Recd. 12 May 1969)
Computer Sciences Department, Purdue University, Lafayette, IN 47907 and College of Arts and Sciences, Virginia Polytechnic Institute, Blacksburg, VA 24061

* Work supported by the National Aeronautics and Space Administration NASA under Grant NGR 15-005-039.

KEY WORDS AND PHRASES: recursive computation, successive derivatives, error control
CR CATEGORIES: 5.11, 5.12

For large values of x, and derivatives of order $n > x$, the first two procedures of Algorithm 282 incur substantial loss of accuracy. The reasons for this, as well as remedial measures, are described in the companion article [1]. The following revised procedures, based on this article, are believed to preserve accuracy as far as seems possible. Both procedures call upon the real procedure t of Algorithm 236 [2].

```
procedure dsubn (x, nmax, acc, machacc, d, error);
  value x, nmax, acc, machacc;  integer nmax, acc, machacc;
  real x;  array d;  label error;
```

```
comment  Given $x \neq 0$, nmax, and the number machacc of decimal
  digits available in the mantissa of machine floating-point num-
  bers, this procedure generates the derivatives

      $d_n(x) = \frac{d^n}{dx^n}\left(\frac{e^x}{x}\right)$,    $n = 0, 1, 2, \cdots, nmax$,

  to an accuracy of acc significant decimal digits, except near a
  zero of $d_n(x)$, where some significance may be lost. The result
  $d_n(x)$ is stored in d[n]. If x = 0, the procedure immediately
  exits to the label error;
begin
  integer n0, min, n, n1;  real x1, e, a, q;
  Boolean bool1, bool2;
  if x = 0 then go to error;
  x1 := abs(x);  n0 := x1;  e := exp(x);
  d[0] := e/x;
  a := 1.1513 × (machacc−acc) − .3466;
  if a < 2 then a := 2;
  bool1 := x < 0 ∨ x1 ≦ a;  bool2 := n0 < nmax;
  min := if bool2 then n0 else nmax;
  for n := 1 step 1 until if bool1 then nmax else min do
    d[n] := (e−n×d[n−1])/x;
  if (¬bool1) ∧ bool2 then
  begin
    n1 := 2.7183 × x1 ×
      t((x1+2.3026×acc+.6932)/(2.7183×x1))−1;
    if n1 < nmax then n1 := nmax;
    q := 1/x;
    for n := 1 step 1 until n1 + 1 do q := −n × q/x;
    for n := n1 step −1 until n0 + 1 do
    begin
      q := (e−x×q)/(n+1);
      if n ≦ nmax then d[n] := q
    end
  end
end dsubn;
procedure csubn (x, nmax, acc, machacc, c, error);
  value x, nmax, acc, machacc;  integer nmax, acc, machacc;
  real x;  array c;  label error;
comment  This procedure generates the derivatives

      $c_n(x) = \frac{d^n}{dx^n}\left(\frac{\cos x}{x} \quad \text{for } n = 0, 1, 2, \cdots, nmax\right)$,

  and stores them in the array c. The parameters acc, machacc
  have the same meaning as in the preceding procedure. There is
  an error exit if x = 0;
begin
  integer n0, min, n, n1;  real x1, a, q;  array tau[1:4];
  Boolean bool1, bool2;
  if x = 0 then go to error;
  x1 := abs(x);  n0 := x1;
  tau[1] := −sin(x);  tau[2] := −cos(x);
  tau[3] := −tau[1];  tau[4] := −tau[2];
  c[0] := tau[4]/x;
  a := 2.3026 × (machacc−acc) − .69315;
  if a < 3 then a := 3;
  bool1 := x1 ≦ a;  bool2 := n0 < nmax;
  min := if bool2 then n0 else nmax;
  for n := 1 step 1 until if bool1 then nmax else min do
    c[n] := (tau[n−4×((n−1)÷4)]−n×c[n−1])/x;
  if (¬bool1) ∧ bool2 then
  begin
    n1 := 2.7183 × x1 × t((2.3026×acc+.6932)/(2.7183×x1))−1;
    if n1 < nmax then n1 := nmax;
    q := 1/x;
    for n := 1 step 1 until n1 + 1 do q := −n × q/x;
    for n := n1 step −1 until n0 + 1 do
    begin
      q := (tau[n+1−4×(n÷4)]−x×q)/(n+1);
      if n ≦ nmax then c[n] := q
    end
  end
end csubn
```

REFERENCES:

1. GAUTSCHI, WALTER, AND KLEIN, BRUCE J. Recursive computation of certain derivatives—A study of error propagation. *Comm. ACM 13* (Jan. 1970), 7-9.
2. GAUTSCHI, WALTER. Algorithm 236, Bessel functions of the first kind [S17]. *Comm. ACM 7* (Aug. 1964), 479-480.

ALGORITHM 283
SIMULTANEOUS DISPLACEMENT OF POLYNOMIAL ROOTS IF REAL AND SIMPLE [C2]

IMMO O. KERNER (Recd. 8 Sept. 1965 and 12 Nov. 1965)
Rechenzentrum Universitaet Rostock

```
procedure Prrs (A, X, n, eps);  value n, eps;
  integer n;  real eps;  array A, X;
comment Prrs (polynomial roots real simple) computes the n
  roots X of the polynomial equation

        $A_n x^n + A_{n-1} x^{n-1} + \cdots + A_0 = 0$

  simultaneously. On entry the array X contains the vector of
  initial approximations to the roots and on exit it contains the
  vector of improved approximations to the roots. The initial
  approximations must be distinct. Accuracy is specified by means
  of a parameter eps. Iteration is continued until the Euclidean
  norm of the correction vector does not exceed eps. The con-
  vergence is quadratic;
begin integer i, k;  real x, P, Q;
      eps := eps ↑ 2;
W:  Q := 0;
      for i := 1 step 1 until n do
  begin  x := P := A[n];
    for k := 1 step 1 until n do
    begin x := x × X[i] + A[n − k];
      if k ≠ i then P := P × (X[i]−X[k])
    end;
    X[i] := X[i]−x/P;
    Q := Q + (x/P) ↑ 2
  end;
  if Q > eps then go to W
end
```

ALGORITHM 284
INTERCHANGE OF TWO BLOCKS OF DATA [K2]

WILLIAM FLETCHER (Recd. 25 Oct. 1965 and 24 Nov. 1965)
Bolt, Beranek and Newman, Inc., Cambridge, Mass.
and
ROLAND SILVER
The Mitre Corp., Bedford, Mass.

```
procedure interchange (a, m, n);
  value m, n;  integer m, n;  array a;
comment  This procedure transfers the contents of a[1] ··· a[m]
  into a[n+1] ··· a[n+m] while simultaneously transferring the
  contents of a[m + 1] ··· a[m + n] into a[1] ··· a[n] without using
  an appreciable amount of auxiliary memory.
    The nonlocal procedure gcd (x, y) has value the greatest
  common divisor of the integers x and y. The nonlocal procedure
  swap (x, y) interchanges the values of the variables x and y.
    Let G be the additive group of integers modulo m+n. The
  multiples 0, n, 2n, ··· of n form a cyclic subgroup C of G. The
  order of C is r = (m + n)/d, where d is the greatest common
  divisor of m and n. The integers 1, ··· , d belong to distinct
  cosets C1 ··· Cd of C. These cosets form a disjoint covering of G.
    The interchange procedure is based on the fact that if we start
  with a member x of the coset Cx , and add n repeatedly modulo
  m + n, we will in r steps have generated each member of Cx
  just once;
begin
  integer d, i, j, k, r;
  real t;
  d := gcd (m, n);
  r := (m + n) ÷ d;
  for i := 1 step 1 until d do
  begin
    j := i;
    t := a[i];
    for k := 1 step 1 until r do
    begin
      ïf j ≦ m then j := j + n else j := j − m;
      swap (t, a[j])
    end k
  end i
end interchange
```

ACM Transactions on Mathematical Software, Vol. 2, No. 4, December 1976, Pages 392-393.

REMARK ON ALGORITHM 284

Interchange of Two Blocks of Data [K2]
[W. Fletcher and R. Silver, *Comm. ACM 9*, 5 (May 1966), 326]

M.R. Ito [Recd 25 July 1975 and 25 May 1976]
Department of Electrical Engineering, University of British Columbia, Vancouver, B.C., Canada, V6T 1W5.

The relocation of two contiguous blocks of data performed by Algorithm 284 can be regarded as a permutation problem. That is, the first m components and the last n components of an $(m + n)$ dimensional vector, a, are interchanged by the transformation, $b = Qa$, where Q is a permutation matrix defined in partitioned form as

$$Q = \left[\begin{array}{c|c} \mathrm{O} & I_n \\ \hline I_m & \mathrm{O} \end{array}\right],$$

and I_k is the identity matrix of order k.

Algorithm 284 is in fact equivalent to the representation [1] of the desired permutation as the product of r disjoint cycles, with each cycle comprising d components, where

d = greatest common denominator of m and n;
$r = (m + n) \div d$.

A more efficient algorithm for performing the permutation is based on the following decomposition of Q. Let P_k be the permutation matrix of order k with ones along the minor diagonal (zeros elsewhere). Then, Q can be decomposed as

$$Q = P_{m+n}RS,$$

where

$$R = \left[\begin{array}{c|c} I_m & \mathrm{O} \\ \hline \mathrm{O} & P_n \end{array}\right], \qquad S = \left[\begin{array}{c|c} P_m & \mathrm{O} \\ \hline \mathrm{O} & I_n \end{array}\right].$$

The partial permutation associated with P_k can be represented as a product of $(k/2 - (k/2) \bmod 1)$ disjoint cycles; each cycle comprising only two components with easily computed indices. This latter property, combined with the above decomposition of Q, leads to an algorithm which avoids the following features present in Algorithm 284:

(i) computation of the greatest common denominator;
(ii) conditional calculation of array element index in inner loop;
(iii) extra storage and variable assignment.

Geometrically, the matrix Q can be interpreted as a rotation matrix, and the matrices P_{m+n} , R, and S can be interpreted as reflection matrices.

The new algorithm is given below.

```
procedure rotatecirclist (a, m, n);
  value m, n;  integer m, n;  array a;
  comment This procedure transfers the contents of a[1] . . . a[m] into a[n + 1] . . . a[n + m]
    while simultaneously transferring the contents of a[m + 1] . . . a[m + n] into a[1] . . . a[n].
      The nonlocal procedure swap (x, y) interchanges the values of the variables x and y.
      Fewer steps occur if the result of integer division is truncated rather than rounded, but
    the procedure also works in the latter case;
  begin
    if m ≠ 0 ∧ n ≠ 0 then
    begin
      integer i, k, l;
      k := m + 1;  l := m ÷ 2;
      for i := 1 step 1 until l do swap (a[i], a[k − i]);
      k := k + n;  l := n ÷ 2;
      for i = 1 step 1 until l do swap (a[m + i], a[k − i])
      l := (m + n) ÷ 2;
      for i := 1 step 1 until l do swap (a[i], a[k − i])
    end;
  end rotatecirclist;
```

REFERENCES

1. Knuth, D.E. *The Art of Computer Programming, Vol. 1.* Addison-Wesley, Reading, Mass., 1969.

ALGORITHM 285
THE MUTUAL PRIMAL—DUAL METHOD [H]

Thomas J. Aird (Recd. 29 June 1964 and 5 Apr. 1965)
Wolf Research and Development Corporation
Manned Spacecraft Center
Houston, Texas

```
procedure Linearprogram (n, p, A, min, psol, dsol, bool);
  value p, n; integer p, n; array A, psol, dsol; real min;
  Boolean bool;
```

comment This procedure solves the linear programming problem by the Mutual Primal-Dual Simplex Method. The problem is assumed to be in the following form:

$$AX + B \le 0$$

$$X \ge 0$$

$$\min u = d + C^T X$$

where A is $p \times n$, B is $p \times 1$ and C is $n \times 1$. The dual problem is then,

$$Y \ge 0$$

$$A^T Y + C \ge 0$$

$$\max v = d + B^T Y.$$

The matrix of coefficients, also called A is formed in the following way:

$$A = \begin{bmatrix} d & C_1 & C_2 & \cdots & C_n \\ b_1 & A_{11} & A_{12} & \cdots & A_{1n} \\ b_2 & A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & & & & \\ b_p & A_{p1} & A_{p2} & \cdots & A_{pn} \end{bmatrix}$$

The input matrix A is declared [0: p, 0: n], *min* is the value of the objective function, *psol* is the solution vector for the primal problem, *dsol* is the solution vector for the dual problem, *bool* will be set to **true** if an optimal solution is found, otherwise *bool* will be set to **false**;

```
begin integer array row [0:2×p,0:p], col [0:2×p,0:n], norow,
  nocol [0:2×p], index [0:n+p];
  integer i, j, k, s, t;
  procedure subschema (k); integer k;
  comment This procedure defines an admissible sequence of
    subschema $S_{k+1}$ $S_{k+2}$ , ··· , assuming that $S_1$ , $S_2$ , ··· $S_k$ ,
    have already been defined;
  begin integer count;
    for i := 1 step 1 until p do if A[i,0] > 0 then go to
      WORK;
    for j := 1 step 1 until n do if A[0,j] < 0 then go to
      WORK; k := 0; go to RETURN;
WORK: if 2 × (k÷2) = k then go to EVEN else go to ODD;
EVEN:
  begin
    if k = 0 then
    begin
      for i := 1 step 1 until p do if A[i,0] > 0 then
      begin
        row[1,0] := i; go to D3
      end;
      row[1,0] := 0; go to D3
    end;
    for j := 1 step 1 until nocol[k] do
      if A[row[k,0],col[k,j]] = 0 then go to D1;
    go to RETURN;
D1: for i := 1 step 1 until norow[k] do
      if A[row[k,i],col[k,0]] > 0 then go to D2;
    go to RETURN;
D2: row[k+1,0] := row[k,i];
    col[k+1,0] := col[k,0];
    count := 0;
    for j := 1 step 1 until nocol[k] do
    if A[row[k,0],col[k,j]] = 0 then
    begin
      count := count + 1;
      col[k+1,count] := col[k,j]
    end;
    nocol[k+1] := count;
D3: count := 0;
    for i := 1 step 1 until norow[k] do
    if A[row[k,i],col[k,0]] ≤ 0 then
    begin
      count := count + 1;
      row[k+1,count] := row[k,i]
    end;
    norow[k+1] := count;
    k := k + 1;
    go to ODD
  end EVEN;
ODD:
  begin
    for i := 1 step 1 until norow[k] do
      if A[row[k,i],col[k,0]] = 0 then go to B1;
    go to RETURN;
B1: for j := 1 step 1 until nocol[k] do
      if A[row[k,0],col[k,j]] < 0 then go to B2;
    go to RETURN;
B2: col[k+1,0] := col[k,j];
    row[k+1,0] := row[k,0];
    count := 0;
    for i := 1 step 1 until norow[k] do
    if A[row[k,i],col[k,0]] = 0 then
    begin
      count := count + 1;
      row[k+1,count] := row[k,i]
    end;
    norow[k+1] := count;
    count := 0;
    for j := 1 step 1 until nocol[k] do
    if A[row[k,0],col[k,j]] ≥ 0 then
    begin
      count := count + 1;
      col[k+1,count] := col[k,j]
    end;
    nocol[k+1] := count;
    k := k + 1;
    go to EVEN
  end ODD;
RETURN:
  end subschema;
```

```
   procedure pivot (s,t);  value s, t;  integer s, t;
   comment  The procedure pivot performs the usual pivot opera-
     tion on the matrix A, A[s,t] is the pivot element;
   begin integer i, j;
     A[s,t] := 1/A[s,t];
     for i := 0 step 1 until s − 1, s + 1 step 1 until p do
     begin
       A[i,t] := −A[i,t] × A[s,t];
       for j := 0 step 1 until t − 1, t + 1 step 1 until n do
         if abs(A[i,j]+A[i,t]×A[s,j])  ≤  abs(A[i,j]×₁₀−8)  then
         A[i,j] := 0
         else A[i,j] := A[i,j] + A[i,t] × A[s,j]
     end;
     for j := 0 step 1 until t − 1, t + 1 step 1 until n do
       A[s,j] := A[s,j] × A[s,t];
     i := index[t];
     index[t] := index[n+s];
     index[n+s] := i
  end pivot;
  procedure pickapivot (k,s,t);  integer k, s, t;
  comment  The procedure pickapivot will choose a pivot ele-
    ment from S_k or S_k−1 in a manner which will guarantee im-
    provement in the goal vector;
  begin real max, test;
    if 2 × (k÷2) = k then go to EVEN else go to ODD;
ODD:
    begin
      for j := 1 step 1 until nocol[k] do
      if A[row[k,0],col[k,j]] < 0 then
      begin
        for i := 1 step 1 until norow[k] do
          if A[row[k,i],col[k,j]] > 0 then go to A1;
        s := row[k,0];
        t := col[k,j];
        k := k − 1;
        go to RETURN;
A1:
      end;
      for j := 1 step 1 until nocol[k] do
      if A[row[k,0],col[k,j]] < 0 then
      begin
        for i := 1 step 1 until norow[k] do
        if A[row[k,i],col[k,j]] > 0 then
        begin s := row[k,i];
          t := col[k,j];
          max := A[row[k,i],col[k,0]]/A[row[k,i],col[k,j]];
          go to A2
        end
      end;
     go to A3;
A2:  for i := i + 1 step 1 until norow[k] do
     if A[row[k,i],col[k,j]] > 0 then
      begin
        test := A[row[k,i],col[k,0]]/A[row[k,i],col[k,j]];
        if test > max then
        begin
          s := row[k,i];
          max := test
        end
      end;
     k := k − 1;
     go to RETURN;
A3:  for j := 1 step 1 until nocol[k−1] do
     if A[row[k,0],col[k−1,j]] < 0 then
      begin
        s := row[k,0];
        t := col[k−1,j];
        max  :=  A[row[k−1,0],col[k−1,j]]/A[row[k,0],col[k−1,j]];
          go to A4
        end;
        s := row[k,0];
        t := col[k,0];
        k := k − 2;
        go to RETURN;
A4:     for j := j + 1 step 1 until nocol[k−1] do
        if A[row[k,0],col[k−1,j]] < 0 then
        begin
          test  :=  A[row[k−1,0],col[k−1,j]]/A[row[k,0],col[k−1,j]];
          if test > max then
          begin
            t := col[k−1,j];
            max := test
          end
        end;
        k := k − 2;
        go to RETURN
      end ODD;
EVEN:
      begin
        for i := 1 step 1 until norow[k] do
        if A[row[k,i],col[k,0]] > 0 then
        begin
          for j := 1 step 1 until nocol[k] do
          if A[row[k,i],col[k,j]] < 0 then
          go to B1;
          s := row[k,i];
          t := col[k,0];
          k := k − 1;
          go to RETURN;
B1:
        end;
        for i := 1 step 1 until norow[k] do
        if A[row[k,i],col[k,0]] > 0 then
        begin
          for j := 1 step 1 until nocol[k] do
          if A[row[k,i],col[k,j]] < 0 then
          begin
            s := row[k,i];
            t := col[k,j];
            max :=  A[row[k,0],col[k,j]]/A[row[k,i],col[k,j]];
            go to B2
          end
        end;
       go to B3;
B2:    for j := j + 1 step 1 until nocol[k] do
        if A[row[k,i],col[k,j]] < 0 then
        begin
          test := A[row[k,0],col[k,j]]/A[row[k,i],col[k,j]];
          if test > max then
          begin
            t := col[k,j];
            max := test
          end
        end;
        k := k − 1;
       go to RETURN;
B3:    for i := 1 step 1 until norow[k−1] do
       if A[row[k−1,i],col[k,0]] > then
        begin
          s := row[k−1,i];
          t := col[k,0];
          max := A[row[k−1,i],col[k−1,0]]/A[row[k−1,i],col[k,0]];
          go to B4
        end;
      s := row[k,0];
      t := col[k,0];
      k := k - 2;
      go to RETURN;
B4:   for i := i + 1 step 1 until norow[k-1] do
      if A[row[k-1,i],col[k,0]] > then
       begin
        test := A[row[k-1,i],col[k-1,0]]/A[row[k-1,i],col[k,0]];
        if test > max then
        begin
          s := row[k-1,i];
          max := test
        end
      end;
      k := k - 2;
      go to RETURN
    end EVEN;
RETURN:
  end pickapivot;
  for i := 1 step 1 until p + n do index[i] := i;
  for i := 0 step 1 until p do row[0,i] := i;
  for j := 0 step 1 until n do col[1,j] := j;
  norow[0] := p;  nocol[1] := n;  k := 0;
  comment  This is a check on the row constraints;
NEXTPIVOT:
  for i := 1 step 1 until p do
  begin
    if A[i,0] ≤ 0 then go to NEXTI;
    for j := 1 step 1 until n do
      if A[i,j] < 0 then go to NEXTI;
    comment  Row constraints are incompatible;
    bool := false;
    go to FINISH;
NEXTI:
  end;
  comment  This is a check on the column constraints;
  for j := 1 step 1 until n do
  begin
    if A[0,j] ≥ 0 then go to NEXTJ;
    for i := 1 step 1 until p do
      if A[i,j] > 0 then go to NEXTJ;
    comment  Column constraints are incompatible;
    bool := false;
    go to FINISH;
NEXTJ:
  end;
  subschema(k);
  if k = 0 then
  begin
    comment  k = 0 indicates that the present solution is opti-
      mal. A[0,0] is value of the objective function;
    min := A[0,0];
    for i := 1 step 1 until p + n do psol[i] := dsol[i] := 0;
    comment  Find the primal solution vector;
    for i := 1 step 1 until p do
      psol[index[n+i]] := -A[i,0];
    comment  Find the dual solution vector;
    for i := 1 step 1 until n do
      if index[i] > n then
      dsol[index[i]-n] := A[0,i]
      else
      dsol[index[i]+p] := A[0,i];
    bool := true;
    go to FINISH;
  end;
  pickapivot(k,s,t);
  if s = 0 ∨ t = 0 then
  begin
    comment  No feasible solution;
    bool := false;
    go to FINISH;
  end;
  pivot(s,t);
  go to NEXTPIVOT;
FINISH:
end Linearprogram
```

CERTIFICATION OF ALGORITHM 285 [H] THE MUTUAL PRIMAL-DUAL METHOD

[Thomas J. Aird, *Comm. ACM 9* (May 1966), 326]

H. Späth (Recd. 13 Feb. 1967)
Institut für Neutronenphysik und Reaktortechnik,
Kernforschungszentrum, Karlsruhe, Germany

The procedure *Linearprogram* has been translated into Fortran II and successfully run on the IBM 7074 Computer. The following corrections had been made (the first two are merely typographical errors).

1. P. 328, left column, 1 line after label *B*3:
reads:
    if *A*[*row*[*k*−1, *i*],*col*[*k*, 0]] > **then**
should read:
    if *A*[*row*[*k*−1, *i*],*col*[*k*, 0]] > 0 **then**

2. P. 328, left column, 1 line after label *B*4:
reads:
    if *A*[*row*[*k*−1, *i*],*col*[*k*, 0]] > **then**
should read:
    if *A*[*row*[*k*−1, *i*],*col*[*k*, 0]] > 0 **then**

3. P. 328, right column, after the end of the procedure *pickapivot* and before the label *NEXTPIVOT* there must be inserted the statement
col[0, 0] := 0;

Otherwise *col*[0, 0] has no assigned value when the procedure *subschema* is entered for the first time.

ALGORITHM 286
EXAMINATION SCHEDULING [ZH]

J. E. L. PECK AND M. R. WILLIAMS (Recd. 17 Mar. 1964, 25 Jan. 1965 and 1 Mar. 1966)
University of Alberta, Calgary, Alta., Canada

```
procedure partition (incidence) graph of order : (m) into : (n)
  parts using weights : (w) bound : (max) preassignment :
  (preassign) of number : (pren);
  Boolean array incidence; integer array w, preassign;
  integer m, n, max, pren;
comment This is an heuristic examination time-tabling procedure
  for scheduling m courses in n time periods. It is essentially
  the problem of graph partitioning and map coloring.
    In the terminology of graph theory: Given a graph of m vertexes
    with a positive integer weight w[i] at the ith vertex,
    partition this graph into no more than n disjoint sets such
    that each set contains no two vertexes joined by an edge,
    and such that the total weight of each set is less than the
    prescribed bound max.
  We represent the graph as an m×m symmetric Boolean matrix
  incidence whose i,jth element is true if and only if vertex i is
  joined to vertex j by an edge (if a student is taking both course i
  and course j), diagonal elements being assigned the value true.
  The weight assigned to the ith vertex (number of students in the
  ith course) is w[i]. We shall see below that preassignment is
  permitted. The number of courses to be preassigned is given in
  pren and the course preassign [i, 1] is to be placed at the time
  preassign [i, 2].
    This procedure does not minimize the second order incidence
  i.e. a vertex i being assigned to the set k, where the set k-1
  contains a vertex j joined to i (a student writing two consecutive
  examinations), but this may be done by rearranging the sets
  after the partitioning is completed. The procedure contains its
  own output statements, but its driver should provide the input;
begin integer array row [1:m], number [1:n];
  integer i, j, sum, course, time;
  Boolean preset, completed;
INITIALIZE:  preset:= false;
  for j := 1 step 1 until n do number [j] := 0;
  for i := 1 step 1 until m do
  begin sum := 0;
    for j := 1 step 1 until m do
    if incidence [i, j] then sum := sum + 1;
    row [i] := sum
  end INITIALIZE. Note that row [i] now contains the multiplicity
    of, or number of edges at the vertex i (number
    of courses which conflict with the course i). Of course since the
    incidence matrix is symmetric, less than half (i > j) need be
    stored. However, this procedure, for the sake of simplicity,
    is written for the whole matrix. Also note that row [i] will
    eventually contain the negative of the set number to which
    the ith vertex is assigned (examination time for the ith course)
    and number [j] will contain the weight of the jth set (number of
    candidates at time j). From here on we drop the allusions to
    graph theory in the comments;
THE PREASSIGNMENT:  for j := 1 step 1 until pren do
  begin comment preassignment of courses to times is now carried
      out. If pren = 0, then there are no preassignments;
    course := preassign [j,1]; time:= preassign [j,2]:
    comment We now attempt to assign this course to the given
      time;
SCRUTINIZE: if row [course] < 0 then
    begin outstring (1, 'This course'); outinteger (1, course);
      outstring (1, 'is already scheduled at time');
      outinteger (1, -row[course]);  go to NEXT
    end;
    if number [time] + w[course] > max then
    begin outstring (1, 'Space is not available for course');
      outinteger (1, course); outstring (1, 'at time');
      outinteger (1, time);  go to NEXT
    end;
    for i := 1 step 1 until m do
      if row [i] = - time then
      begin if incidence [i, course] then
        begin outstring (1, 'course number');
          outinteger (1, course);  outstring (1, 'conflicts with');
          outinteger (1,i);
          outsiring (1, 'which is already scheduled at');
          outinteger (1, time),
          go to NEXT
        end if incidence
      end if row;
SATISFACTORY: row[course] := -time;
    number [time] := number [time] + w [course];
    preset := true;
NEXT:
  end THE PREASSIGNMENT;
MAIN PROGRAM: begin Boolean array available [1:m];
    integer next;
    procedure check (course);  integer course;
    begin integer j;  comment This procedure renders unavailable
        those courses conflicting with the given course;
      for j := 1 step 1 until m do
        if incidence [course,j] then available [j] := false
    end of procedure check.
      For each of the n time periods we select a suitable set of nonconflicting
      courses whose students will fit the examination
      room;
START OF MAIN PROGRAM:
    for time := 1 step 1 until n do
      if preset ≡ number[time] > 0 then
      begin comment The preceding Boolean equivalence directs
          the attention of the program initially only to
          those times where prescheduling has occurred. We now
          determine the available courses (i.e. unscheduled and
          nonconflicting). If course i is already scheduled, then
          row[i] is negative;
        completed := true;
        for i := 1 step 1 until m do if row [i] > 0 then
        begin available [i] := true; completed := false end
          else available [i] := false;
        if completed then go to OUTPUT;
        if preset then
        begin comment Some courses were prescheduled at
            this time. It is necessary to render their conflicts unavailable;
          for i := 1 step 1 until m do
            if row[i] = -time then check (i)
        end prescheduled courses.
```

```
          We now select the available course with the most con-
          flicts. This is essentially the heuristic step and there-
          fore the place where variations on the method may be
          made;
AGAIN:
         sum := 0;
        for i := 1 step 1 until m do
          if available [i] ∧ row [i] > sum then
          begin next := i;  sum := row [i] end most conflicts;
        if sum > 0 then
        begin comment There exists an available course, so
              we test it (viz next) for size. If it does not fit we look
              for another;
          available [next] := false;
          if number [time] + w[next] > max then go to AGAIN;
          comment  If we are here the course will fit so we use it;
          row [next] := −time;
          number [time] := number [time] + w[next];
          check (next); go to AGAIN
        end sum > 0
    end of the time loop;
    if preset then
      begin preset := false; go to START OF MAIN
        PROGRAM end
        In case of prescheduling this takes us back to try the re-
        maining time periods.
          If we have reached here with completed true then all
        courses are scheduled, but the converse may not be true,
        therefore;
    if ¬ completed then
    begin completed := true;
      for i := 1 step 1 until m do
        if row [i] > 0 then completed := false
    end ¬ completed and
  end of the main program;
OUTPUT: if ¬ completed then
  begin comment The following for statement outputs the
        courses that were not scheduled;
    outstring (1, 'courses not scheduled');
    for i := 1 step 1 until m do
      if row [i] > 0 then outinteger (1,i)
  end not scheduled.
    The following outputs the time period j, the number of stu-
    dents number[j] and the courses i written at time j;
TIMETABLE: outstring(1, 'time  enrolment  courses');
  for j := 1 step 1 until n do
  begin outinteger (1,j); outinteger (1, number[j]);
    for i := 1 step 1 until m do
      if row[i] = −j then outinteger (1,i)
  end j.
    The following outputs the courses, the times at which they are
    written, and their enrolment;
  outstring (1, 'course  time  enrolment');
  for i := 1 step 1 until m do
    if row [i] < 0 then outinteger (1, i); outinteger (1, row [i]);
      outinteger (1, w[i])
    else
    begin outinteger(1,i); outstring(1, 'unscheduled');
      outinteger (1, w[i])
    end
end of the procedure
```

REMARK ON ALGORITHM 286 [H] EXAMINATION SCHEDULING [J. E. L. Peck and M. R. Williams, *Comm. ACM 9* (June 1966), 433].

The 6th and 7th lines from the end of the procedure should be corrected by the insertion of a **begin end** pair so that they read

```
if row [i] < 0 then
  begin outinteger (1, i); outinteger (1, row [i]); outinteger
    (1, w[i])
  end
```

ALGORITHM 287
MATRIX TRIANGULATION WITH INTEGER ARITHMETIC [F1]

W. A. Blankinship

(Recd. 19 May 1965 and 17 Sept. 1965)

National Security Agency, Ft. Geo. G. Meade, Md.

```
integer procedure INTRANK (mat, m, n, e); value m, n, e;
  integer m, n, e; integer array mat;
```

comment This procedure operates on an m by $n+e$ matrix whose name is *mat* and whose elements are integers. If *mat* is considered as composed of two submatrices U and V, where U comprises the first n columns of *mat* and V comprises the last e columns, then the effect of the procedure is as follows:

(1) The rank of the submatrix U is returned as the value of *INTRANK* (designated by r in the following discussion).

(2) *mat* is transformed by a sequence of elementary row operations in such a manner that U is reduced to triangular form. Triangular form means that the leading, or first nonzero, element of each row appears to the right of the leading element of the preceding row.

(3) It is easy to deduce from the proof in [1, p. 72, Th. 12] that for any set of k columns of *mat*, the greatest common divisor of all kth order minors selected from those columns is preserved. In particular, the product of all leading elements in U (final) (which are preserved as the first r elements of the local array a) will be equal to the gcd of all nth order minors of U.

(4) It is also easy to show, by the methods of [2] that if *mat* contains an $m \times m$ identity matrix, I, then I ends up as a record of the row operations actually performed, specificallv:

$$mat\ (\text{final}) = I\ (\text{final}) \times mat\ (\text{initial})$$

(5) Since (3) implies that the rank of U is preserved, and the rank of U (final) is obviously equal to the number of nonzero rows that it contains, this number, r, is returned as the value of *INTRANK*.

(6) Under the conditions of (4), it follows that the last $m-r$ rows of I (final) comprise a complete, linearly independent set of left-annihilators (row-dependences) of the matrix U.

The preceding properties are the basis of the claims for the procedure *SOLVEINTEGER* [Algorithm 288, *Comm. ACM 9* (July 1966), 514] which calls this procedure.

INTRANK is designed to minimize the likelihood of overflow, the detection of which is left to the user. The best method is to include an identity matrix in *mat* and check the relation described in 4 (above). In many instances overflow doesn't matter. In particular, if (a) the machine-compiler combination does integer addition, subtraction and multiplication modulo $2i+1$ where i is the maximum integer representable in the machine, (b) division is done by the usual long-division algorithm, and (c) the answers sought are either known to be less than i in absolute value, or only desired modulo $2i+1$, then, short of interference by an over-zealous monitor, the procedure will produce satisfactory results in spite of overflow. (Although the CDC 1604 does not satisfy (a), the same effect can be achieved by using a suitable subroutine in place of the multiplication sign in the procedure *REDUCE*.)

Overflow is generally dependent upon the magnitude of the greatest common divisor of all $r \times r$ minors contained in U, as this number, or a large divisor of it will appear in the rth row of *mat* (final) and as $a[r]$. Thus if U is a square matrix whose determinant is a prime greater than the capacity of the machine, there is obviously no way to avoid overflow. Even if the determinant is composite, it is most likely that only small factors will be left on the diagonal and overflow will still occur. When elements of U are chosen from a flat-random population of integers in the closed interval $[-13, +13]$ it has been found empirically that overflow almost never occurs for $m=n=11$ when run on the CDC 1604 where $i = 2^{46}-1$. See also the discussions on overflow in the procedure *SOLVEINTEGER*;

```
begin integer i, j, k, Q, T, topel, nextel, itop, inext;
  integer array a [1:m];
  procedure FINDNEXT;
  begin nextel := 0;
    for k := i step 1 until m do
      if a[k]>nextel∧k ≠ itop then
      begin nextel := a[k] ; inext := k
      end
  end;
  procedure SWAPROWS;
  begin for k := j step 1 until T do
    begin Q := − mat[i,k];
      mat [i,k] := mat [itop, k];
      mat [itop, k] := Q
    end;
    a[i] := a [itop];
    comment The last statement is a luxury which ensures that,
      at the end of the algorithm, a will contain the leading ele-
      ments of the first INTRANK rows of mat;
  end;
  procedure REDUCE;
  begin Q := mat [itop,j] ÷ mat [inext, j];
    for k := j step 1 until T do
      mat [itop,k] := mat [itop,k] −Q × mat [inext,k];
    a [itop] := if mat [itop,j] < 0 then − mat [itop,j] else
      mat [itop,j];
  end;
  i := j:= itop := 0; T := n+e;
NEXTROW: if itop ≠ i then SWAPROWS;
  i := i+1; if i > m then go to OUT;
NEXTCOL: j := j+1; if j > n then go to OUT;
  for k := i step 1 until m do
    a[k] := if mat [k,j] < 0 then − mat [k,j] else mat [k,j];
  comment Find the value and location of the largest element at
    or below position (i,j) of mat.;
  itop := i−1; FINDNEXT;
  if nextel = 0 then go to NEXTCOL;
CONTINUE: itop := inext; topel := nextel;
  comment Find the value and location of the next largest
    element at or below position (i,j);
  FINDNEXT;
  if nextel = 0 then go to NEXTROW;
  comment Subtract row containing next highest element from
    that containing highest element. Repeat until highest element
    no longer ranks highest;
  REDUCE;
  go to CONTINUE;
OUT:  INTRANK := i−1;
end
```

REFERENCES:

1. ALBERT, A. A. *Fundamental Concepts of Higher Algebra.* U. of Chicago Press., Chicago, Ill., 1956.
2. BLANKINSHIP, W. A. A new version of the Euclidean algorithm. *Amer. Math. Month. 70* (1963), 742–745.

ALGORITHM 288
SOLUTION OF SIMULTANEOUS LINEAR DIOPHANTINE EQUATIONS [F4]

W. A. Blankinship

(Recd. 19 May 1965 and 17 Sept. 1965)

National Security Agency, Ft. Geo. G. Meade, Md.

```
Boolean procedure SOLVEINTEGER (A) times: (x) equals
    the vector: (b) times a least integer: (d) where A is a matrix of
    dimension one to: (m) by one to: (n) Also find: (k) linearly
    independent auxiliary solutions and store in the matrix: (Y);
  value m, n;
  integer m, n, d, k;
  integer array A, x, b, y;
comment Seeks the smallest positive integer, d, for which an
    integer solution to the equation Ax = bd exists.
```

If no solution exists then *SOLVEINTEGER* is returned as **false.** Otherwise *SOLVEINTEGER* is returned as **true** and the values of d and the solution vector x are returned.

If more than one solution exists then auxiliary solutions are returned in the matrix Y. The additional solutions are obtained by adding any linear combination of the first k rows of Y to the solution x.

It is assumed that

A is dimensioned $[1{:}m,1{:}n]$,
x is dimensioned $[1{:}n]$,
b is dimensioned $[1{:}m]$,
Y is dimensioned $[1{:}n,1{:}n]$.

Note that a diophantine solution exists if and only if d is returned as 1 and *SOLVEINTEGER* is returned as **true.**

The procedure relies entirely on the action of the procedure *INTRANK* [Algorithm 287, *Comm. ACM 9* (July 1966), 513]. In particular, a matrix, *mat*, is formed by adjoining $-b$ to the transpose of A, and then adjoining an $(n + 1)$th order identity matrix as follows:

$$mat = \begin{pmatrix} -b & \\ A^T & I \end{pmatrix}$$

INTRANK is then called upon to triangularize the first $m+1$ columns of *mat* (reaching into the first column of I). The value of *INTRANK* will be returned as an integer r which is 1 greater than the rank of A. Furthermore, as a consequence of properties (4) and (6) claimed under *INTRANK*, the last $n-r+1$ rows of I (final) will comprise a complete set of left annihilators of the matrix $\begin{pmatrix} -b \\ A^T \end{pmatrix}$. Since only the first of these rows (if any) will have a nonzero element in the first column, it follows that this first row expresses the value d and the desired solution (if $d \neq 0$), and the succeeding $n-r$ rows constitute solutions to the homogeneous equation. If any linear combination of these last $n-r+1$ rows were to yield a vector whose elements have a greatest common divisor not equal to 1, this would imply that *det* (I (final)) = *det* (I (initial)) $\neq 1$, which is false. This ensures that d is the smallest value, as claimed.

Overflow cannot occur in this procedure except as inherited from the procedure *INTRANK*. Overflow seems to be no problem when solutions (x,d) exist which are within the machine's capacity to verify. I am unable to fully explain this but numerous cases have been run on the CDC-1604 (47-bit integers plus sign bit) with elements of A chosen randomly between -13 and $+13$ inclusive and for $m=n=5$ through 20 (10 or more cases each). Only a single failure (in the case $m=n=20$) occurred. These cases were devised by preassigning integer values to x, calculating b and then calling *SOLVEINTEGER*. It is difficult to devise significant test cases where $det(A) \neq d \gg 1$ as this involves assigning values of x satisfying $Ax=0 \pmod d$). This implies d must be a divisor of *det* (A) which must therefore be precalculated. But *det* (A) may overflow even though there may be a d for which solution is possible. When $m=n$ the values of x and d will usually be, according to Cramer's rule, nth order determinants, or high divisors thereof, which may exceed machine capacity. When the elements of both b and A are chosen equiprobably between $-\alpha$ and $+\alpha$, inclusive, it can be shown that the standard deviation of such a determinant is $(n!\alpha^n (\alpha+1)^n/3)^{\frac{1}{2}}$. Since this is an upper bound for the expected absolute value of such a determinant, it may be used as a rule of thumb to predict overflow. If $\alpha=13$, then for $n=11$ this value is $10^{13.6}$ and for $n=12$ it is $10^{15.0}$. 1604 capacity is $10^{14.1}$. In test cases, the procedure invariably succeeded for $n=11$ and invariably failed for $n=12$. (Remember, we are referring to cases where b is chosen randomly so that an integer solution will hardly ever exist.)

Note that if $m=1$, this algorithm solves the gcd problem in much the same way as Algorithm 237 [J. E. L. Peck, *Comm. ACM 8* (Aug. 1964), 481];

```
begin integer i, j, rank, s;
  integer array mat [1 : n+1, 1 : m+n+1];
  for j := 1 step 1 until m do
  begin mat [1,j] := -b [j];
    for i := 1 step 1 until n do mat [i+1, j] := A [j,i]
  end;
  for j := 1 step 1 until n+1 do
  for i := 1 step 1 until n+1 do
    mat [i, j+m] := if i = j then 1 else 0;
  rank := INTRANK (mat, n+1, m+1, n);
  d := mat [rank, m+1];
  if d = 0 then begin SOLVEINTEGER := false;  go to OUT
                end;
  for i := rank step 1 until m do
    if mat [rank, i] ≠ 0 then
    begin SOLVEINTEGER := false;  go to OUT
    end;
SOLVEINTEGER := true;
  s := if d < 0 then -1 else 1;  d := s × d;
  k := n - rank + 1;
  for i := 1 step 1 until n do
  begin x[i] := mat [rank, m+i+1] × s;
    for j := 1 step 1 until k do
      Y [j,i] := mat [rank+j, m+i+1]
  end;
OUT:
end of procedure SOLVEINTEGER
```

ALGORITHM 289
CONFIDENCE INTERVAL FOR A RATIO [G1]

I. D. Hill and M. C. Pike (Recd. 8 Oct. 1965)
Statistical Research Unit, Medical Research Council,
London, England

```
procedure Fieller (y, x, Vyy, Vxy, Vxx, t, r1, r2, inclusive);
  value y, x, Vyy, Vxy, Vxx, t;
  real y, x, Vyy, Vxy, Vxx, t, r1, r2;
  Boolean inclusive;
comment This procedure finds the (1−2×a) confidence limits
  for θ/φ where y and x are estimates of θ and φ respectively,
  subject to random errors 'normally' distributed with zero means,
  variance estimates Vyy and Vxx, and covariance estimate Vxy,
  each based on f degrees of freedom, and t is the upper (100×a)
  percent point of the t distribution on f degrees of freedom.
    At exit, if inclusive is true then the confidence interval in-
  cludes all values such that r1 ≦ value ≦ r2. Otherwise the
  confidence interval includes all values such that − infinity ≦
  value ≦ r2 and additionally all values such that r1 ≦ value ≦
  infinity.
    Where the interval is such that the value of r1 or r2 should be
  ± infinity, the procedure sets the value to ± the largest available
  real number.
    Reference: E. C. Fieller, A fundamental formula in the
  statistics of biological assay, and some applications, Quart. J.
  Pharm. Pharmacol. 17 (1944), 117–123;
begin
  real c, r, infinity;
  inclusive := true; infinity := ⏨114;
  comment Set infinity to largest available positive real number;
  c := t ↑ 2; r := x ↑ 2 − c × Vxx;
  r1 := x × y − c × Vxy; c := y ↑ 2 − c × Vyy;
  if r ≠ 0 then
  begin
    c := r1 ↑ 2 − r × c;
    if r > 0 ∧ c < 0 then c := 0;
    if c < 0 then go to unbounded;
    inclusive := r > 0; r := 1.0/r; c := sqrt (c);
    r2 := (r1+c) × r; r1 := (r1−c) × r
  end else
  begin
    if r1 ≠ 0 then
    begin
      c := c/(2.0×r1);
      if r1 > 0 then
      begin
        r1 := c; r2 := infinity
      end else
      begin
        r1 := −infinity; r2 := c
      end
    end else
    begin
unbounded: r1 := −infinity; r2 := infinity
    end
  end
end Fieller
```

ALGORITHM 290
LINEAR EQUATIONS, EXACT SOLUTIONS [F4]

J. Boothroyd* (Recd. 7 Sept. 1965 and 21 Mar. 1966)
U. of Tasmania, Hobart, Tas., Australia

* Thanks are due to the referee for useful criticism and awkward test cases.

```
procedure exactle(a, b, n, det);  value n;  integer n, det;
   integer array a, b;
comment  solves the matrix equation Ax = b for A = a [1:n,
   1:n] and x, b[1:n] where the elements of A, b are small integers
   and the results are required as ratios of integers. The solution
   vector overwrites b and has values given by det A × x where det
   A is the determinant of A and x is the true solution vector. The
   user is warned that this procedure, of limited though useful
   application, is not a substitute for other well-established
   methods of solving general sets of linear equations owing to the
   inherent danger of integer overflow. This may occur in the
   reduction if the elements of the matrix are large or in the back
   substitution if the determinant and/or the elements of the right-
   hand side are large and may even occur with small elements and
   determinant if the order of the matrix and the nature of the
   equations combine to produce large solution values. Four
   devices intended to avoid integer overflow are incorporated.
   These are, (1) choice of column pivots having the smallest non-
   zero absolute value, (2) division by previous pivots (both after
   Fox, L., An Introduction to Numerical Linear Algebra, Oxford
   U. Press, New York, 1965, p. 82), and (3) the local procedures
   crossmpy and abdivc which respectively evaluate integer expres-
   sions of the form (a×b−c×d) ÷ e and a × b ÷ c by performing
   the divisions before the multiplications. The output parameter
   det yields the determinant of A. If A is singular det := 0;
begin integer piv, pivot, sum, arii, aki, i, j, k, pivi, ri, rk, m;
   integer array r [1:n]; boolean zpiv;
   integer procedure iabs (it); value it; integer it;
     iabs := if it < 0 then − it else it;
   integer procedure crossmpy(a)times:(b)minus:(c)times:(d)all
     over:(e);
     value a,b,c,d,e;  integer a,b,c,d,e;
   begin integer qab,qcd,r,res;
     if iabs(a) > iabs(b) then
     begin
       qab := a ÷ e;  r := a − qab × e;
       qab := qab × b;  res := r × b
     end
     else
     begin
       qab := b ÷ e;  r := b − qab × e;
       qab := qab × a; res := r × a
     end;
     if iabs(c) > iabs(d) then
     begin
       qcd := c ÷ e;  r := c − qcd × e;
       qcd := qcd × d;  res := res − r × d
     end
     else
     begin
       qcd := d ÷ e;  r := d − qcd × e;
       qcd := qcd × c;  res := res − r × c
     end;
     crossmpy := qab − qcd + res ÷ e
   end crossmpy;
   integer procedure abdivc(a,b,c,sum);  value a,b,c;  integer
     a,b,c,sum;
   comment  evaluates expressions of the form a × b ÷ c by
     performing divisions before multiplications, assigning the
     quotient to abdivc and accumulating the remainder in sum;
   begin integer q,r,temp;
     if iabs(a) > iabs(b) then
     begin q := a ÷ c;  temp := q × b;
       r := a − c × q;
       q := b ÷ c;
       abdivc := temp + q × r;
       sum := sum + (b−q×c) × r
     end
     else
     begin q := b ÷ c;  temp := q × a;
       r := b − c × q;
       q := a ÷ c;
       abdivc := temp + q × r;
       sum := sum + (a−q×c) × r
     end
   end abdivc;
   procedure permb(b,r,n);  value n;  integer array b,r;  inte-
     ger n;
   comment rearranges the elements of b[1:n] so that b[i] :=
     b[r[i]], i = 1, 2, ···, n;
   begin integer i,k,w;
     for i := n step −1 until 2 do
     begin k := r[i];
L:
       if k ≠ i then
       begin
         if k > i then begin k := r [k];  go to L end;
         w := b[i];  b[i] := b[k];  b[k] := w
       end
     end
   end permb;
   m := 1;
   for i := 1 step 1 until n do r[i] := i;
   for i := 1 step 1 until n do
   begin pivot := 0;  zpiv := true;
     for k := i step 1 until n do
     begin aki := iabs(a[r[k],i]);
       if zpiv ∧ aki > 0 ∨ aki ≠ 0 ∧ aki < iabs(pivot) then
       begin zpiv := false; pivi := k;  pivot := a[r[k],i] end
     end;
     if pivot = 0 then begin det := 0;  go to out end;
     ri := r[pivi];  r[pivi] := r[i];  r[i] := ri;  if pivi ≠ i then
         m := − m;
     for k := i + 1 step 1 until n do
     begin rk := r[k];  aki := a[rk,i];
       for j := i + 1 step 1 until n do
         a[rk,j] := if i = 1 then a[rk,j] × pivot − aki × a[ri,j]
               else crossmpy(a[rk,j],pivot,aki,a[ri,j],piv);
       b[rk] := if i = 1 then b[rk] × pivot − aki × b[ri]
                          else crossmpy(b[rk],pivot,aki,b[ri],piv)
     end;
     piv := pivot
   end;
```

```
 ri := r[n];
 if m ≠ 1 then
 begin det := aki := − a[ri,n];  b[ri] := − b[ri] end
 else det := aki := a[ri,n];
  for i := n − 1 step −1 until 1 do
  begin ri := r[i];  arii := a[ri,i];
     sum := 0;  piv := abdivc(b[ri],aki,arii,sum);
     sum := − sum;
     for j := i + 1 step 1 until n do
        piv := piv − abdivc(b[r[j]],a[ri,j],arii,sum);
     b[ri] := piv − sum ÷ arii
 end;
 permb(b,r,n);
out:
end exactle
```

ALGORITHM 291
LOGARITHM OF GAMMA FUNCTION [S14]

M. C. Pike and I. D. Hill (Recd. 8 Oct. 1965 and 12 Jan. 1966)
Medical Research Council's Statistical Research Unit, University College Hospital Medical School, London, England

```
real procedure loggamma (x);
  value x;  real x;
comment This procedure evaluates the natural logarithm of
  gamma(x) for all x > 0, accurate to 10 decimal places. Stirling's
  formula is used for the central polynomial part of the procedure.;
begin
  real f, z;
  if x < 7.0 then
  begin f := 1.0;  z := x − 1.0;
    for z := z + 1.0 while z < 7.0 do
    begin x := z;  f := f × z
    end;
    x := x + 1.0;  f := − ln(f)
  end
  else f := 0;
  z := 1.0/x ↑ 2;
  loggamma := f + (x−0.5) × ln(x) − x + .91893 85332 04673 +
    (((−.00059 52380 95238×z+.00079 36507 93651) × z −.00277
    77777 77778)×z+.08333 33333 33333)/x
end loggamma
```

REMARKS ON:
ALGORITHM 34 [S14]
GAMMA FUNCTION

[M. F. Lipp, *Comm. ACM 4* (Feb. 1961), 106]

ALGORITHM 54 [S14]
GAMMA FUNCTION FOR RANGE 1 TO 2

[John R. Herndon, *Comm. ACM 4* (Apr. 1961), 180]

ALGORITHM 80 [S14]
RECIPROCAL GAMMA FUNCTION OF REAL ARGUMENT

[William Holsten, *Comm. ACM 5* (Mar. 1962), 166]

ALGORITHM 221 [S14]
GAMMA FUNCTION

[Walter Gautschi, *Comm. ACM 7* (Mar. 1964), 143]

ALGORITHM 291 [S14]
LOGARITHM OF GAMMA FUNCTION

[M. C. Pike and I. D. Hill, *Comm. ACM 9* (Sept. 1966), 684]

M. C. Pike and I. D. Hill (Recd. 12 Jan. 1966)
Medical Research Council's Statistical Research Unit,
University College Hospital Medical School,
London, England

Algorithms 34 and 54 both use the same Hastings approximation, accurate to about 7 decimal places. Of these two, Algorithm 54 is to be preferred on grounds of speed.

Algorithm 80 has the following errors:
(1) *RGAM* should be in the parameter list of *RGR*.
(2) The lines

```
  if x = 0 then begin RGR := 0;  go to EXIT end
```

and

```
  if x = 1 then begin RGR := 1;  go to EXIT end
```

should each be followed either by a semicolon or preferably by an **else**.
(3) The lines

```
  if x = 1 then begin RGR := 1/y;  go to EXIT end
```

and

```
  if x < − 1 then begin y := y × x;  go to CC end
```

should each be followed by a semicolon.
(4) The lines

```
  BB:  if x = −1 then begin RGR := 0;  go to EXIT end
```

and

```
  if x > −1 then begin RGR := RGAM(x);  go to EXIT end
```

should be separated either by **else** or by a semicolon and this second line needs terminating with a semicolon.
(5) The declarations of **integer** *i* and **real array** *B*[0:13] in *RGAM* are in the wrong place; they should come immediately after

```
  begin real z;
```

With these modifications (and the replacement of the array *B* in *RGAM* by the obvious nested multiplication) Algorithm 80 ran successfully on the ICT Atlas computer with the ICT Atlas ALGOL compiler and gave answers correct to 10 significant digits.

Algorithms 80, 221 and 291 all work to an accuracy of about 10 decimal places and to evaluate the gamma function it is therefore on grounds of speed that a choice should be made between them. Algorithms 80 and 221 take virtually the same amount of computing time, being twice as fast as 291 at $x = 1$, but this advantage decreases steadily with increasing x so that at $x = 7$ the speeds are about equal and then from this point on 291 is faster—taking only about a third of the time at $x = 25$ and about a tenth of the time at $x = 78$. These timings include taking the exponential of *loggamma*.

For many applications a ratio of gamma functions is required (e.g. binomial coefficients, incomplete beta function ratio) and the use of algorithm 291 allows such a ratio to be calculated for much larger arguments without overflow difficulties.

REMARK ON ALGORITHM 291 [S14]
LOGARITHM OF GAMMA FUNCTION [M.C. Pike and I. D. Hill, *Comm. ACM 9* (Sept. 1966), 684]

Miss M. R. Hoare (Recd. 24 Aug. 1967)
℅ C. Hoare and Co., 37 Fleet St., London, E.C.4.

```
(1)  if x < 7.0 then
     begin f := 1.0;  z := x − 1.0;
        for z := z + 1.0 while z < 7.0 do
```

would be better written as:

```
     if x < 7.0 then
     begin f := 1.0;
```

for $z := x, z + 1.0$ **while** $z < 7.0$ **do**

This avoids unnecessary operations.

(2) In the final statement, **loggamma** should read *loggamma*

ALGORITHM 292
REGULAR COULOMB WAVE FUNCTIONS

WALTER GAUTSCHI (Recd. 8 Oct. 1965)
Purdue University, Lafayette, Indiana and Argonne National Laboratory, Argonne, Illinois

Work performed under the auspices of the U. S. Atomic Energy Commission.

```
real procedure t(y);  value y;  real y;
comment  This procedure evaluates the inverse function t = t(y)
   of y = t ln t in the interval y ≧ −1/e, to an accuracy of about
   4 percent, or better. Except for the addition of the case
   −1/e ≦ y ≦ 0, and an error exit in case y < −1/e, the procedure
   is identical with the real procedure t of Algorithm 236;
begin real p, z;
   if y < −.36788 then go to alarm 1;
   if y ≦ 0 then t := .36788 + 1.0422 × sqrt(y + .36788) else
   if y ≦ 10 then
   begin
      p := .000057941 × y − .00176148;  p := y × p + .0208645;
      p := y × p − .129013;  p := y × p + .85777;
      t := y × p + 1.0125
   end
   else
   begin
      z := ln(y) − .775;  p := (.775−ln(z))/(1+z);
      p := 1/(1+p);  t := y × p/z
   end
end t;
procedure minimal (eta, omega, eps, la1, dm);
   value eta, omega, eps; real eta, omega, eps, la1, dm;
comment  This procedure assigns the value of λ1' to la1, accu-
   rately to within a relative error of eps, where {λL'} is the minimal
   solution (normalized by λ0'=1) of the difference equation
```

$$\lambda_{L+1} - \frac{2L+1}{L+1}\omega\lambda_L - \frac{L^2+\eta^2}{L(L+1)}\lambda_{L-1} = 0 \quad (\omega \neq 0).$$

(For terminology, see [3].) If $\{\lambda_L\}$ denotes the solution corresponding to initial values $\lambda_0 = 1, \lambda_1 = \omega - \eta$, the procedure also assigns to *dm* the value $\lambda_1 - \lambda_1'$. The negative logarithm of $|\lambda_1 - \lambda_1'|$ may be considered a measure of the "degree of minimality" of the solution $\{\lambda_L\}$;

```
begin integer L, nu;  real eta2, r, ra;
   eta2 := eta ↑ 2;
   nu := 20;  ra := 0;
L1: r := 0;
   for L := nu step −1 until 1 do
   r := −(L ↑ 2+eta2)/(L×((2×L+1)×omega−(L+1)×r));
   if abs(r−ra) > eps × abs(r) then
   begin
      ra := r;  nu := nu + 10;  go to L1
   end;
   la1 := r;  dm := omega − eta − r
end minimal;
procedure Coulomb (eta, ro, Lmax, d, F);
   value eta, ro, Lmax, d;  integer Lmax, d;  real eta, ro;
   array F;
comment  This procedure generates to d significant digits the
```

regular Coulomb wave functions $F_L(\eta, \rho)$ for fixed $\eta \gtreqless 0$, $\rho \geqq 0$, and for $L = 0(1)Lmax$. (For notation, see [2, Ch. 14]). The results are put into the array F. Letting

$$f_L = \frac{2^L L!}{(2L)!C_L(\eta)} F_L(\eta, \rho), \quad C_L(\eta) = \frac{2^L e^{-\pi\eta/2} \,|\, \Gamma(L+1+i\eta)|}{(2L+1)!},$$

the procedure first obtains f_L as the minimal solution of the recurrence relation

$$\frac{L[(L+1)^2+\eta^2]}{(L+1)(2L+3)} y_{L+1} - \left[\eta + \frac{L(L+1)}{\rho}\right] y_L + \frac{L(L+1)}{2L-1} y_{L-1} = 0,$$

using for normalization the identity

$$\rho e^{\omega\rho} = \sum_{L=0}^{\infty} \lambda_L f_L \,, \quad \lambda_L = i^L P_L^{(i\eta,-i\eta)}(-i\omega),$$

where $P_L^{(\alpha,\beta)}(z)$ denotes the Jacobi polynomial of degree L. The parameter ω is so chosen as to avoid undesirable cancellation effects. The final results F_L are obtained recursively, by

$$F_L(\eta, \rho) = c_L f_L \,,$$

$$c_L = \frac{2L-1}{L(2\mathrm{L}+1)}[L^2+\eta^2]^{\frac{1}{2}} c_{L-1} (L = 1, 2, 3 \cdots), \quad c_0 = \left(\frac{2\pi\eta}{e^{2\pi\eta}-1}\right)^{\frac{1}{2}}.$$

A detailed justification of the process is to appear elsewhere ([3]). For large positive η and ρ, the generation of the coefficients λ_L is subject to some loss of accuracy. If $0 \leqq \eta \leqq 20, 0 \leqq \rho \leqq 20$, none, or only a few decimal digits will be lost, however. Writing the procedure *minimal* in double precision will resolve the problem for η, ρ up to about 50, for normal accuracy requirements. In any case, if higher precision is desirable, the procedure puts out a message to this effect. There is an error exit, if $\rho < 0$;

```
begin integer L, nu, nu1, mu, mu1, i, k;
   real epsilon, ro1, eta2, omega, d1, sum, r, r1, s, t1, t2;
   array lambda, lmin[0:1], Fapprox, Rr[0:Lmax];
   switch coefficients := L2, L1, M1;
   if ro < 0 then go to alarm2;
   if ro = 0 then
   begin
      for L := 0 step 1 until Lmax do F[L] := 0;
      go to L5
   end;
   epsilon := .5 × 10 ↑ (−d);  ro1 := 1/ro;  eta2 := eta ↑ 2;
   t1 := if eta > 0 then .5 × ro/eta else 0;
   omega := if eta < 1 then 0 else
   if t1 ≧ 1 then 1.570796327/t1 else
      (1.570796327 − arctan(sqrt(1/t1−1)) + sqrt(t1×(1−t1)))/t1;
   lambda [0] := lmin[0] := 1;  lambda[1] := omega − eta;
   sum := ro × exp(omega×ro);
   for L := 0 step 1 until Lmax do Fapprox[L] := 0;
   d1 := 2.3026 × d + 1.3863;
   t1 := 1.3591 × ro;
   L := if Lmax < t1 then 1 + entier(t1) else Lmax;
   t1 := exp(1.5708×eta);  s := sqrt(1+omega ↑ 2);
   t1 := if  omega = 0  then  t1 + 1/t1  else
      exp(−eta×arctan(1/omega));
   t2 := omega + s;
   r := 1.3591 × ro × t2;
   s := (d1+ln(t1×sqrt(t2/s))−omega× ro)/r;
   nu := if s ≧ −.36788 then entier(r×t(s)) else 1;
   nu1 := entier(L×t(.5×d1/L));
   nu := if nu < nu1 then nu1 else nu;
   nu1 := 1;
   if omega = 0 then i := 1 else i := 2;
L0: begin own array lambda[0:nu];
     comment Dynamic own array declarations are not per-
       mitted in most of the current ALGOL compilers. It can be
       avoided here, at the cost of extra storage, if lambda is de-
       clared as an array of dimension [0:300] at the beginning of
       the procedure Coulomb. The same remark applies to the
       array lmin declared later in the block labeled M1;
     go to coefficients [i];
L1: minimal (eta, omega, 10−m, r1, d1);
     comment The letter m in 10−m is a place holder for a ma-
       chine-dependent integer, namely one less than the number of
       decimal digits carried in the precision mode (single, or
       double precision) of the procedure minimal. Similarly for
       the letter n in the next statement, which is a place holder
       for the integer m + 1. Both m and n are to be properly
       substituted by the user;
     if abs(d1×epsilon) ≧ 10−n then begin i := 1; go to L2 end;
     outstring (1, 'The requested accuracy cannot be guaranteed.
     Use of the procedure minimal in a higher precision mode
     appears indicated');
      i := 3; mu1 := 0;
M1: begin array Rra, lam[0:nu]; own array lmin[0:nu];
      mu := entier (1.25×nu);
      for L := mu1 step 1 until nu do lam[L] := 0;
M2:   r := 0;
      for L := mu step −1 until mu1 + 1 do
      begin
        r := −(L↑2+eta2)/(L×((2×L+1)×omega−(L+1)×r));
        if L ≦ nu then Rra[L−1] := r
      end;
      for L := mu1 + 1 step 1 until nu do
        lmin[L] := Rra[L−1] × lmin[L−1];
      for L := mu1 step 1 until nu do
      if abs(lmin[L]−lam[L]) > epsilon × abs(lmin[L]) then
      begin
        for k := mu1 step 1 until nu do lam[k] := lmin[k];
        mu := mu + 5;
        if mu < 5 × nu then go to M2 else
        begin
          outstring (1, 'convergence difficulty in the generation of
            the coefficients lambda sub L');
          go to L5
        end
      end;
      lam[0] := −r1; lam[1] := 1; t1 := d1/(1 + r1↑2);
      for L := 2 step 1 until nu do
      begin
        lam[L] := ((2×L−1)×omega×lam[L−1]+
          ((L−1)↑2+eta2)×lam[L−2]/(L−1))/L;
        lambda[L] := lmin[L] + t1 × (lam[L]+r1×lmin[L])
      end
    end;
    go to L3;
L2: for L := nu1 step 1 until nu − 1 do
      lambda[L+1] := ((2×L+1)×omega×lambda[L]+
          (L↑2+eta2)×lambda[L−1]/L)/(L+1);
L3: r := s := 0;
    for L := nu step −1 until 1 do
    begin
      t1 := eta/(L+1);
      r := 1/((2×L−1)×(t1/L+ro1−(1+t1↑2)×r/(2×L+3)));
      s := r × (lambda[L]+s);
      if L ≦ Lmax then Rr[L−1] := r
    end;
    F[0] := sum/(1+s);
    for L := 1 step 1 until Lmax do F[L] := Rr[L−1] × F[L−1];
    comment The for-statement which follows is of purely
      precautionary nature, making sure that the results have the
      required accuracy. If speed is important, the statement
      may be omitted;
    for L := 0 step 1 until Lmax do
    if abs(F[L]−Fapprox[L]) > epsilon × abs(F[L]) then
    begin
      for k := 0 step 1 until Lmax do Fapprox[k] := F[k];
      nu1 := mu1 := nu; nu := nu + 10;
      if nu < 300 then go to L0 else
      begin
        outstring (1, 'convergence difficulty in Coulomb');
        go to L5
      end
    end
  end;
  t1 := 6.2831853072 × eta;
  comment The constant 2π in the preceding statement must be
    supplied more accurately if more than 11 significant digits are
    desired in the final results;
  if abs(t1) < 1 then
  begin
    t2 := s := 1; L := 1;
L4: L := L + 1;
    t2 := t1 × t2/L; s := s + t2;
    if abs(t2) > epsilon × abs(s) then go to L4;
    s := sqrt(1/s)
  end
  else
    s := sqrt(t1/(exp(t1)−1));
  F[0] := s × F[0];
  for L := 1 step 1 until Lmax do
  begin
    s := (L−.5) × sqrt(L↑2+eta2) × s/(L×(L+.5));
    F[L] := s × F[L]
  end;
L5: end Coulomb;
```

comment The procedure *Coulomb* was tested on the CDC 3600 computer, with the procedure *minimal* in single precision (unless stated otherwise). The tests included the following:

(i) Generation of $\Phi_L(\eta, \rho) = [C_L(\eta)\rho^{L+1}]^{-1}F_L(\eta, \rho)$, $L = 0(1)21$, to 8 significant digits (d=8) for $\eta = 0$, $-5(2)5$, $\rho = .2$, $1(1)5$. The results were in complete agreement with values tabulated in [4].

(ii) Computation of $F_0(\eta, \rho)$, $F_0'(\eta, \rho) = (d/d\rho)F_0(\eta, \rho)$ to 6 significant digits for $\eta = 0(2)12$, $\rho = 0(5)40$, using $F_0' = (\rho^{-1}+\eta)F_0 - (1+\eta^2)^{\frac{1}{2}}F_1$. Comparison with [5] revealed frequent discrepancies of one unit in the last digit. In addition, beginning with $\eta = 8$, the results became progressively worse for $\rho = 30, 35, 40$, being correct to only 2–3 digits when $\eta = 12$, $\rho = 40$. With the procedure *minimal* in double precision, however, these errors disappeared.

(iii) Computation to 8 significant digits of $F_0(\eta, \rho)$, $F_0'(\eta, \rho)$ for $\rho = 2\eta$, $\rho = .5(.5)20(2)50$. The results agreed with those published in [1] for $\rho \leqq 16$, but became increasingly inaccurate for larger values of ρ. Complete agreement was observed, however, when the procedure *minimal* was operating in the double-precision mode;

References:

1. Abramowitz, M., and Rabinowitz, P. Evaluation of Coulomb wave functions along the transition line. *Phys. Rev. 96* (1954), 77–79.

2. Abramowitz, M., and Stegun, I. A. (Eds.). *Handbook of Mathematical Functions.* NBS Appl. Math. Ser. 55, U. S. Gov't. Printing Off., Washington, D. C., 1964.

3. Gautschi, W. Computational aspects of three-term recurrence relations. *SIAM Rev.*, to appear.

4. National Bureau of Standards. *Tables of Coulomb Wave Functions, Vol. I.* Appl. Math. Ser. 17, U. S. Gov't. Printing Office, Washington, D. C., 1952.

5. Tubis, A. Tables of nonrelativistic Coulomb wave functions. LA-2150, Los Alamos Scientific Lab., Los Alamos, New Mexico, 1958.

REMARK ON ALGORITHM 292 [S22]
REGULAR COULOMB WAVE FUNCTIONS [Walter Gautschi, *Comm. ACM 9* (Nov. 1966), 793]

Walter Gautschi (Recd. 5 July 1967)
Computer Sciences Department, Purdue University, Lafayette, Indiana, and Argonne National Laboratory, Argonne, Illinois

* This work was performed under the auspices of the United States Atomic Energy Commission.

KEY WORDS AND PHRASES: Coulomb wave functions, wave functions, regular Coulomb wave functions
CR CATEGORIES: 5.12

The following changes are suggested to eliminate the need for multiple-precision arithmetic. The underlying theory will be published in *Aequationes Math.*

1. Remove the procedure *minimal*.

2. Change the statement (near the bottom of page 794)

   nu := **if** $s \geq -.36788$ **then** *entier* $(r \times t(s))$ **else** 1

   to read:

   nu := **if** $s \geq -.36788$ **then** *entier* $(r \times t(s))$ **else** $r/2.7183$

3. Change the statement labeled *L*1 to read

   *L*1: *d*1 := 2 × *eta*/(*exp*(2×*eta*×*arctan*(1/*omega*))−1)

   and rephrase the comment following this statement to read:

   comment The letter *n* in the following statement is a place holder for a machine-dependent integer, namely, the number of (equivalent) decimal digits carried in the mantissa of floating-point numbers. This integer must be properly substituted by the user;

4. Omit the output statement

   outstring (1, 'The requested accuracy cannot be guaranteed. Use of the procedure *minimal* in a higher precision mode appears indicated');

5. Insert the statement

   *r*1 := *lmin*[1];

   between the two lines

   end;

   and

   lam[0] := −*r*1; *lam*[1] := 1; *t*1 := *d*1/(1+*r*1 ↑ 2);

6. Change the line (near the middle of page 795)

   s := *sqrt*(*t*1/(*exp*(*t*1)−1));

   to read

   s := *exp*(−*t*1/4)/*sqrt*((*exp*(*t*1/2) − *exp*(−*t*1/2))/*t*1);

   (These statements are mathematically equivalent, but the latter delays overflow as the value of *t*1 becomes large.)

7. If large values of $|\eta|$ and/or ρ, say exceeding 100, are contemplated, it may be necessary to increase the dimension of the arrays *lambda* and *lmin* (if they are declared at the beginning of the procedure *Coulomb*) and to correspondingly increase the upper limit for *nu* in the conditional clause

   if *nu* < 300

   near the top of page 795. The user, in this case, should also be prepared to encounter overflow difficulties, especially in the later entries of the array *lam*.

With these revisions the algorithm produced correct results on the CDC 3600 for the three tests described at the end of Algorithm 292. It was also used (with input parameter $d = 10$) to compute miscellaneous values of $F_0(\eta, \rho)$ and $\Phi_0(\eta, \rho)$ published in a paper by C. E. Fröberg (Numerical treatment of Coulomb wave functions. *Rev. Mod. Phys. 27* (1955), 399–411). The results are summarized in the table below.

η	ρ	Algorithm 292 (revised)	Fröberg
9	50	$F_0 = 9.357085680_{10}-1$	$9.3570855_{10}-1$
50	80	$F_0 = 1.203662491_{10}-3$	$1.203665_{10}-3$
50	120	$F_0 = 2.002599349_{10}-1$	$2.00255_{10}-1$
100	4	$\Phi_0 = 5.722985154_{10}21$	$5.722985155_{10}21$
200	1	$\Phi_0 = 7.236604732_{10}14$	$7.236604731_{10}14$

In addition, the algorithm was run (with $d = 6$, and *lambda*, *lmin* being declared as arrays of dimension [0 : 600]) for $\eta = -200(20)200$, $\rho = 20(20)200$, $Lmax = 0(50)100$. Apparently valid results were obtained as long as $\eta \leq 100$, though no tables seem to exist to check these results against. Overflow was observed in some of the entries of the array *lam*, for $\eta = 120$, $\rho \geq 120$; $\eta = 140$, $\rho \geq 60$; $\eta = 160$, $\rho \geq 40$; and $\eta = 200$, $\rho \geq 20$. (For the purpose of this test, a number is considered to overflow if its modulus exceeds $_{10}300$.)

CERTIFICATION OF ALGORITHM 292 [S22]
REGULAR COULOMB WAVE FUNCTIONS [Walter Gautschi, *Comm. ACM 9* (Nov. 1966), 793]
AND OF
REMARK ON ALGORITHM 292 [S22]
REGULAR COULOMB WAVE FUNCTIONS [Walter Gautschi, *Comm. ACM 12* (May 1969), 280]

K. S. Kölbig (Recd. 10 Oct. 1967)
Applied Mathematics Group, Data Handling Division, European Organization for Nuclear Research (CERN), 1211 Geneva 23, Switzerland

KEY WORDS AND PHRASES: Coulomb wave functions, wave functions, regular Coulomb wave functions
CR CATEGORIES: 5.12

Both the original and the revised version of the procedure *Coulomb* have been translated into Fortran and tested on a Control Data 6600 computer. It became apparent that the following changes in the original version are necessary:

1. The second sentence in the **comment** following the statement labeled *L*1 in procedure *Coulomb* should be replaced by:

Similarly for the letter *n* in the next statement, which is a place holder for the number of digits carried in the main program.

2. The second statement after this **comment** (beginning "*outstring* . . .") should be changed to

```
if abs(d1 × epsilon) < 10-m-1 then
outstring (1, 'The requested accuracy cannot be guaranteed.
Use of the procedure minimal in a higher precision mode appears indicated.');
```

Since the original version of *Coulomb* is to be superseded by the revised one (see Remark), detailed test results are given here only for the latter. Most of the tests have already been described in the Algorithm itself or in the Remark. Those presented here are obtained on a different machine, and the results differ slightly in some cases from the previous ones. The tests included the following:

(i) Generation of $\Phi_L(\eta,\rho) = [C_L(\eta)\rho^{L+1}]^{-1} F_L(\eta,\rho)$, $L = 0(1)21$, to 8 significant digits ($d = 8$) for $\eta = -5(1)5$, $\rho = .2(.2)5$. The results were in complete agreement with the values tabulated in [4] of Algorithm 292. In the cases where more than 8 significant digits are tabulated, the highest discrepancy was one unit in the last digit; e.g. for $L = 0$, $\eta = 5$, $\rho > 3.4$, 10 to 11 correct significant digits have been found.

(ii) Computation of $F_0(\eta,\rho)$, $F_0'(\eta,\rho) = (d/d\rho)\, F_0(\eta,\rho)$ to 5 significant digits for $\eta = 0(2)12$, $\rho = 0(5)40$, using $F'_0 = (\rho^{-1} + \eta)F_0 - (1+\eta^2)^{\frac{1}{2}} F_1$. Comparison with [5] of Algorithm 292 revealed frequent discrepancies of one unit in the fifth digit. For $\eta = 2$, $\rho = 40$ the discrepancy in F_0 is 80 units of the fifth digit. This is probably an error in the table.

(iii) Computation to 8 significant digits of $F_0(\eta,\rho)$, $F_0'(\eta,\rho)$ for $\rho = 2\eta$, $\rho = .5(.5)20(2)50$. The results agreed completely with those published in [1] of Algorithm 292.

(iv) Computation (with $d = 10$) of the miscellaneous values of $F_0(\eta,\rho)$ and $\Phi_0(\eta,\rho)$ given in the Remark on Algorithm 292. The results obtained differ slightly from those given in the Remark. In the worst case, $\eta = 50$, $\rho = 120$, the discrepancy is 16 units in the tenth digit.

(v) After changing the dimensions of the arrays *lambda*, *lmin* into [0:600] and adjusting the upper limit for *nu* to 600 (see Remark on Algorithm 292), $F_L(\eta,\rho)$ has been calculated with $d = 6$ for $\eta = -200(20)\,200$, $\rho = 20(20)\,200$, $Lmax = 0(50)100$ merely to test whether overflow occurs or not. The following table indicates where overflow, indefinite results, or convergence difficulties in the generation of λ_L (see Algorithm 292) have been observed.

η	$\rho \geq$
20	200
40	200
60	180
80	100
100	80
120	60
140	60
160	60
180	40
200	40

(vi) Calculation of $F_L(\eta,\rho)$ for $L = 0(50)100$ with $d = 7$ for $\eta = 1$, $\rho = 10^{-n}$, $n = -20(1)-1$. Underflow occurred for $L = 50$, $n \leq 5$; $L = 100$, $n \leq 2$. The valid results have been compared with those obtained by summation of the power series for $\Phi_L(\eta,\rho)$ (see [4, (1.3) and (4.4)] of Algorithm 292). Agreement has been found to 7 significant digits.

(vii) Calculation of $\Phi_L(\eta,\rho)$ to 13 significant digits (d=13) for $\rho = 5$, $\eta = 0(1)5$, $L = 0(10)100$. The results have been compared with those obtained by summation in double-precision mode (27 digits) of the power series mentioned in (vi). Agreement was found to at least 12 significant digits. The constant 2π in the statement *t*1 := . . . on page 795 of Algorithm 292 was supplied here with 14 significant digits, as required by the **comment**.

Acknowledgment. I wish to thank Professor Gautschi for useful remarks and comments.

REMARK ON ALGORITHM 292 [S22]*
REGULAR COULOMB WAVE FUNCTIONS [Walter Gautschi, *Comm. ACM 9* (Nov. 1966), 793]
AND ON
REMARK ON ALGORITHM 292 [S22]
REGULAR COULOMB WAVE FUNCTIONS [Walter Gautschi, *Comm. ACM 12* (May 1969), 280]

W. J. Cody and Kathleen A. Paciorek (Recd. 8 Sept. 1969 and 8 May 1970)
Argonne National Laboratory, Argonne, IL 60439

* Work performed under the auspices of the US Atomic Energy Commission.

KEY WORDS AND PHRASES: Coulomb wave functions, wave functions, regular Coulomb wave functions
CR CATEGORIES: 5.12

The revised version of the procedure *Coulomb* was translated into IBM System/360 Algol and tested on an IBM S/360 Model 75 Computer. When $\eta > 12$ overflow problems were encountered in the generation of intermediate arrays. These were due to the smaller exponent range of the S/360, $-64 \leq exp \leq 63$. The following changes, while not completely eliminating the overflow probe ms, greatly alleviate them.

```
Insert   real scale;
after   begin integer L, nu, nu1, mu, mu1, i, k;

Insert   scale := 16 ↑ (−57);
  comment  This value of scale is appropriate for the IBM S/360.
    On a machine with a different base and a different exponent
    range, say α ≤ exp ≤ β, the value of scale should be base ↑
    (6−β);
between   end;
and   epsilon := .5 × 10 ↑ (−d);

Change   lambda [0] := lmin [0] := 1;  lambda [1] := omega-eta;
         sum := ro × exp (omega × ro);
to   lambda [0] := scale;  lmin [0] := 1;
     lambda [1] := (omega-eta) × scale;
     sum := ro × exp(omega × ro) × scale;

Change   lmin [L] := Rra [L − 1] × lmin [L − 1];
to   begin
       t1 := Rra [L − 1] × lmin [L − 1];
  comment  The following constant 5 ↑ (−10) is approximately
    2 × base ↑ α/scale, where base is the base of the floating-
    point number system and α ≤ exp ≤ β;
  lmin[L] := if abs(t1) > 5 ↑ (−10) then
       t1 else 0
  end;

Change   lam [0] := −r1; lam [1] := 1;
to   lam [0] := −r1 × scale;  lam [1] := scale;

Change   lambda [L] := lmin [L] + t1 × (lam [L] + r1 × lmin [L])
to   lambda [L] := lmin [L] × scale + t1 ×
         (lam [L] + r1 × scale × lmin [L])

Change   F[0] := sum/(1+s);
to   F[0] := sum/(scale+s);
```

The authors gratefully acknowledge the referee's helpful suggestions.

ALGORITHM 293
TRANSPORTATION PROBLEM [H]

G. Bayer (Recd. 9 July 1965 and 22 Aug. 1966)
Technische Hochschule, Braunschweig, Germany

```
procedure transp1 (m, n, inf, c, a, b, x, kw);  value m, n, inf;
  integer m, n, inf, kw;  integer array c, a, b, x;
comment transp1 is derived from Algorithm 258, transport,
  [Comm. ACM 8 (June 1965), 381] in order to reduce running time
  by about 50 percent. The following notation is used.
  c     m, n-matrix of unit costs,
  a     array of quantities available,
  b     array of quantities required, following the usual descrip-
        tion of the transportation problem,
  inf   greatest positive integer within machine capacity,
  x     m, n-matrix of flows,
  kw    optimal total costs (computed by procedure).
  c, a, b are disturbed by the procedure. Sum of a[i] = sum of b[i].
  Multiple solutions are left out of account. [Ref.: G. Hadley,
  Linear Programming, Reading, London, 1962, p. 351];
begin integer i, j, u, v, k, l, s, t, gd, h, p, cij, xij, ai, bj, lsvj, nlvi;
  Boolean zg;
  integer array g, listu, nlv[1:m], r, listv[1:n], ls[0:m+n−1],
    nl[1:m×n], lsv[0:n];
  comment in the for-statement u := ··· after s33, operate on
    all pairs i, j with c[i,j] = 0. To win time the array nl supervises
    those zeros; the j-indices of zeros in row i are kept in
    nl[(i−1)×n+1] ··· nl[nlv[i]]. In the for-statement v := ···
    after s33, operate on all pairs i, j with x[i,j] ≠ 0 (and c[i,j]=0).
    ls supervises those essential zeros, the i-indices of essential
    zeros in column j are kept in ls[lsv[j−1]+1] ··· ls[lsv[j]
    Procedure in adds to list ls, procedure out takes out from list
    ls an essential zero in position i, j;
  procedure in;
  begin
    lsvj := lsv[j];
    for t := lsv[n] step −1 until lsvj do ls[t+1] := ls[t];
    for t := j step 1 until n do lsv[t] := lsv[t] + 1;
    ls[lsvj+1] := i
  end ;
  procedure out ;
  begin
    lsvj := lsv[j];
    for t := lsv[j−1]+1 step 1 until lsvj do
    begin
      if ls[t] ≠ i then go to next;
      s := t;  go to ex;
  next:
    end ;
  ex:
    for t := j step 1 until n do lsv[t] := lsv[t]−1;
    lsvj := lsv[n];
    for t := s step 1 until lsvj do ls[t] := ls[t+1]
  end ;
  for i := 1 step 1 until m do
    for j := 1 step 1 until n do x[i,j] := 0;
  for i := 1 step 1 until m do nlv[i] := (i−1)×n;
  lsv[0] := 0;
  for j := 1 step 1 until n do
  begin
    listv[j] := 1;
    lsv[j] := 0
  end ;
s1:
  kw := gd := 0;
  comment gd is the defect, i.e., the sum of quantities not yet
    transported;
  for i := 1 step 1 until m do
  begin
    h := inf;
    for j := 1 step 1 until n do
      if c[i, j] < h then h := c[i, j];
    for j := 1 step 1 until n do
    begin
      cij := c[i, j] := c[i, j] − h;
      if cij = 0 then
      begin
        listv[j] := 0;
        nlvi := nlv[i] := nlv[i] + 1;
        nl[nlvi] := j
      end
    end;
    kw := h × a[i] + kw
  end see next comment;
  for j := 1 step 1 until n do
  begin
    if listv[j] = 0 then go to nextj1;
    h := inf;
    for i := 1 step 1 until m do
      if c[i, j] < h then h := c[i, j];
    for i := 1 step 1 until m do
    begin
      cij := c[i, j] := c[i, j] − h;
      if cij = 0 then
      begin
        nlvi := nlv[i] := nlv[i] + 1;
        nl[nlvi] := j
      end
    end;
    kw := h × b[j] + kw;
nextj1:
  end;
  comment in step 1 the usual reduction of the matrix of costs
    is achieved (dual problem), zeros are listed in nl;
s2:
  for i := 1 step 1 until m do
  begin
    ai := a[i]; nlvi := nlv[i];
    for u := (i−1) × n + 1 step 1 until nlvi do
    begin
      if ai = 0 then go to nexti2;
      j := nl[u];
      bj := b[j];
      if bj = 0 then go to nextj4;
      h := x[i, j] := if ai < bj then ai else bj;
      ai := ai − h; b[j] := bj − h; in;
```

```
nextj4:
    end;
nexti2:
    a[i] := ai; gd := gd + ai
  end;
  comment applying a usual rule to all zeros we get an initial
    flow (restricted primal problem) in step 2;
s31:
  if gd = 0 then go to s6;
  comment problem is solved if defect has become zero;
s32:
  for j := 1 step 1 until n do r[j] := 0;
  k := 0;
  for i := 1 step 1 until m do
  begin
    if a[i] ≠ 0 then
    begin
      k := k + 1; listu[k] := i; g[i] := inf
    end
    else g[i] := 0
  end;
  comment r[j] = 0 if column j is unlabeled, = i if labeled
    from row i. g[i] = 0 if row i is unlabeled, = inf if a[i] ≠ 0,
    i.e., a[i] is a possible source of flow. The indices i of labeled
    rows are kept in listu[1] ··· listu[k]. In step 3, consisting of
    step 32 and step 33, the maximal flow is found by the la-
    beling process. Labeling ends in only two ways: (a) a column j
    with b[j] > 0 has been labeled: go to step 4, (b) all labeling is
    done, but a positive flow has not been found: go to s5;
s33:
  l := 0;
  for u := 1 step 1 until k do
  begin
    i := listu[u]; nlvi := nlv[i];
    begin
      j := nl[s];
      if r[j] ≠ 0 then go to nextj5;
      r[j] := i; l := l + 1; listv[l] := j;
      if b[j] > 0 then go to s4;
nextj5:
    end
  end in each newly labeled row, see listu, look for zeros in
    unlabeled columns, list them in listv;
  if l = 0 then go to s5;
  k := 0;
  for v := 1 step 1 until l do
  begin
    j := listv[v]; lsvj := lsv[j];
    for s := lsv[j-1]+1 step 1 until lsvj do
    begin
      i := ls[s];
      if g[i] = 0 then
      begin
        g[i] := j; k := k + 1;
        listu[k] := i
      end
    end
  end in each newly labeled column, see listv, look for essential
    zeros in unlabeled rows, label these rows, list them in listu;
  if k = 0 then go to s5;
  go to s33;
  comment step 4. A column j with b[j] has been labeled, b[j]
    is the sink of a possible positive flow, the path of which is
    indicated by labels. Find the minimum flow h along the path;
  n := b[j]; p := j;
mark:
  i := r[j]; j := g[i];
  if j = inf then
  begin
    if a[i] < h then h := a[i]; go to re
  end;
  if x[i, j] < h then h := x[i, j];
  go to mark;
re: ;
  comment flow h along the labeled path thus reduces defect
    without changing total costs. Correct list of essential zeros
    if necessary. Start labeling anew, optimizing the restricted
    primal problem;
  j := p; b[j] := b[j] - h; a[i] := a[i] - h;
  gd := gd - h;
re1:
  i := r[j]; xij := x[i, j]; x[i, j] := xij + h;
  if xij = 0 then in;
  j := g[i];
  if j = inf then go to s31;
  xij := x[i, j] := x[i, j] - h;
  if xij = 0 then out;
  go to re1;
s5: ;
  comment step 5. Flow is maximal. To find a new solution to
    the dual, take the part of matrix c which is the intersection
    of labeled rows and unlabeled columns, reduce matrix in a
    certain way;
  k := 0; l := n + 1;
  for j := 1 step 1 until n do
  begin
    if r[j] = 0 then
    begin
      k := k + 1; listv[k] := j
    end
    else
    begin
      l := l - 1; listv[l] := j
    end
  end list all labeled resp. unlabeled columns in listv;
  h := inf;
  for i := 1 step 1 until m do
  begin
    if g[i] = 0 then go to nexti6;
    for s := 1 step 1 until k do
    begin
      j := listv[s];
      if c[i, j] < h then h := c[i, j]
    end;
nexti6:
  end find minimum h in partial matrix;
  for i := 1 step 1 until m do
  begin
    zg := g[i] ≠ 0; nlvi := (i-1) × n;
    for s := 1 step 1 until n do
    begin
      j := listv[s];
      if zg then cij := c[i, j]
      else
      cij := c[i, j] := c[i, j] + h;
      if cij = 0 then
      begin
        nlvi := nlvi + 1;
        nl[nlvi] := j
      end
    end;
    for s := 1 step 1 until k do
    begin
```

```
      j := listv[s];
      if zg then cij := c[i, j] := c[i, j] − h
      else cij := c[i, j];
      if cij = 0 then
      begin
        nlvi := nlvi + 1;
        nl[nlvi] := j
      end
    end;
    nlv[i] := nlvi
  end reduction, add h to labeled columns, subtract h from
    labeled rows. Construct new list of zeros;
  kw := h × gd + kw;
  comment total costs for new solution of dual;
  go to s32;
s6: ;
  comment solution, defect has become zero;
end
```

CERTIFICATION OF:

ALGORITHM 258 [H] TRANSPORT

[G. Bayer, *Comm. ACM 8* (June 1965), 381]

ALGORITHM 293 [H] TRANSPORTATION PROBLEM

[G. Bayer, *Comm. ACM 9* (Dec. 1966), 869]

Lee S. Sims (Recd. 21 Feb. 1967 and 17 Mar. 1967)
Kates, Peat, Marwick & Co., Toronto, Ont., Canada

Both of these algorithms were coded in Extended Algol 60 and tested on a Burroughs B5500. Three problems were solved correctly, one of them being of medium size (55 × 167). On this larger problem *transp*1 was found to be about twice as fast as *transport*.

In coding and debugging *transp*1 three apparent errors were found. In the right-hand column on page 870, after line 27 which is

```
i := listu[u]; nlvi := nlv[i];
```

a line is missing. This line should read

```
for s := (i−1) × n + 1 step 1 until nlvi do
```

Also in the right-hand column, the line

```
s4: ;
```

should be inserted ahead of line −12, which begins

```
comment Step 4. A column j with b[j] has been labeled, b[j]
```

On page 871, in the left-hand column, line --22 which reads

```
for s := 1 step 1 until n do
```

should read

```
for s := l step 1 until n do
```

REMARK ON ALGORITHM 293 [H] TRANSPORTATION PROBLEM [G. Bayer, *Comm. ACM 9* (Dec. 1966), 869]

G. Bayer (Recd. 24 Aug. 1967, 30 Oct. 1967 and 8 Jan. 1968)
Technische Hochschule Braunschweig, Germany

KEY WORDS AND PHRASES: transportation problem, linear programming
CR CATEGORIES: 5.41

There is an error in the algorithm concerning the number of essential zeros which can be greater than $m + n - 1$. An example is:

$$\begin{array}{rcccc} c: & 1 & 1 & 2 & 1 \\ & 2 & 1 & 1 & 1 \\ & 1 & 2 & 2 & 2 \\ a: & 4 & 6 & 1 & \\ b: & 2 & 4 & 2 & 3 \end{array}$$

The difficulty may be overcome in two ways.

1. Declare array *ls* by:

```
integer array ls[0 : m×n]
```

instead of:

```
integer array ls[0 : m+n−1]
```

2. As the case of more than $m + n - 1$ essential zeros will seldom arise in practical problems, it may be enough to have

```
ls[0 : 2×m+n−1];
```

(It is assumed that $m \leq n$). To make sure that list *ls* does not overflow, add a statement to **procedure** *in* and remove *inf* from the **value** part.

```
procedure in;
begin if lsv[n] = 2 × m + n − 1 then
  begin inf := 0;  go to s6 end;
```

Thus in the case of overflow of *ls*, the procedure is left with *inf* = 0 signalling that the optimum has not been reached and that the solution is possibly incomplete. (One would wish then to run the procedure anew with more space for *ls* and using the solution obtained as an initial flow. This would only be possible by partly rewriting the algorithm.)

ALGORITHM 294
UNIFORM RANDOM [G5]

W. MURRAY STROME (Recd. 26 May 1966)
Carnegie Institute of Technology, Pittsburgh, Pa.

```
real procedure UNIFORM (A, B, X0, C, M);
  value A, B, X0, C, M;  real A, B;  integer X0, M, C;
```

comment This procedure generates the next uniformly distributed pseudorandom number on (A, B). The "multiplicative congruential" method is used, namely

$$Z_{n+1} = C \times Z_n (\text{mod } M)$$

M and C are chosen to maximize the period and minimize the correlation of the sequence generated. To accomplish this, M should be as large as possible subject to the following conditions [1]:

(i) $C \simeq \sqrt{M}$ and suitably chosen.

(ii) The expression $X := X0/M$ followed by $X := X \times C$ within the procedure must be evaluated with no roundoff or truncation error for every positive integer $X0 < M$.

For most applications, M and C may be chosen as follows. Let D denote the number base of the machine (e.g., $D = 10$ for a decimal machine) and n the number of significant D-digits of a **real** variable of the ALGOL implementation. Then let $M = D^k$ and $C = D^{n-k} - q$ where $k = \textit{entier}\ ((2n + 1)/3)$. For $D = 2, 4, 5, 8, 10$ or 16 and $D^{n-k} > 100$, $q = 3$ is suitable. In general, choosing M and C in the above fashion will guarantee that condition (ii) be met, but this should be verified for the particular implementation. See [1] for a more detailed discussion on the choice of C and M. The first time *UNIFORM* is used in a program, $X0$ should be a positive integer less than and relatively prime to M. Subsequently, use $X0 = 0$.

UNIFORM was translated into C.I.T's ALGOL-20 and run on a CDC G-20 computer with $M = 2^{28}$ and $C = 2^{14} - 3$. Some scaling was required to prevent roundoff in the multiplications since the G-20 is a 14-octal digit machine rather than a 42-bit binary one (the scaling would have been unnecessary had we used $M = 8^9$, $C = 8^5 - 3$, but the period of the sequence would have been shorter). In order to test the algorithm, the following statistical tests were performed for sequences of pseudorandom numbers generated on $(-1, 1)$.

1. *Distribution.* We divided $(-1, 1)$ into 10 equal subintervals. Denote by f_i the number of numbers of a sequence of length 1000 in the ith interval. The statistic

$$\chi^2 = .01 \sum_{i=1}^{10} (f_i - 100)^2$$

was computed for each of 62 different such sequences. For numbers drawn from a uniform distribution, this statistic has a χ^2-distribution, with 9 degrees of freedom [2]. The results obtained were entirely consistent with the hypothesis that the numbers were distributed uniformly.

2. *Independence.* Define the serial correlation (lag j) by

$$\rho_j = \frac{\frac{1}{N} \sum_{i=1}^{N} X_i X_{i+j} - \left(\frac{1}{N} \sum_{i=1}^{N} X_i\right)^2}{\frac{1}{N-1} \sum_{i=1}^{N} X_i^2 - \left(\frac{1}{N} \sum_{i=1}^{N} X_i\right)^2}.$$

If X_i, X_{i+j} are independent, then for large N, ρ_j is distributed normally with mean $-1/N$ and standard deviation $1/\sqrt{N}$ [3]. ρ_1 was estimated for 16 different sequences each of length 5000. The average, -0.004, and the standard deviation, 0.011, are consistent with the hypothesis of independence. ρ_j was estimated for 3 different sequences each of length 9900 for $j = 1, 2, \cdots, 49$. These results were consistent with the hypothesis that X_i, X_{i+j} are independent for these values of j.

The Von Neumann ratio test [4] for 16 sequences of length 1000 also yielded results consistent with the hypothesis of independence. The results of other tests for many values of C and M using this method are outlined in [1];

```
begin own real X;
  if X0 ≠ 0 then X := X0/M;
  X := X × C;  X := X − entier (X);
UNIFORM := X × (B − A) + A
end procedure UNIFORM
```

REFERENCES:

1. HULL, T. E., AND DOBELL, A. R. Random number generators. *SIAM Rev. 4* (July 1962), 230–254.
2. YAMANE, T. *Statistics, An Introductory Analysis.* Harper & Row, New York, 1964, pp. 584–593.
3. ANDERSON, R. L. Distribution of the serial correlation coefficient, *Ann. Math. Stat. 13* (1942), 1–13.
4. HART, B. I. Tabulation of the probability for the ratio of the mean square successive difference to the variance. *Ann. Math. Stat. 13* (1942), 207–214.

ALGORITHM 295
EXPONENTIAL CURVE FIT [E2]

H. Späth (Recd. 29 Apr. 1966)
Institut für Neutronenphysik und Reaktortechnik,
Kernforschungszentrum Karlsruhe, Germany

```
procedure expfit (x, y, p, n, ca, ce, eps, a, b, c, s, fx, exit);
  value n, ca, ce, eps;  integer n;  real ca, ce, eps, a, b, c, s;
  label exit;  array x, y, p, fx;
```

comment If the method of least squares is used to determine the parameters a, b, c of a curve $f(x) = a + be^{-cx}$ which approximates n data points (x_i, y_i) with associated weights p_i, then

$$s(a, b, c) = \sum_{i=1}^{n} p_i(y_i - f(x_i))^2 \qquad \text{(I)}$$

must be a minimum. A necessary condition for this is that

$$\frac{\partial s}{\partial a} = \frac{\partial s}{\partial b} = \frac{\partial s}{\partial c} = 0. \qquad \text{(II)}$$

Usually (see [1]) it is attempted to solve this system of nonlinear equations by an iterative method which is based upon the linearization of f in (II) and the convergence of which depends on the given starting values for a, b, c.

A simpler and more effective way which can always be chosen if there is only one nonlinear parameter in f is the following: It is always possible to eliminate $a = a(c)$ and $b = b(c)$ from the equations $\partial s/\partial a = 0$ and $\partial s/\partial b = 0$ and to put these expressions into $\partial s/\partial c = 0$. This gives only one equation in one variable

$$F(c) := \frac{\partial s}{\partial c}(a(c), b(c), c) = 0.$$

If a value c' is calculated with $F(c') = 0$ then the corresponding values of a and b are obtained from $a' = a(c')$ and $b' = b(c')$.

The following procedure is based upon this idea which is fully treated in [2]. It allows to find a triple (a, b, c) which solves (II) if you make available a nonlocal procedure *Rootfinder* which is able to get a zero c of a function $F(c)$ in the interval $[ca, ce]$ with the relative accuracy *eps*, if *sign* $(F(ca)) \neq$ *sign* $(F(ce))$ otherwise leaving to the global label *exit*. As the above $F(c)$ is discontinuous at $c = 0$, $[ca, ce]$ must not contain 0. [The speed and efficiency of the algorithm depend on the choice of the procedure *Rootfinder*.—Ref.]

Most of the symbols are self-explanatory. The array *fx* finally contains the values $a + be^{-cx_i}$;

```
begin integer i;  real t, u, v, w, fc, h0, h1, h2, h3, h4, h5, h6, h7;
  procedure fronc (c, fc);  value c;  real c, fc;
  comment  computes for a given c the value fc = F(c) and
    a = a(c), b = b(c);
  begin h0 := h1 := h2 := h3 := h4 := h5 := h6 := h7 := 0;
    for i := 1 step 1 until n do
    begin
      t := x[i];  u := exp(−c×t);  v := p[i];  w := y[i];
      h0 := h0 + v;  h1 := h1 + u × v;  h2 := h2 + u × u × v;
      h3 := h3 + v × w;  h4 := h4 + u × v × w;
        h5 :=h5 + t × u × v;
      h6 := h6 + t × u × u × v;  h7 := h7 − u × v × w × t
    end i;
    t := 1.0/(h0×h2−h1×h1);  a := t × (h2×h3−h1×h4);
    b := t × (h0×h4−h1×h3);  fc := h7 + (h5×a+h6×b)
  end fronc;
  Rootfinder (fronc, ca, ce, eps, c, exit);  t := 0;
  for i := 1 step 1 until n do
  begin
    v := fx[i] := a + b × exp(−c×x[i]);  v := v − y[i];
    t := t + p[i] × v × v
  end i;
  s := t
  end expfit
```

References:

1. Deily, G. R. Algorithm 275, Exponential curve fit. *Comm. ACM 9* (Feb. 1966), 85.
2. Oberländer, S. Die Methode der kleinsten Quadrate bei einem dreiparametrigen Exponentialansatz. *ZAMM 43* (1963), 493–506.

ALGORITHM 296
GENERALIZED LEAST SQUARES FIT BY ORTHOGONAL POLYNOMIALS [E2]

G. J. Makinson (Recd. 30 Sept. 1965 and 29 Aug. 1966)
University of Liverpool, Liverpool 3, England

```
procedure LSFITUW (f, x, w, m, k, si, p, l, al, be, s);  value m, k;
  integer m, k;  array f, w, si, p, x, al, be, s;  Boolean l;
comment  LSFITUW accepts m observations x[i], f[i], i = 1, 2,
  · · · , m each with its associated weight w[i]. The weights should
  be provided inversely proportional to the standard error of the
  observations.
    x[1] should be algebraically the smallest abscissa and x[m] the
  largest.
    The coefficients of the best fitting polynomial of degree k or
  less, where k < m − 1, are obtained in p[0:k], with p[0] the
  independent term. si[0:k] contains the measures of the goodness
  of fit of each polynomial tested. The si[t] are examined suc-
  cessively and the best polynomial is chosen of degree h if h is
  the first value of t found such that si[h] < si[h+1] provided
  that si[j] > 0.6 × si[h] for k ≧ j > h + 1. If h is the first value
  of t found such that si[h] < si[h+1] but then a j is found that
  satisfies si[j] ≦ 0.6 × si[h] for j > h + 1 the procedure will choose
  the polynomial of degree j as best fit.
    If an h such that si[h] < si[h+1] is not found then the poly-
  nomial is chosen of degree k. LSFITUW uses the procedure
  POLYX (a, b, c, d, n) [Algorithm 29, Comm. ACM 3 (Nov. 1960),
  604] to transform its results from the interval (−2,2) to the
  interval (x[1], x[m]).
    Normally l should be false but if the choice made is to be
  overruled after consideration of the si and the best fitting
  polynomial is required to be strictly of degree k, then l should be
  true.
    The programming is as outlined by G. E. Forsythe, [J. Soc.
  Indust. Appl. Math. 5 (1957), 74-88] and originally programmed
  by J. G. Mackinney [Algorithm 28/29, Comm. ACM 3 (Nov.
  1960), 604]. LSFITUW incorporates remarks made by D. B.
  MacMillan [Comm. ACM 4 (Dec. 1961), 544].
    The variables in the paper of Forsythe have been abbreviated
  as follows.
      al[i] is alpha[i], be[i] is beta[i], si[i] is (sigma[i]) ↑ 2, s[i] is
      the same, om is omega, lw is w[i, i], tw is w[i+1, i+1],
      ctp[j] is the coefficient of x ↑ j in This (the current)
      orthogonal polynomial, clp[j] is the coefficient of x ↑ j in
      the Last (previous) orthogonal polynomial, cp[j] is the co-
      efficient of x ↑ j in the most recently calculated polynomial
      of best fit, tp[i] is the value at x[i] of the present orthogonal
      polynomial, lp[i] is the value at x[i] of the last orthogo-
      nal polynomial, simin is the least value of (sigma[i]) ↑ 2
      found so far, swx becomes false as soon as (sigma[i+1]) ↑
      2 ≧ (sigma[i]) ↑ 2 one time, comp becomes true if swx
      is false and some (sigma[i]) ↑ 2 < 0.6 × simin;
begin integer i, j;  real du, delsq, om, lw, tw, simin, a, b;
  array ctp, cpsave, cp[0:k], clp[−1:k], lp, tp[1:m];
    Boolean swx, comp;
  comment  initialization;
  for i := 0 step 1 until k do cp[i] := 0;  simin := 0;
  swx := true;  be[0] := clp[0] := clp[−1] := delsq := om := 0;
    ctp[0] := 1;  tw := 0;  comp := false;
  for i := 1 step 1 until m do
  begin
    delsq := delsq + w[i] × f[i] ↑ 2;  tp[i] := 1;
    lp[i] := 0;  om := om + w[i] × f[i];  tw := tw + w[i]
  end;
  s[0] := cp[0] := om/tw;  delsq := delsq − s[0] × om;
    si[0] := delsq/(m−1);
  comment  transformation of abscissa;
  a := 4/(x[m]−x[1]);  b := − 2 − a × x[1];
  for i := 1 step 1 until m do x[i] := a × x[i] + b;
  comment  main computation loop;
  for i := 0 step 1 until k − 1 do
  begin
    du := 0;
    for j := 1 step 1 until m do du := du + w[j] × x[j] × tp[j] ↑ 2;
    al[i + 1] := du/tw;  lw := tw;  tw := om := 0;
    for j := 1 step 1 until m do
    begin
      du := be[i] × lp[j];
      lp[j] := tp[j];
      tp[j] := (x[j]−al[i+1]) × tp[j] − du;
      tw := tw + w[j] × tp[j] ↑ 2;
      om := om + w[j] × f[j] × tp[j]
    end;
    be[i+1] := tw/lw;  s[i+1] := om/tw;
    delsq := delsq − s[i+1] × om;  si[i+1] := delsq/(m−i−2);
    if l then go to L1;
    if ¬ comp then
    begin
      if swx then
      begin
      if si[i+1] ≧ si[i] then
        begin
          comment higher power appears not to improve fit;
          swx := false;
          simin := si[i];
          for j := 0 step 1 until k do
            cpsave[j] := cp[j]
        end;
        go to L1
      end;
      if si[i+1] < 0.6 × simin then comp := true;
      comment  termination of main loop at superior fit to first
        one found;
      comment  recursion to obtain the coefficients cp of the
        polynomial of best fit of degree i + 1;
L1:   for j := 0 step 1 until i do
      begin
        du := clp[j] × be[i];
        clp[j] := ctp[j];
        ctp[j] := clp[j−1] − al[i+1] × ctp[j] − du;
        cp[j] := cp[j] + s[i+1] × ctp[j]
      end;
      cp[i+1] := s[i+1];  ctp[i+1] := 1;  clp[i+1] := 0;
      if ¬ (comp ∨ swx) then
    begin
      if $i = k - 1$ then
      for $j := 0$ step 1 until $k$ do
        $cp[j] := cpsave[j]$
      end
    end
  end end of main computation loop. Transformation of poly-
    nomial follows;
  $POLYX(a, b, cp, p, k)$
end $LSFITUW$
```

REMARK ON ALGORITHM 296 [E2] GENERALIZED LEAST SQUARES FIT BY ORTHOGONAL POLYNOMIALS

[G. J. Makinson, *Comm. ACM 10* (Feb. 1967), 87]

G. J. Makinson (Recd. 21 Mar. 1967)
University of Liverpool, Liverpool 3, England

The second sentence of the first comment should read "The weights should be provided inversely proportional to the square of the standard error of the observations."

instead of

"The weights should be provided inversely proportional to the standard error of the observations."

CERTIFICATION OF ALGORITHM 296 [E2] GENERALIZED LEAST SQUARES FIT BY ORTHOGONAL POLYNOMIALS [G. J. Makinson, *Comm. ACM 10* (Feb. 1967), 87]

Wayne T. Watson (Recd. 11 Feb. 1969 and 21 Mar. 1969)
Service Bureau Corp., Development Laboratory, 111 West St. John Street, San Jose, CA 95113

KEY WORDS AND PHRASES: least squares, curve fitting, orthogonal polynomials, three-term recurrence, polynomial regression, approximation, Forsythe's method
CR CATEGORIES: 5.13, 5.5

LSFITUW was compiled and tested in CALL/360:PL/I. No modifications were made to the algorithm, and the computations were made in long precision (about 15 significant floating point digits). In addition, *POLYX* [2] was used to transform the results of *LSFITUW* from the interval $(-2,2)$ to the interval (x_1, x_m).

To generally test the algorithm, several small sets of data were used with *LSFITUW* and the results were compared with those obtained from an independently written polynomial curve fitting algorithm which does not use the method of orthogonal polynomials. Only polynomials of degree less than 5 were used to fit the data. Agreement between coefficients and standard errors was good.

As a more comprehensive test of the algorithm, all experiments that could be duplicated from the article by Ascher and Forsythe [1] were performed; a slight modification to *LSFITUW* was required to transform the data to the interval $(-1,1)$ instead of $(-2,2)$. Briefly, the experiments included:

(1) For certain equally spaced data, a comparison of the α_i and β_i calculated by the program against those values of α_i and β_i obtained from known formulas ($\alpha_i = 0$ for equally spaced data).

(2) A fit of the function $f(x) = |x|$ over the interval $(-1,1)$ for equally spaced data for polynomials of degree as high as 30.

(3) A fit of the function $f(x) = e^x$ for unequally spaced data inside the interval $(-1,1)$ for polynomials of degree as high as 32.

The results of experiment (1) showed that *LSFITUW* produced values of β_i differing only in the last significant digit (15) from those calculated by the known formula. The values of α_i produced were in the range of the floating point round-off error (10^{-15}). The results of duplicating experiments (2) and (3) were better than those reported in [1] because of the greater precision used in the calculations (about 10.8 versus about 15 significant floating digits). While conducting the last two experiments, it was noted that for data values of x symmetric about the origin, the value of b in the transformation equation $x = at + b$ may be computed to be a number in the floating point round-off range instead of exactly zero. When fitting polynomials of a sufficiently high degree, this may cause an underflow at line 4 of *POLYX*, the transformation routine. The user may find it desirable to branch on an underflow in *POLYX* and reset b to zero.

To check the computations of the σ_k^2 obtained by the recursive definition of σ_k^2 used in the algorithm, the σ_k^2 were compared with results computed directly from the equation

$$\sigma_k^2 = \sum_{j=1}^{m} (f_j - y_k(x_j))^2/(m-k-1) \qquad (*)$$

where y_k is the best fitting polynomial of degree k for the data x_j, f_j. Experience with the algorithm indicates that a loss of accuracy in computing σ_k^2 occurs at smaller values of k when using the recursive definition than when using (*). If the values of σ_k^2 are of importance to the user, he may find it useful to compute them using (*) instead.

A comprehensive test of the algorithm's feature which uses the σ_k^2 to automatically select the best fitting polynomial was not made, but the feature did work properly for the polynomials used. In connection with this feature, the user should be aware, though, of the possible difficulty mentioned above in computing σ_k^2 accurately using the recursive definition. In this case, the user should not expect the algorithm to select the best fitting polynomial. This difficulty was experienced several times while testing the algorithm, but was circumvented by using (*) to calculate σ_k^2. In order to detect a possible loss in accuracy, the σ_k^2 should be examined carefully or compared with those obtained by (*).

Comprehensive tests were not made using weights; however, no problems were encountered with a moderate usage of this feature.

References:

1. Ascher, M., and Forsythe, G. E. SWAC experiments on the use of orthogonal polynomials for data fitting. *J. ACM 5* (Jan. 1958), 9–21.
2. MacKinney, John G. Algorithm 29, Polynomial transformer. *Comm. ACM 3* (Nov. 1960), 604.

ALGORITHM 297
EIGENVALUES AND EIGENVECTORS OF THE SYMMETRIC SYSTEM $(A-\lambda B)X = 0$ [F2]

J. Boothroyd (Recd. 19 Aug. 1965, 7 Feb. 1966, 1 Aug. 1966, and 14 Nov. 1966)
University of Tasmania, Hobart, Tas., Australia

```
procedure eigensolve(a, b, x, n, nondef); value n; integer n;
  label nondef; array a, b, x;
comment solves the equation (A − λB)X = 0 for symmetric
  A, B in a, b[1:n, 1:n] provided one of these is either positive or
  negative-definite. B is decomposed symmetrically so that B
  = LL′ and the equation transformed to (C−λI)Y = 0 where
  C = (L)⁻¹A(L′)⁻¹ is symmetric and Y = L′X. If B is negative-
  definite (A−(−λ)(−B))X = 0 is solved. If B is neither positive
  nor negative-definite the original equation is rearranged as
  (B−(1/λ)A) X = 0 and solved as such for positive-definite A or
  as (B−(−1/λ)(−A))X = 0 for negative-definite A. Failure to
  achieve one useful transformation from the four possibilities
  leads to exit via the label nondef.
    The procedure calls procedure symmetric QR 2 [P. A. Businger,
    Algorithm 254, Eigenvalues and eigenvectors of a real
    symmetric matrix by the QR method. Comm. ACM 8 (April,
    1965), 218–219] to evaluate the roots and vectors of
    (C−λI) Y = 0. That procedure leaves untouched the
    strictly upper triangle of C. In conformity with this,
    eigensolve preserves the strictly upper triangles of A and B.
    If, before entry to eigensolve, the user saves the diagonals
    of A, B, both these arrays may, if necessary, be fully
    restored after exit.
    On exit from the procedure the eigenvalues occupy the diago-
    nal elements a[i, i] with the eigenvectors in corresponding
    columns of x[1:n, 1:n];
begin integer i, j, k, jless1, iless1, adi, bdi;
  real t, sum, xij, length;
  Boolean recip;
  procedure LCHOLESKI(a, n, fail); value n; integer n;
    label fail; array a;
  comment performs the symmetric decomposition A = LL′
    for positive definite A in a[1:n, 1:n]. The lower triangle of A
    is overwritten by L. The strictly upper triangle of A is intact.
    For nonpositive-definite A the procedure exits via label pa-
    rameter fail;
  begin integer i, j, k, jless1;
    real ajj, ajk, aij;
    jless1 := 0;
    for j := 1 step 1 until n do
    begin ajj := a[j, j];
      for k := 1 step 1 until jless1 do
      begin ajk := a[j, k];
        ajj := ajj − ajk × ajk
      end;
      if ajj ≤ 0.0 then go to fail;
      ajj := a[j, j] := sqrt(ajj);
      for i := j + 1 step 1 until n do
      begin aij := a[i, j];
        for k := 1 step 1 until jless1 do
        aij := aij − a[i, k] × a[j, k];
        a[i, j] := aij/ajj
      end;
      jless1 := j
    end j
  end LCHOLESKI;
  comment scan diagonals of A, B setting adi, bdi respectively
    to +1, −1, 0 if the diagonal elements are all positive and non-
    zero, all negative or neither. Save the diagonal of B in X;
  adi := sign(a[1, 1]);
  x[1, 1] := t := b[1, 1];
  bdi := sign(t);
  for i := 2 step 1 until n do
  begin t := a[i, i];
    if t = 0.0 ∨ (t>0≡adi<0) then adi := 0;
    x[i, i] := t := b[i, i];
    if t = 0.0 ∨ (t>0≡bdi<0) then bdi := 0
  end;
  recip := false; comment prepare to solve (A−λB)X = 0;
  if bdi = 0 then go to swap; comment B is nondefinite;
  if bdi < 0 then
  begin comment prepare to solve (A−(−λ)(−B))X = 0;
    for i := 1 step 1 until n do
    for j := 1 step 1 until i do b[i, j] := − b[i, j]
  end;
newtry: LCHOLESKI(b, n, swap);
  go to ok;
swap: if recip then go to nondef;
  recip := true;
  comment prepare to solve (B−(1/λ)A)X = 0;
  if adi = 0 then go to swap; comment to escape, since A is
    also nondefinite;
  if adi < 0 then
  begin comment prepare to solve (B−(−1/λ)(−A))X = 0;
    for i := 1 step 1 until n do
    begin b[i, i] := a[i, i]; a[i, i] := x[i, i];
      for j := i + 1 step 1 until n do
      begin b[j, i] := −a[i, j]; a[j, i] := b[i, j] end
    end
  end
  else
  begin comment prepare to solve (B−(1/λ)A)X = 0;
    for i := 1 step 1 until n do
    begin b[i, i] := a[i, i]; a[i, i] := x [i, i];
      for j := i + 1 step 1 until n do
      begin b[j, i] := a[i, j]; a[j, i] := b[i, j] end
    end
  end;
  go to newtry;
  comment form C=(L)⁻¹A(L′)⁻¹ by LX = A, CL′ = X. C re-
    places A;
ok: jless1 := 0;
  for j := 1 step 1 until n do
  begin iless1 := 0;
    for i := 1 step 1 until j do
    begin sum := a[j, i];
      for k := 1 step 1 until iless1 do
      sum := sum− x[k, j] × b[i, k];
      sum := x[i, j] := sum/b[i, i];
      for k := 1 step 1 until jless1 do
```

```
        sum := sum − (if k≤i then a[i, k] else a[k, i]) × b[j, k];
      a[j, i] := sum/b[j, j];
      iless1 := i
    end;
    jless1 := j
  end;
  comment  global call of symmetric QR 2 to solve (C−λI) Y = 0.
    symmetric QR 2 includes a built-in precision tolerance. For
    use with eigensolve this constant should be changed to the
    value appropriate to whatever computer is used. Those in-
    terested in using JACOBI [Thomas G. Evans, Algorithm 85,
    JACOBI, Comm ACM 5 (April 1962), 208] in place of symmetric
    QR 2 may do so by copying the lower triangle of A to the upper
    triangle and making suitable changes to accommodate the
    parameter rho of that procedure before it is called. In this case
    the strictly upper triangle of A will not be preserved on exit
    from eigensolve;
  symmetric QR 2 (n, a, x);
  comment  change the Y vectors, now in x by L′ X = Y and
    normalize to unit length;
  for j := 1 step 1 until n do
  begin length := 0.0;
    for i := n step −1 until 1 do
    begin sum := x[i, j];
      for k := i + 1 step 1 until n do
        sum := sum −b[k, i] × x[k, j];
      xij := x[i, j] := sum/b[i, i];
      length := length + xij × xij
    end;
    length := sqrt(length);
    for i := 1 step 1 until n do x[i, j] := x[i, j]/length
  end;
  comment take the reciprocals and/or change the signs of the
    roots if necessary;
  for i := 1 step 1 until n do
  if recip then
  begin
    if adi <0 then a[i, i] := −1.0/a[i, i]
    else a[i, i] := 1.0/a[i, i] end
  else if bdi <0 then a[i, i] := −a[i, i]
end eigensolve
```

ALGORITHM 298
DETERMINATION OF THE SQUARE-ROOT OF A POSITIVE DEFINITE MATRIX [F1]

H. Späth (Recd. 20 Sept. 1966)
Institut für Neutronenphysik und Reaktortechnik
Kernforschungszentrum Karlsruhe, Germany

```
procedure WURZEL(A, B, N, theta, eps);
  value N, theta, eps;  integer N;  real theta eps;  array A, B;
comment  Let A be a symmetric positive-definite matrix of the
```

order N. Further let $\lambda_{\min}$ be the smallest and $\lambda_{\max}$ be the greatest eigenvalue of A.

It is known [1] that for all θ with $0 < \theta < 1$ the sequence

$$B_{k+1} = B_k + c(A - B_k{}^2), \qquad B_0 = 2cA \tag{1}$$

with

$$c = \frac{\theta}{2\sqrt{\lambda_{\max}}}$$

converges to $\sqrt{A}$. The rate of convergence of the above sequence is given by the rate of convergence to zero of the sequence

$$x_k = \left(1 - \theta\sqrt{\lambda_{\min}/\lambda_{\max}}\right)^k. \tag{2}$$

As $\| A \| = \alpha\lambda_{\max}$ with $\alpha \geq 1$, we set

$$c_1 = \frac{\theta_1}{2\sqrt{\|A\|}}.$$

(In the program we choose $\| A \| = \max_i \| \sum_k | a_{ik} |$). Then the sequence (1) with $c = c_1$ converges for all θ_1 with

$$0 < \theta_1 < \sqrt{\alpha} = \sqrt{\| A \| / \lambda_{\max}}$$

and therefore in any case for θ_1 with $0 < \theta_1 < 1$. Because of (2) it is favorable to choose θ close to 1 and θ_1 close to $\sqrt{\alpha}$, respectively. If nothing at all is known about α, the optimum is to choose θ_1 close to 1. The computing time is proportional to $f(\theta_1)N^3$, where $f(\theta_1)$ decreases as θ_1 increases.

Meaning of symbols in the formal parameter list:

$A = A[1{:}N, 1{:}N]$ must be symmetric and positive-definite.
A is not destroyed after leaving *WURZEL*
$B = B[1{:}N, 1{:}N]$ contains $\sqrt{A}$ when *WURZEL* is left
N is the order of A and B
theta $= \theta_1$ is an input parameter as described above
eps is an accuracy parameter. The iteration is stopped when

$$\max_{i,j} | b_{ij}^{(k+1)} - b_{ij}^{(k)} | < eps;$$

```
begin integer i, j, k;  real delta, s, c;  array bb[1:N];
  comment determination of c;  c := 0;
  for i := 1 step 1 until N do
  begin s := 0;
    for j := 1 step 1 until N do s := s + abs(A[i, j]);
    c := if c < s then s else c
  end;
  c := .5 × theta/sqrt(c);
  comment  now B0 is set;
  for i := 1 step 1 until N do
  for j := i step 1 until N do
    B[i, j] := B[j, i] := 2.0 × c × A [i, j];
  comment  start of iteration;
REPEAT: delta := 0;
  for i := 1 step 1 until N do
  begin for j := i step 1 until N do
    begin s := 0;
      for k := 1 step 1 until N do s := s − B[i, k] × B[k, j];
      bb[j] := B[i, j] + c × (A[i, j] + s)
    end;
    for j := i step 1 until N do
    begin s := abs(B[i, j] − bb[j]);
      if s > delta then delta := s;
      B[i, j] := bb[j]
    end
  end;
  for i := 1 step 1 until N − 1 do
  for j := i + 1 step 1 until N do B[j,i] := B[i, j];
  if delta > eps then go to REPEAT
end WURZEL
```

Reference:

1. Babuška, I., Práger, M., and Vitásek, E. *Numerical Processes in Differential Equations.* John Wiley & Sons, Ltd., London, 1966, p. 31 ff.

CERTIFICATION OF ALGORITHM 298 [F1] DETERMINATION OF THE SQUARE ROOT OF A POSITIVE DEFINITE MATRIX [H. Späth, *Comm. ACM 10* (Mar. 1967), 182]

B. J. Duke (Recd. 26 Apr. 1967, 16 July 1968 and 10 Oct. 1968)
Department of Chemistry, University of Lancaster, Bailrigg, Lancaster, England

KEY WORDS AND PHRASES: matrix, symmetric matrix, positive definite matrix, matrix square root
CR CATEGORIES: 5.14

Algorithm 298 has been tested in ICT Algol and used successfully on a number of matrices. One minor modification seems advisable. To avoid the procedure looping if an error occurs in its call, a maximum number of iterations should be set, with the procedure exiting through a label if this number is reached. The modifications to the procedure are obvious.

Comparisons with an alternative method using a binomial series are interesting. If

$$C = I - \frac{\theta}{\|A\|} A,$$

$$A^{\frac{1}{2}} = \left\{\frac{\|A\|}{\theta}\right\}^{\frac{1}{2}} \left\{I - \frac{1}{2} C - \frac{1}{8} C^2 - \frac{1}{16} C^3 \cdots \right\}.$$

For convergence,

$$\theta < 2 \| A \| / \lambda_{\max} ,$$

and thus a sufficient condition is $\theta < 2$.

Optimum convergence is for

$$\theta_{\text{opt}} = \frac{2\ \|A\|}{\lambda_{\max} + \lambda_{\min}}\ .$$

Thus

$$1 < \alpha < \theta_{\text{opt}} < 2\alpha$$

where $\alpha = \|A\|/\lambda_{\max}$. The choice of θ is difficult, as the method is particularly slow for values of θ not close to θ_{opt}. Unless other information is available, it seems preferable to choose θ in the range 1.4–1.8.

Both methods have been tested on over 30 positive definite matrices of order 2 to 12 arising from physical problems. In about half the cases studied all diagonal elements of A were equal; two typical examples are illustrated below. There was no significant difference between the behavior of these matrices and matrices with diagonal elements differing in magnitude.

(a)

$$A = \begin{pmatrix} 1.0 & 0.259952 & 0.03886876 & 0.01772265 & 0.03886876 \\ 0.259952 & 1.0 & 0.259952 & 0.03886876 & 0.01772265 \\ 0.03886876 & 0.259952 & 1.0 & 0.259952 & 0.03886876 \\ 0.01772265 & 0.03886876 & 0.259952 & 1.0 & 0.259952 \\ 0.03886876 & 0.01772265 & 0.03886876 & 0.259952 & 1.0 \end{pmatrix}$$

$$A^{\frac{1}{2}} = \begin{pmatrix} 0.9911413 & 0.1309132 & 0.0104918 & 0.0063647 & 0.0187119 \\ 0.1309132 & 0.9826457 & 0.1308604 & 0.0102163 & 0.0063647 \\ 0.0104918 & 0.1308604 & 0.9826144 & 0.1308604 & 0.0104918 \\ 0.0063647 & 0.0102163 & 0.1308604 & 0.9826457 & 0.1309132 \\ 0.0187119 & 0.0063647 & 0.0104918 & 0.1309132 & 0.9911413 \end{pmatrix}$$

(b)

$$A = \begin{pmatrix} 1.0 & 0.74917 & 0.48985 \\ 0.74917 & 1.0 & 0.74917 \\ 0.48985 & 0.74917 & 1.0 \end{pmatrix}$$

$$A^{\frac{1}{2}} = \begin{pmatrix} 0.9017878 & 0.3893683 & 0.1875400 \\ 0.3893683 & 0.8347366 & 0.3893683 \\ 0.1875400 & 0.3893683 & 0.9017878 \end{pmatrix}$$

In both methods iteration was continued until, at iteration k, the estimate of $A^{\frac{1}{2}}D^{(k)}$ changed by less than 10^{-7}, i.e.

$$|\ D_{ij}^{k} - D_{ij}^{k-1}\ | < 10^{-7} \text{ for all } i \text{ and } j.$$

Algorithm 298—No. of Iterations

	N	α	θ				
			0.8	0.9	0.95	0.999	1.05
(a)	5	1.054	22	18	17	16	14
(b)	3	1.071	60	52	49	47	44

Series Method—No. of Iterations

	N	θ_{opt}	θ										
			1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.0
(a)	5	1.59	21	19	17	16	14	13	15	19	26	37	67
(b)	3	2.006	118	108	100	93	87	81	77	72	69	65	63

The behavior of Algorithm 298 was found to be similar in all cases studied. The best choice of θ is as close to α as possible. Normally, 0.999 must be chosen. The performance of the series method is well illustrated by the two examples chosen. It is difficult to determine a good value of θ, and even if a value very close to θ_{opt} is accidentally used, the performance of the series method can be inferior to the method used in Algorithm 298.

The series method has one other disadvantage. For an efficient algorithm, several extra arrays are required as intermediate storage. The only clear advantage is that the series method can be readily modified for powers other than square root. Algorithm 298 is the most efficient method of the two.

ALGORITHM 299
CHI-SQUARED INTEGRAL [S15]

I. D. Hill and M. C. Pike (Recd. 9 Sept. 1965 and 3 Oct. 1966)
Medical Research Council, Statistical Research Unit, 115 Gower St., London W.C.1., England

real procedure *chiprob* (*x*, *f*, *bigx*, *normal*, *wrong*);
  value *x*, *f*, *bigx*; **real** *x*; **integer** *f*; **Boolean** *bigx*;
  real procedure *normal*; **label** *wrong*;
comment Finds the probability that χ^2, on f degrees of freedom exceeds x, i.e.,

$$\frac{1}{2^{\frac{1}{2}f}\Gamma(\frac{1}{2}f)}\int_x^\infty z^{\frac{1}{2}f-1}\, e^{-\frac{1}{2}z}\, dz \quad (x \geqq 0,\ f \geqq 1)$$

The algorithm is based upon the recurrence formula

$$P\ (\chi_f{}^2 > x) = P\ (\chi_{f-2}^2 > x) + \frac{(\frac{1}{2}\chi)^{\frac{1}{2}f-1}\ e^{-\frac{1}{2}x}}{\Gamma(\frac{1}{2}f)}$$

[*Handbook of Mathematical Functions*, National Bureau of Standards, Appl. Math. Series 55 (1964), formula 26.4.8] by means of which any χ^2-integral can be reduced to the sum of

(i) a series of terms that can be directly evaluated, and
(ii) a χ^2-integral on 2 degrees of freedom (if f is even), or on 1 degree of freedom (if f is odd).

To evaluate (ii) we have either

$$P\ (\chi_2{}^2 > x) = e^{-\frac{1}{2}x}$$

or

$$P\ (\chi_1{}^2 > x) = (2/\sqrt{2\pi}) \int_{\sqrt{x}}^\infty e^{-\frac{1}{2}z^2}\, dz.$$

The evaluation of the latter expression is performed by the formal **real procedure** *normal* which must evaluate the lower tail area of the standardized normal curve (**real procedure** *Gauss* [D. Ibbetson, Alg. 209, *Comm. ACM 6* (Oct. 1963), 616] may be used as the actual parameter).

The parameter *bigx* should be set to **true** if the value of x is too big for *exp* ($-0.5\times x$) to be accurately represented by the machine, or **false** otherwise.

For even degrees of freedom the method is exact, and the algorithm is essentially accurate to the accuracy of the machine. For odd degrees of freedom the accuracy will be dictated by the accuracy of the **real procedure** *normal*.

For large degrees of freedom, if speed is more important than great accuracy, it may be found preferable to use an approximation, e.g., the Wilson-Hilferty cubic formula [Wilson, E. B., and Hilferty, M. M., *Proc. Nat. Acad. Sci. 17* (1931), 684] which may be expressed as

chiprob := *normal* (−*sqrt* (4.5×*f*)×((*x*/*f*) ↑ (1/3)+2/(9×*f*)−1)).

This is accurate to 3 decimal places for $f > 40$.

The authors thank the referee and the editor for helpful criticisms and suggestions;

```
begin
  if x < 0 ∨ f < 1 then go to wrong else
  begin
    real a, y, s;
    Boolean even;
    a := 0.5 × x;  even := 2 × (f÷2) = f;
    if even ∨ f > 2 ∧ ¬bigx then y := exp(−a);
    s := if even then y else 2.0 × normal (−sqrt (x));
    if f > 2 then
    begin
      real e, c, z;
      x := 0.5 × (f−1.0);  z := if even then 1.0 else 0.5;
      if bigx then
      begin
        e := if even then 0 else 0.572364942925;  c := ln (a);
        comment 0.572364942925 = ln (sqrt(π));
        for z := z step 1.0 until x do
        begin
          e := ln (z) + e;
          s := exp(c×z−a−e) + s
        end;
        chiprob := s
      end else
      begin
        e := if even then 1.0 else 0.564189583548/sqrt(a);  c := 0;
        comment 0.564189583548 = 1/sqrt(π);
        for z := z step 1.0 until x do
        begin
          e := e × a/z;
          c := c + e
        end;
        chiprob := c × y + s
      end
    end else chiprob := s
  end
end chiprob
```

CERTIFICATION OF ALGORITHM 299 [S15]
CHI-SQUARED INTEGRAL [I. D. Hill and M. C. Pike, *Comm. ACM 10* (Apr. 1967), 243]

William M. O'Brien and Joan Wood (Recd. 17 Oct. 1967 and 1 Dec. 1967)
Department of Preventive Medicine, University of Virginia School of Medicine, Charlottesville, Virginia

KEY WORDS AND PHRASES: chi-squared integral, probability, special functions
CR CATEGORIES: 5.5, 5.12

Chi-Squared Integral compiled and ran in Burroughs B5500 Algol with the following revisions:

(i) *wrong* was removed from the formal parameter list;
(ii) **label** *wrong*; was removed from the specification part
(iii) the last two lines were modified to read:

```
        end,
    wrong:  end chiprob
```

These modifications were necessary since the heading of a typed procedure may not contain a label in Burroughs Extended ALGOL. [Editor's note: The question of whether a function procedure in ALGOL 60 may have a label as a formal parameter providing an exit from the procedure via a go to statement is not completely answered in the ALGOL 60 report. See D. E. Knuth, The remaining trouble spots in ALGOL 60, *Comm. ACM 10* (Oct. 1967), 611-618 (614). The use of *wrong* as a formal parameter in *chiprob* may cause trouble in many compilers. Perhaps the best way to handle the problem of error exits is to provide a formal parameter, *error*, which is a procedure name and let the user provide his own procedure for error recovery.—JGH].

bigx was set to **true** if $exp(-0.5\times x) < {}_{10}-10$ and Algorithm 209 [D. Ibbetson, Gauss, *Comm. ACM 6* (Oct. 1963), 616] was used for the formal **real procedure** *normal*.

The following were calculated:

Chi squared	*Degrees of freedom*
0.001	1 to 3
2.2	1 to 17
8.2	1 to 32
82.0	even values 34 to 70

The results were checked against E. S. Pearson and H. O. Hartley, *Biometrika Tables for Statisticians, vol. 1*, 2nd ed., Cambridge, 1962, pp. 122-129, which gives values of chi squared to five decimal places. The computer calculations, which were carried to nine places, gave identical results except in three instances, which were $\chi^2 = 2.2$ with df = 10, $\chi^2 = 8.2$ with df = 24, and $\chi^2 = 82$ with df = 38. In all three cases the sixth figure would have rounded to a 5 and the discrepancies appear to be due to inconsistencies in the rounding of the original *Biometrika Tables*, rather than errors in the procedure.

ACM Transactions on Mathematical Software, Vol. 2, No. 4, December 1976, Pages 393-395.

REMARK ON ALGORITHM 299

Chi-Squared Integral [S15]
[I.D. Hill and M.C. Pike, *Comm. ACM 10*, 4 (April 1967), 243]

Mohamed el Lozy, M.D. [Recd 20 May 1976 and 15 July 1976]
Department of Nutrition, Harvard School of Public Health, 665 Huntington Ave., Boston, MA 02115.

This work was supported in part by the Agency for International Development under Contract AID/afr-650 and in part by the Fund for Research and Teaching, Department of Nutrition, Harvard School of Public Health.

This algorithm suggests the use of the Wilson-Hilferty formula [3] if an approximation is desired for large degrees of freedom. Peizer and Pratt [2] have since then described a family of normal approximations far superior to the cube-root family, their formula for the chi-square distribution being [eqs. (2.24b) to (2.27) of their paper]:

$$z = d[(1 + g(s/x))/2x]^{1/2}$$

where z represents the corresponding normal deviate, x represents the chi-squared value, n represents the degrees of freedom, and

$$\begin{aligned} s &= n - 1 \\ d &= x - n + \tfrac{2}{3} - 0.08/n \\ g(t) &= (1 - t^2 + 2t \ln t)/(1 - t)^2, \quad t > 0,\ t \neq 1 \\ g(0) &= 1, \quad g(1) = 0. \end{aligned}$$

The two approximations were compared for degrees of freedom $n = 1\ (1)\ 20\ (5)\ 100\ (2)\ 200$ using, for each value of n, a grid of 500 chi-squared values uniformly distributed over the interval from $P = 0.00001$ to $P = 0.99999$. The "true" values of P were calculated using an IMSL subroutine, MDCHDI [1] which is essentially a double precision Fortran version of Algorithm 299 not using any approximation. Table I shows the maximum difference between the "true" results and those obtained with both approximations; the superiority of the Peizer and Pratt approximation is clear. For only 4 degrees of freedom it will give 3 correct decimals; for 11 degrees of freedom it will give 4 correct decimals; for 31 degrees of freedom it will give 5 correct decimals; and for 120 degrees of freedom it will give 6 correct decimals. In contrast, the Wilson-Hilferty approximation will give 3 correct decimals for 25 or more degrees of freedom, and calculations with a coarse grid show that 4 correct decimals are achieved somewhere between 200 and 300 degrees of freedom.

Since full word length accuracy is rarely, if ever, needed in the evaluation of the integrals of probability functions, it is suggested that for more than 30 degrees of freedom the Peizer and Pratt approximation be used in place of the iterative algorithm. There would appear to be no justification for using the Wilson and Hilferty approximation.

The calculations were done on an IBM 370/168 using double precision throughout, as it was desired to test the accuracy of the approximation without having to worry about inaccuracies due to the short word length of the machine. Single precision calculations gave almost identical results for the Wilson-Hilferty approximation. In the case of the Peizer and Pratt approximation very similar results were obtained up to about 30 degrees of freedom, after which the maximum error obtained with single precision was greater than that obtained with double precision, and never fell below .2E − 5. However, like the double precision version, the single precision routine gave 5 correct decimal places for 31 or more degrees of freedom.

Table I. Maximum Absolute Errors for the Wilson-Hilferty and Peizer-Pratt Approximations to the Chi-Squared Integral

Degrees of freedom	Maximum error using approximation of	
	Wilson-Hilferty	Peizer-Pratt
5	.26E−2	.33E−3
10	.13E−2	.58E−4
15	.82E−3	.22E−4
20	.61E−3	.12E−4
25	.48E−3	.73E−5
30	.39E−3	.50E−5
50	.23E−3	.19E−5
100	.11E−3	.55E−6
120	.92E−4	.41E−6
200	.54E−4	.18E−6

The use of the g function avoids inaccuracies that would arise if the simpler equation (2.24a) of [2] were used. In evaluating it, care must be taken near the two singularities. For 1 degree of freedom $s = 0$; so the argument to $g(t)$ will be zero and $g(t)$ must be set to 1. In the testing done, the smallest nonzero value of the argument to $g(t)$ was 0.04, which did not lead to any numerical problems. On the other hand, values very close to 1 were obtained, the smallest absolute difference from 1 being .14E − 4. In single precision at least such arguments can lead to great loss of accuracy; so for values of the argument close to 1 the power expansion given by Peizer and Pratt [2, eq. (10.3)] should be used:

$$g(t) = \sum_{j=1}^{j=\infty} 2(1 - t)^j/(j + 1)(j + 2).$$

It is not clear what the optimal value of the crossover point from the logarithmic to the power series form of $g(t)$ is, but in the single precision version we have used an absolute value of $(1 - t)$ less than 0.1 as the crossover criterion, taking the first 5 terms of the series.

REFERENCES

1. Library I Reference Manual, Vol. 2. Int. Math. and Stat. Libraries, 3rd edition, 1974.
2. Peizer, D.B., and Pratt, J.W. A normal approximation for binomial, F, beta, and other common, related tail probabilities, I. *J. Amer. Stat. Assn. 63* (1968), 1416–1456.
3. Wilson, E.B., and Hilferty, M.M. The distribution of chi-square. Proc. Nat. Acad. Sci., 1931, pp. 684–688.

ALGORITHM 300
COULOMB WAVE FUNCTIONS [S22]

J. H. Gunn (Recd. 19 Feb. 1965)
Nordisk Institut for Teoretisk Atomfysik
Blegdamsvej 15, Copenhagen, Denmark

```
procedure Coulomb(F, Fd, G, Gd, sig, rho, eta, lmax, exit);
  value rho, eta, lmax;
  real rho, eta; integer lmax; array F, Fd, G, Gd, sig; label
    exit;
comment The Coulomb wave functions $F_L$ and $G_L$ are defined
  as the two independent solutions of the differential equation

      $\frac{d^2y}{d\rho^2} + (1-2\eta/\rho-L(L+1)/\rho^2)y = 0$

  having the asymptotic behavior for large $\rho$

      $F_L \sim \sin\left(\rho-\eta \ln 2\rho-\frac{L}{2}\pi+\sigma_L\right),$

      $G_L \sim \cos\left(\rho-\eta \ln 2\rho-\frac{L}{2}\pi+\sigma_L\right)$

  where $\sigma_L = \arg \Gamma\ (i\eta+L+1)$. The procedure calculates for a
  given $\rho$ = rho and $\eta$ = eta, the functions $F_L$ and $G_L$ , their de-
  rivatives $F_L'$ and $G_L'$, and $\sigma_L$ for all $L$ from 0 up to lmax (>0) and
  places the results in the arrays F, G, Fd, Gd, sig respectively,
  which must have bounds 0:lmax. rho must lie in the range 5-30
  and eta in the range 0.1-30: values outside this range cause the
  procedure to leave via the label exit. This range is one that is
  often used in scattering and reaction problems in physics. De-
  tails of the methods used are to be found in: C. E. Fröberg,
  "Numerical treatment of Coulomb wave functions," Rev. Mod.
  Phys. 27 (1955), 399-411, and in: H. F. Lutz and M. D. Karvelis,
  "Numerical calculation of Coulomb wave functions for repulsive
  Coulomb fields," Nucl. Phys. 43 (1963), 31-44. The author grate-
  fully acknowledges the extensive help of Miss Margaret Wirt
  in the preparation of this procedure;
begin
  integer n; real rhom;
  comment jump to label exit if rho and eta lie outside range of
    procedure;
  if rho < 5 ∨ rho > 30 ∨ eta < 0.1 ∨ eta > 30 then
    go to exit;
  begin real sto; integer i;
    comment phase shifts $\sigma_L$ are calculated using formulae
      44-45 of Lutz and Karvelis;
    sto := 16 + eta ↑ 2;
    sig[0] := -eta + eta/2 × ln(sto) + 3.5 × arctan(eta/4) -
      (arctan(eta)+arctan(eta/2)+arctan(eta/3)) - eta/(12×sto) ×
      (1+1/30 × (eta↑2-48)/sto↑2 + 1/105
      × (eta↑4-160×eta↑2+1280)/sto↑4);
    for i := 1 step 1 until lmax do
      sig[i] := sig[i-1] + arctan(eta/i)
  end;
  if rho ≦ (5×eta-15)/3 ∨ rho ≦ eta then
  begin comment G[0] and Gd[0] are calculated using the Riccati
    method ($\rho<2\eta$) ref. formulae 9.1-9.4, Fröberg;
    integer i; real q, psi, psid, f; array g, gd[0:7], t, s[1:10];
    t[1] := rho/(2×eta); s[1] := 1 - t[1]; q := sqrt(t[1]×s[1]);
    for i := 2 step 1 until 10 do
    begin t[i] := t[1] × t[i-1];
      s[i] := s[1] × s[i-1]
    end;
    g[0] := q + arctan(t[1]/q) - 1.5707963;
    g[1] := 0.25 × ln(t[1]/s[1]);
    g[2] := -(8×t[2]-12×t[1]+9)/(48×q×s[1]);
    g[3] := (8×t[1]-3)/(64×t[1]×s[3]);
    g[4] := (2048×t[6]-9216×t[5]+16128×t[4]-13440×t[3]-12240
      ×t[2]+7560×t[1]-1890)/(92160×q×t[1]×s[4]);
    g[5] :=  3×(1024×t[3]-448×t[2]+208×t[1]-39)/(8192×t[2]×
      s[6]);
    g[6] = -(262144×t[10]-1966080×t[9]+6389760×t[8]-11714560
      ×t[7]+13178880×t[6]-9225216×t[5]+13520640×t[4]-
      3588480×t[3]+2487240×t[2]-873180×t[1]+130977)/(10321920
      ×q×t[2]×s[7]);
    g[7] := (1105920×t[5]-55296×t[4]+314624×t[3]-159552×t[2]
      +45576×t[1]-5697)/(393216×t[3]×s[9]);
    gd[0] := q/t[1];
    gd[1] := 0.25/(t[1]×s[1]);
    gd[2] := -(8×t[1]-3)/(32×q×t[1]×s[2]);
    gd[3] := 3×(8×t[2]-4×t[1]+1)/(64×t[2]×s[4]);
    gd[4] := -(1536×t[3]-704×t[2]+336×t[1]-63)/(2048×q×t[2]
      ×s[5]);
    gd[5] := 3×(2560×t[4]-832×t[3]+728×t[2]-260×t[1]+39)/
      (4096×t[3]×s[7]);
    gd[6] := (-368640×t[5]-30720×t[4]+114944×t[3]-57792×t[2]
      +16632×t[1]-2079)/(65536×q×t[3]×s[8]);
    gd[7] := 3×(860160×t[6]+196608×t[5]+308480×t[4]-177280×
      t[3]+73432×t[2]-17724×t[1]+1899)/(131072×t[4]×s[10]);
    f := 2 × eta; psi := psid := 0; q := -1;
    for i := 0 step 1 until 7 do
   begin psi := psi + q × f × g[i];
    psid := psid + q × f × gd[i];
    f := f/(2×eta); q := -q
   end;
   G[0] := exp(psi); Gd[0] := G[0] × psid/(2× eta); rhom :=
     rho
end else
if rho ≧ (30×eta+75)/13 ∧ rho < 2 × eta ↑ 2 then
begin comment G[0] and Gd[0] are calculated using the second
  Riccati method ($2\eta< \rho$) ref. formulae 9.6-9.8, Fröberg;
  integer i; real A, B, psi, phi, M, q; array x, y, e[1:10];
  x[1] := 2 × eta/rho; y[1] := 1 - x[1]; q := sqrt(y[1]); e[1]
    := 2 × eta;
  for i := 2 step 1 until 10 do
  begin x[i] := x[1] × x[i-1]; e[i] := e[1] × e[i-1];
    y[i] := y[1] × y[i-1]
  end;
  psi := -(8×x[3]-3×x[4])/(64×e[2]×y[3]) + 3 × x[5] ×
    (1024 - 448×x[1]+208×x[2]-39×x[3])/(8192×e[4]×y[6]) -
    x[7] × (1105920-55296×x[1]+314624×x[2]-159552×x[3]+
    45576×x[4]-5697×x[5])/(393216×e[6]×y[9]);
  phi := e[1] × (q/x[1] + 0.5 × ln((1-q)/(1+q))) + 0.7853982
    - (9×x[2]-12×x[1]+8)/(48×e[1]×q×y[1])
    - (2048-9216×x[1]+16128×x[2]-13440×x[3]-12240
    ×x[4]+7560×x[5]−1890×x[6])/(92160×e[3]×q×y[4])
    − (130977×x[10]−873180×x[9]+2487240×x[8]−3588480
    ×x[7]+13520640×x[6]−9225216×x[5]+15178880 × x[4]
    −11714560×x[3]+6389760×x[2]−1966080×x[1]
    +262144)/(10321920×e[5]×q×y[7]);
  A := q/x[2] + (8×x[1]−3×x[2])/(32×e[2]×q×y[2])
    − x[3] × (1536−704×x[1]+336×x[2]−63×x[3])/
    (2048×e[4]×q×y[5]) + x[5] × (368640−30720×x[1]
    +114944×x[2]−57792×x[3]+16632×x[4]− 2079×x[5])/
    (65536×e[6]×q×y[8]);
  B := 1/(4×e[1]×y[1]) − 3 × x[2] × (x[2]−4×x[1]+8)/
    (64×e[3]×y[4]) + 3 × x[4] × (2560−832×x[1]+728
    ×x[2]−260×x[3]+39×x[4])/(4096×e[5]×y[7]) − 3
    × x[6] × (1899×x[6]−17724×x[5]+73432×x[4]−177280
    ×x[3]+308480×x[2]+196608×x[1]+860160)/(131072
    ×e[7]×y[10]);
  M := sqrt(1/q) × exp(psi);
  G[0] := M × cos(phi);
  Gd[0] := −x[2] × (A×M×sin(phi)+B×G[0]);  rhom := rho
end else
if eta < 4 then
begin comment G[0] and Gd[0] are calculated using an asymp-
    totic expansion, ref. formulae 12.3–12.7, Fröberg;
  real ss, s1, tt, t1, SS, S1, TT, T1, sn, tn, Sn, Tn, An, Bn, theta,
    cth, sth;  integer i;
  rhom := if rho ≧ 2 × eta ↑ 2 then rho else 2 × eta ↑ 2;
  comment a suitable value of rhom is chosen for which the
    expansion is valid;
  ss := sn := 1;  tt := tn := 0;
  SS := Sn := 0;  TT := Tn := 1 − eta/rhom;
  for i := 0 step 1 until 10, 11, i + 1 while (abs(sn)>₁₀−7
    ×abs(ss)∨abs(tn)>₁₀−7×abs(tt)∨abs(Sn)>₁₀−7
    ×abs(SS)∨abs(Tn)>₁₀−7×abs(TT))∧(abs(sn)
    <abs(s1)∧abs(tn)<abs(t1)∧abs(Sn)<abs(S1)∧abs(Tn)
    <abs(T1)) do
  begin An := (2×i+1) × eta/(2×(i+1)×rhom);
        Bn := (eta↑2−i×(i+1))/(2×(i+1)×rhom);
        s1 := sn;  t1 := tn;  S1 := Sn;  T1 := Tn;
        sn := An × s1 − Bn × t1;
        tn := An × t1 + Bn × s1;
        Sn := An × S1 − Bn × T1 − sn/rhom;
        Tn := An × T1 + Bn × S1 − tn/rhom;
        ss := ss + sn;  tt := tt + tn;
        SS := SS + Sn;  TT := TT + Tn
  end;
  theta := −eta × ln(2×rhom) + rhom + sig[0];
  cth := cos(theta);  sth := sin(theta);
  G[0] := ss × cth − tt × sth;  Gd[0] := SS × cth − TT × sth
end else
begin comment G[0] and Gd[0] are calculated on the transition
    line for rhom = 2 × eta, ref. formulae 10.3–10.4, Fröberg;
  G[0] := 1.22340416 × eta ↑ (1/6) × (1+0.0495957017/eta↑(4/3)
    −0.0088888889/eta↑2+0.00245519918/eta↑(10/3)
    −0.000910895806/eta↑4+0.000253468412/eta↑(16/3));
  Gd[0] := −.707881773 × eta ↑ (−1/6) × (1−0.172826037/
    eta↑(2/3)+0.000317460317/eta↑2−0.00358121485/eta↑(8/3)
    +0.000311782468/eta↑4−0.000907396643/eta↑(14/3));
  rhom := 2 × eta
end;
if rhom ≠ rho then
begin comment Integrate the solutions G[0] and Gd[0] from
    the value of rhom at which they were evaluated to the value
    of rho required using Runge-Kutta formula;
  integer nh, i;  real k1, k2, k3, k4, k1p, k2p, k3p, k4p, y, yp,
    x, h;
  nh := entier(abs(rhom−rho)×10+1);
  h := (rho−rhom)/nh;
  x := rhom;  y := G[0];  yp := Gd[0];
  for i := 1 step 1 until nh do
  begin k1 := h × yp;  k1p := −h × (1−2×eta/x) × y;
    k2 := h × (yp+k1p/2);  k2p := −h × (1−2×eta/ (x+h/2))
    × (y+k1/2);
    k3 := h × (yp+k2p/2);  k3p := −h × (1−2×eta/ (x+h/2))
    × (y+k2/2);
    k4 := h × (yp+k3p);  k4p := −h × (1−2×eta/(x+h)) ×
    (y+k3); y := y + (k1+2×k2+2×k3+k4)/6;
    yp := yp + (k1p+2×k2p+2×k3p+k4p)/6;
    x := x + h
  end;
  G[0] := y;  Gd[0] := yp
end;
n := if rho > lmax then entier(rho+10) else lmax + 10;
begin comment Use downward recurrence relation (Millers
    method) and normalisation condition to obtain solutions
    F[L];
  array f[0:n];  real fd0, alpha, sto;  integer L;
  f[n] := 0;
  f[n−1] := 1;
  for L := n − 1 step −1 until 1 do
    f[L−1] := L/sqrt(eta↑2+L↑2) × (((2×L+1)×eta/
      (L×(L+1))+(2×L+1)/rho)× f[L]−sqrt(eta↑2
      +(L+1)↑2)/(L+1)×f[L+1]);
  fd0 := (eta+1/rho) × f[0] − sqrt(eta↑2+1) × f[1];
  G[1] := (−Gd[0]+(1/rho+eta)×G[0])/sqrt(1+eta↑2);
  alpha := 1/(sqrt(1+eta↑2)×(f[0]×G[1]−f[1]×G[0]));
  F[0] := alpha × f[0];
  Fd[0] := alpha × fd0;
  comment Upward recurrence relations for remaining
    solutions;
  for L := 0 step 1 until lmax−1 do
  begin F[L+1] := alpha × f[L+1];
    sto := sqrt(eta↑2+(L+1)↑2)/(L+1);
    Fd[L+1] := sto × F[L] − (eta/(L+1)+(L+1)/rho) ×
    F[L+1]; G[L+1] := 1/sto × ((eta/(L+1)+(L+1)/rho)
    ×G[L]−Gd[L]); Gd[L+1] := sto × G[L] − (eta/(L+1)+
    (L+1)/rho) × G[L+1]
  end
 end
end Coulomb
```

CERTIFICATION OF ALGORITHM 300 [S22] COULOMB WAVE FUNCTIONS [J. H. Gunn, *Comm. ACM 10* (Apr. 1967), 244]

K. S. Kölbig (Recd. 8 Feb. 1968)
Applied Mathematics Group, Data Handling Division,
European Organization for Nuclear Research (CERN),
1211 Geneva 23, Switzerland

KEY WORDS AND PHRASES: Coulomb wave functions, wave functions
CR CATEGORIES: 5.12

The procedure *Coulomb* was checked for a few parameter values using the Algol compiler of the CDC 3800 computer at CERN. It was found that for $\rho = \eta$ better results were obtained if the first line of the second **if** statement was altered to read:

```
if rho ≤ (5 × eta − 15)/3 ∨ rho < eta then
```

It was also necessary to correct a misprint in the first constant following the **comment** "G[0] and Gd[0] are calculated on the transition line for *rhom* = 2 × *eta*, ref. formulas 10.3–10.4, Fröberg." The line following this **comment** should read:

```
G[0] := 1.223404016 × eta ↑ (1/6) × (1 + 0.0495957017/eta ↑ (4/3))
```

The procedure was then translated into FORTRAN and tested in more detail on a CDC 6600 computer. The tests included the following:

(i) Generation of $\Phi_L(\eta,\rho) = [C_L(\eta)\rho^{L+1}]^{-1} F_L(\eta,\rho)$, $L = 0(1)21$ for $\eta = 1(1)5$, $\rho = 5$. The results were compared with values tabulated in [1]. In most cases, 6 to 7 significant digits agreed, except for $\eta = 1$, where agreement was found to 3 to 4 significant digits. It is interesting to compare some results for $\rho = \eta = 5$ obtained with and without the first of the above corrections:

$L \diagdown \Phi_L$	Without correction	With correction	Table [1] and Gautschi [2]
0	$6.554097_{10}3$	$6.552297_{10}3$	$6.552292_{10}3$
5	$1.865738_{10}1$	$1.865226_{10}1$	$1.865225_{10}1$
10	$5.354953_{10}0$	$5.353482_{10}0$	$5.353478_{10}0$
20	$2.440859_{10}0$	$2.440188_{10}0$	$2.440187_{10}0$

(ii) Computation of $F_0(\eta,\rho)$, $F_0'(\eta,\rho) = (d/d\rho)F_0(\eta,\rho)$ for $\eta = 2(2)12$, $\rho = 5(5)30$. Comparison with the table of Tubis [3] revealed frequent discrepancies of 1 (occasionally 2) units of the fifth significant digit. However, disagreement was observed in many fewer cases when comparing the calculated results with those obtained by Gautschi's algorithm [2].

(iii) Computation of $F_0(\eta,\rho)$, $F_0'(\eta,\rho)$, $G_0(\eta,\rho)$, and $G_0'(\eta,\rho)$ for $\rho = 2\eta$, $\rho = 5(.5)20(2)30$. Comparing the results with the table of Abramowitz and Rabinowitz [4] or with the values obtained with Gautschi's algorithm, the following discrepancies were found in units of the seventh decimal place:

F_0 —frequently 1, occasionally 2, units for $\rho \leq 10$;
F_0'—frequently 1 unit for $\rho \leq 8.5$;
G_0 —for $\rho \leq 8$ up to 40 units, for $8 < \rho \leq 14.5$ up to 2 or 3 units;
G_0'—for $\rho \leq 7.5$ up to 13 units.

(iv) Calculation of $G_0(\eta,\rho)$, $G_0'(\eta,\rho)$ for $\eta = .5(.5)20$, $\rho = 5(1)20$. The results have been compared with the tables given by Abramowitz [5]. Agreement was found in most cases to 5 significant digits. Discrepancies of 1, occasionally more, units of the fifth significant digit were found, mainly for arguments near a line separating two methods used in the algorithm. In some cases (in the immediate neighborhood of a zero of G_0 or G_0') there was agreement to only 2 or 3 significant digits.

(v) Generation of $F_L(\eta,\rho)$, $F_L'(\eta,\rho)$, $G_L(\eta,\rho)$, $G_L'(\eta,\rho)$, $\sigma_L(\eta)$ for $L = 0(1)10$, $\rho = 5,10$, $\eta = 1(1)5,10,25$. As a first step, the results were compared with values given in a table by Lutz and Karvelis [6]. Since important discrepancies were noted for $\eta = 1$, $\rho = 5$ and $\eta \geq 4$, the values for F_L and F_L' were also calculated by Gautschi's algorithm, known to be correct by checking it against the table [1]. Lutz and Karvelis give 6 significant digits, but without commenting on a possible error tolerance. They state, "we test [the generated functions] to see how closely the Wronskian relation $F_L'G_L - F_LG_L' = 1$ is obeyed." Comparison of their values with those obtained from Gautschi's algorithm shows, for $\eta < 4$, occasional discrepancies of 1 unit in the sixth significant digit. For $\eta \geq 4$ [disregarding some obvious misprints, e.g. for $G_1(2,10)$ and $G_{10}'(10,10)$] there are discrepancies which in a few cases exceed a 100 units in the sixth significant digit. Because of this, the table of Lutz and Karvelis was used for checking the procedure *Coulomb* only for $\eta < 4$. For $\eta \geq 4$ check values were obtained from Gautschi's algorithm (F_L and F_L' only). The following discrepancies were found in units of the sixth significant digit:

$\eta = 1, \rho = 5$: F_L—up to 119 units ($L = 8$).
F_L'—up to 87 units ($L = 0$).
G_L—up to 350 units ($L = 2$).

G_L'—up to 247 units ($L = 0$).

$\eta = 1, \rho = 10$;
$\eta = 2,3$: 1 or 2 units in several cases, exceptionally more; one isolated case $G_3(3,10)$ with 23 units. Comparison with Gautschi's values (where possible) gives better agreement.

$\eta \geq 4$: Occasionally 1 unit for F_L and F_L' .

$\sigma_L(\eta)$ nearly always agreed to 6 significant digits for all tested η.

To complete the check, values of the functions at $\eta = 1$, $\rho = 5$, and $\eta = \rho = 5$ were calculated using the ALGOL procedure. The results agreed with those calculated by the FORTRAN program to the 6 significant digits which were compared.

REFERENCES:

1. NATIONAL BUREAU OF STANDARDS. *Tables of Coulomb Wave Functions, Vol. I.* Appl. Math. Ser. 17, U.S. Govt. Printing Office, Washington, D.C., 1952.
2. GAUTSCHI, W. Algorithm 292. Regular Coulomb wave functions. *Comm. ACM 9* (Nov. 1966), 793–795.
3. TUBIS, A. *Tables of Nonrelativistic Coulomb Wave Functions.* LA-2150, Los Alamos Sci. Lab., Los Alamos, New Mexico, 1958.
4. ABRAMOWITZ, M., AND RABINOWITZ, P. Evaluation of Coulomb wave functions along the transition line. *Phys. Rev. 96* (1954), 77–79.
5. ——, AND STEGUN, I. A. (Eds.) *Handbook of Mathematical Functions.* NBS Appl. Math. Ser. 55, U.S. Govt. Printing Office, Washington, D.C., 1965.
6. LUTZ, H. F., AND KARVELIS, M.D. Numerical calculation of Coulomb wave functions for repulsive Coulomb fields. *Nucl. Phys. 43* (1963), 31–44.

REMARK ON ALGORITHM 300 [S22] COULOMB WAVE FUNCTIONS [J. H. Gunn, *Comm. ACM 10* (Apr. 1967), 244]; CERTIFICATION OF ALGORITHM 300 [K. S. Kölbig, *Comm. ACM 12* (May 1969), 279]

K. S. KÖLBIG (Recd. 14 Apr. 1969)
Data Handling Division, European Organization for Nuclear Research (CERN), 1211 Geneva 23, Switzerland

KEY WORDS AND PHRASES: Coulomb wave functions, wave functions, special functions, function evaluation
CR CATEGORIES: 5.12

Recently, Isacson [1] pointed out that the coefficient of $\eta^{-16/3}$ in the known asymptotic expansion for the irregular Coulomb wave function $G_0(\eta, \rho)$ on the transition line $\rho = 2\eta$ was erroneous.

In addition, he gave the expansions for F_0, G_0, F_0' and G_0' up to order η^{-8}, whereas the old expansions were given to order $\eta^{-16/3}$ only.

Therefore, and for reasons of speed, the relevant part of Algorithm 300 should be changed as follows:

```
begin comment G[0] and Gd[0] are calculated on the transition
  line for rhom = 2 × eta, ref. Isacson in remark;
  array et[1:12];  real et1;
  et[1] := eta ↑ (−2/3);
```

```
    for i := 2 step 1 until 12 do et[i] := et[1] × et[i−1];
    et1 := eta ↑ (1/6);
    G[0] := 1.223404016 × et1 × (1 + 0.04959570165 × et [2]
    −0.008888888889 × et [3] + 0.002455199181 × et [5]
    −0.0009108958061 × et [6] + 0.0008453619999 × et [8]
    −0.0004096926351 × et [9] + 0.0007116506205 × et [11]
    −0.00002439615603 × et [12]);
    Gd[0] := (−0.7078817734/et1) × (1 − 0.1728260369 × et [1]
    + 0.0003174603174 × et [3] − 0.003581214850 × et [4]
    +0.0003117824680 × et [6] − 0.0009073966427 × et [7]
    +0.0002128570749 × et [9] − 0.0006215584171 × et [10]
    +0.00003685244766 × et [12]);
    rhom := 2 × eta
end;
```

Furthermore, it was found in this connection that replacing the first line of the fourth *if* statement of the algorithm by

if *eta* < 4 ∧ *eta* < *rho*/2 **then**

gives, together with the above expansions, better results for $\rho = 2\eta$ in test (iii) and for $\rho = 3, \eta = 5$ in test (i) of the Certification.

The relevant statements in test (iii) of the Certification should therefore be replaced by the following ones:

F_0 − 1 unit for $\rho = 5$, $\rho = 6$, and $\rho = 8.5$.
F_0' − 1 unit for $\rho = 6$.
G_0 − 1 unit for $\rho = 5.5$, $\rho = 16$, and $\rho = 30$.
G_0' − 1 unit for $\rho = 5.5$.

REFERENCE:

1. ISACSON, T. Asymptotic expansion of Coulomb wave functions on the transition line. *BIT 8* (1968), 243–245.

Remark 2 on Algorithm 300 [S22]

Coulomb Wave Functions [J.H. Gunn, *Comm. ACM 10* (Apr. 1967), 244]; Certification of Algorithm 300 [K.S. Kölbig, *Comm. ACM 12* (May 1969), 279)]; Remark on Algorithm 300 [K.S. Kölbig, *Comm. ACM 12* (Dec. 1969), 692].

H. Vos [Recd. 9 Aug. 1971 and 8 Feb. 1972]
Natuurkundig Laboratorium der Vrije Universiteit,
Amsterdam, The Netherlands

Key Words and Phrases: Coulomb wave functions, wave functions, special functions, function evaluation
CR Categories: 5.12

The procedure *Coulomb* can be used very well to generate the Coulomb wave functions F_L and G_L and their derivatives, needed in elastic scattering calculations in nuclear physics. When the procedure is used many times for many values of *rho* and *eta*, it is not only very useful but also necessary to have in each instance an indication about the accuracy of the results. It is obvious to use the Wronskian relations $F_L'G_L - F_LG_L' \equiv 1$ for the purpose of checking the results, as Fröberg [1] states after formula (3.4). However, one has to be very careful in using these relations. The most significant check is given later on, but first it is shown what can go wrong.

This investigation was part of the research program of the "Stichting voor Fundamental Onderzoek der Materie (F.O.M.)," which is financially supported by the "Nederlandse Organisatie voor Zuiver Wetenschappelijk Onderzoek (Z.W.O.)".

Kölbig pointed out already in the certification that Lutz and Karvelis [2] failed to notice discrepancies exceeding 100 units in the sixth significant digit in their tables although they state "when all the functions are generated we test to see how closely the Wronskian relation $F_L'G_L - F_LG_L' = 1$ is obeyed." The way Lutz and Karvelis generate the functions goes as follows. First they calculate G_0 and G_0'; then they use recurrence relations to get G_L and G_L' for $L > 0$; and lastly them use backward recurrence relations together with the relation $F_0G_1 - G_0F_1 = (\eta^2 + 1)^{-\frac{1}{2}}$ to get F_L and F_L' for all L. This last relation is in fact a different form of the Wronskian relation, see e.g. Fröberg [1] formula (3.5). The use of the Wronskian relations to check the results now gives information only about the stability in the use of the recurrence relations, not about the accuracy of the Coulomb wave functions.

As an independent check on the function values, the following procedure can be used. It is easy to calculate F_0 and F_0' directly, that is in the same way as G_0 and G_0' are calculated (see Fröberg [1] and Isacson [3]). We call the results F_0 (*dir*) and F_0' (*dir*). These values can be compared with the F_0 (*rec*) and F_0' (*rec*) calculated via the recurrence relations, Wronskian relation, and G_0 and G_1 as in the procedure *Coulomb*. This direct test has to be preferred above a test via the Wronskian relation for the direct results G_0F_0' (*dir*) − $G_0'F_0$ (*dir*) = 1 because errors in F_0 (*dir*) and F_0' (*dir*) sometimes cancel in the Wronskian. The other Wronskian relations (i.e. for $L > 0$ and F_0 (*rec*) and F_0' (*rec*)) are hardly needed as a test because they only check the recurrence relations used. The experience is that errors herein are completely negligible (always less than one unit in the tenth digit for all values of L for the 12-digit EL-X8 computer of the Mathematisch Centrum in Amsterdam).

To include this check, Algorithm 300 should be changed as follows:

1. The line following the first **begin** should read

```
integer n; real rhom, q;
```

2. The line following the fourth **comment** ($G[0]$ and $Gd[0]$ are calculated using the Riccati method ($\rho < 2\eta$) formulas 9.1–9.4, Fröberg;) should be altered, according to Fröberg [1] formulas (9.1) and (9.2), to read:

```
integer i; real q, psi, psid, phi, phid, f; array g, gd[0:7], t,
   s[1:10];
```

3. The relevant lines after the statement starting with $gd[7]$:= . . . should read:

```
f := 2 × eta; psi := psid := phi := phid := 0; q := −1;
for i := 0 step 1 until 7 do
begin psi := psi + q × f × g[i];
   psid := psid + q × f × gd[i];
   phi := phi + f × g[i];
   phid := phid + f × gd[i];
   f := f/2/eta; q := −q
end;
G[0] := exp(psi); Gd[0] := G[0] × psid/2/eta;
F[0] := 0.5 × exp(phi); Fd[0] := F[0] × phid/2/eta;
rhom := rho;
```

4. The line just before the fourth **if** statement (**if** *eta* < 4 ∧ *eta* < *rho*/2 **then**), i.e. **end else,** should according to Fröberg formula (9.8) be replaced by:

```
  ;F[0] := M × sin(phi);
  Fd[0] := −x[2] × (B×F[0]−A×G[0])
end else
```

5. Insert after the last line of the calculation using an asymptotic expansion, just before the third **end else,** according to Fröberg formula (12.7) , the following lines:

```
  F[0] := tt × cth + ss × sth;
  Fd[0] := TT × cth + SS × sth;
```

6. The two statements after the line

```
et1 := eta ↑ (1/6);
```

i.e. $G[0]$:= . . . , and $Gd[0]$:= . . . , should be replaced by:

```
q := 1;
here 1:
  G[0] := 1.223 404 016 × et1 × (1+ q × 0.049 595 701 65 × et[2]
        −0.008 888 888 889×et[3]+q×0.002 455 199 181
        ×et[5]−0.000 910 895 806 1×et[6]+q
        ×0.000 845 361 999 9×et[8]−0.000 409 692 635 1
        ×et[9]+q×0.000 711 650 620 5×et[11]
        −0.000 024 396 156 03×et[12]);
  Gd[0] := (−q×0.707 881 773 4/et1) × (1−q
        ×0.172 826 036 9×et[1]+0.000 317 460 317 4
        ×et[3]−q×0.003 581 214 850×et[4]
        +0.000 311 782 468 0×et[6]−q×0.000 907 396 642 7
        ×et[7]+0.000 212 857 074 9×et[9]−−q
        ×0.000 621 558 417 1×et[10]+0.000 036 852 447 66
        ×et[12]);
  if q < 0 then begin
    q := + 1;
    F[0] := G[0] × 0.706 332 637 3 / 1.223 404 016;
    Fd[0] := Gd[0] × 0.408 695 732 3 / 0.707 881 773 4;
    go to here 1
    end;
```

7. Replace the line

```
x := rhom; y := G[0]; yp := Gd[0];
```

after **comment** Integrate the solutions $G[0]$ . . . by the lines

```
x := rhom; y := G[0]; yp := Gd[0]; q := +1;
here 2:
```

8. Replace the line following the next **for** statement; i.e.

```
G[0] := y; Gd[0] := yp
```

by the lines

```
if q > 0 then
begin G[0] := y; Gd[0] := yp; q := −1;
  y := F[0]; yp := Fd[0]; x := rhom; go to here 2
end else
begin F[0] := y; Fd[0] := yp
end;
```

9. Insert after the next **end**; before the line

n := **if** *rho* > *lmax* **then** . . . the following lines:

```
outreal (F[0]); outreal (Fd[0]); outreal
  (Fd[0]×G[0]−F[0]×Gd[0]);
comment F0 (direct), F0' (direct) and the Wronskian for the
  direct results W(direct) are printed;
```

10. Insert just before the **comment** (Upward recurrence relations for remaining solutions) the lines:

```
outreal (F[0]); outreal (Fd[0]);
comment F0 (rec) and F0' (rec) are printed;
```

The tests of the procedure *Coulomb* with these changes included all the computations mentioned in the Certification except those under (ii), and those in the Remark. The tests gave the same results as in the Certification and in the Remark. Moreover the following results were obtained:

The maximum M of the absolute differences

$$M = max\,(|\,[F_0(dir)-F_0(rec)]\,/\,F_0(rec)\,|\,,\quad |\,[F_0'(dir)--F_0'(rec)]\,/\,F_0'(rec)\,|\,)$$

was always greater than the absolute difference between the Wronskian for the direct results $W(dir)$ and 1; i.e.

$M \geq \Delta W = |\,1-W(dir)\,|$.

In some cases $W(dir)$ differed not significantly from 1, while the test with M indicated considerable discrepancies (see Table I, ρ,η = 6,1.5; 7,3.5 and 19,5.5). It was found that for all discrepancies stated by Kölbig in the Certification and in the Remark, the relative error was smaller than or of the same order as M, so M gives a good indication about the accuracy of the results (see Table ρ,η = 7,3.5(cert) and 7,3.5(remark)). So discrepancies of several units in the fourth or fifth significant digit were found near some lines in the (ρ,η) plane separating two methods used in the Algorithm: namely, the lines $\rho = \eta$ for $5 \leq \rho \leq 7.5$, $5\eta = 3\rho + 15$, $30\eta = 13\rho - 75$ and $\eta = 4$, where integration of the Coulomb wave functions from the transition line to the desired arguments turned out to be the best method (see e.g. Table ρ, η = 5,5.5). In some cases in the neighborhood of a zero of G_0 or G_0', the check with M indicated discrepancies in the third or fourth significant digit (see e.g. Table ρ, η = 19,5.5).

These examples show that when the procedure *Coulomb* is used as a standard procedure in calculations where an accuracy of three or more digits is required, it is necessary to have in each instance an indication about the accuracy of the results. The quantity M introduced above can be used very well for such a check.

Acknowledgment. We would like to thank Prof. Dr. C.C. Jonker for valuable discussions and comments.

Table I.

ρ	η	$\Delta W\times10^6$	$M\times10^6$				*Tabulated*
5	5.5	128	200	G_0	=	.38701(+2)	.38704(+2)
6	1.5	1.3	50	G_0	=	−.60187	−.60177
6	2	14.6	20	G_0	=	.57306(−1)	.57313(−1)
7	3.5*	.4	5	G_0	=	1.520489	1.520492
7	3.5†	.007	.05	G_0	=	1.5204917	1.520492
19	5.5	5	2000	G_0	=	−.16442	−.16427

* Certification.
† Remark.

References

1. Fröberg, C.E. Numerical treatment of Coulomb wave functions. *Rev. Mod. Phys. 27* (1955), 399-411.
2. Lutz, H.F., and Karvelis, M.D. Numerical calculation of Coulomb wave functions for repulsive Coulomb fields. *Nucl. Phys. 43* (1963), 31-44.
3. Isacson, T. Asymptotic expansion of Coulomb wave functions on the transition line. *BIT* (Nordisk Tidskrift for Informations-Behandling) *8* (1968), 243-245.

ALGORITHM 301
AIRY FUNCTION [S20]

GILLIAN BOND AND M. L. V. PITTEWAY

(Recd. 7 Apr. 1966 and 19 Oct. 1966)

Cripps Computing Centre, University of Nottingham, England

```
procedure Airy (Ai, Bi, Aid, Bid, x, xia, control);
  value x, xia, control;  real Ai, Aid, Bi, Bid, x, xia;
  integer control;
comment This procedure evaluates the real Airy functions and
  their derivatives by solution of the differential equation y″ = xy.
  The solutions Ai and Bi satisfy the Wronskian relation Ai Bi′ −
  Bi Ai′ = 1/π. Ai decreases exponentially for large positive
  values of x. For large negative values of x, Ai and Bi have simi-
  lar amplitudes but differ by π/2 in phase.
    The solution is tabulated in the interval −6.6 < x < 6.6 by
  Taylor integration of the differential equation in the stable
  directions (towards negative x for Ai and away from the origin
  for Bi) with step size 0.1. Alternate values are stored using 268
  locations so that any point is within Taylor range for subse-
  quent interpolation in the table. Asymptotic series are used
  outside this range. The solutions are accurate to eight decimal
  figures.
    For extensive use, computation times can be reduced by ex·
  tending the tabular range to −10 < x < 10 and changing the
  step size to 0.05, using 804 locations. The coefficients A[7] to
  A[10] may then be dropped from the asymptotic series, and
  tor [9] and tor [10] from the Taylor series (J. C. P. Miller, The
  Airy Integral, British Association Mathematical Tables, part-
  volume B, Cambridge, 1946).
    The operation of the procedure is controlled by the integer
  code. A negative value should be assigned to code to set up the
  Airy function tables on the first call for the procedure, or when-
  ever the tables have been disturbed. A subsequent entry with
  code greater than 0 will form:

    Ai = exp(xia) × Ai(x)        Aid = exp(xia) × Ai′(x)

    Bi = exp(−xia) × Bi(x)       Bid = exp(−xia) × Bi′(x)

  If the derivatives are not required, code should be set to zero.
  This will avoid asymptotic series calculations, but Aid and Bid
  are set if |x| < 6.6 even if code = 0;
begin
  real rtmdx, xi, factor, p, q, scale, s, c, xtab, h, pi;
  integer n, r, j;
  array A[0:10];
  own real array Aitab, Bitab, Aidtab, Bidtab[−33:33];
  procedure Taylor(y1, derivy1, x, h, y, derivy);
    value x, h, y, derivy;  real y1, derivy1, x, h, y, derivy;
  comment Calculates y(x+h) from y(x) by series expansion of
    dy^2/dx^2 = xy;
  begin
    real square;
    array tor[0:10];
    integer n;
    if h = 0 then
    begin
      y1 := y;
      derivy1 := derivy;
      go to zerostep
    end shortcut
    else
    begin
      tor[0] := y;
      tor[1] := h × derivy;
      square := h × h;
      tor[2] := 0.5 × square × x × tor[0];
      y1 := tor[0] + tor[1] + tor[2];
      derivy1 := tor[1] + 2 × tor[2];
      for n := 3 step 1 until 10 do
      begin
        tor[n] := square × (x×tor[n−2]+h×tor[n−3])/
          ((n−1)×n);
        y1 := y1 + tor[n];
        derivy1 := derivy1 + n × tor[n]
      end;
      derivy1 := derivy1/h
    end calculation of coefficients in series expansion;
zerostep:
end Taylor;
  pi := 3.14159 26536;
  if control < 0 then
  begin
    Bitab[0] := 0.61492 66274;
    Bidtab[0] := 0.44828 83574;
    Aitab[33] := 2.15659 99525₁₀ − 6;
    Aidtab[33] := −5.61931 9442₁₀ − 6;
    xtab := 0;
    for n := 0 step 1 until 32 do
    begin
      Taylor(Bi, Bid, xtab, 0.1, Bitab[n], Bidtab[n]);
      Taylor(Bitab[n+1], Bidtab[n+1], xtab+0.1, 0.1, Bi, Bid);
      Taylor(Bi, Bid, −xtab, −0.1, Bitab[−n], Bidtab[−n]);
      Taylor(Bitab[−n−1], Bidtab[−n−1], −xtab−0.1, −0.1, Bi,
        Bid);
      xtab := xtab + 0.2
    end setting up Bi tables;
    for n := 33 step −1 until −32 do
    begin
      Taylor(Ai, Aid, xtab, −0.1, Aitab[n], Aidtab[n]);
      Taylor(Aitab[n−1], Aidtab[n−1], xtab−0.1, −0.1, Ai, Aid);
      xtab := xtab − 0.2
    end setting Ai tables
  end;
  if abs(x) ≦ 6.6 then
  begin
    j := 5 × x;
    xtab := j/5;
    h := x − xtab;
    scale := exp(−xia);
    Taylor(Ai, Aid, xtab, h, Aitab[j], Aidtab[j]);
    Taylor(Bi, Bid, xtab, h, Bitab[j], Bidtab[j]);
    Ai := Ai/scale;
    Aid := Aid/scale;
    Bi := Bi × scale;
    Bid := Bid × scale;
    go to finish
    end interpolation in previously established table;
    rtmdx := sqrt(abs(x));
    xi := rtmdx ↑ 3/1.5;
    factor := 1/(12×xi);
    A[0] := 1/sqrt(pi×rtmdx);
    r := 6;
    for n := 0 step 1 until 9 do
     begin
       A[n+1] := (r−1) × (r−5) × factor × A[n]/r;
       r := r + 6
     end calculation of asymptotic series coefficients;
     if x < 0 then go to neg;
     p := A[0] + A[2] + A[4] + A[6] + A[8] + A[10];
     q := A[1] + A[3] + A[5] + A[7] + A[9];
     scale := exp(xi−xia);
     Ai := (p−q)/(2×scale);
     Bi := (p+q) × scale;
     go to continue;
  neg:
     p := A[0] − A[2] + A[4] − A[6] + A[8] − A[10];
     q := A[1] − A[3] + A[5] − A[7] + A[9];
     s := sin (xi+pi/4);
     c := cos(xi+pi/4);
     scale := exp(−xia);
     Ai := (p×s−q×c)/scale;
     Bi := (p×c+q×s) × scale;
  continue:
     if control = 0 then go to finish
     else if x < 0 then
     begin
       p := −(rtmdx/xi) ×
           (−2×A[2]+4×A[4]−6×A[6]+8×A[8]−10×A[10]);
       q := −(rtmdx/xi) ×
           (A[1]−3×A[3]+5×A[5]−7×A[7]+9×A[9]);
       Aid := −(rtmdx×Bi)/(scale×scale)−Ai/(4×x)
           − (p×s−q×c)/scale;
       Bid := rtmdx × Ai × scale × scale − Bi/(4×x)
           − (p×c+q×s) × scale;
       go to finish
    end calculation of derivatives;
    p := (rtmdx/xi) ×
         (2×A[2]+4×A[4]+6×A[6]+8×A[8]+10×A[10]);
    q := −(rtmdx/xi) ×
         (A[1]+3×A[3]+5×A[5]+7×A[7]+9×A[9]);
    Aid := (p−q)/(2×scale) − Ai × (rtmdx+1/(4×x));
    Bid := (p+q) × scale + Bi × (rtmdx−1/(4×x));
  finish:
  end Airy
```

REMARK ON ALGORITHM 301 [S20]
AIRY FUNCTION [Gillian Bond and M.L.V. Pitteway, *Comm. ACM 10* (May 1967), 291]

M.L.V. Pitteway (Recd. 19 May 1967)
Brunel University, ACTON, W.3., England

The initial minus sign has been omitted from the line immedi ately following the line

```
                    end calculation of derivatives;
```

The statement should read

```
p:= − (rtmdx/xi) × (2 × A[2] + 4 × A[4] + 6 × A[6]
                                        + 8 × A[8] + 10 × A[10]);
```

ALGORITHM 302
TRANSPOSE VECTOR STORED ARRAY [K2]

J. Boothroyd (Recd. 12 Sept. 1966, 28 Nov. 1966, and 6 Feb. 1967)

U. of Tasmania, Hobart, Tas., Australia

procedure *transpose*(*a*, *m*, *n*); **value** *m*, *n*; **integer** *m*,*n*; **array** *a*, **comment** performs an in-situ transposition of an $m \times n$ array $A[1{:}m, 1{:}n]$ stored by rows in the vector $a[1{:}m \times n]$. The method is essentially that of Windley [1], modified for use with vectors having unit lower subscript bounds.

The algorithm processes only elements $A[1, 2]$ through $A[m, n-1]$ since $A[1, 1]$ and $A[m, n]$ retain their original positions. Elements $A[q, p]$ of the transposed matrix are placed in $a[i]$, in the order $i = 2, 3, \cdots, mn - 2$, by an exchanging process. At the last step two elements are correctly placed which accounts for the value $mn - 2$ as the upper bound on i. Valid subscripts of the vector $a[1{:}m \times n]$ are elements in the 1-origin index set $[1, 2, \cdots, mn]$. Computationally, however, it is more convenient to use the zero-origin set $[0, 1, \cdots, mn-1]$. Denoting by i_0 $(i_0 = i-1)$ the corresponding zero-origin index of $a[i]$, to be occupied by $A[q, p]$, we have $i = m(q-1) + (p-1)$. The corresponding zero-origin index j_0 of the $A[p, q]$ element now in $a[j]$, which must be transferred to $a[i]$, is:

$$j_0 = j - 1 = n(p-1) + (q-1) = n \times i_0 \bmod (mn-1).$$

For each value of $i = 2, 3, \cdots, mn - 2$ (or $i_0 = 1, 2, \cdots, mn - 3$) we compute the index j of $a[j]$ and exchange $a[i]$ and $a[j]$ provided $j \geqq i$ (i.e., $j_0 \geqq i_0$). The case $j < i$ indicates that the element originally in $a[j]$ is now elsewhere following previous exchanges. Its present position is given by the first $j_r \geqq i_0$ in the series of zero-origin indices:

$$j_0,\ j_{r+1} = n \times j_r \bmod (mn-1).$$

The two sequences modulo$(mn-1)$ are generated by different methods. An additive process generates the first, using k to duplicate the function of j, in case this is adjusted in the second recurrence-generated sequence if $j < i$.

Unlike the similar problem [3], transposition does not appear to be completely soluble on wholly group-theoretic lines. A general discussion of transposition and a reference to its formulation as a problem in Abelian-Groups is given in [2].

[1] P. F. Windley, Transposing matrices in a digital computer. *Comp. J. 2* (1959), 47–48. [2] G. A. Heuer, Control Data Technical Report T.R.53, pp. 3–5. [3] Fletcher, W., and Silver, R. Algorithm 284. *Comm. ACM 9* (May 1966), 326;

```
begin integer i, j, k, iless1, mnless1, done, jn, modlessn;
  real t;
  mnless 1 := m × n − 1;  modlessn := mnless1 − n;
  done := mnless1 − 1;  k := 0;  iless1 := 1;
  for i := 2 step 1 until done do
  begin comment computes $j = k = n \times i_0 \bmod (mn-1)$;
    j := k := if k ≤ modlessn then k + n else k − modlessn;
  test: if j < iless1 then
    begin comment computes $j_{r+1} = n \times j_r \bmod (mn-1)$;
          jn := j × n;
          j := jn − jn ÷ mnless1 × mnless1;
          go to test
    end;
    comment avoid unnecessary exchanges;
    if j ≠ iless1 then
    begin j := j + 1;
      t := a[i];  a[i] := a[j];  a[j] := t
    end;
    iless1 := i
  end
end transpose
```

CERTIFICATION OF ALGORITHM 302 [K2]
TRANSPOSE VECTOR STORED ARRAY [J. Boothroyd, *Comm. ACM 10* (May 1967), 292]

I. D. G. MacLeod (Recd. 8 Jan. 1968)

Department of Engineering Physics, Australian National University, Canberra 2600, Australia

KEY WORDS AND PHRASES: matrix transposition, array transposition, vector stored array
CR CATEGORIES: 5.39

Algorithm 302 has been tested using both Fortran IV and Algol on A.N.U's IBM System 360 model 50, with satisfactory results in each case.

There is a misprint in line 2 of the procedure: the comma between *a* and **comment** should be replaced by a semicolon.

This compact algorithm can be written even more briefly and with improved efficiency by making the following changes:

1. Delete *jn* from the list of declared integers.
2. Replace lines 8 through 13 of the procedure body by

```
if j < iless1 then
begin comment computes $j_{r+1} = n \times j_r \bmod (mn-1)$;
newj:  j := j × n − j ÷ m × mnless1;
       if j < iless1 then go to newj
end;
```

In-situ transposition of a vector stored array may be considered as a permutation which decomposes into a set of unique cycles. Accessing arrays may be a relatively slow process (as in Algol with subscript-bound checks) and, in general, unnecessary accesses should be avoided. The test in Algorithm 302 for unnecessary exchanges has been inserted for this purpose but it should be pointed out that only one exchange is saved in each cycle. The inclusion of this test yields a useful gain in efficiency only for those situations in which: (i) the implementation is such that array access time is dominant; and (ii) the required transposition decomposes into a high proportion of short cycles, e.g. transposition of a square matrix of order n decomposes into n cycles of length 1 and $n(n-1)/2$ cycles of length 2.

If the implementation is such that accessing arrays is efficient, and the algorithm is to be used for rectangular as well as square matrices, replacement of lines 14 through 18 of the procedure body by

```
j := j + 1;
t := a[i];  a[i] := a[j];  a[j] := t;
```

may make the algorithm more efficient and even more compact.

ALGORITHM 303
AN ADAPTIVE QUADRATURE PROCEDURE WITH RANDOM PANEL SIZES [D1]

L. J. Gallaher (Recd. 8 Nov. 1966 and 1 Feb. 1967)
Georgia Institute of Technology, Engineering Experiment Station, Atlanta, Ga.

```
real procedure Integral(a, x, b, fx, random number, error);
  value a, b, error;
  real a, x, b, fx, error;
  real procedure random number;
```

comment This procedure approximates the quadrature of the function *fx* on the interval $a < x < b$ to an estimated accuracy of *error*. It does this by sampling the function *fx* at appropriate points until the estimated error is less than *error*. The points to be sampled are determined by a combination of random sampling and of estimating what regions are more in need of sampling, this need being determined by the samples already taken. This process goes under the name "importance sampling" in nuclear reactor literature [for example, see J. M. Hammersley and D. C. Handscomb, *Monte Carlo Methods*, John Wiley, Inc., 1964, p. 57]. The form of importance sampling used here is based on estimates of the error contributed to the quadrature by the second derivative. That is, random samples of the average value of the second derivative of *fx* in a region are taken and used to decide if more samples are needed in that region.

Randomness here is achieved through the real procedure *random number*. This procedure is not given explicitly here but can be any random number generator available, provided only that the numbers given are distributed on the interval 0 to 1. The random numbers given need not be of particularly high quality (i.e., need not have low correlation). Further the random number generator need not be passed as a parameter but could be either global or local to the procedure *Integral*.

This procedure is meant to be used for low-accuracy estimates of quadratures, especially large dimensional multiple integrals for which the high-accuracy methods would be too time consuming and expensive. It can achieve high accuracies but not as efficiently as algorithms already in the literature. The general form of this algorithm is similar to Algorithm 145 [W. M. McKeeman, Adaptive Numerical Integration by Simpson's Rule, *Comm. ACM 5* (Dec. 1962), 604] (and others) except that in subdividing the region of integration the panel sizes are determined in part by the random-number generator.

This quadrature procedure has been found particularly effective in integrating ill-behaved functions of the following type.

A. Functions having singularities on the boundary of the region of integration. Such integrals as

$$\int_0^1 x^{-1/2}\,dx,$$

$$\int_0^2 dx \int_0^{\sqrt{1-(1-x)^2}} dy(x^2+y^2)^{-1/2},$$

and

$$\int_0^1 dx \int_0^{\sqrt{1-x^2}} dy(x^2+y^2)^{-1/2},$$

have been successfully integrated with this procedure to 1% accuracy.

B. Functions having an infinite number of zeros in the interval of integration. Such integrals as

$$\int_0^1 dy\,\sqrt{y}\sin(1.5\ln y),$$

$$\int_0^1 dy\,y^{-1/2}\sin(0.5\ln y),$$

and

$$\int_1^2 dx \int_0^1 dy\,xy^{(x-1)}\sin(x\ln y),$$

have been successfully integrated with this procedure to 1% accuracy.

C. Functions having high-frequency oscillations or a large number of discontinuities. The function

$$f(x) = \begin{cases} 2 \text{ if the least significant bit of } x \text{ is } 1 \\ 0 \text{ otherwise} \end{cases}$$

is almost as discontinuous as can be represented in a binary number computer. One hundred attempts at integrating this function on the interval 0 to 1 gave an average of the absolute value of the error ≈ 0.13.

The main limitation in integrating anomalous functions of the above type is in the hardware or software of the particular machine being used. The procedure will fail when the interval is subdivided to a point where it is smaller than the smallest in magnitude nonzero number representable in the machine.

A histogram is given below of the errors in the evaluation of the integrals

$$\int_0^1 dy\,xy^{(x-1)}\sin(x\ln y)$$

and

$$\int_0^1 dy\,xy^{(x-1)}\cos(x\ln y)$$

for $x = 1.04(0.04)2.00$, with error tolerances 10^{-3} and 10^{-4}.

Number of occurrences	0	0	1	1	53	41	4	0	0	0	
$\log_{10}(\epsilon/\epsilon_0)$	−5	−4	−3	−2	−1	0	1	2	3	4	5

Here ϵ_0 is the error requested, ϵ is the error obtained.

The formal parameter *fx* is an arithmetic expression dependent on *x*. In translating to another language it may be desirable to make this parameter a procedure identifier with appropriate modifications in the body of the program;

```
if a = b then Integral := 0
```

```
else
begin real fl, fr, c;
  real procedure Int(a, x, b, fx, fc2, error);
    value a, b, fc2, error;
    real a, x, b, fx, fc2, error;
  begin real dx, dxc, fc1, fc3;
    error := error × 0.577;
    comment The factor 0.577 is an approximation to 1/√3.
      The assumption here is that error contributed by the indi-
      vidual panels is random and not additive, thus the error
      from three panels is assumed to be √3 (not 3) times the
      error of one panel;
    dxc := (random number+0.5) × (b−a)/3;
    dx := (b−a−dxc)/2;
    x := a + dx/2;  fc1 := fx;
    x := b − dx/2;  fc3 := fx;
    Int :=
    if abs(dx×(fc1−2×fc2+fc3)) ≦ error then
      dx × (fc1+fc3) + dxc × fc2
    else
      Int (a, x, a+dx, fx, fc1, error)
      +Int (a+dx, x, b−dx, fx, fc2, error)
      +Int (b−dx, x, b, fx, fc3, error)
  end;
  c := a + (random number+0.5) × (b−a)/2;
  x := (a+c)/2;  fl := fx;
  x := (c+b)/2;  fr := fx;
  error := abs(error) × 14.6;
  comment The factor 14.6 can be thought of as an empirical
    constant. There is some theoretical justification for calculat-
    ing an optimum value for this factor, but in practice it was
    determined empirically;
  Integral :=
    Int(a, x, c, fx, fl, error)
    +Int(c, x, b, fx, fr, error)
end
```

ALGORITHM 304
NORMAL CURVE INTEGRAL [S15]

I. D. Hill and S. A. Joyce (Recd. 21 Nov. 1966)
Medical Research Council, Statistical Research Unit,
115 Gower Street, London W.C.1., England

```
real procedure normal (x, upper);
  value x, upper;  real x;  Boolean upper;
comment calculates the tail area of the standardized normal
  curve, i.e.,
```

$$\frac{1}{\sqrt{2\pi}} \int e^{-1/2t^2}\, dt .$$

If *upper* is **true** the limits of integration are x and ∞.
If *upper* is **false** the limits are $-\infty$ and x.
If x lies in the central area of the curve the method used is the convergent series

$$e^{(1/2)x^2} \int_0^x e^{-(1/2)t^2}\, dt = x + \frac{x^3}{} + \frac{x^5}{3 \times 5} + \frac{x^7}{3 \times 5 \times 7} + \cdots .$$

(See [1, 26.2.11].)
If x lies in one of the tails the method used is the continued fraction

$$e^{(1/2)x^2} \int_x^\infty e^{-(1/2)t^2}\, dt = \frac{1}{x+}\,\frac{1}{x+}\,\frac{2}{x+}\,\frac{3}{x+}\,\frac{4}{x+} \cdots .$$

(See [1, 26.2.14].)

The changeover point between the two methods is at $abs(x) = 3.5$ if the required area is greater than 0.5. This value is chosen on grounds of speed. If, however, the required area is less than 0.5, a changeover as far out as 3.5 will lead to the loss of three significant decimal figures due to cancellation error upon making a subtraction. In this case speed is sacrificed to accuracy and the changeover point is at $abs(x) = 2.32$, chosen as the point at which the area is 0.01. The value 2.32 may be changed to 1.28 (the point at which the area is 0.1) if the full accuracy of the machine is desired over the range $1.28 < abs(x) \leqslant 2.32$, but this leads to a considerable loss of speed and the accuracy lost by using 2.32 is only one decimal place.

Except for this subtraction error, the procedure works virtually to the accuracy of the machine (provided that the constant $1/sqrt(2\pi)$ is given to this accuracy) for $x \leqslant 7$ but to 1 decimal place less than the accuracy of the machine for $x > 7$.

Reference: [1]. Abramovitz, M. and Stegun, I. A. *Handbook of Mathematical Functions*, National Bureau of Standards, Appl. Math. Ser. 55, US Government Printing Office, Washington, D.C., 1964;

```
if x = 0 then normal := 0.5 else
begin
  real n, x2, y;
  upper := upper ≡ x > 0;
  x := abs(x);  x2 := x × x;
  y := 0.3989422804014 × exp (−0.5×x2);
  comment 0.3989422804014 = 1/sqrt(2×π);
  n := y/x;
  if ¬ upper ∧ 1.0 − n = 1.0 then normal := 1.0 else
  if upper ∧ n = 0 then normal := 0 else
  begin
    real s, t;
    if x > (if upper then 2.32 else 3.5) then
    begin
      real p1, p2, q1, q2, m;
      q1 := x;  p2 := y × x;
      n := 1.0;  p1 := y;
      q2 := x2 + 1.0;
      if upper then
      begin
        s := m := p1/q1;
        t := p2/q2
      end else
      begin
        s := m := 1.0 − p1/q1;
        t := 1.0 − p2/q2
      end;
      for n := n + 1.0 while m ≠ t ∧ s ≠ t do
      begin
        s := x × p2 + n × p1;
        p1 := p2;  p2 := s;
        s := x × q2 + n × q1;
        q1 := q2;  q2 := s;
        s := m;  m := t;
        t := if upper then p2/q2 else 1.0 − p2/q2
      end;
      normal := t
    end else
    begin
      s := x := y × x;  n := 1.0;  t := 0;
      for n := n + 2.0 while s ≠ t do
      begin
        t := s;  x := x × x2/n;
        s := s + x
      end;
      normal := if upper then 0.5 − s else 0.5 + s
    end
  end
end normal
```

REMARKS ON:

ALGORITHM 123 [S15]
REAL ERROR FUNCTION, ERF(x)
[Martin Crawford and Robert Techo *Comm. ACM 5* (Sept. 1962), 483]

ALGORITHM 180 [S15]
ERROR FUNCTION—LARGE X
[Henry C. Thacher Jr. *Comm. ACM 6* (June 1963), 314]

ALGORITHM 181 [S15]
COMPLEMENTARY ERROR FUNCTION—
LARGE X
[Henry C. Thacher Jr. *Comm. ACM 6* (June 1963), 315]

ALGORITHM 209 [S15]
GAUSS
[D. Ibbetson. *Comm. ACM 6* (Oct. 1963), 616]

ALGORITHM 226 [S15]
NORMAL DISTRIBUTION FUNCTION
[S. J. Cyvin. *Comm. ACM 7* (May 1964), 295]

ALGORITHM 272 [S15]
PROCEDURE FOR THE NORMAL DISTRIBUTION FUNCTIONS
[M. D. MacLaren. *Comm. ACM 8* (Dec. 1965), 789]

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

I. D. Hill and S. A. Joyce (Recd. 21 Nov. 1966)
Medical Research Council,
Statistical Research Unit, 115 Gower Street, London
W.C.1., England

These algorithms were tested on the ICT Atlas computer using the Atlas Algol compiler. The following amendments were made and results found:

ALGORITHM 123

(i) **value** x; was inserted.
(ii) $abs(T) \leqslant {}_{10}-10$ was changed to $Y - T = Y$ both these amendments being as suggested in [1].
(iii) The labels 1 and 2 were changed to $L1$ and $L2$, the **go to** statements being similarly amended.
(iv) The constant was lengthened to 1.12837916710.
(v) The extra statement $x := 0.707106781187 \times x$ was made the first statement of the algorithm, so as to derive the normal integral instead of the error function.

The results were accurate to 10 decimal places at all points tested except $x = 1.0$ where only 2 decimal accuracy was found, as noted in [2]. There seems to be no simple way of overcoming the difficulty [3], and any search for a method of doing so would hardly be worthwhile, as the algorithm is slower than Algorithm 304 without being any more accurate.

ALGORITHM 180

(i) $T := -0.56418958/x/exp(v)$ was changed to $T := -0.564189583548 \times exp(-v)/x$. This is faster and also has the advantage, when v is very large, of merely giving 0 as the answer instead of causing overflow.
(ii) The extra statement $x := 0.707106781187 \times x$ was made as in (v) of Algorithm 123.
(iii) **for** $m := m + 1$ was changed to **for** $m := m + 2$. $m+1$ is a misprint, and gives incorrect answers.

The greatest error observed was 2 in the 11th decimal place.

ALGORITHM 181

(i) Similar to (i) of Algorithm 180 (except for the minus sign).
(ii) Similar to (ii) of Algorithm 180.
(iii) m was declared as **real** instead of **integer**, as an alternative to the amendment suggested in [4].

The results were accurate to 9 significant figures for $x \leqslant 8$, but to only 8 significant figures for $x = 10$ and $x = 20$.

ALGORITHM 209

No modification was made. The results were accurate to 7 decimal places.

ALGORITHM 226

(i) $10 \uparrow m/(480 \times sqrt(2 \times 3.14159265))$ was changed to $10 \uparrow m \times 0.000831129750836$.
(ii) **for** i := 1 **step** 1 **until** $2 \times n$ **do** was changed to $m := 2 \times n$; **for** i := 1 **step** 1 **until** m **do**.
(iii) $-(i \times b/n) \uparrow 2/8$ was changed to $-(i \times b/n) \uparrow 2 \times 0.125$.
(iv) **if** $i = 2 \times n - 1$ was changed to **if** $i = m - 1$
(v) $b/(6 \times n \times sqrt(2 \times 3.14159265))$ was changed to $b/(15.0397696478 \times n)$.

Tests were made with $m = 7$ and $m = 11$ with the following results:

x	*Number of significant figures correct*		*Number of decimal places correct*	
	$m = 7$	$m = 11$	$m = 7$	$m = 11$
−0.5	7	11	7	11
−1.0	7	10	7	10
−1.5	7	10	8	10
−2.0	7	9	8	10
−2.5	6	9	8	11
−3.0	6	7	8	9
−4.0	5	7	10	11
−6.0	2	1	12	10
−8.0	0	0	11	9

Perhaps the comment with this algorithm should have referred to decimal places and not significant figures. To ask for 11 significant figures is stretching the machine's ability to the limit, and where 10 significant figures are correct, this may be regarded as acceptable.

ALGORITHM 272

The constant .99999999 was lengthened to .9999999999.

The accuracy was 8 decimal places at most of the points tested, but was only 5 decimal places at $x = 0.8$.

ALGORITHM 304

No modification was made. The errors in the 11th significant figure were:

$abs(x)$	$x > 0 \equiv upper$	$x > 0 \not\equiv upper$
0.5	1	1
1.0	1	2
1.5	21[a](5)	2
2.0	25[a](0)	4
3.0	0	0
4.0	2	3
6.0	6	0
8.0	14	0
10.0	23	0
20.0	35	0

[a] Due to the subtraction error mentioned in the comment section of the algorithm. Changing the constant 2.32 to 1.28 resulted in the figures shown in brackets.

To test the claim that the algorithm works virtually to the accuracy of the machine, it was translated into double-length instructions of Mercury Autocode and run on the Atlas using the EXCHLF compiler (the constant being lengthened to 0.398942280401432677939946). The results were compared with hand calculations using Table II of [5]. The errors in the 22nd significant figure were:

abs(x)	$x > 0 \equiv upper$	$x > 0 \not\equiv upper$
1.0	2	3
2.0	7	1
4.0	2	0
8.0	8	0

Timings. Timings of these algorithms were made in terms of the Atlas "Instruction Count," while evaluating the function 100 times. The figures are not directly applicable to any other computer, but the relative times are likely to be much the same on other machines.

Instruction Count for 100 Evaluations

abs(x)	Algorithm number							
	123	180	181	209	226 $m = 7$	272	304[a]	304[b]
0.5	58			8	97	24	25	24
1.0	65[c]			8	176	24	29	29
1.5	164	128	127	9	273	25	35	35
2.0	194	78	90	8	387	24	39	39
2.5	252	54	68	10	515	24	131	44
3.0		42	51	9	628	25	.97	50
4.0		27	39	9	900[d]	25	67	44
6.0		15	30	6	1400[d]	16	49	23
8.0		9	28	7	2100[d]	18	44	11
10.0		10	25	5	2700[d]	16	38	11
20.0		9	22	5	6500[d]	16	32	11
30.0		9	9	5	10900[d]	16	11	11

[a] Readings refer to $x > 0 \equiv upper$.

[b] Readings refer to $x > 0 \not\equiv upper$.

[c] Time to produce incorrect answer. A count of 120 would fit a smooth curve with surrounding values.

[d] 100 times Instruction Count for 1 evaluation.

Opinion. There are advantages in having two algorithms available for normal curve tail areas. One should be very fast and reasonably accurate, the other very accurate and reasonably fast. We conclude that Algorithm 209 is the best for the first requirement, and Algorithm 304 for the second.

Algorithms 180 and 181 are faster than Algorithm 304 and may be preferred for this reason, but the method used shows itself in Algorithm 181 to be not quite as accurate, and the introduction of this method solely for the circumstances in which Algorithm 180 is applicable hardly seems worth while.

Acknowledgment. Thanks are due to Miss I. Allen for her help with the double-length hand calculations.

References:

1. Thacher, Henry C. Jr. Certification of Algorithm 123. *Comm. ACM 6* (June 1963), 316.
2. Ibbetson, D. Remark on Algorithm 123. *Comm. ACM 6* (Oct. 1963), 618.
3. Barton, Stephen P., and Wagner, John F. Remark on Algorithm 123. *Comm. ACM 7* (Mar. 1964), 145.
4. Clausen, I., and Hansson, L. Certification of Algorithm 181. *Comm. ACM 7* (Dec. 1964), 702.
5. Sheppard, W. F. *The Probability Integral.* British Association Mathematical Tables VII, Cambridge U. Press, Cambridge, England, 1939.

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

A. Bergson (Recd. 11 Aug. 1967 and 9 Nov. 1967)
Computing Laboratory, Sunderland Technical College, Sunderland, Co. Durham, England

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

Algorithm 304 was coded in 803 Algol and run on a National-Elliott 803 (with automatic floating-point unit).

There are typographical errors in the first two integrals contained in the **comment**.

The integrals should read:

(i) $$\frac{1}{\sqrt{2\pi}} \int e^{-(\frac{1}{2})t^2}\, dt$$

(ii) $$e^{(\frac{1}{2})x^2} \int_0^x e^{-(\frac{1}{2})t^2}\, dt = x + \frac{x^3}{3} + \frac{x^5}{3 \times 5} + \frac{x^7}{3 \times 5 \times 7} + \cdots.$$

The algorithm was run as published and gave answers within the accuracy of the machine [1] for a random selection of values of x and *upper*.

With the following alterations, however, the algorithm was made 0.2 percent more efficient in speed, and gave identical results as above.

(a) $n := 1.0;$ was omitted from the line $n := 1.0;\ \ p1 := y;$

(b) the ten lines after $q2 := x2 + 1.0;$ were replaced by:

```
m := n;  t := p2 / q2;
if ¬ upper then
begin
  m := 1.0 − m;  t := 1.0 − t
end;
for n := 2.0, n + 1.0 while m ≠ t ∧ s ≠ t do
```

(c) in the line beginning $s := x := y \times x;$, $n := 1.0;$ and $t := 0;$ were omitted and the next line written:

```
for n := 3.0, n + 2.0 while s ≠ t do
```

Reference:

1. A specification of 803 ALGOL; Description of 803 Library Program A104. Elliott-NCR Ltd., Borehamwood, Hertfordshire, England. (Jan. 1965, issue 4).

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

Bo Holmgren (Recd. 30 Apr. 1970)
Dept. KDO, ASEA, S-721 83 Västerås, Sweden

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

Algorithm 304 with the remark of Adams was translated into Fortran IV and run on a GE-625 computer. The GE-625 has a 28-bit mantissa and allows exponents up to 10^{38}. With *upper* = **false** and $x < -2.32$, the routine ran into overflow at several values of x. To avoid this the following lines

```
  if q2 > 10^30 then
  begin
    p1 := p1 × ₁₀-30;  p2:= p2 × ₁₀-30;
    q1 := q1 × ₁₀-30;  q2 := q2 × ₁₀-30
  end;
```

were inserted after the line

```
  s := m;  m := t;
```

ALGORITHM 305
SYMMETRIC POLYNOMIALS [C1]

P. Bratley and J. K. S. McKay (Recd. 23 Sept. 1966, 15 Feb. 1967 and 10 Mar. 1967)
Department of Computer Science, University of Edinburgh, Edinburgh, Scotland

```
real procedure express(b, unit, n);   value n;   integer n;
integer array b;   array unit;
comment  express expresses the symmetric sum $\sum x_{i_1}^{b_1} x_{i_2}^{b_2} \cdots x_{i_n}^{b_n}$
  over n variables as a sum of determinants in the unitary sym-
  metric functions $\sum x_{i_1} x_{i_2} x_{i_3} \cdots x_{i_r}$. The non-negative ex-
  ponents $b_i$ ($i = 1, \cdots, n$) are assumed to be in b[1:n] on entry
  to express. (The elements of this array are altered by the pro-
  cedure.) The symmetric sum is first expressed in terms of Schur
  functions which are then evaluated as determinants in the
  unitary symmetric functions. The Schur functions are generated
  in the local array c[1:i] with the sign in the local integer sig.
  The unitary functions of degree $r = 1, \cdots, n$ should be in
  unit[1:n] on entry to express.
    This procedure may be used to determine the coefficients of a
  polynomial with roots the kth (k a positive integer) powers of
  the roots of a given monic polynomial. Use is made of the
  procedures determinant [Algorithm 224, Comm. ACM 12 (Apr.
  1964), 243)] and perm [Algorithm 306, Comm. ACM 10 (July
  1967), 450]
    REFERENCES:
  1. LITTLEWOOD, D. E.  The Theory of Group Characters. Claren-
       don Press, Oxford, England 1958, 2nd ed., Ch. 6.
  2. McKAY, J. K. S.  On the representation of symmetric poly-
       nomials. Comm. ACM 10 (July 1967), 428–429;
begin integer array c, d[1:n];
  integer sig, p, q, i, j;   Boolean finish;   real sigma;
  procedure sort (x, c, n);   value n;   integer c, n;
    integer array x;
  comment  sorts the integer array x[1:n] into descending order.
    c is set to ±1 according to whether the number of transposi-
    tions made is even or odd;
  begin integer i, j, k;
    c := 1;
L4: i := 1;   k := 0;   j := x[1];
L1: i := i + 1;   if i > n then go to L3;
    if x[i] ≤ j then
      begin x[i−1] := j;   j := x[i] end
    else begin x[i−1] := x[i];   k := 1;   c := −c end;
    go to L1;
L3: x[n] := j;   if k ≠ 0 then go to L4
  end sort;
  procedure conjugate(p, long1, q, long2);   value long1;
    integer array p, q;   integer long1, long2;
  comment  conjugate forms in q[1:long2] the partition conju-
    gate to that in p[1:long1];
  begin
    integer r, i, j;
    long2 := 0;
    for r := long1 step −1 until 1 do
    begin i := if r = long1 then p[r] else p[r] − p[r+1];
      for j := 1 step 1 until i do
        begin long2 := long2 + 1;   q[long2] := r end
    end
  end conjugate;
  finish := true;   sigma := 0;
  sort (b, sig, n);
  if b[1] = 0 then begin sigma := 1;   go to L99 end;
L3: perm (b, n, finish);
  if finish then go to L99;
  for i := 1 step 1 until n do
  begin c[i] := b[i] + n − i;
    for j := 1 step 1 until i − 1 do
    if c[i] = c[j] then go to L3
  end;
  sort (c, sig, n);
  for i := 1 step 1 until n do
  begin c[i] := c[i] + i − n;
    if c[i] = 0 then
      begin i := i − 1;   go to L7 end
  end;
  i := n;
  comment  each Schur function and its sign are to be found in
    c[1:i] and sig respectively;
L7: conjugate (c, i, d, q);
  begin
    array x[1:q, 1:q];
    for i := 1 step 1 until q do
    for j := 1 step 1 until q do
    begin p := d[i] − i + j;
      x[i, j] := if p < 0 ∨ p > n then 0 else
      if p = 0 then 1 else unit[p]
    end;
    sigma := sigma + sig × determinant (x, q)
  end;
  go to L3;
L99: express := sigma
end express
```

REMARK ON ALGORITHM 305 [C1]
SYMMETRIC POLYNOMIALS [P. Bratley and J. K. S. McKay, *Comm. ACM 10* (July 1967), 450]

J. K. S. McKay (Recd. 13 Sept. 1967 and 18 Dec. 1967)
Atlas Laboratory, Science Research Council, Chilton, Didcot, Berks., England

KEY WORDS AND PHRASES: symmetric polynomials, symmetric sum, unitary symmetric functions, Schur functions
CR CATEGORIES: 5.39

The published algorithm fails with subscript overflow if $\sum_{i=1}^{n} b_i$ is greater than n and the partition conjugate to that in $c\,[1:i]$ has more than n parts.

The symmetric sum is defined ambiguously in the initial comment.

The following alterations are suggested to remove the ambiguitv and correct the algorithm.

(1) In line 4,

  ··· over n variables ···

should be replaced by

  ··· over all distinct terms in n variables ···

to remove any ambiguity in the definition of the symmetric sum.

(2) In line 8, before

  The symmetric sum ···

insert

  Three examples to clarify the value of the symmetric sum are:

  If $n = 3$ and the b_i are 3, 2, 0 in any order the sum is $x_1{}^3\, x_2{}^2 + x_2{}^3\, x_3{}^2 + x_3{}^3\, x_1{}^2 + x_1{}^3\, x_3{}^2 + x_2{}^3\, x_1{}^2 + x_3{}^3\, x_2{}^2$.

  If $n = 3$ and the b_i are 2, 2, 0 in any order the sum is $x_1{}^2\, x_2{}^2 + x_2{}^2\, x_3{}^2 + x_3{}^2\, x_1{}^2$.

  If all b_i are zero the procedure will return the value 1.

(3) In lines 17–18, the reference to Algorithm 224 should read:

  Comm. ACM 7 (Apr. 1964), 243 and (Dec. 1964), 702.

(4) Lines 25–26

  integer array *c*,*d*[1:*n*];
  integer *sig*,*p*,*q*,*i*,*j*; ···

should be replaced by

  integer *sig*,*p*,*q*,*i*,*j*; *j* := 0;
  for *i* := 1 **step** 1 **until** *n* **do** *j* := *j*+*b*[*i*];
  begin integer array *c*[1:*n*], *d*[0:*j*]; ···

(5) In line 72,

  comment each Schur function ···

should be replaced by

  comment at *L7* each Schur function ···

(6) In line 87, an **end** should be inserted immediately before

  end *express*

ALGORITHM 306
PERMUTATIONS WITH REPETITIONS [G6]

P. Bratley (Recd. 23 Sept. 1966 and 15 Feb. 1967)
Department of Computer Science, University of Edinburgh, Edinburgh Scotland

```
procedure perm(a, n, last);  value n;  integer n;
  integer array a;  Boolean last;
comment  a[1:n] is an integer array. Initially the elements of
  a[1:n] must be arranged in descending order and last must be
  set true. If the elements of a are not initially in descending
  order the effect of the procedure is undefined. Successive calls of
  perm generate in a all permutations of its elements in reverse
  lexicographical order.
    last is set false if the procedure has generated a new permuta-
  tion, but if the procedure is entered after all the permutations
  have been generated, last will be set true. Neither a nor n should
  be altered between successive calls of the procedure;
begin integer i, p, q, r;
  own integer m;  own integer array b[1:n];
  if ¬ last then go to L12;  last := false;
  for i := 1 step 1 until n do b[i] := a[i];
  p := b[n];
  for i := n step −1 until 1 do
    if p ≠ b[i] then
      begin m := i;  go to L99 end;
  m := 0;  go to L99;
L12:  if m = 0 then go to L10;
  p := b[m];  q := m;  r := 0;
L9:  i := n;
L4:  if a[i] = p then go to L2;
  if a[i] < p then r := i;
L5:  i := i − 1;  go to L4;
L2:  a[i] := b[n] − 1;  if r = 0 then go to L8;
L1:  a[r] := p;  q := q + 1;
L3:  r := r + 1;  if r > n then go to L11 else if a[r] > p
  then go to L3;
L11:  if b[q] = p then go to L1;  r := 0;
L6:  r := r + 1;  if a[r] ≥ p then go to L6;
  a[r] := b[q];  if q = n then go to L7;
  q := q + 1;  go to L6;
L7:  last := false;  go to L99;
L8:  q := q − 1;  if q = 0 then go to L10;
  if b[q] = p then go to L5;
  p := b[q];  go to L9;
L10:  last := true;
L99:
end perm
```

ALGORITHM 307
SYMMETRIC GROUP CHARACTERS [A1]

J. K. S. McKay (Recd. 23 Sept. 1966, 15 Feb. 1967, and 10 Mar. 1967)
Department of Computer Science, University of Edinburgh, Edinburgh, Scotland

```
integer procedure character (n, rep, longr, class, longc, first);
  value n, rep, longr, class, longc;
  integer n, longr, longc;  Boolean first;
  integer array rep, class;
```

comment *character* produces the irreducible character of the symmetric group corresponding to the partitions of the representation and the class of the group S_n stored with parts in descending order in arrays *rep*[1:*longr*] and *class*[1:*longc*], respectively. Both arrays are preserved. The method is similar to that described by Bivins et al. [1]. Comét describes a later method.

On first entry to *character*, *first* should be set **true** in order to initialize the own array $p[0{:}n, 0{:}n]$. This single initialization is sufficient for all symmetric groups of degree less than or equal to n. *character* is intended for computing individual characters. If a substantial part of the character table is required it is suggested that procedure *generate* [Algorithm 263, *Comm. ACM 8* (Aug. 1965), 493)] be used to produce the partitions prior to use of *character*. If this is done, then the own array p should be replaced by a suitable global array, and *first* should be set **false** to avoid unwanted initialization. *character* uses procedures *set*, *generate*, and *place* [Algorithms 262, 263, 264, *Comm. ACM 8* (Aug. 1965), 493].

References:

1. Bivins, R. L., Metropolis, N., Stein, P. R., and Wells, M. B. Characters of the symmetric groups of degree 15 and 16. *MTAC 8* (1954), 212-216.
2. Littlewood, D. E. *The Theory of Group Characters*. Clarendon Press. Oxford, England 1958, 2d ed., Ch. 5.
3. Comét, S. Improved methods to calculate the characters of the symmetric group. *MTAC 14* (1960), 104-117.;

```
begin
  integer procedure degree (n, rep, length);  value n, length;
    integer n, length;  integer array rep;
  comment degree gives the degree of the representation of the
    symmetric group on n symbols defined by the partition
    rep[1:length] with parts in descending order;
  begin
  own integer array p[0:n, 0:n];
    integer array q[1:length];  integer i, j, deg;
    integer procedure fac(n);  value n;  integer n;
    fac := if n = 1 then 1 else n × fac(n−1);
    for i := 1 step 1 until length do
      q[i] := rep[i] + length − i;
    deg := fac(n);
    for i := 1 step 1 until length do
    for j := i + 1 step 1 until length do
      deg := deg × (q[i]−q[j]);
    for i := 1 step 1 until length do
      deg := deg ÷ fac(q[i]);
    degree := deg
  end degree;
  if first then
    begin set (p, n);  first := false end;
  begin
    integer array pr[1:n], r[0:1, 0:p[n, n]−1];
    integer length, m, t, old, new, index, i, char, k, coeff, u, pos,
      j1, j2;
    m := longc;
    new := n;
    index := 1;
    for i := 0 step 1 until p[n, n] − 1 do
      r[index, i] := 0;
    r[index, place(p, n, rep)] := 1;
    for t := 1 step 1 until m do
    begin if class[t] = 1 then go to identity;
      index := 1 − index;  old := new;  new := new − class[t];
      for i := 0 step 1 until p[new, new] − 1 do
        r[index, i] := 0;
      for u := p[old, old] − 1 step − 1 until 0 do
      begin if r[1 − index, u] = 0 then go to B;
        generate (p, old, u, pr, length);
        k := length;  j1 := 1;
G:      j2 := j1;  coeff := r[1−index, u];
        for i := 1 step 1 until k do rep[i] := pr[i];
        if rep[1] = old then go to H;
        rep[j2] := rep[j2] − class[t];
        if rep[j2] + k − j2 < 0 then go to B;
        if rep[j2] ≥ if (j2 = k then 0 else rep[j2+1]) then go to F;
        if rep[j2+1]= rep[j2] + 1 then go to J;
        i := rep[j2+1];  rep[j2+1] := rep[j2] + 1;
        rep[j2] := i − 1;  coeff:= − 1 coeff;  j2 := j2 + 1;
        go to E;
H:      rep[1] := rep[1] − class[t];
F:      pos := place(p, new, rep);
        r[index, pos] := r[index, pos] + coeff;
J:      j1 := j1 + 1;  if j1 ≤ k then go to G;
B:
      end
    end;
A:  char := r[index, 0];  go to Z;
identity:  char := 0;
    for u := p[new,new] − 1 step − 1 until 0 do
    begin if r[index, u] = 0 then go to BB;
      generate(p, new, u, pr, length);
      char := char + r[index, u] × degree (new, pr, length);
BB:
    end;
Z:  character := char
  end
end character
```

REMARK ON ALGORITHM 307 [A1]
SYMMETRIC GROUP CHARACTERS

[J. K. S. McKay, *Comm. ACM 10* (July 1967), 451]

J. K. S. McKay (Recd. 13 Sept. 1967)
Dept. of Computer Science, University of Edinburgh, Edinburgh, Scotland

Three corrections are noted.

(1) Line 39:

```
        own integer array p[0:n,0:n];
```

should be moved to the line after the **begin** in line 32.

(2) At *E* the line should read

```
E:  if rep[j2] ≥ (if j2=k then 0 else rep[j2+1])
    then go to F;
```

(3) Three lines, later

```
              coeff := −1 coeff;
```

should read

```
              coeff := −coeff;
```

ALGORITHM 308
GENERATION OF PERMUTATIONS IN PSEUDO-LEXICOGRAPHIC ORDER [G6]

R. J. Ord-Smith (Recd. 11 Nov. 1966, 1 Dec. 1966, 28 Dec. 1966 and 27 Mar. 1967)
Computing Laboratory, University of Bradford, England

Lexicographic generation has the advantage of producing an order easily followed by the user, but its real value in certain combinatorial applications is that a $(k-1)$-th intransitive subgroup of permutations is generated before the kth element is moved. By not insisting on strict lexicographic generation, though preserving the latter property, an enormous reduction in the total number of transpositions is obtained. The total number of transpositions in this algorithm can be shown to tend asymptotically to (sinh 1) $n!$ which is less than in Algorithm 86 [J. E. L. Peck and G. F. Schrack, Permute, *Comm. ACM 5* (Apr. 1962), 208] and almost as good as Algorithm 115 [H. F. Trotter, Perm, *Comm. ACM 5* (Aug. 1962), 434]. The algorithm offers a further useful facility. Like several others it uses a nonlocal Boolean variable called *first*, which may be assigned the value **true**, to initialize generation. On procedure call this is set **false** and remains so until it is again set **true** when complete generation of permutations has been achieved. At any subsequent call after initializing generation of permutations of degree n, one may set parameter $n = n'$ where $n' \leq n$. Further calls with this value may continue until the completion of the subgroup of degree $(n' - 1)$ when *first* will be set **true.** The process can be continued by resetting *first* **false** and calling with a larger value of n. This gives the user complete control over the main attribute which lexicographic order offers. There is no restriction on the elements permuted. Table I gives results obtained for *ECONOPERM*. Times given in seconds are for an ICT 1905 computer. The algorithm has also been tested successfully on IBM 7094, Elliott 503 and STC Stantec computers. t_n is the time for complete generation of $n!$ permutations. r_n has the usual definition $r_n = t_n/(n \cdot t_{n-1})$.

TABLE I

Algorithm	t_6	t_7	t_8	r_6	r_7	r_8	*Number of transpositions*
ECONOPERM	0.85	6.2	50.6	—	1.04	1.02	→ 1.175$n!$

```
procedure ECONOPERM (x, n);   value n;   integer n;
  array x;
begin own integer array q[2:n];
  comment  own dynamic arrays are not often implemented.
    The upper bound will then have to be given explicitly;
  integer k, l, m;  real t;
  l := 1;  k := 2;
  if first then
    begin first := false;  go to label end;
  comment  the above is the initialization process;
loop:  if q[k] = k then
  begin if k < n then
  begin k := k + 1;  go to loop end
  else begin first := true;  go to finish end
end;
n := k - 1;
comment  note n called by value;
label:  for m := 2 step 1 until n do q[m] := 1;
  comment  after the initialization the for statement sets all
    elements of q array to 1. Otherwise only the first k-2 elements
    are reset 1;
  q[k] := q[k] + 1;
transpose:  t := x[l];  x[l] := x[k];  x[k] := t;
  l := l + 1;  k := k - 1;
  if l < k then go to transpose;
  comment  when k < 4 only one transposition occurs. On final
    exit when first is reset true, no transposition occurs at all;
finish:
end of procedure ECONOPERM
```

REMARK ON ALGORITHM 308 [G6]
GENERATION OF PERMUTATIONS IN PSEUDO-LEXICOGRAPHIC ORDER [R. J. Ord-Smith, *Comm. ACM 10* (July 1967), 452]

R. J. Ord-Smith (Recd. 21 May 1969)
Computing Laboratory, University of Bradford, England

KEY WORDS AND PHRASES: permutations, lexicographic order, lexicographic generation, permutation generation
CR CATEGORIES: 5.39

Following the construction of the very fast lexicographic permutation Algorithm 323 [1] it has become clear that the permutation sequence generated by the Algorithm 308 can be obtained more quickly. In fact, replacement of

```
trstart:m := q[k];   t := x[m];   x[m] := x[k];   x[k] := t;
  q[k] := m + 1;  k := k - 1;
```

by

```
trstart:  q[k] := q[k] + 1;
```

in Algorithm 323 produces the *ECONOPERM* sequence of Algorithm 308.

The times are as follows on an ICT 1905, in seconds.

	t_7	t_8
Algorithm 323	6	47
New *ECONOPERM*	5.9	45
Old *ECONOPERM*	6.2	50.6

Reference:

1. Ord-Smith, R. J. Algorithm 323: Generation of permutations in lexicographic order. *Comm. ACM 11* (Feb. 1968), 117.

ALGORITHM 309
GAMMA FUNCTION WITH ARBITRARY PRECISION [S14]

ANTONINO MACHADO SOUZA FILHO AND GEORGES SCHWACHHEIM (Recd. 12 Apr. 1966 and 14 Apr. 1967)

Centro Brasileiro de Pesquisas Fisicas, Rio de Janeiro, ZC82, Brazil

```
procedure gamma(z,y,msize,error);
 value z, msize; real z; integer msize; label error;
comment This procedure computes the value y of the gamma
  function for any real argument z for which the result can be
  represented within the computer, working with msize decimal
  digits. An exit is made thru the label error when the argument is
  a pole or is too large, while a zero result is returned when the
  argument is too small for a correct internal representation of the
  result.
  This procedure is especially useful for variable field length
  computers and for double- or multiple-precision computations,
  when a simple power series algorithm is no longer applicable.
  It computes the gamma function thru the Stirling asymptotic
  series for the logarithm of the gamma function with an argu-
  ment increased by an appropriate integer to insure the required
  precision with the least computation work.
  Negative arguments are reduced to positive ones by:
```

$$\Gamma(z) = \frac{\pi}{\sin(\pi z) \times \Gamma(1-z)}$$

```
  This procedure is not recursive and uses no own variable.
  It was translated to FORTRAN II and run on an IBM 1620. The
  errors were at most of a few hundred units in the last digit of the
  mantissa, being due to the use of logarithms;
begin
  real procedure loggamma (t); value t; real t;
  comment The loggamma auxiliary procedure computes the
    logarithm of the gamma function of a positive argument t.
    If its argument is below a value tmin, loggamma first increases
    the argument by an integer value, using the relation:
```

$$\ln \Gamma(t) = \ln \Gamma(t+k) - \ln(\prod_{i=0}^{k-1} (t+i))$$

```
    where ln Γ(t + k) is computed by the procedure lgm.
      The formula we use for tmin is a rough empirical relation
    to minimise computation time.
      Indeed an increase of k while decreasing the number of
    terms of the series, results in more computation for the fac-
    tor ln (∏_i( t + i ));
  begin integer tmin;
    tmin := if msize ≥ 18 then msize − 10 else 7;
    if t > tmin then loggamma := lgm(t)
    else
    begin real f;
      f := t;
L:    t := t+1;
      if t < tmin then
      begin f := f × t;
        go to L
      end;
      loggamma := lgm(t) − ln (f)
    end
  end of procedure loggamma;
  real procedure lgm(w); value w; real w;
  comment This procedure evaluates the logarithm of the
    gamma function according to the Stirling asymptotic series:
```

$$\ln \Gamma(w) \simeq (w - \tfrac{1}{2}) \times \ln (w) - w + \ln \sqrt{2\pi} + \sum_i \frac{c_i}{z^{2i-1}}$$

```
      The coefficients c_i = B_2i/(2i(2i−1)), B_2i being the
    Bernoulli numbers, are rational numbers given here as irre-
    ducible fractions.
      Twenty terms are sufficient for a precision of up to 50
    decimal digits;
  begin array c[1:20]; real w2, presum, const, den, sum;
    integer i;
    c[1]  := 1/12;           c[2]  := −1/360;
    c[3]  := 1/1260;         c[4]  := −1/1680;
    c[5]  := 1/1188;         c[6]  := −691/360360;
    c[7]  := 1/156;          c[8]  := −3617/122400;
    c[9]  := 43867/244188;   c[10] := −174611/125400;
    c[11] := 77683/5796;     c[12] := −236364091/1506960;
    c[13] := 657931/300;     c[14] := −3392780147/93960;
    c[15] := 1723168255201/2492028;
    c[16] := −7709321041217/505920;
    c[17] := 151628697551/396;
    c[18] := −26315271553053477373/2418179400;
    c[19] := 154210205991661/444;
    c[20] := −261082718496449122051/21106800;
    const := .918938533204672741780329736405617639861397473637783;
    comment const = ln√2π;
    den := w; w2 := w × w; presum := (w−.5) × ln(w) −
    w + const;
    for i := 1 step 1 until 20 do
    begin sum := presum + c[i]/den;
      if sum = presum then go to exit;
      den := den × w2;
      presum := sum
    end;
exit : lgm := sum
  end of procedure lgm;
  comment: main procedure gamma starts here;
  real pi;
  pi := 3.14159265358979323846264338327950288419716939937510;
  comment argov, argund, lnunder are hardware dependent con-
    stants that are compared to the arguments of intermediate
    results, setting error exit or zero result to prevent exponent
    over or underflow. Should be replaced in the procedure by
    the appropriate numbers;
  if z > argov then go to error else if z = entier (z) then
  begin if z ≤ 0 then go to error; y := 1;
    if z > 2 then
    begin loop: z := z − 1; y := y × z;
      if z > 2 then go to loop
    end
  end when z is integer
  else if abs(z) < 10 ↑ (−msize) then y := 1/z
  else if z < 0 then
  begin if z < argund then y := 0
    else
    begin comment  As the use of the sine subroutine for large
        arguments might introduce errors, some reductions of
        the argument are made before using it;
      Boolean procedure parity (m);  real m;
      begin integer j;
        j := entier(m);  parity := j = j ÷ 2 × 2
      end parity;
      real procedure decimal(x);  real x;
      begin integer n;
        n := x;
        decimal := abs(x−n)
      end decimal;
      real delta, ex;
      delta := decimal(z) × pi;
      ex := (if delta<10 ↑ (−msize/2) then − ln(decimal(z)) else
      ln(pi/(sin(delta)))) − loggamma(1−z);
      y := if ex < lnunder then 0 else
        if parity (z) then exp(ex) else
        −exp(ex)
    end
  end when z is negative
  else y := exp(loggamma(z))
end of procedure gamma
```

ALGORITHM 310
PRIME NUMBER GENERATOR 1 [A1]

B. A. Chartres (Recd. 25 Oct. 1966 and 13 Apr. 1967)
Computer Science Center, University of Virginia,
Charlottesville, Virginia

```
integer procedure sieve1(m, p); value m; integer m; in-
  teger array p;
comment sieve1(m, p) generates the prime numbers less than
  or equal to m, and places them in the array p, setting p[1] = 2,
  p[2] = 3, p[3] = 5, ···, p[k] = (largest prime found). The value
  of the procedure is k, the number of primes less than or equal to
  m.
    The method used is a modification of the Sieve of Eratosthenes.
  In its customary form this method requires a repeated sweeping
  over m numbers (or m/2 odd numbers), crossing out all multiples
  of the ith prime on the ith sweep. The variation of the method
  used here condenses all these sweeps into one. When the odd
  integer n is being tested ("if n=q[i]") to see whether it should
  be crossed out ("t := false"), q[i], for i = 3, 4, ···, j, contains
  the smallest odd multiple of p[i] which is no smaller than either
  n or p[i] ↑ 2. The sequence of values taken on by q[i] defines the
  set of numbers crossed out because they are multiples of p[i].
  The initial value of q[i] is p[i] ↑ 2 because all smaller odd multi-
  ples of p[i] have at least one other odd prime factor smaller than
  p[i]. For the same reason, q[j+1] does not become active ("j
  := j+1") until n has become equal to p[j] ↑ 2. The dimension of
  the arrays q and dq is therefore the number of primes less than
  or equal to the square root of m. Thus we have replaced repeated
  sweeps over the array p by (many more) repeated sweeps over
  part of the much smaller array q. This does not reduce the
  amount of computation, but does lead to a much more efficient
  computer implementation, as only the arrays q and dq need be
  held in a random access store.;
begin
  integer array q, dq[2 : 2.7×sqrt(m)/ln(m)];
  integer i, j, k, n;
  Boolean t;
  p[1] := j := k := 2;  p[2] := 3;  q[2] := 9;  dq[2] := 6;
  for n := 5 step 2 until m do
  begin
    t := true;
    for i := 2 step 1 until j do
    begin
      if n = q[i] then
      begin
        q[i] := n + dq[i];  t := false;
        if i = j then
        begin
          j := j + 1;  q[j] := p[j] ↑ 2;
          dq[j] := 2 × p[j];  go to A
        end
      end
    end;
    if t then
    begin
      k := k + 1;  p[k] := n
    end;
A: end;
  sieve1 := k
end sieve1
```

REMARKS ON:
ALGORITHM 35 [A1]
Sieve [T. C. Wood, *Comm. ACM 4* (Mar. 1961), 151]
ALGORITHM 310 [A1]
PRIME NUMBER GENERATOR 1 [B. A. Chartres, *Comm. ACM 10* (Sept. 1967), 569]
ALGORITHM 311 [A1]
PRIME NUMBER GENERATOR 2 [B. A. Chartres, *Comm. ACM 10* (Sept. 1967), 570]

B. A. Chartres (Recd. 13 Apr. 1967)
Computer Science Center, University of Virginia,
Charlottesville, Virginia

The three procedures *Sieve*(m,p), *sieve*1(m,p), and *sieve*2(m,p), which all perform the same operation of putting the primes less than or equal to m into the array p, were tested and compared for speed on the Burroughs B5500 at the University of Virginia. The modification of *Sieve* suggested by J. S. Hillmore [*Comm. ACM 5* (Aug. 1962), 438] was used. It was also found that *Sieve* could be speeded up by a factor of 1.95 by avoiding the repeated evaluation of $sqrt(n)$. The modification required consisted of declaring an integer variable s, inserting the statement $s := sqrt(n)$ immediately after $i := 3$, and replacing $p[i] \leq sqrt(n)$ by $p[i] \leq s$.

The running times for the computation of the first 10,000 primes were:

Sieve (Algorithm 35)	845 sec
Sieve (modified)	434 sec
*sieve*1	220 sec
*sieve*2	91 sec

The time required to compute the first k primes was found to be, for each algorithm, remarkably accurately represented by a power law throughout the range $500 \leq k \leq 50{,}000$. The running time of *Sieve* varied as $k^{1.40}$, that of *sieve*1 as $k^{1.53}$, and that of *sieve*2 as $k^{1.35}$. Thus the speed advantage of *sieve*2 over the other algorithms increases with increasing k. However, it should be noted that *sieve*2 took approximately 33 minutes to find the first 100,000 primes, and, if the power law can be trusted for extrapolation past this point (there is no reason known why it should be), it would take about 12 hours to find the first million primes.

CERTIFICATION OF ALGORITHM 310 [A1]
PRIME NUMBER GENERATOR 1 [B. A. Chartres, *Comm. ACM 9* (Sept. 1967), 569]

Donald G. Rapp and Larry D. Scott (Recd. 21 Apr. 1969 and 13 Aug. 1969)
Computer Science Group, Texas A&M University, College Station, TX 77843

KEY WORDS AND PHRASES: prime numbers, generator
CR CATEGORIES: 5.0

Algorithm 310 was coded in Algol 60 reference language and run on an IBM 360/65. The algorithm was tested for a large range of values including m = 5, 10, 501, and 2000. Reference [1] was utilized to verify the theory involved in the algorithm before actual machine testing.

The intention of Algorithm 310 is to give only the number of primes less than or equal to m. Actual confirmation in the initial phases was accomplished through additional instructions that printed the array of prime numbers, p, and the number of primes, k. Both references listed were useful in substantiation of the prime numbers given. These references were again useful in verifying that all the primes in the array had been discovered and printed.

Each test produced the correct number of primes, k, for the specified range, m. When the primes were listed, the total taken by hand agreed with the number, k, given by the algorithm.

REFERENCES:

1. ESTERMANN, T. *Introduction to Modern Prime Number Theory*. Cambridge U. Press, Cambridge, England, 1952.
2. LEHMER, D. N. Carnegie Institution of Washington, Publication No. 165. Hafner, New York, 1956.

ALGORITHM 311
PRIME NUMBER GENERATOR 2 [A1]

B. A. Chartres (Recd. 25 Oct. 1966 and 13 Apr. 1967)
Computer Science Center, University of Virginia,
Charlottesville, Virginia

```
integer procedure sieve2(m, p);  value m;
  integer m;  integer array p;
comment  sieve2 is a faster version of sieve1. Two changes were
  made to obtain higher speed.
   (1) The multiples q[i] are sorted, smallest first, so that each
  value of n does not need to be compared with every q[i]. The
  sorted order of the q[i] is indicated by an index array r. The
  ith sorted element of q is q[r[i]]. It was found empirically that
  greater speed is obtained when the q[r[i]] are not kept con-
  stantly sorted, but are re-sorted only at the time a new prime is
  discovered. The integer jj indicates which of the q[r[i]] are sorted:
  q[r[3]] through q[r[jj-1]] are out of order, whereas q[r[jj]] through
  q[r[j]] are in order. Sorting is performed in two stages. A sift
  sort first rearranges r[3] through r[jj-1] into rr[3] through
  rr[jj-1]. Then a single merge sort combines rr[3] through rr[jj-1]
  and r[jj] through r[j] into r[1] through r[j].
   (2) All multiples of 3 are automatically excluded from con-
  sideration by stepping n alternately by 2 and 4, and, in a similar
  way, by stepping q[i] alternately by 2 × p[i] and 4 × p[i].;
begin
  integer array q, dq, sq, r, rr[2: 2.7×sqrt(m)/ln(m)];
  integer i, j, jj, k, n, ir, jr, dn;
  Boolean t;
  p[1] := dn := 2;  p[2] := j := jj := k := r[3] := 3;
  p[3] := 5;  q[3] := 25;  dq[3] := 10;  sq[3] := 30;
  for n := 7 step dn until m do
  begin
    t := true;  dn := 6 − dn;
    for i := 3 step 1 until jj do
    begin
      ir := r[i];
      if n = q[ir] then
      begin
        q[ir] := n + dq[ir];
        dq[ir] := sq[ir] − dq[ir];
        t := false;
        if i = jj then
        begin
          jj := jj + 1;
          if ir = j then
          begin
            j := j + 1;  r[j] := j;
            q[j] := p[j] ↑ 2;
            sq[j] := 6 × p[j];
            dq[j] := sq[j] × (1+(p[j] ÷ 3)) − 2× q[j]
          end
        end
      end
    end;
    if t then
    begin
      k := k + 1;  p[k] := n;
A:    if jj = 3 then go to F;
      jj := jj − 1;
      if q[r[jj]] < q[r[jj+1]] then go to A;
      comment  sift sort;
      rr[3] := r[3];
      for ir := 4 step 1 until jj do
      begin
        i := ir − 1;
B:      if q[r[ir]] < q[rr[i]] then
        begin
          rr[i+1] := rr[i];  i := i − 1;
          if i ⩾ 3 then go to B
        end;
        rr[i+1] := r[ir]
      end;
      comment  merge sort;
      i := ir := 3;  jr := jj + 1;
C:    if q[rr[ir]] ⩽ q[r[jr]] then
      begin
        r[i] := rr[ir];  ir := ir + 1;
        if ir > jj then go to E
      end
      else
      begin
        r[i] := r[jr];  jr := jr + 1;
        if jr > j then go to D
      end;
      i := i + 1;  go to C;
D:    i := i + 1;  r[i] := rr[ir];  ir := ir + 1;
      if ir ⩽ jj then go to D;
E:    jj := 3
    end;
F: end;
  sieve2 := k
end sieve2
```

REMARKS ON:

ALGORITHM 35 [A1]
Sieve [T. C. Wood, *Comm. ACM 4* (Mar. 1961), 151]
ALGORITHM 310 [A1]
PRIME NUMBER GENERATOR 1 [B. A. Chartres, *Comm. ACM 10* (Sept. 1967), 569]
ALGORITHM 311 [A1]
PRIME NUMBER GENERATOR 2 [B. A. Chartres, *Comm. ACM 10* (Sept. 1967), 570]

B. A. Chartres (Recd. 13 Apr. 1967)
Computer Science Center, University of Virginia,
Charlottesville, Virginia

The three procedures *Sieve(m,p)*, *sieve1(m,p)*, and *sieve2(m,p)*, which all perform the same operation of putting the primes less than or equal to *m* into the array *p*, were tested and compared for speed on the Burroughs B5500 at the University of Virginia. The modification of *Sieve* suggested by J. S. Hillmore [*Comm. ACM 5* (Aug. 1962), 438] was used. It was also found that *Sieve* could be speeded up by a factor of 1.95 by avoiding the repeated evaluation of *sqrt*(n). The modification required consisted of declaring an integer variable s, inserting the statement $s := sqrt(n)$ immediately after $i := 3$, and replacing $p[i] \leq sqrt(n)$ by $p[i] \leq s$.

The running times for the computation of the first 10,000 primes were:

Sieve (Algorithm 35)	845 sec
Sieve (modified)	434 sec
*sieve*1	220 sec
*sieve*2	91 sec

The time required to compute the first k primes was found to be, for each algorithm, remarkably accurately represented by a power law throughout the range $500 \leq k \leq 50{,}000$. The running time of *Sieve* varied as $k^{1.40}$, that of *sieve*1 as $k^{1.53}$, and that of *sieve*2 as $k^{1.35}$. Thus the speed advantage of *sieve*2 over the other algorithms increases with increasing k. However, it should be noted that *sieve*2 took approximately 33 minutes to find the first 100,000 primes, and, if the power law can be trusted for extrapolation past this point (there is no reason known why it should be), it would take about 12 hours to find the first million primes.

ALGORITHM 312
ABSOLUTE VALUE AND SQUARE ROOT OF A COMPLEX NUMBER, [A2]

PAUL FRIEDLAND (Recd. 13 Feb. 1967 and 16 June 1967)
Burroughs Corporation, Pasadena, California

```
real procedure cabs (x,y);
  value x, y;   real x, y;
comment This procedure returns the absolute value of the com-
  plex number x + iy. The procedure provides for the possible
  overflow on x² + y² in | x + iy | = √(x² + y²);
begin
  x := abs (x);  y := abs (y);
  cabs := if x = 0 then y else if y = 0 then x else
          if x > y then x × sqrt (1+(y/x) ↑ 2)
                   else y × sqrt (1+(x/y) ↑ 2)
end cabs;
procedure csqrt (x,y,a,b);
  value x, y;   real x, y, a, b;
comment This procedure computes a and b where a + ib =
  √(x + iy). For x = y = 0 we have that a = b = 0 so we will assume
  that x and y are not both zero.
    Solving simultaneously for a and then b ···
```

$$\text{(1)} \qquad a = \pm \sqrt{\frac{x \pm |x + iy|}{2}}, \quad b = y/(2a)$$

and for b and then a. . .

$$\text{(2)} \qquad b = \pm \sqrt{\frac{-x \pm |x + iy|}{2}}, \quad a = y/(2b)$$

```
    To keep the radical real, we will always use the positive sign
  with | x + iy | and use equation (1) with the sign of "a" taken
  positive for x ≥ 0 and (2) when x < 0, with the sign of "b"
  taken positive for y ≥ 0 and negative for y < 0;
begin
  if x = 0 ∧ y = 0 then a := b := 0 else
  begin
    a := sqrt ((abs (x) + cabs (x, y)) × 0.5);
    if x ≥ 0 then b := y/(a + a) else
    begin
      b := if y < 0 then −a else a;
      a := y/(b + b)
    end
  end
end csqrt
```

ALGORITHM 313
MULTI-DIMENSIONAL PARTITION GENERATOR [A1]

P. BRATLEY AND J. K. S. MCKAY (Recd. 23 Aug. 1966, 15 Feb. 1967 and 14 Apr. 1967)
Dept. of Computer Science, University of Edinburgh

```
procedure partition (N, dim, use);
  value N, dim; integer N, dim; procedure use;
```

comment A partition of N is an ordered sequence of positive integers, $n_1 \geq n_2 \geq n_3 \geq \cdots \geq n_k$, such that $\sum_{i=1}^{k} n_i = N$. Such a partition may be represented by a Ferrers-Sylvester graph of nodes with n_i nodes in the ith row, e.g.,

```
* * * * *
* * * *
* *
* *
```

represents 5, 4, 2, 2. This two-dimensional diagram may be generalized in a natural way to three, or more, dimensions. More formally, we regard a d-dimensional partition of n as a set S of n nodes, each defined by its non-negative integer coordinates such that
$(x_1, x_2, \cdots, x_d) \in S$ if and only if $(x_1', x_2', \cdots, x_d') \in S$
whenever

$$0 \leq x_i' \leq x_i \quad \text{for all} \quad i = 1, 2, \cdots, d.$$

This generalization reduces to the usual definition when $d = 2$. There is little literature on these generalized partitions. It is with a view to facilitating numerical studies that this algorithm is published.

After generation, each partition is presented to the procedure *use*, which should be supplied by the user for the purpose he requires. *use* has three formal parameters, the first being the name of a two-dimensional integer array, and the second and third being integers giving the size of this array. When the procedure is called by

use (*current*, *dim*, *N*)

then the coordinates of the nodes entering into the newly generated multi-dimensional partition will be found in *current* [1:*dim*,1:*N*]. The parameters of *use* should be called by value, or alternatively care should be taken that neither *dim*, *N*, nor the contents of the array *current* are disturbed.

REFERENCES:

1. GUPTA, H., GWYTHER, C. E., AND MILLER, J. C. P. *Tables of Partitions.* Royal Society Mathematical Tables, Vol. 4, Cambridge Univ. Press, 1958.
2. MACMAHON, P. A. *Combinatory Analysis*, Vol. 2, Cambridge Univ. Press, 1916.
3. CHAUNDY, T. W. Partition generating functions. *Quart. J. Math. 2* (1931), 234–240.
4. ATKIN, A. O. L., BRATLEY, P., MACDONALD, I. G., AND McKAY, J. K. S. Some computations for *m*-dimensional partitions. *Proc. Cambridge Phil. Soc.* (to appear);

```
begin
  integer i; integer array current [1:dim, 1:N],
    x[1:dim,0:(N−1)×dim];
  procedure part (n,q,r); value n, q, r; integer n, q, r;
  begin integer s, i, j, k, p, m, z;
    for p:= q step 1 until r − 1 do
    begin
      for i := 1 step 1 until dim do current [i,n] := x[i,p];
      if n = N then begin use (current,dim,N); go to L2 end;
      s := r;
      for i := 1 step 1 until dim do
      begin
        for j := 1 step 1 until dim do x[j,s] := x[j,p];
        x[i,s] := x[i,s] + 1;
        for j := 1 step 1 until dim do
        begin
          if x[j, s] = 0 then go to L3;
          for k := 1 step 1 until n do
          begin
            for m := 1 step 1 until dim do
            begin
              z := if j = m then 1 else 0;
              f ciurrent [m, k] ≠ x[m,s] − z then go to L4
            end;
            go to L3;
L4:
          end k;
          go to L5;
L3:
        end j;
        s := s + 1;
L5:
      end i;
      part (n+1,p+1,s);
L2: end p
  end part;
  for i := 1 step 1 until dim do x[i,0] := 0; part (1,0,1)
end partition
```

ALGORITHM 314
FINDING A SOLUTION OF *N* FUNCTIONAL EQUATIONS IN *N* UNKNOWNS [C5]

D. B. Dulley and M. L. V. Pitteway (Recd. 7 Apr. 1966, 19 Oct. 1966 and 5 July 1967)
Cripps Computing Centre, University of Nottingham, England

```
procedure ndinvt (functions, initstep, error, cycles, x, f, accest, n);
  value n; procedure functions; real initstep, error;
  integer cycles, n; array x, f, accest;
```

comment This procedure performs inverse interpolation in *n* dimensions, i.e., it will find a set of values for *n* variables *x*, such that *n* functions *f*(*x*) are zero. A more sophisticated technique, suitable for large values of *n*, has been developed by S. M. Robinson (Interpolative Solution of Systems of Nonlinear Equations, *SIAM Journal of Numerical Analysis*, *3* (1966), 650–658). It can also be used to fit a curve with *n* arbitrary parameters to a set of points, the *n* functions being formed, in this case, by equating to zero the differential of the sum of the squares of the residues with respect to each parameter in turn.

The functions required are specified by a procedure of the form *functions* (*f*, *x*) where *f* and *x* are declared as arrays from 1 to *n*. This procedure should calculate the *n* functions from a set of values given in *x*, placing the results in *f*. The first step is made by forming partial derivatives over an interval *initstep*. $1_{10} - 6$ should be suitable for values of *x* of the order 1 to 10. Exit from the procedure will occur if:

(i) the root sum square of the *x* increments is less than *error*. If *error* is negative, this condition must be satisfied for | *error* |, and in addition this process is continued until the root sum square of the incrementsfails to decrease

or (ii) the number of iterations is greater than *cycles*, implying that too much accuracy has been requested

or (iii) the specified equations are singular. In this case exit is by a jump to a label *fails*.

On entry, the array *x* should contain the starting values. On exit, the array *x* will contain the accurate root, *f* the residues and *accest* the last increments made to *x* as a measure of the accuracy.

This procedure calls on a global procedure *eqnsolve* (*A*, *b*, *n*, *label*), which solves *n* linear simultaneous equations in *n* unknowns *Ax* = *b*, placing the result in *b*. If *A* is singular, it is assumed that an exit is made by a jump to *label*;

```
begin
  real work, sumsqres, prevres;
  integer i, j, count;
  Boolean switch;
  array prevf[1:n], copydelf[1:n, 1:n], delx, delf[1:n, 1:n+1];
  functions(prevf, x);
  for i := 1 step 1 until n do
  begin
    x[i] := x[i] + initstep;
    functions (f, x);
    for j := 1 step 1 until n do
    begin
      delf[i, j] := f[j] − prevf[j];
      delx[i, j] := 0;
    end differencing initial point;
    delx[i, i] := initstep;
    x[i] := x[i] − initstep;
  end setting up the initial matrix of points;
  sumsqres := $1_{10}30$;
  count := 0;
iterate:
  switch := true;
  prevres := sumsqres;
tryagain:
  for i := 1 step 1 until n do
  begin
    f[i] := prevf[i];
    for j := 1 step 1 until n do copydelf[i, j] := delf[i, j]
  end copying delf for destructive use in procedure eqnsolve;
  eqnsolve (copydelf, f, n, inline);
  sumsqres := 0;
  for := 1 step 1 until n do
  begin
    work := 0;
    for j := 1 step 1 until n do work := work − delx[i, j] × f[j];
    accest[i] := work;
    x[i] := x[i] + work;
    sumsqres := sumsqres + work × work
  end calculation of next point;
  count := count + 1;
  functions (f, x);
  if count > cycles ∨ sumsqres < error × error ∧
    (error > 0 ∨ sumsqres > prevres) then go to exit;
  for i := 1 step 1 until n do
  begin
    work := f[i] − prevf[i];
    prevf[i] := f[i];
    for j := n step − 1 until 1 do
    begin
      delx[i, j+1] := delx[i, j] − accest[i];
      delf[i, j+1] := delf[i, j] − work
    end calculation of new differences;
    delx[i, 1] := −accest[i];
    delf[i, 1] := −work
  end moving points up one place in tables;
  go to iterate;
inline:
  for i := 1 step 1 until n do
  begin
    delx[i, n] := delx [i, n+1];
    delf[i, n] := delf[i, n+1]
  end discarding alternative point;
  switch := ¬ switch;
  if switch then go to fails else go to tryagain;
exit:
end ndinvt
```

REMARK ON ALGORITHM 314 [C5]
FINDING A SOLUTION OF *N* FUNCTIONAL

EQUATIONS IN N UNKNOWNS [D. B. Dulley and M. L. V. Pitteway, *Comm. ACM 10* (Nov. 1967), 726].

JAMES VANDERGRAFT AND CHARLES MESZTENYI

(Recd. 12 Aug. 1968)

Computer Science Center, University of Maryland, College Park, MD 20742

KEY WORDS AND PHRASES: functional equations, interpolation, nonlinear equations, secant method
CR CATEGORIES: 5.13, 5.15

The algorithm, as published, requires four iterations to find the solution to a pair of linear equations. The difficulty seems to lie in the last statement of the first column. If this is replaced by

$$delf\ [j,i] := f[j] - prevf[j];$$

then the algorithm works well. In fact, however, it is now simply an n-dimensional secant method, which can be described by the iteration

$$x^{k+1} = x^k - \delta x_k(\delta F_k)^{-1} F(x^k), \quad k = 0,1,2,. . .,$$

where δF_k and δx_k are matrices whose ith columns are $f(x^{k-i}) - f(x^k)$ and $x^{k-i} - x^k$, respectively. The iteration is started by setting

$$x^{-i} = x^0 + h\ e_i$$

where x^0 is a given vector, h is a small positive constant, and e_i is the ith unit coordinate vector.

It should be observed, also, that the algorithm will not break down if δx_k becomes singular. However, if this should happen it means that $x^k, x^{k-1}, . . ., x^{k-n}$ lie in a proper subspace S of E^n, Euclidean n-space, and all successive iterates will also lie in S. Hence the algorithm may converge to a point in S which is not a solution to $f(x) = 0$. To prevent this, the norm of $f(x)$ should be checked before leaving the procedure.

ALGORITHM 315
THE DAMPED TAYLOR'S SERIES METHOD FOR MINIMIZING A SUM OF SQUARES AND FOR SOLVING SYSTEMS OF NONLINEAR EQUATIONS [E4, C5]

H. Späth (Recd. 25 Oct. 1966 and 19 June 1967)
Institut für Neutronenphysik und Reaktortechnik
Kernforschungszentrum Karlsruhe, Germany

```
procedure TAYLOR (n, m, x, h, f, itmax, eps1, eps2, der, S, KENN,
  EXIT);
  value n, m, eps1, eps2; integer n, m, itmax, KENN;
    real eps1, eps2, S;
  Boolean der; array x, h, f; label EXIT;
comment
```

Let

$$S(x_1,\cdots,x_n) = \sum_{i=1}^{m} f_i^2(x_1,\ldots,x_n) \qquad (m \geqq n) \tag{1}$$

the function to be minimized. Such functions always appear if you apply the method of least squares to estimate nonlinear parameters. The following sequence

$$x^{(k+1)} = x^{(k)} - \beta\Delta x^{(k)} = x^{(k)} - \beta(F'^{T}_{x^{(k)}}F'_{x^{(k)}})^{-1}F'^{T}_{x^{(k)}}F(x^{(k)})$$

$$F = (f_1,\cdots,f_m), \quad F'_x = \left(\frac{\partial f_i}{\partial x_j}\right) i = 1,\cdots,m, j = 1,\cdots,n \tag{2}$$

where β, which is always possible, is chosen to be such that

$$S(x^{(k)}-\beta\Delta x^{(k)}) \leqslant (1-\beta\lambda)S(x^{(k)}) \qquad (0<\lambda<1) \tag{3}$$

is known to converge [1] for any $x^{(0)}$ to a stationary point of S ($grad\, S = 2F'^{T}_{x}F(x) = 0$), if on the carrying out of the iteration the matrix $F'^{T}_{x}F'_x$ does not become singular.

For $m = n$ you have $\Delta x = F'^{-1}_{x}F(x)$ and (2) becomes a damped version of Newton's method for solving the system of nonlinear equations

$$F(x) = 0 \tag{4}$$

All zeros of (4) are stationary points of (1). Thus we are able to generate a sequence which converges for any $x^{(0)}$ to a stationary point of (1) and the possible divergence of Newton's method ($\beta=1$) is avoided. It is not assured, however, that the method will always converge to a solution of (4). Numerical experience has shown that though Newton's method ($\beta=1$) diverges for a certain $x^{(0)}$ the damped sequence converges to a solution of (4) for the same $x^{(0)}$.

In the program we have chosen $\lambda = .2$. At each iteration we set first $\beta = 1$ and then, if (3) is not valid, $\beta = 2^{-j}$ ($j=1,2,\ldots,16$). If j is greater than 16 then $\beta < .00002$ and we assume to have reached a stationary point of S.

Meaning of the formal parameters:

n the number of variables x_i

m the number of functions f_i

x the array $x[1:n]$ which must first contain a starting value $x^{(0)}$ and finally will contain a stationary point of S, if $F'^{T}_{x}F'_x$ or for $m = n$ F'_x, respectively, has not become singular

h $h[1:n]$ is a step size vector for the approximation of F'_x (see below)

f the array $f[1:m]$ will contain the function values at the last x calculated in *TAYLOR*

itmax must initially contain the maximum number of iterations to be performed. Leaving *TAYLOR* regularly, *itmax* contains the actual number of performed iterations

*eps*1 the iteration is stopped when $S < eps1$

*eps*2 the iteration is discontinued when $\sum_{i=1}^{n} | \Delta x_i^{(k)}| < eps2 \times \sum_{i=1}^{n} | x_i^{(k+1)} |$

der if *der* = **true** the matrix F'_x must be produced by a global procedure named *DERIVE*(*x*, *dfdx*) which adjoins to the vector $x[1:n]$ the array $dfdx[1:m, 1:n]$. In this case the array *h* can be loaded by an arbitrary vector, for instance *x*.

if *der* = **false** the matrix F'_x is approximated by

$$\frac{\partial f_i}{\partial x_j} = \frac{f_i(x_1,\ldots,x_j+h_j,\ldots,x_n) - f_i(x_1,\ldots,x_j-h_j,\ldots,x_n)}{2h_j}$$

where h is a given step size vector. With a suitable choice of the h_j the convergence behavior of the sequence (2) is not destroyed. *DERIVE*(*x*, *dfdx*) must be formally declared outside of *TAYLOR* in this case.

[In some cases, particularly when solving nonlinear equations, the extra accuracy achieved by using central differences to estimate the derivatives is not necessary. A considerable saving in execution time can be obtained by using one-sided differences which means only minor changes in the program below. —Ref.]

S should initially contain the greatest positive number that the employed computer can store. Finally S contains $S = S(x^{(itmax)})$, if *TAYLOR* is regularly left.

KENN if after having called *TAYLOR*

KENN = 0 then one of the above interruptions applies (*eps*1, *eps*2),

KENN = 1 then *itmax* iterations were carried out and *TAYLOR* is left,

KENN = −1 then $\beta = 2^{-17}$ and *TAYLOR* is left.

EXIT *TAYLOR* goes to this global label if i encounters a singular matrix.

Further two global procedures must be made available to *TAYLOR*:

i) *FUNCTION*(*x*, *f*) which is able to calculate for a given vector $x[1:n]$ the function values $f[1:m]$

ii) *GAUSS*(*n*, *A*, *b*, *x*, *EXIT*) which solves the linear system of *n* equations $Ax = b$ for *x*. If *A* is singular then *GAUSS* returns to the global label *EXIT*. Any linear equation solver may be used for *GAUSS*;

```
begin integer i, j, k, z, l;  real hf, hl, hs, hz;
  array fp, fm[1:m], b, dx[1:n], dfdx[1:m, 1:n], aa[1:n, 1:n];
  hs := S;  KENN := z := 0;
ITERATION: z := z + 1;
  if z > itmax then begin KENN := 1;  go to ENDE end;
    l := 0;  hl := 1.0;
DAMP: l := l + 1;
  if l > 16 then begin KENN := −1;  go to ENDE end;
  FUNCTION(x, f);  hf := 0;
  for i := 1 step 1 until m do hf := hf + f[i] × f[i];
  if hf > hs × (1.0 − .2 × hl) then
  begin hl := hl × .5;
    for k := 1 step 1 until n do x[k] := x[k] + hl × dx[k];
    go to DAMP
  end;
  hs := hf;  if hs < eps 1 then go to ENDE;
  if der then DERIVE(x, dfdx) else
  begin
    for i := 1 step 1 until n do
    begin hf := h[i];  hz := 2.0 × hf;
      x[i] := x[i] + hf;  FUNCTION(x, fp);
      x[i] := x[i] − hz;  FUNCTION(x, fm);
      x[i] := x[i] + hf;  hz := 1.0/hz;
      for k := 1 step 1 until m do
      dfdx[k, i] := hz × (fp[k] − fm[k])
    end
  end;
if m = n then GAUSS(n, dfdx, f, dx, EXIT) else
begin
  for i := 1 step 1 until n do
  begin hf := 0;
    for k := 1 step 1 until m do
      hf := hf + dfdx[k, i] × f[k];  b[i] := hf;
    for k := i step 1 until n do
    begin hf := 0;
      for j := 1 step 1 until m do
        hf := hf + dfdx[j, i] × dfdx[j, k];
      aa[i, k] := aa[k, i] := hf
    end
  end;
  GAUSS(n, aa, b, dx, EXIT)
  end;
  hz := hf := 0;
  for i := 1 step 1 until n do
  begin
    x[i] := x[i] − dx[i];  hz := hz + abs(x[i]);
    hf := hf + abs(dx[i])
  end;
  if hf ≥ eps2 × hz then go to ITERATION;
ENDE: FUNCTION(x, f);  S := 0;  itmax := z;
  for i := 1 step 1 until m do S := S + f[i] × f[i]
end TAYLOR
```

REFERENCE:

[1] BRAESS, D. Über Dämpfung bei Minimalisierungsverfahren. *Computing 1* (1966), 264–272.

REMARK ON ALGORITHM 315 [E4, C5] THE DAMPED TAYLOR'S SERIES METHOD FOR MINIMIZING A SUM OF SQUARES AND FOR SOLVING SYSTEMS OF NONLINEAR EQUATIONS [H. Späth, *Comm. ACM 10* (Nov. 1967), 726].

GARY SILVERMAN (Recd. 4 Mar. 1969, 14 Apr. 1969 and 11 June 1969)
IBM Scientific Center, Los Angeles, CA 90067.

KEY WORDS AND PHRASES: solution of equations, least squares approximation, Newton's method
CR CATEGORIES: 5.13, 5.14, 5.1 5

The algorithm, as published, may introduce unnecessary truncation error into the solution. If the matrix F'_x is approximated by central differences (*der* = **false**) then the value of the iterate is used to compute these differences. This involves two additions to and one subtraction from the iterate, each of which may result in truncation error. To correct this, the following statements on page 727

```
x[i] := x[i] + hf;  FUNCTION (x, fp);
x[i] := x[i] − hz;  FUNCTION (x, fm);
x[i] := x[i] + hf;  hz := 1.0/hz;
```

may be replaced by

```
hh := x[i];
x[i] := x[i] + hf;  FUNCTION (x, fp);
x[i] := x[i] − hz;  FUNCTION (x, fm);
x[i] := hh;  hz := 1.0/hz;
```

after declaring an additional real variable *hh*.

In solving two equations in two unknowns the published algorithm converged to a solution with $S = 8.83653 \times 10^{-13}$ and $KENN = -1$. After the above modification convergence was with $S = 0$ and $KENN = 0$.

ALGORITHM 316
SOLUTION OF SIMULTANEOUS NON-LINEAR EQUATIONS [C5]

K. M. Brown (Recd. 27 Oct. 1966, 31 Mar. 1967, 17 July 1967, and 26 July 1967)
Department of Computer Science, Cornell University, Ithaca, New York

```
procedure nonlinearsystem (n, maxit, numsig, singular, x);
  value n, numsig; integer n, maxit, numsig, singular; array x;
comment This procedure solves a system of n simultaneous
  nonlinear equations. The method is roughly quadratically con-
  vergent and requires only ((n^2/2)+(3n/2)) function evaluations
  per iterative step as compared with (n^2+n) evaluations for
  Newton's Method. This results in a savings of computational
  effort for sufficiently complicated functions. A detailed de-
  scription of the general method and proof of convergence are
  included in [1]. Basically the technique consists in expanding
  the first equation in a Taylor series about the starting guess,
  retaining only linear terms, equating to zero and solving for
  one variable, say x_k , as a linear combination of the remaining
  n − 1 variables. In the second equation, x_k is eliminated by
  replacing it with its linear representation found above, and
  again the process of expanding through linear terms, equating
  to zero and solving for one variable in terms of the now remain-
  ing n − 2 variables is performed. One continues in this fashion,
  eliminating one variable per equation, until for the nth equa-
  tion, we are left with one equation in one unknown. A single
  Newton step is now performed, followed by back-substitution
  in the triangularized linear system generated for the x_i's. A
  pivoting effect is achieved by choosing for elimination at any
  step that variable having a partial derivative of largest absolute
  value. The pivoting is done without physical interchange of
  rows or columns.
    The vector of initial guesses x, the number of significant digits
  desired numsig, the maximum number of iterations to be used,
  maxit, and the number of equations n, should be set up prior to
  the procedure call which activates nonlinearsystem. After execu-
  tion of the procedure, the vector x is the solution of the system
  (or best approximation thereto), maxit is now the number of
  iterations used and singular = 0 is an indication that a Jaco-
  bian-related matrix was singular—indicative of the process
  "blowing-up," whereas singular = 1 is an indication that no
  such difficulty occurred. Storage space may be saved by imple-
  menting the algorithm in a way which takes advantage of the
  fact that the strict lower triangle of the array pointer and the
  same number of positions in the array coe are not used;
begin integer converge, m, j, k, i, jsub, itemp, kmax, kplus, tally;
  real f, hold, h, fplus, dermax, test, factor, relconvg;
  integer array pointer[1:n, 1:n], isub[1:n−1];
  array temp, part[1:n], coe[1:n, 1:n+1];
  procedure backsubstitution (k, n, x, isub, coe, pointer);
    value k, n;
    integer k, n; integer array isub, pointer; array x, coe;
  comment This procedure back-solves a triangular linear
    system for improved x[i] values in terms of old ones;
  begin integer km, kmax, jsub;
    for km := k step −1 until 2 do
    begin kmax := isub[km−1]; x[kmax] := 0;
      for j := km step 1 until n do
      begin jsub := pointer[km, j];
        x[kmax] := x[kmax] + coe[km−1, jsub] × x[jsub]
      end;
      x[kmax] := x[kmax] + coe[km−1, n+1]
    end;
  end backsubstitution;
  procedure evaluatekthfunction (x, y, k);
    integer k; real y; array x;
  begin comment the body of this procedure must be provided
    by the user. One call of the procedure should cause the value
    of the kth function at the current value of the vector x to be
    placed in y;
  end evaluatekthfunction;
  converge := 1; singular := 1; relconvg := 10 ↑ (-numsig);
  for m := 1 step 1 until maxit do
  begin
    comment An intermediate output statement may be in-
      serted at this point in the procedure to print the successive
      approximation vectors x generated by each complete itera-
      tive step;
    for j := 1 step 1 until n do pointer [1, j] := j;
    for k := 1 step 1 until n do
    begin if k > 1 then backsubstitution (k, n, x, isub, coe, pointer);
      evaluatekthfunction (x, f, k); factor := .001;
AAA:      tally := 0; for i := k step 1 until n do
      begin itemp := pointer[k, i]; hold := x[itemp];
        h := factor × hold; if h = 0 then h := .001;
        x[itemp] := hold + h;
        if k > 1 then backsubstitution (k, n, x, isub, coe, pointer);
        evaluatekthfunction (x, fplus, k);
        part[itemp] := (fplus−f)/h;
        x[itemp] := hold; if (abs(part[itemp])=0)∨
        (abs(f/part[itemp]) > 1.0⏨20) then tally := tally + 1;
      end;
      if tally ≦ n − k then go to AA; factor := factor × 10.0;
      if factor > .5 then go to SING; go to AAA;
AA:   if k < n then go to A; if abs (part[itemp]) = 0
      then go to SING;
      coe[k, n+1] := 0; kmax := itemp; go to ENDK;
A:    kmax := pointer[k, k]; dermax := abs(part[kmax]);
      kplus := k + 1;
      for i := kplus step 1 until n do
      begin jsub := pointer[k, i]; test := abs(part[jsub]);
        if test < dermax then go to B; dermax := test;
        pointer [kplus, i] := kmax; kmax := jsub;
        go to ENDI;
B:      pointer [kplus, i] := jsub;
ENDI:
      end;
      if abs(part[kmax]) = 0 then go to SING; isub[k] := kmax;
      coe[k, n+1] := 0;
      for j := kplus step 1 until n do
      begin jsub := pointer[kplus, j];
        coe[k, jsub] := −part[jsub]/part[kmax];
        coe[k, n+1] := coe[k, n+1] + part[jsub] × x[jsub]
      end;
ENDK:
      coe[k, n+1] := (coe[k, n+1]−f)/ part[kmax] + x[kmax]
    end k;
    x[kmax] := coe[n, n+1];
    if n > 1 then backsubstitution (n, n, x, isub, coe, pointer);
    if m = 1 then go to D;
    for i := 1 step 1 until n do
      if abs((temp[i]-x[i])/x[i]) > relconvg then go to C;
     converge := converge + 1;
     if converge ≧ 3 then go to TERMINATE else go to D;
C:   converge := 1;
D:   for i := 1 step 1 until n do temp[i] := x[i]
  end m;
  go to THROUGH;
TERMINATE:
  maxit := m;  go to THROUGH;
SING:
  singular := 0;
THROUGH:
  end nonlinearsystem
```

APPENDIX

We include a sample procedure *evaluatekthfunction* for the 2 × 2 system:

$$\left(1 - \frac{1}{4\pi}\right)(e^{2x_1} - e) + \frac{e}{\pi} x_2 - 2ex_1 = 0$$

$$\frac{1}{2}\sin(x_1 x_2) - \frac{x_2}{4\pi} - \frac{x_1}{2} = 0,$$

one solution of which is (.5, π) see [2]

```
procedure evaluatekthfunction (x, y, k);
  integer k;  real y;  array x;
begin switch functionnumber := F1, F2;
  go to functionnumber [k];
F1:  y := 2.71828183 × (.920422528 × (exp(2×x[1]-1)-1)+
         x[2]/3.14159265-2×x[1]);
  go to RETURN;
F2:  y := .5 × sin(x[1]×x[2]) - x[2]/12.5663706 - x[1]/2;
RETURN:
end evaluatekthfunction;
```

REFERENCES:

1. BROWN, K. M. A quadratically convergent method for solving simultaneous non-linear equations. Doctoral Thesis, Dept. Computer Sciences, Purdue U., Lafayette, Ind., Aug., 1966.
2. BROWN, K. M., AND CONTE, S. D. The solution of simultaneous nonlinear equations. Proc. ACM 22nd Nat. Conf., pp 111–114.

Remark on Algorithm 316 [C5]

Solution of Simultaneous Nonlinear Equations
(K.M. Brown, *Comm. ACM 10* (Nov. 1967), 728–729)

William J. Raduchel (Recd. 12 Aug. 1970
and 8 Jan. 1971)
Project for Quantitative Research in Economic
Development, Harvard University,
Cambridge, MA 02138

Key Words and Phrases: nonlinear equations
CR Categories: 5.15

The procedure was coded in both Burroughs 5500 ALGOL and IBM FORTRAN-IV and ran correctly on the sample problem provided. However, two changes seem appropriate: In the loop to compute the partial derivatives following *AAA* replace

```
if h = 0 then h := 0.001;
```

with

```
if h = 0 then h := factor;
```

for otherwise the purpose of the loop is lost for variables currently having the value zero. To avoid an interrupt for a zero-divide replace

```
if abs ((temp [i]-x[i])/x[i]) > relconvg
  then go to C;
```

with

```
if abs ((temp [i]-x[i])/(if x[i] ≠ 0 then x[i] else if temp [i] ≠ 0
  then temp [i] else 1)) > relconvg then go to C
```

As the author indicates there are unused positions in the arrays *pointer* and *coe* because of the triangularity of the method. Implementing the algorithm to use this fact to conserve storage is much easier if, in both the main procedure and in backsubstitution, values are stored and retrieved in natural order rather than according to the current pivot scheme.

ALGORITHM 317*
PERMUTATION [G6]

Charles L. Robinson (Recd. 12 Apr. 1967, 2 May 1967 and 10 July 1967)

Institute for Computer Research, U. of Chicago, Chicago, Ill.

* This work was supported by AEC Contract no. AT (11-1)-614.

```
procedure permute(n, k, v); value n, k; integer array v;
    integer n, k;
comment This procedure produces in the vector v the kth
  permutation on n variables. When k = 0, v takes on the value
  1, 2, 3, 4, ··· , n. This algorithm is not as efficient as pre-
  viously published algorithms [1], [2], [3] for generating a
  complete set of permutations but it is significantly better
  for generating a random permutation, a property useful in
  certain simulation applications. Any non-negative value of
  k will produce a valid permutation. To generate a random
  permutation, k should be chosen from the uniform distribu-
  tion over the integers from 0 to n! - 1 inclusive;
begin integer i, q, r, x, j;
  for i := 1 step 1 until n do v[i] := 0;
  for i := n step -1 until 1 do
  begin
q := k ÷ i;  r := k - q × i;  x := 0;  j := n;
no: if v[j] = 0 then
      begin
        if x = r then go to it else x := x + 1
      end;
      j := j - 1;  go to no;
it:  v[j] := i;  k := q;
    end
end
```

References:

1. Coveyou, R. R., and Sullivan, J. G. Algorithm 71, Permutation. *Comm. ACM 4* (Nov. 1961), 497.
2. Peck, J. E. L., and Schrack, G. F. Algorithm 86, Permute. *Comm. ACM 5* (Apr. 1962), 208.
3. Trotter, H. F. Algorithm 115, Perm. *Comm. ACM 5* (Aug. 1962), 434.

ALGORITHM 318
CHEBYSCHEV CURVE-FIT (REVISED) [E2]

J. Boothroyd (Recd. 15 May 1967)
University of Tasmania, Hobart, Tas., Australia

```
procedure chebfit(x, y, n, a, m);  value n, m;
  array x, y, a;  integer n, m;
comment evaluates, in a[0] through a[m] of a[0:m+1], the co-
  efficients of an mth order polynomial $P(x) = a_0 + a_1x + \cdots a_mx^m$
  such that the maximum error $abs(P(x_i)-y_i)$ is a minimum over
  the n(n>m+1) sample points x, y[1:n]. The x[i] must form a
  strictly monotonic sequence.
    This procedure is an extensive revision of Algorithm 91 (Albert
  Newhouse, Chebyshev Curve-Fit, Comm. ACM 5 (May 1961),
  281). The polynomial P(x) is a best-fit polynomial in the Cheby-
  shev sense as described by Stiefel (Numerical Methods of Tcheby-
  cheff Approximation), in Langer (Ed.), On Numerical Approxi-
  mation, U. of Wisconsin Press, 1959, pp. 217-232. Stiefel (p. 221)
  shows that the procedure must terminate after a finite number
  of steps. This is not always so with imperfect arithmetic, where
  roundoff errors may cause cycling of the chosen reference sets.
  This condition is detected by checking that the reference devia-
  tion is always raised monotonically. At exit the absolute value
  of a[m+1] yields the final reference deviation. Negative a[m+1]
  indicates that the procedure has been terminated following the
  detection of cycling;
begin
  integer i, j, k,mplus1, ri, i1, imax, rj, j1;
  real d, h, ai1, rhi1, denom, ai, rhi, xj, hmax, himax, xi, hi, abshi,
    nexthi, prevh;
  integer array r[0:m+1];  array rx, rh[0:m+1];
  mplus1 := m + 1;  prevh := 0;
  comment index vector for initial reference set;
  r[0] := 1;  r[mplus1] := n;
  d := (n-1)/mplus1;  h := d;
  for i := 1 step 1 until m do
  begin r[i] := h + 1;  h := h + d end;
start: h := -1.0;
  comment select m + 2 reference pairs and set alternating
    deviation vector;
  for i := 0 step 1 until mplus1 do
  begin
    ri := r[i];
    rx[i] := x[ri];  a[i] := y[ri];
    rh[i] := h := -h
  end i;
  comment compute m + 1 leading divided differences;
  for j := 0 step 1 until m do
  begin
    i1 := mplus1;  ai1 := a[i1];
    rhi1 := rh[i1];
    for i := m step -1 until j do
    begin
      denom := rx[i1] - rx[i-j];
      ai := a[i];  rhi := rh[i];
      a[i1] := (ai1-ai)/denom;
      rh[i1] := (rhi1-rhi)/denom;
      i1 := i;  ai1 := ai;  rhi1 := rhi
    end i
  end j;
  comment equate (m+1)th difference to zero to determine h;
  h := -a[mplus1]/rh[mplus1];
  comment with h known, combine the function and deviation
    differences;
  for i := 0 step 1 until mplus1 do
    a[i] := a[i] + rh[i] × h;
  comment compute polynomial coefficients;
  for j := m - 1 step -1 until 0 do
  begin
    xj := rx[j];  i := j;  ai := a[i];
    for i1 := j + 1 step 1 until m do
    begin
      ai1 := a[i1];
      a[i] := ai - xj × ai1;
      ai := ai1;  i := i1
    end i1
  end j;
  comment if the reference deviation is not increasing mono-
    tonically then exit;
  hmax := abs(h);
  if hmax ≤ prevh then
  begin a[mplus1] := -hmax;  go to fit end;
  comment find the index, imax, and value, himax, of the largest
    absolute error for all sample points;
  a[mplus1] := prevh := hmax;  imax := r[0];  himax:= h;
  j := 0;  rj := r[j];
  for i := 1 step 1 until n do
    if i ≠ rj then
  begin
    xi := x[i];  hi := a[m];
    for k := m - 1 step -1 until 0 do
    hi := hi × xi + a[k];
    hi := hi - y[i];  abshi := abs(hi);
    if abshi > hmax then
    begin hmax := abshi;  himax := hi;  imax := i end
  end
  else
  if j < mplus1 then
  begin j := j + 1;  rj := r[j] end;
  comment if the maximum error occurs at a nonreference
    point, exchange this point with the nearest reference point
    having an error of the same sign and repeat;
  if imax ≠ r[0] then
  begin
    for i := 0 step 1 until mplus1 do
    if imax < r[i] then go to swap;
    i := mplus1;
swap: nexthi := if i - i ÷ 2 × 2 = 0 then h else -h;
    if himax × nexthi ≥ 0 then r[i] := imax
    else
    if imax < r[0] then
    begin
      j1 := mplus1;
      for j := m step -1 until 0 do
      begin r[j1] := r[j];  j1 := j end;
      r[0] := imax
    end
    else
    if imax > r[mplus1] then
    begin
```

```
      j := 0;
      for j1 := 1 step 1 until mplus1 do
      begin r[j] := r[j1]; j := j1 end;
      r[mplus1] := imax
    end
    else r[i-1] := imax;
    go to start
  end;
fit:
end chebfit
```

ALGORITHM 319
TRIANGULAR FACTORS OF MODIFIED MATRICES [F1]

DAVID R. GREEN (Recd. 26 Apr. 1965, 19 Oct. 1965 and 30 Aug. 1967)*
Mount Isa Mines Ltd., Queensland, Australia

KEY WORDS AND PHRASES: matrix decomposition, matrix factors, matrix modifier, matrix perturbation
CR CATEGORIES: 5.14

```
procedure modifacs (l,c,x,m,n,epsilon,fail);
  value epsilon,m,n;  array l,c,x;  integer m,n;
  real epsilon;  label fail;
comment Suppose that the symmetric, positive definite, n × n
  matrix a has been decomposed into the matrix product l.l^T
  where l is a lower triangular matrix and T denotes transpose.
  If a is to be modified by the addition of a matrix triple product
  x.c.x^T, this procedure will modify l, in its own space, to pro-
  duce the triangular factors of a + x.c.x^T in approximately mn^2
  operations (x is an n × m matrix, c is a symmetric, m × m
  matrix, m ≥ 1, m ≪ n).
    This situation can arise, for example, in some treatments of
  network flow problems and the elastic plastic analysis of plane
  frames. The referee has pointed out that a further very useful
  application would be updating least squares solutions when
  additional readings have been obtained. A full description of
  the algorithm for general matrices is given by J. M. Bennett,
  Triangular Factors of Modified Matrices, Numer. Math. 7 (1965),
  217-221.
    On entry, array l should hold the lower triangular matrix l.
  Elements above the diagonal of l are ignored by the procedure.
  On exit the modified values of l are held in the same format.
  The method will fail if the resulting matrix a + x.c.x^T is not
  positive definite, so should the absolute value of any pivot be
  less than the parameter epsilon, or should a pivot be negative,
  then exit through fail will occur;
begin
  array p[1:m];
  integer i,j,k;
  real d,t;
  i := 1;
repeat:
  d := l[i,i];
  t := d↑2;
  for k := 1 step 1 until m do
  begin
    p[k] := 0;
    for j := 1 step 1 until m do
    p[k] := p[k] + x[i,j] × c[j,k];
    t := t + x[i,k] × p[k]
  end;
  if t < epsilon then go to fail;
  l[i,i] := sqrt(t);
  if i = n then go to exit;
  for j := 1 step 1 until m do
    p[j] := p[j]/l[i,i];
  for j := i + 1 step 1 until n do
  begin
    l[j,i] := l[j,i]/d;
    t := 0.0;
    for k := 1 step 1 until m do
    begin
      x[j,k] := x[j,k] − x[i,k] × l[j,i];
      t := t + x[j,k] × p[k]
    end;
    l[j,i] := l[i,i] × l[j,i] + t
  end;
  for j := 1 step 1 until m do
  begin
    c[j,j] := c[j,j] − p[j]↑2;
    if j < m then
    for k := j + 1 step 1 until m do
    c[j,k] := c[k,j] := c[j,k] − p[j] × p[k]
  end;
  i := i + 1;
  go to repeat;
exit:
end
```

ALGORITHM 320
HARMONIC ANALYSIS FOR SYMMETRICALLY DISTRIBUTED DATA [C6]

D. B. Hunter (Recd. 1 June 1965, 4 Jan. 1966, and 26 June 1967)
Department of Mathematics, University of Bradford, Yorkshire, England

KEY WORDS AND PHRASES: harmonic analysis, cosine series, sine series, function approximation, curve fitting, trigonometric series
CR CATEGORIES: 5.13

```
procedure trigfit (index, n, m, h, e, x, f, mt, a);
value index, n, m, h, e; integer index, n, m, mt; real h, e; array
    x, f, a;
```

comment Approximates a function y of x by a half-range cosine or sine series of period $2h$ from values specified at discrete points, not necessarily equally-spaced, in the range $(0, h)$. The input parameters are:

index—if $index = 0$, a cosine series is fitted, if $index = 1$, a sine series. No other value is permitted.
n—number of function-values given.
m—order of the highest harmonic required.
h—half-period of the fitted series.
e—used to terminate the process if rounding errors start to accumulate excessively (see note below).
x—the given values of x are stored on $x[1], x[2], \cdots, x[n]$.
f—the value of y corresponding to $x == x[i]$ is stored on $f[i]$ $(i=1, 2, \cdots, n)$.

The procedure then calculates the coefficients $a[r]$ in the approximation

$$S(x) = \begin{cases} \frac{1}{2}a[0] + \sum_{r=1}^{mt} a[r] \cos (r\pi x/h) & \text{if } index = 0, \\ \sum_{r=1}^{mt} a[r] \sin (r\pi x/h) & \text{if } index = 1. \end{cases}$$

Here normally $mt = m$, but provision is included to calculate fewer harmonics if rounding errors begin to accumulate excessively (see note below).

Method of calculation. The coefficients $a[r]$ are calculated so as to minimize the sum

$$\sum_{i=1}^{n} w_i(f[i]-S(x[i]))^2, \quad w_i = \begin{cases} \frac{1}{2} \text{ if } x[i] = 0 \text{ or } h, \\ 1 \text{ otherwise.} \end{cases}$$

The method used is similar to that of [1]. First $S(x)$ is expanded in the form

$$S(x) = \sum_{i=index}^{mt} b_i p_i(x)$$

where

$$p_i(x) = \begin{cases} \frac{1}{2}a_{i0} + \sum_{j=1}^{i} a_{ij} \cos (j\pi x/h) & \text{if } index = 0, \\ \sum_{j=1}^{i} a_{ij} \sin (j\pi x/h) & \text{if } index = 1. \end{cases}$$

Then

$$a[r] = \sum_{i=r}^{mt} b_i a_{ir} .$$

The polynomials $p_j(x)$ are chosen so as to be orthogonal w.r.t. summation over $x = x[i]$, with weights w_i . This implies that

$$b_i = \sum_{j=1}^{n} w_j f[j] p_i(x[j]) / \sum_{j=1}^{n} w_j [p_i(x_j)]^2.$$

The $p_i(x)$ are generated by a recurrence relation

$$p_{i+1}(x) = (2 \cos (\pi x/h) - \alpha_i) p_i(x) - \beta_i p_{i-1}(x)$$

where

$$\alpha_i = \frac{2 \sum_{j=1}^{n} w_j \cos (\pi x[j]/h) \cdot [p_i(x[j])]^2}{\sum_{j=1}^{n} w_j [p_i(x[j])]^2} \qquad (i \geq index),$$

$$\beta_i = \frac{\sum_{j=1}^{n} w_j [p_i(x[j])]^2}{\sum_{j=1}^{n} w_j [p_{i-1}(x[j])]^2} \qquad (i > index).$$

The initial forms are

$p_0(x) = \frac{1}{2}$ if $index = 0$

or $p_1(x) = \sin (\pi x/h)$ if $index = 1$.

Thus if the $x[i]$ are equally spaced, i.e. if $x[i] = (i-1)h/(n-1)$, it follows that
$p_k(x) = \cos (k\pi x/h)$ or $\sin (k\pi x/h)$ according as $index = 0$ or 1.

The values of the $p_i(x)$ are calculated by the method of [2].

Note. If the $x[i]$ are verp irregular in their distribution serious rounding errors may accumulate, and it is recommended that the points be as nearly as possible equally spaced. However the procedure includes provision, under control of parameter e, to reduce the number of harmonics calculated, mt, if rounding errors do start to build up.

Rounding error is controlled by estimating the error which would occur in the analysis of a standard function $q(x)$ for the given points, where

$$q(x) = \begin{cases} 1 & \text{if } index = 0, \\ n \sin (\pi x/h) / \sum_{j=1}^{n} | \sin (\pi x[j]/h) | & \text{if } index = 1. \end{cases}$$

The estimate used for the rounding error in the rth harmonic is

$$e_r = \sum_{i=index+1}^{r} c_i \times d_i,$$

where

$$c_i = \max | a_{ij} | \text{ for } index \leq j \leq i,$$

$$d_i = | \sum_{j=1}^{n} w_j q(x[j]) p_i(x[j]) / \sum_{j=1}^{n} w_j [p_i(x[j])]^2 |.$$

If for any $r, e_r > e$, the procedure is terminated with $mt = r - 1$.

REFERENCES:

1. CLENSHAW, C. W. Curve-fitting with a digital computer, *Comput. J.* 2, 170–173.
2. WATT, J. M. A note on the evaluation of trigonometric series. *Comput. J. 1*, 162;

```
begin
  integer i, j;  real s1, s2, s3, alpha, beta, c, d, u, v, w, g, s, mean,
    p, coeff, er, cer;
  array c1[0:m], c2[0:m+1];
  g := 3.1415926536/h;
  if index = 0 then mean := 1 else
  begin mean := 0;
    for i := 1 step 1 until n do
      mean := mean + abs(sin(g×x[i]));
    mean := n/mean
  end;
  for i := index step 1 until m do a[i] := 0;
  c2[m+1] := alpha := cer := 0;
  for i := 0 step 1 until m do c1[i] := c2[i] := 0;
  c1[index] := −1;
  beta := s3 := 1;  mt := index;
loop:  coeff := 0;  for i := index step 1 until mt do
  begin
    d := (if i=0 then c2[1] else c1[i−1]) + c2[i+1] − beta ×
      c1[i] − alpha × c2[i];
    c1[i] := c2[i];  c2[i] := d;  d := abs(d);
  if d > coeff then coeff := d
  end;
  s1 := s2 := d := er := 0;
  for i := 1 step 1 until n do
  begin
    c := 2 × cos(g×x[i]);
    if mt = 0 then begin p := 0.5;  go to sum end;
    u := v := 0;
    for j := mt step − 1 until 1 do
    begin
      w := c × u − v + c2[j];  v := u;  u := w
    end;
    if index = 0 then
    begin
      s := 1;  p := 0.5 × (u×c+c2[0]) − v
    end
    else
    begin
      s := sin(g×x[i]);  p := u × s
    end;
sum:  w := if x[i] = 0 ∨ x[i] = h then 0.5 else 1;
      d := d + w × p × f[i];
      if mt > index then er := er + w × p × s × mean;
      p := w × p ↑ 2;  s1 := s1 + c × p;  s2 := s2 + p
  end;
  cer := cer + coeff × abs(er)/s2;
  if cer > e then go to exit;  alpha := s1/s2;
  beta := s2/s3;  d := d/s2;  s3 := s2;
  for i := index step 1 until mt do
    a[i] := a[i] + d × c2[i];
  mt := mt + 1;  if mt ≤ m then go to loop;
exit:  mt := mt − 1
end trigfit;
procedure harmanalsymm (n, m, h, e, x, ypos, yneg, mc, ms, a, b);
  value n, m, h, e;  integer n, m, mc, ms;  real h, e;  array x,
    ypos, yneg, a, b;
```

comment Approximates a function y of x by a finite trigonometric series of period $2h$ from values specified at discrete points in the range $(-h, h)$. Those points need not be equally spaced, but must be symmetrically distributed about the value $x = 0$. Thus only the values of x in the range $0 \leq x \leq h$ need be given.

The input parameters are:

n—number of values of x in the range $0 \leq x \leq h$.

m—order of the highest harmonic required.

h—half-period of the fitted series.

e—used to terminate the process if rounding errors start to accumulate excessively (see note on *trigfit*).

x—the given values of x in the range $(0, h)$ are stored on $x[1]$, $x[2], \cdots, x[n]$.

ypos—the value of y corresponding to $x = +x[i]$ is stored on $ypos[i]$ $(i=1, 2, \cdots, n)$.

yneg—the value of y corresponding to $x = -x[i]$ is stored on $yneg[i]$ $(i=1, 2, \cdots, n)$.

The procedure then calculates the coefficients $a[r]$ and $b[r]$ in the approximation

$$S(x) = \tfrac{1}{2}a[0] + \sum_{r=1}^{mc} a[r] \cos (r\pi x/h) + \sum_{r=1}^{ms} b[r] \sin (r\pi x/h).$$

Here normally $mc = ms = m$, but provision is included to calculate fewer harmonics if rounding errors begin to accumulate excessively (see note on *trigfit*), or if m exceeds its maximum permissible value. For the cosine terms this maximum value is $n - 1$. For the sine terms it is n, this figure being reduced by 1 for each $x[i]$ equal to 0 or h. The cosine and sine series are calculated separately by *trigfit*, with

$$f[i] = \begin{cases} 0.5 \times (ypos[i] + yneg[i]) \text{ for cosine series,} \\ 0.5 \times (ypos[i] - yneg[i]) \text{ for sine series;} \end{cases}$$

```
begin
  integer i, md;  array f[1:n];  procedure trigfit;
  for i := 1 step 1 until n do
    f[i] := 0.5 × (ypos[i] + yneg[i]);
  trigfit (0, n, if m ≥ n then n − 1 else m, h, e, x, f, mc, a);
  md := n;
  for i := 1 step 1 until n do
  begin
    f[i] := 0.5 × (ypos[i] − yneg[i]);
    if x[i] = 0 ∨ x[i] = h then md := md − 1
    end;
  trigfit (1, n, if md ≥ m then m else md, h, e, x, f, ms, b)
end harmanalsymm
```

ALGORITHM 321
t-TEST PROBABILITIES [S14]

JOHN MORRIS (Recd. 6 Jan. 1967, 18 July 1967, and 10 Oct. 1967)
Computer Institute for Social Science Research, Michigan State University, East Lansing, Michigan

KEY WORDS AND PHRASES: *T*-test, Student's *t*-statistic, distribution function
CR CATEGORIES: 5.5

```
real procedure ttest (x, df, maxn, gauss, error);
  value x, df, maxn; real x; integer df, maxn; real procedure
    gauss; label error;
comment This procedure gives the probability that t will be
  greater in absolute value than the absolute value of x, where t
  is the Student t-statistic, as defined and tabled by R. A. Fisher
  [2], evaluated at df degrees of freedom: that is, 2 times the inte-
  gral of the distribution function of t, evaluated from abs(x) to
  infinity. The procedure may also be used, e.g., to estimate the
  two-tailed probability of a simple correlation, r, where N = the
  number of pairs of observations, df = N − 2, and t = r × sqrt
  (df/(1.0 − r ↑ 2))(cf. e.g. [5]).
```

For reasonably small *df*, Student's cosine formula is used [3, 4]:

$$ttest = 1.0 - coef \int_0^\theta \cos^{df-1} \theta \, d\theta$$

where $\theta = \arctan\ (t/sqrt(df))$ and

$$coef = (df-1)/(df-2) \times (df-3)/(df-4)$$

$$\cdots \begin{cases} (\frac{4}{3}) \times (2/\pi) & \text{for odd } df, \\ (\frac{5}{4}) \times (\frac{3}{2}) \times (\frac{1}{2}) & \text{for even } df. \end{cases}$$

Integrated in series, this gives results which appear to be correct to very nearly the full single precision accuracy of the machine (in terms of the number of digits after the decimal point, not necessarily significant digits).

An approximation due to R. A. Fisher [1] gives results accurate to within ±3 × 10^{-7} when *maxn* has been set at 30. The tradeoff on time is also optimal at about this point. The **real procedure** *gauss* computes the area under the left-hand portion of the normal curve. Algorithm 209 [6] may be used for this purpose.

Thanks to the referee for many helpful suggestions, most of which have been incorporated, and to David F. Foster, who wrote an early version of part of the program.

REFERENCES:
1. FISHER, R. A. *Metron 5* (1925), 109–112.
2. ——. *Statistical Methods for Research Workers*. Oliver and Boyd, Edinburgh, 1965.
3. GOSSET, W. S. (Student). The probable error of a mean. *Biometrika 6* (1908), 1.
4. ——. New tables for testing the significance of observations. *Metron 5* (1925), 105.
5. GUILFORD, J. P. *Fundamental Statistics in Psychology and Education*. McGraw-Hill, New York, 1956, pp. 219–221.
6. IBBETSON, D. Algorithm 209, Gauss. *Comm. ACM, 6* (Oct. 1963), 616.

```
begin
  if df < 1 then go to error;
  if x = 0 then ttest := 1.0 else
  begin real t;
    t := abs (x);
    if df < maxn then
    begin integer i, nh;  real cth, sth, cthsq, xi, coef, z;
      z := t/sqrt(df);
      cth := 1.0/sqrt(z ↑ 2+1.0);
      sth := z × cth;
      cthsq := cth ↑ 2;
      nh := (df−1) ÷ 2;
      if df = 2 × (df÷2) then
      begin
        t := sth;
        if nh = 0 then go to g;
        cth := cthsq;  xi := 1.0;
        coef := 0.5 × sth
      end else
      begin
        t := 0.6366197724 × arctan(z);
        comment 0.63661977236758134307553 51··· = 2/π;
        if nh = 0 then go to g;
        xi := 0;  coef := 0.6366197724 × sth
      end;
      for i := 1 step 1 until nh do
      begin
        t := t + coef × cth;  cth := cth × cthsq;
        xi := xi + 2.0;
        coef := coef × xi/(xi+1.0)
       end;
     g: t := 1.0 − t
    end else
      if t > 6.0 then t := 0 else
      if df < $_{10}6$ then
    begin real f, t2, t4, t6, t8, t10, t12, t14, t16, t18;
      f := df;  t2 := t × t;  t4 := t2 × t2;  t6 := t4 × t2;
      t8 := t6 × t2;  t10 := t8 × t2;  t12 := t10 × t2;
      t14 := t12 × t2;  t16 := t14 × t2;  t18 := t16 × t2;
      comment 0.3989422804014326779399461··· = 1/sqrt (2×π);
      t := 2.0 × (gauss(−t)+t×0.3989422804 ×exp(−0.5×t2)×
        ((t2+1.0)/(4.0×f)+(3.0×t6–7.0×t4−5.0×t2−3.0)/
        (96.0×f×f)+(t10−11.0×t8+14.0×t6+6.0×t4−3.0×t2−
        15.0)/(384.0×f ↑ 3)+(15.0×t14−375.0×t12+2225.0×t10−
        2141.0×t8−939.0×t6−213.0×t4−915.0×t2+945.0)/
        (92160.0×f ↑ 4)+(3.0×t18−133.0×t16+1764.0×t14−
        7516.0×t12+5994.0×t10+2490.0×t8+1140.0×t6+180.0×
        t4+5355.0×t2+17955.0)/(368640.0×f ↑ 5)))
    end else t := 2.0 × gauss(−t);
    ttest := if t <0 then 0 else t
  end
end ttest
```

REMARKS ON
ALGORITHM 321 [S14] *t*-TEST PROBABILITIES
[John Morris, *Comm. ACM 11* (Feb. 1968), 115-6]
ALGORITHM 344, STUDENT'S *t*-DISTRIBUTION
[David Levine, *Comm. ACM 12* (Jan. 1969), 37-8]

G. W. Hill, and Mary Loughhead* (Recd. 16 Apr. 1969 and 29 Sept. 1969)
Commonwealth Scientific and Industrial Research Organization, Division of Mathematical Statistics, Glen Osmond, South Australia
* Present address: Monash University, Clayton, Victoria, Australia

KEY WORDS AND PHRASES: *t*-test, Student's *t*-statistic, distribution function, approximation
CR CATEGORIES: 5.12, 5.5

Algorithm 321, as published, was coded in CSIRO 3200 Algol and run on a CDC 3200 with programmed floating point operations. A Fortran equivalent of Algorithm 321 was run for comparison with the Fortran Algorithm 344, which uses the same recurrence relation based on Student's cosine formula as that used in Algorithm 321 for *df* degrees of freedom less than *maxn*. Numerical results agreed with 6-digit tabulated values [1] and double precision calculations indicate that accuracy is limited by truncation of intermediate results to the precision of the processor, with error in the final result increasing as the square root of *df*. Timing tests rated Algorithm 344 at approximately ($\frac{3}{4}$ *df*+$1\frac{1}{2}$) msec; slightly faster than Algorithm 321, which required approximately ($\frac{3}{4}$ *df*+$2\frac{1}{2}$)msec for *df* < *maxn*.

For *df* ≥ *maxn* Algorithm 321 uses Fisher's [2] fifth order approximation, whose accuracy is summarized in the diagram for *df* = 10(10)50 (see Figure 1). The shaded regions indicate values of *t* for which the claimed accuracy of 3×10^{-7} for *maxn* = 30 is not attained. For *t* > 6.0 this algorithm returns zero values, giving errors up to 1.39×10^{-6}. The following alterations avoid this error and, by "nesting" Fisher's polynomial approximation, reduced the time from about 25msec to 20msec and reduced the store requirement by 27%.

Replace the 19 lines beginning "*g*: *t* := 1.0 – *t*" by

```
g: x := 1.0 − t
end else
begin x := 2.0 × gauss (−t);
if df < $_{10}6$ then
begin real f, t2;
f := 0.25/df;  t2 := t × t;
```

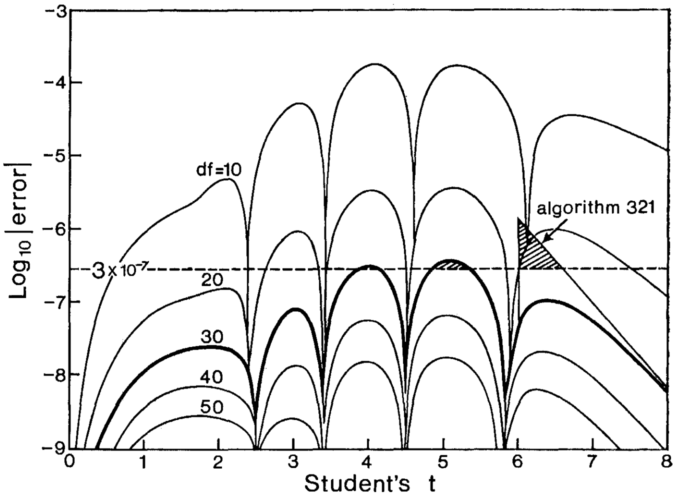


Fig. 1

```
x := (((((((((((((3.0×t2−133.0)×t2
  +1764.0)×t2−7516.0)×t2+5994.0)×t2+2490.0)×t2
  +1140.0)×t2+180.0)×t2+5355.0)×t2+17955.0) × f
  + ((((((15.0×t2−375.0)×t2+2225.0)×t2−2141.0)×t2
  −939.0)×t2−213.0)×t2−915.0)×t2+945.0) × f/60.0
  + ((((t2−11.0)×t2+14.0)×t2+6.0)×t2−3.0)×t2−15.0) × f
  +((3.0×t2−7.0)×t2−5.0)×t2−3.0)×f/6.0
  +(t2+1.0)) × f × t × 0.7978845608 × exp (−0.5×t2) + x
end;
ttest := if x < 0.0 then 0.0 else x
```

The last statement, recommended by the referee, avoids negative results due to rounding errors when the answer is small.

In Algorithm 344 the three statements beginning "1 T = ABS(T)" were replaced by:

```
1 T2 = T*T/FLOAT(DF)
  T1 = SQRT(T2)
  T2 = 1./(1.+T2)
```

to avoid changing the calling parameter T.

Although Algorithm 321 occupies about twice the store space needed for Algorithm 344, and is slightly slower for *df* < *maxn* = 30, it is about three times faster for *df* = 100.

References:

1. Smirnov, N. V. *Tables for the Distribution and Density Functions of t-distribution.* Pergamon Press, New York, 1961.
2. Fisher, R. A. Expansion of "Student's" integral in powers of n^{-1}. *Metron. 5*, 3 (1926), 109–112.

ALGORITHM 322
F-DISTRIBUTION [S14]

Egon Dorrer (Recd. 25 Jan. 1967, 3 July 1967, and 17 Oct. 1967)
Institut für Photogrammetrie und Kartographie, Technische Hochschule München, W. Germany; now: Department of Surveying Engineering, University of New Brunswick, Fredericton, N.B., Canada

KEY WORDS AND PHRASES: Fisher's *F*-distribution, Student's *t*-distribution
CR CATEGORIES: 5.5

```
real procedure Fisher (m, n, x);
  value m, n, x;  integer m, n;  real x;
comment  Fisher's F-distribution with m and n degrees of
  freedom. Computation of the probability
```

$$Pr(F < x) = \frac{\Gamma\left(\frac{m+n}{2}\right)}{\Gamma\left(\frac{m}{2}\right)\cdot\Gamma\left(\frac{n}{2}\right)} \cdot \int_0^w \frac{\xi^{m/2-1}}{(\xi+1)^{(m+n)/2}}\,d\xi,$$

where $w = (m/n)x$ and $F = (\sum_{i=1}^{m} x_i^2/m)/(\sum_{j=1}^{n} y_j^2/n)$. The solution results recursively from the basic integrals

$$Fisher\ (1,1,x) = 2\cdot\arctan\sqrt{w}/\pi,\quad Fisher\ (1,2,x) = (w/(w+1))^{\frac{1}{2}},$$

$$Fisher\ (2,1,x) = 1 - 1/(w+1)^{\frac{1}{2}},\quad Fisher\ (2,2,x) = w/(w+1).$$

π is introduced by $0.3183098862 = 1/\pi$. By calling *Fisher* $(1, n, t\uparrow 2)$, Student's *t*-distribution will be obtained;

```
begin integer a, b, i, j;  real w, y, z, d, p;
  a := 2 × (m÷2) − m + 2;  b := 2 × (n÷2) − n+ 2;
    w := x × m/n;  z := 1/(1+w);
  if a = 1 then
  begin
    if b = 1 then
    begin
      p := sqrt(w);  y := 0.3183098862;
      d := y × z/p;  p := 2 × y × arctan(p)
    end else
    begin
      p := sqrt(w×z);  d := 0.5×p × z/w
    end
  end else
  if b = 1 then
  begin
    p := sqrt(z);  d := 0.5 × z × p;  p := 1 − p
  end else
  begin
    d := z × z;  p := w × z
  end;
  y := 2 × w/z;
  for j := b + 2 step 2 until n do
  begin
    d := (1 + a/(j−2)) × d × z;
    p := if a = 1 then p + d × y/(j−1) else (p+w) × z
  end j;
  y := w × z;  z := 2/z;  b := n − 2;
  for i := a + 2 step 2 until m do
  begin
    j:= i + b;  d := y × d × j/(i−2);  p := p − z × d/j
  end i;
  Fisher := p
end Fisher
```

CERTIFICATION OF ALGORITHM 322 [S14]
F-DISTRIBUTION [Egon Dorrer, *Comm. ACM 11* (Feb. 1968), 116]

J. B. F. Field (Recd. 15 Aug. 1968)
Commonwealth Scientific and Industrial Research Organisation, Adelaide, South Australia

KEY WORDS AND PHRASES: Fisher's *F*-distribution, Student's *t*-distribution
CR CATEGORIES: 5.5

Algorithm 322 was coded into Fortran and run on a CDC 3200, and its accuracy for moderate probability levels was tested using (a) 5-figure critical values of the *F*-distribution at the .95 and .99 levels, taken from [1], and (b) 6-figure probability values of the *t*-distribution, taken from [2]. In both cases, limitations in the results appeared to be due to limitations in the tables, rather than in the algorithm.

232 values of the *F*-distribution were tested, for $m = 1$ and 12 using all tabulated values of n, and for $n = 10$ and 21 using all tabulated values of m. All the results agreed with the tabulated probability level to 4 significant figures, 89% to 5 figures, and over half the results agreed to 6 or more figures.

300 values of the *t*-distribution were tested, for $n = 1(1)30$ and $t = .5(.5)5$. All the results agreed with the tabulated probability to 5 significant figures, and 90% to the full 6 figures given in the tables.

To test extreme probability levels, another 100 values of the *F*-distribution were used: for $m = n = 2, 10, 50, 75, 100, 120, 150, 200, 300$, and 400 for each of the values $x = 10^i$, where $i = 5(1)5$. It was found that for probabilities which are extremely close to 0 or 1, the algorithm may produce probabilities which are slightly less than zero, or slightly greater than 1. It is recommended that a "guard" be inserted in the program to set these values equal to 0 or 1. For example, this could be done by inserting before *Fisher*:=*p* the additional statement

p := **if** *p* > 1 **then** 1 **else if** *p* < 0 **then** 0 **else** *p*;

The time taken by the algorithm was directly proportional to the sum of the degrees of freedom. The constant of proportionality depended mainly on whether m was even or odd (the time taken for m even being .81 of the time taken for m odd, using a CDC 3200 with programmed floating point). To a much lesser extent, it was influenced by whether n was even or odd (the time taken for n even being .99 of that for n odd).

REFERENCES

1. Owen, D. B. *Handbook of Statistical Tables.* Addison-Wesley, Reading, Mass., 1962.
2. Smirnov, N. V. *Tables for the Distribution and Density Functions of t-distribution.* Pergamon Press, Oxford, 1961.

REMARK ON ALGORITHM 322 [S14]
F-DISTRIBUTION [Egon Dorrer, *Comm. ACM 11* (Feb. 1968), 116]

HUBERT TOLMAN (Recd. 7 Apr. 1970 and 13 Oct. 1970)
Department of Mathematics, Northeast Louisiana State College, Monroe, LA 71201

KEY WORDS AND PHRASES: Fisher's F-distribution, Student's t-distribution
CR CATEGORIES: 5.5

Replacing the statements

```
  for j := b + 2 step 2 until n do
  begin
    d := (1 +a/(j−2)) × d × z;
    p := if a = 1 then p + d × y/(j−1) else (p + w) × z
  end j;
```

by the algebraically equivalent statements

```
  if a = 1 then
  begin
    for j := b + 2 step 2 until n do
    begin
      d := (1 + a/(j−2)) × d × z;
      p := p + d × y/(j−1)
    end j;
  end
  else
  begin
    zk := z ↑ ((n−1) ÷ 2);
    d := d × zk × n/b;
    p := p × zk + w × z × (zk−1)/(z−1);
  end;
```

substantially reduces the execution time when *m* is even, and did not change the speed when *m* is odd. For the resulting algorithm, the execution time is proportional to *m* when *m* is even, and proportional to *m* + *n* when *m* is odd.

TABLE I. PERCENT TIME SAVINGS

				m		
		4	8	16	32	64
	8	34	3	2	1	1
	16	37	32	20	12	7
	32	62	54	45	30	19
n	64	79	73	63	51	36
	128	88	85	78	68	54
	256	94	92	88	81	71
	512	96	95	93	90	83
	1024	98	97	96	94	90

Both the new and original forms of the algorithm were coded in Fortran and timed. The percentage reductions in execution time are given in Table I. The greatest reduction came when *n* is large and *m* is small. In many statistical applications *n* is substantially larger than *m* and seldom smaller, thereby falling in the region of the greatest saving in execution time.

ALGORITHM 323 GENERATION OF PERMUTATIONS IN LEXICOGRAPHIC ORDER [G6]

R. J. Ord-Smith (Recd. 27 Apr. 1967 and 26 July 1967)
Computing Laboratory, University of Bradford, Bradford, Yorkshire, England

KEY WORDS AND PHRASES: permutations, lexicographic order, lexicographic generation, permutation generation
CR CATEGORIES: 5.39

Author's Remark. Lexicographic generation involves more than the minimum of $n!$ transpositions for generation of the complete set of $n!$ permutations of n objects. The actual number of transpositions required can be shown to tend asymptotically to (cosh 1) $n! \doteqdot 1.53n!$ However, lexicographic generation can be described by an algorithm requiring very simple book-keeping. The author is indebted to Professor H. F. Trotter for suggesting an improvement to an original algorithm, which now results in a process more than twice as fast as the previously fastest lexicographic Algorithm 202 [*Comm. ACM 6* (Sept. 1963), 517]. Tabulated results below show *BESTLEX* to be only 9.3 percent slower than the transposition Algorithm 115 [*Comm. ACM 5* (Aug. 1962), 434] when $n = 8$.

The usual practice is adopted of using a nonlocal Boolean variable called *first* which may be assigned the value *true* to initialize generation. On procedure call this is set *false* and remains so until it is again set *true* when complete generation of permutations has been achieved. Table I gives results obtained for *BESTLEX*. The times given in seconds are for an I.C.T. 1905 computer. t_n is the time for complete generation of $n!$ permutations. r_n has the usual definition $r_n = t_n/(n \cdot t_{n-1})$.

TABLE I

Algorithm	t_7	t_8	r_8	*Number of transpositions*
BESTLEX	6	47	0.98	→ $1.53n!$
202	12.4	100	1.00	?
115	5.6	43	0.98	$n!$

```
procedure BESTLEX (x, n);  value n;  integer n;  array x;
begin own integer array q[2:n];  integer k, m;  real t;
comment own dynamic arrays are not often implemented. The
  upper bound will then have to be given explicitly;
  if first then
  begin first := false;
    for m := 2 step 1 until n do q[m] := 1
  end of initialization process;
  if q[2] = 1 then
  begin q[2] := 2;
    t := x[1];  x[1] := x[2];  x[2] := t;
    go to finish
  end;
  for k := 2 step 1 until n do
      if q[k] = k then q[k] := 1  else go to trstart;
first := true;  k := n;  go to trinit;
trstart:  m := q[k];  t := x[m];  x[m] := x[k];  x[k] := t;
  q[k] := m + 1;  k := k − 1;
trinit:  m := 1;
transpose:  t := x[m];  x[m] := x[k];  x[k] := t;
  m := m + 1;  k := k − 1;
  if m < k then go to transpose;
finish:
end of procedure BESTLEX
```

CERTIFICATION OF ALGORITHM 323 [G6] GENERATION OF PERMUTATIONS IN LEXICOGRAPHIC ORDER [R. J. Ord-Smith, *Comm. ACM 11* (Feb. 1968), 117]

I. M. Leitch (Recd. 9 July 1968, 6 Jan. 1969 and 17 Mar. 1969)
Department of Medicine, University of Newcastle upon Tyne, Newcastle upon Tyne, England

KEY WORDS AND PHRASES: permutations, direct lexicographic order, reverse lexicographic order, lexicographic generation
CR CATEGORIES: 5.39

The ranking function $R_d(a_1, a_2, \cdots, a_n)$ which specifies the position of a permutation $(a_1, a_2, \cdots, a_n)$ of the numbers 0 (1) $n-1$ in a direct lexicographic order is commonly defined recursively[1] by

$$R_d(0) = 0$$

and

$$R_d(a_1, a_2, \cdots, a_n) = a_1 \cdot (n-1)! + R_d\ (M(a_1, a_2, \cdots, a_n))$$

where $M(a_1, a_2, \cdots, a_n)$ is the permutation of the numbers 0 (1) $n-2$ obtained from $a_1, a_2, \cdots, a_n$ by deleting a_1 and reducing by unity all those elements which exceed a_1.

Reverse lexicographic order of a permutation $(b_1, b_2, \cdots, b_n)$ is defined by a similar ranking function,

$$R_r(b_1, b_2, \cdots, b_n) = n! - 1 - R_d(b_n, \cdots, b_2, b_1).$$

As reverse lexicographic order has the property (which direct order does not) that all the permutations which involve only the first K elements are generated before the $(K + 1)$-th element is moved, it is sometimes preferred above the direct order. The two are closely related since in any n-element permutation vector **a** typical element a_i of the direct order corresponds to element a_{n-i+1} of the reverse order. As both of these orderings are in common use, it is inappropriate to describe either as lexicographic without further qualification.

After replacement of the dynamic upper bound of the **own integer array** by a constant (necessitated by a compiler implementation restriction), Algorithm 323 was compiled by the Kidsgrove Algol compiler and run on an English Electric KDF9 computer. The full permutation was generated for values of $n = 2$ (1)

9. The permutations generated by *BESTLEX* (Algorithm 323) were compared automatically with those of Algorithm 202 [*Comm. ACM 6* (Sept. 1963), 517]. It was known that Algorithm 202 generated permutations in a direct lexicographic order, and it was found that permutations were produced by *BESTLEX* in a reverse lexicographic order.

The order in which the permutations of *BESTLEX* are generated is governed by the **own integer array** q of that procedure and its integer counters m and k. Because of the simple relationship which exists between direct and reverse lexicographic order, the published algorithm may be modified so that it will generate permutations in direct lexicographic order by systematic application of the following three rules:

1. Wherever the value 1 or 2 occurs either as a subscript expression or an integer constant which is not part of a more complex expression, replace it by n or $n-1$, respectively.

2. Redefine the bounds of q and the limits of both **for** loops to be from 1 to $n-1$. Reverse the direction of the k **for** loop.

3. In the last seven lines of the algorithm, the integer counter k must be incremented by 1 from 1 (rather than decremented from n), and, similarly, wherever $m+1$ appears in an assignment statement it is replaced by $m-1$. Consequently m and k must be reversed in the comparison on the penultimate line of the algorithm.

At each call of the algorithm these modifications redirect attention from the beginning of the permutation vector to the end, and so cause permutations to be generated in direct order. However, because of the nature of these changes, no loss in computational efficiency should be expected (since the only extra arithmetic incurred is the evaluation of $n-1$, which need be performed only once for each procedure call). This was confirmed at run times as the times taken to generate a full permutation in reverse order by the published algorithm and in direct order by the modified algorithm were identical.

Table I gives the time in seconds (t_n) which is required by each procedure for the complete generation of the $n!$ permutations, r_n has the usual definition of $t_n/(n \cdot t_{n-1})$.

TABLE I

Algorithm	t_7	t_8	r_8
BESTLEX	10.01	80.08	1.00
202	20.84	166.75	1.00

Both algorithms were also tested under the Whetstone ALGOL interpreter on the KDF9, an ALGOL compiler for the 1130, and the IBM 360 Model 67 Operating System ALGOL "F" compiler. As the last two implementations do not recognize the concept of **own**, results were obtained by inserting an **integer array** into the procedure heading as an additional parameter and by not declaring the **own integer array** in the procedure body. For comparison, execution times for the $n!$ permutations which were recorded when the procedure was run on the IBM 360/67 are given in Table II.

TABLE II

	t_7	t_8	r_8
BESTLEX	7.6	61.01	0.99

REFERENCES

1. LEHMER, D. H. Teaching combinational tricks to a computer. Proc. of Symp. in Appl. Math., Vol. 10, Amer. Math. Soc., Providence, R. I., 1960, pp. 179–193.

Remark on Algorithm 323 [G6]

Generation of Permutations in Lexicographic Order [R.J. Ord-Smith, *Comm. ACM 11* (Feb. 1968), 117]

Mohit Kumar Roy [Recd. 15 May 1972]
Computer Centre, Jadavpur University, Calcutta 32, India

In presenting Algorithm 323, *BESTLEX*, for generating permutations in lexicographic order, the author has mentioned the number of transpositions. It may be remarked here that equal numbers of transpositions are required by both *BESTLEX* and the previously fastest algorithm, Algorithm 202 [1]. The exact number of transpositions (T_n) necessary to generate the complete set of $n!$ permutations is given by

$$T_n = n!\,(\psi_{n-1}) - (n+1)/2, \quad \text{if } n \text{ is odd, and}$$
$$T_n = n!\,(\psi_{n-2}) - n/2, \quad \text{if } n \text{ is even,}$$

$$\text{where } \psi_{2n} = 1 + \frac{1}{2!} + \frac{1}{4!} + \cdots + \frac{1}{(2n)!} \doteqdot 1.543 \text{ for } n \geq 3.$$

The above expressions do not include the few extra transpositions (equal to the integral part of $n/2$) required by *BESTLEX* to generate the initial arrangement from the final one, as this portion has not been included in Algorithm 202. Therefore, the number of transpositions has no importance in the context of the claim that *BESTLEX* is more than twice as fast as Algorithm 202.

The main factor contributing to the speed of *BESTLEX* is the substantial reduction in the number of comparisons required, by the introduction of the **own integer array** q. Taking into account only those comparisons which involve array elements, the number of comparisons (C_n) required to generate all the $n!$ permutations can be shown to be equal to

$$C_n \text{ (Algorithm 202)} = \frac{n!}{2}[1 + 3\varphi_{n-2}] + n,$$
$$C_n \ (\textit{BESTLEX}) = n!\,[\tfrac{1}{2} + \varphi_{n-1}],$$

$$\text{where } \varphi_n = 1 + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{n!} \doteqdot 1.718 \text{ for } n \geq 6.$$

This shows that the number of comparisons required by *BESTLEX* is lower by .859($n!$) (approximately) in the case of the generation of all the $n!$ arrangements.

Finally, a modification of the *BESTLEX* algorithm is suggested which will reduce the number of comparisons again by ($n!$)/2. The modification involves replacement of lines 2–14 of Algorithm 323 by the following.

```
begin own integer array q[3:n];  integer k, m;
   real t;  own Boolean flag;
comment  Own dynamic arrays are not often implemented. The
   upper bound will have to be given explicitly;
if first then
begin first := false;  flag := true
for m := 3 step 1 until n do q[m] := 1
end of initialization process;
if flag then
begin flag := false;
   t := x[1];  x[1] := x[2];  x[2] := t;
   go to finish
end;
flag := true;
for k := 3 step 1 until n do
```

References

1. Shen, Mok-Kong. Algorithm 202, generation of permutations in lexicographical order. *Comm. ACM* 6 (Sept. 1963), 517.

Added in proof: An improved version of *BESTLEX*, viz. Algorithm 323A, Generation of Permutation Sequences: Part 2, by R.J. Ord-Smith [*Comp. J. 14*, 2 (May 1971), 136–139], which also incorporates the modification suggested here, has come to the author's attention.

ALGORITHM 324
MAXFLOW [H]

G. BAYER (Recd. 31 July 1967)
Technische Hochschule, Braunschweig, Germany

KEY WORDS AND PHRASES: network, linear programming, maximum flow
CR CATEGORIES: 5.41

```
procedure maxflow (from, to, cap, flow, v, n, mflow, source, sink,
inf, eps);
  value v, n, source, sink, inf;
  integer v, n, source, sink;  real inf, eps, mflow;
  integer array from, to;  array cap, flow;
comment  The nodes of the network are numbered from 1 to sn.
  It is not necessary but reasonable that each number represent a
  node. The data of the network are given by arrays from, to, cap
  in the following manner. There is a maximum possible flow of
  cap[i], nonnegative, leading from from[i] to to[i], i = 1, ··· , v.
    Compute the maximum flow  mflow  from  source  to  sink,
  (source and sink given by their node numbers). inf represents
  the greatest positive real number within machine capacity.
  flow[i] gives the actual flow from from[i] to to[i]. Flows abso-
  lutely less than eps are considered to be zero. Literature: G.
  Hadley, Linear Programming, Addison-Wesley, Reading (Mass.)
  and London, 1962, pp. 337–344.
    Multiple solutions are left out of account;
begin integer l, j, k, r, lk, ek, u, s;  real gjk, d;
  integer array low, up, klist, labj[1:n], ind[1:v];  real array
  labf[1:n];
comment  Note structure of data lists in up and low;
  l := 1;
  for j := 1 step 1 until n do
  begin low[j] := l;
    for r := 1 step 1 until v do
    begin if from[r] = j then
      begin ind[l] := r;
        flow[l] := cap[l];  l := l + 1
      end
    end;
    up[j] := l − 1
  end;
  mflow := 0.0;
lab:;
  comment  Prepare lists for new labeling;
  for j := 1 step 1 until n do
  begin labj[j] := klist[j] := 0;
    labf[j] := 0.0
  end;
  labf [source] := inf;
  comment labeling;
  j := source;  lk := ek := 0;
path:
  u := up[j];
  for s := low[j] step 1 until u do
  begin l := ind[s];
    k := to[l];  gjk := flow[l];
    if labj[k] ≠ 0 ∨ abs(gjk) < eps
      then go to end;
      labj[k] := j;
      labf[k] := if gjk < labf[j] then gjk else labf[j];
      if k = sink then go to reached;
      lk := lk + 1;  klist[lk] := k;
end:
  end;
  ek := ek + 1;  j := klist[ek];
  if j ≠ 0 then go to path else go to max;
  comment  sink is labeled, find path and possible
    flow, reduce excess capacities along path;
reached:
  j := sink;  d := labf[j];  mflow := mflow + d;
look:  k := labj[j];  u := up[k];
  for s := low[k] step 1 until u do
  begin l := ind[s];
    if to[l] = j then flow[l] := flow[l] − d
  end;
  u := up[j];
  for s := low[j] step 1 until u do
  begin l := ind[s];
    if to[l] = k then flow[l] := flow[l] + d
  end;
  j := k;  if j ≠ source then go to look;
  go to lab;
max:;  comment maximal flow found;
  for l := 1 step 1 until v do
    flow[l] := cap[l] − flow[l]
end
```

Remark on Algorithm 324 [H]

Maxflow [G. Bayer, *Comm. ACM 11* (Feb. 1968), 117]

G. Bayer [Recd. 5 Aug. 1971]
Technische Universität, 33 Braunschweig, Germany

It is necessary to clarify the meaning of input parameters *from*, *to* and *cap* describing the given network.

A connection between two nodes, say *a* and *b*, must be given by two arcs like this: At two index-positions, say *ia* and *ib*, the input arrays have values

```
from [ia] = a          from [ib] = b
  to [ia] = b            to [ib] = a
 cap [ia] = capab       cap [ib] = capba
```

Even if one of the two flows, say *capab* from node *a* to node *b*, is zero, it must not be omitted, for otherwise the algorithm goes wrong.

If there is no connection between two nodes, then no arcs are to be given. In this case another input yields the same result: Two arcs are given, each with a maximum possible flow of zero. (But this case is not physically, or in the sense of the algorithm, the same as the first one.)

ALGORITHM 325
ADJUSTMENT OF THE INVERSE OF A SYMMETRIC MATRIX WHEN TWO SYMMETRIC ELEMENTS ARE CHANGED [F1]

GERHARD ZIELKE (Recd. 24 Aug. 1967)
Institut für Numerische Mathematik der Martin Luther Universität Halle-Wittenberg, German Democratic Republic

KEY WORDS AND PHRASES: symmetric matrix, matrix inverse, matrix perturbation, matrix modification
CR CATEGORIES: 5.14

```
procedure INVSYM 2 (n, i, j, c, a, b);
  value n, i, j, c;  integer n, i, j;  real c;  array a, b;
```

comment *INVSYM* 2 computes the inverse $A^{-1} = a$ of a nonsingular symmetric nth order matrix $A = B + c(e_i e_j' + e_j e_i')$ which arises from a symmetric matrix B by a change c in two elements B_{ij} and $B_{ji} = B_{ij}$ $(i \neq j)$. The inverse matrix $B^{-1} = b$ is assumed to be known. The calculation with the new formula

$$a = b - \frac{c}{d}\,[b_{.i}(h_1 b_{j.} + h_2 b_{i.}) + b_{.j}(h_3 b_{j.} + h_1 b_{i.})]$$

where

$$h_1 = 1 + cb_{ij}\,, \quad h_2 = -cb_{jj}\,, \quad h_3 = -cb_{ii}\,, \quad d = h_1^2 - h_2 h_3$$

requires $n^2 + O(n)$ multiplications, therefore only about the same number of operations as if the well-known Sherman-Morrison formula for a change in one element (see Algorithm 51 [*Comm. ACM 4* (Apr. 1961), 180]) is used. In these equations e_i denotes the ith column and e_i' the ith row of the unit matrix, $b_{.i} = be_i$ denotes the ith column and $b_{i.} = e_i'b$ the ith row of the matrix b;

```
begin integer k, l;  real h1, h2, h3, d;
  array r, s[1:n];
  h1 := 1 + c × b[i, j];  h2 := −c × b[j, j];
  h3 := −c × b[i, i];  d := h1 ↑ 2 − h2 × h3;  d := c/d;
  h1 := h1 × d;  h2 := h2 × d;  h3 := h3 × d;
  for k := 1 step 1 until n do
  begin
    r[k] := h1 × b[j, k] + h2 × b[i, k];
    s[k] := h3 × b[j, k] + h1 × b[i, k]
  end;
  for k := 1 step 1 until n do
  for l := 1 step 1 until k do
    a[k, l] := a[l, k] := b[k, l] − b[k, i] × r[l] − b[k, j] × s[l]
end INVSYM 2
```

ALGORITHM 326
ROOTS OF LOW-ORDER POLYNOMIAL EQUATIONS [C2]

TERENCE R. F. NONWEILER (Recd. 14 Apr. 1967)
James Watt Engineering Laboratories, The University,
Glasgow W2, Scotland

KEY WORDS AND PHRASES: rootfinders, polynomial equation roots, quadratic equation roots, cubic equation roots, biquadratic equation roots, polynomial zeros
CR CATEGORIES: 5.15

```
ROOTFINDERS:
begin
comment  suite of procedures finding the (complex) roots of the
   lower order polynomial equations by the familiar algebraic
   methods;
procedure BIQUADROOTS(p, r);  value p;  array p, r;
comment  finds the roots x = r[1, k] + sqrt(−1) × r[2, k] of
   the biquadratic equation p[0] × x ↑ 4 + ··· + p[4] = 0;
comment  array r defined for subscript bounds [1:2, 1:4] and p
   for [0:4]. Failure occurs (in overflow) if p[0] = 0 and in other
   cases. Uses nonlocal procedures QUADROOTS and CUBIC-
   ROOTS;
begin real e, b, d, c, a;
  integer k, j;
  if p[0] ≠ 1.0 then
  begin
    for k := 1 step 1 until 4 do p[k] := p[k]/p[0];  p[0] := 1.0
  end;
  e := 0.25 × p[1];  b := e + e;  c := b × b;  d := 0.75 × c;
  b := p[3] + b × (c−p[2]);  a := p[2] − d;
  c := p[4] + e × (e×a−p[3]);  a := a − d;  p[1] := 0.5 × a;
  p[2] := (p[1]×p[1]−c)/4.0;  p[3] := b × b/(−64.0);
  if p[3| < 0 then
  begin
    CUBICROOTS(p, r);
    for k := 1 step 1 until 3 do
      if r[2, k] = 0 and r[1, k] > 0 then
      begin
        d := r[1, k] × 4.0;  a := a + d;
        p[1] := if a ≥ 0 ≡ b ≥ 0 then  sqrt(d)  else  -sqrt(d);
        b := 0.5 × (a+b/p[1]);  go to QUAD
      end the general case jumping to QUAD;
  end nonzero p[3];
  if p[2] < 0 then
  begin
    b := sqrt(c);  d := b + b − a;
    p[1] := if d ≤ 0 then 0 else sqrt(d)
  end
  else
  begin
    b := sqrt(p[2]) × (if p[1]>0 then +2.0 else −2.0) + p[1];
    if b ≠ 0 then p[1] := 0 else
    begin
      for k := 1 step 1 until 4 do
      begin
        r[1, k] := −e;  r[2, k] := 0
      end;
      go to END
    end
  end;
QUAD:  p[2] := c/b;  QUADROOTS(p, r);
  for k := 1, 2 do
    for j := 1, 2 do r[j, k+2] := r[j, k];
  p[1] := −p[1];  p[2] := b;  QUADROOTS(p, r);
  for k := 1 step 1 until 4 do r[1, k] := r[1, k] − e;
END:
end BIQUADROOTS;
procedure CUBICROOTS(p, r);  value p;  array p, r;
comment  finds the roots x = r[1, k] + sqrt(−1) × r[2, k], ar-
   ranged in order (k=1, 2, 3) of increasing modulus, of cubic equa-
   tion p[0] × x ↑ 3 + ··· + p[3] = 0;
comment  array r defined for subscript bounds [1:2, 1:3] and p
   for [0:3]. Failure occurs (in overflow) if p[0] = 0 and in other
   cases. Assumes 0 < arctan(x) < pi/2 for x > 0;
begin real s, t, b, c, d;
  integer k;
  if p[0] ≠ 1.0 then
    for k := 1 step 1 until 3 do p[k] := p[k]/p[0];
  s := p[1]/3.0;  t := s × p[1];
  b := 0.5 × (s×(t/1.5−p[2])+p[3]);  t := (t−p[2])/3.0;
  c := t ↑ 3;  d := b × b − c;
  if d ≥ 0 then
  begin
    d := (sqrt(d)+abs(b)) ↑ (1.0/3.0);
    if d ≠ 0 then
    begin
      b := if b > 0 then −d else d;  c := t/b;
    end;
    d := r[2, 2] := sqrt(0.75) × (b−c);  b := b + c;
    c := r[1, 2] := −0.5 × b − s;
    if b > 0 ≡ s ≤ 0 then
    begin
      r[1, 1] := c;  r[2, 1] := −d;  r[1, 3] := b − s;
      r[2, 3] := 0
    end
    else
    begin
      r[1, 1] := b − s;  r[2, 1] := 0;  r[1, 3] := c;
      r[2, 3] := −d
    end
  end the case of two equal or complex roots
  else
  begin
    d := if b = 0 then arctan(1.0)/1.5 else arctan(sqrt(−d)/
    abs(b))/3.0;
    b := sqrt(t) × (if b<0 then 2.0 else −2.0);
    c := cos(d) × b;  t := −sqrt(0.75) × sin(d) × b − 0.5 × c;
    d := −t − c − s;  c := c − s;  t := t − s;
    if abs(c) > abs(t) then r[1, 3] := c
    else
    begin
      r[1, 3] := t;  t := c
    end;
    if abs(d) > abs(t) then r[1, 2] := d else
    begin
      r[1, 2] := t;  t := d
    end;
    r[1, 1] := t;
    for k := 1 step 1 until 3 do r[2, k] := 0;
  end the irreducible case;
end CUBICROOTS;
procedure QUADROOTS(p, r);  array p, r;
comment finds the roots x = r[1, k] + sqrt(−1) × r[2, k]
  arranged in order (k=1, 2) of ascending modulus, of the quadra-
  tic equation p[0] × x ↑ 2 + p[1] × x + p[2] = 0;
comment array p defined for subscript limits [0:2] and r for
  [1:2, 1:2]. The entry values of the array p are preserved. Fails
  (in overflow) if p[0] = 0 and in other cases;
begin real b, c, d;
  b := −p[1]/p[0]/2.0;  c := p[2]/p[0];  d := b × b − c;
  if d > 0 then
  begin
    b := r[1, 2] := if b > 0 then sqrt(d) + b else b − sqrt(d);
    r[1, 1] := c/b;  r[2, 1] := r[2, 2] := 0
  end
  else
  begin
    d := r[2, 1] := sqrt(−d);  r[2, 2] := −d;
    r[1, 1] := r[1, 2] := b
  end
end QUADROOTS;
end
```

ALGORITHM 327
DILOGARITHM [S22]

K. S. Kölbig (Recd. 10 Oct. 1967)
Applied Mathematics Group, Data Handling Division,
European Organization for Nuclear Research (CERN),
1211 Geneva 23, Switzerland

KEY WORDS AND PHRASES: dilogarithm function, special functions
CR CATEGORIES: 5.12

real procedure *dilog*(x); **value** x; **real** x;
comment This procedure evaluates the dilogarithm function

$$d(x) = -\int_0^x (ln\,|1 - y\,|/y)\,dy$$

for real arguments x. 13 to 14 significant digits are correct, except for values of x near to the zero of $d(x)$ on the positive axis ($x \approx 12.6$). This function appears in several fields of theoretical physics. The method of computation is described by Mitchell [1]. For $0 < x \leq 0.5$, a Chebyshev approximation is used, which was obtained by economizing the power series $\sum_{n=1}^{\infty} x^n/n^2$ with a multiprecision CERN library program [2].

References:

1. Mitchell, K. Tables of the function $\int_0^z (-\log|\,1-y\,|/y)\,dy$, with an account of some properties of this and related functions. *Phil. Mag. 40* (1949), 351–368.
2. Carlson, J. R. TCHEBY—telescoping of a polynomial. CERN 6600 Computer Program Library E203 (1966), unpublished;

```
begin real f, u, y, z;
comment  3.289868 ··· = π²/3, 1.644934 ··· = π²/6;
  if x ≥ 2 then
  begin
    z := 1/x;  u := −0.5 × ln(x) ↑ 2 + 3.289868133696453;
    f := −1
  end
  else if x > 1 then
  begin
    z := (x−1)/x;
    u := −0.5 × ln(x) × ln(z×x−z) + 1.644934066848226;
    f := 1
  end
  else if x = 1 then
  begin
    dilog := 1.644934066848226;  go to L1
  end
  else if x > 0.5 then
  begin
    z := 1 − x;  u := −ln(x) × ln(z) + 1.644934066848226;
    f := −1
  end
  else if x > 0 then
  begin
    z := x;  u := 0;  f := 1
  end
  else if x = 0 then
  begin
    dilog := 0;  go to L1
  end
  else if x ≥ −1 then
  begin
    z := x/(x−1);  u := −0.5 × ln(1 − x) ↑ 2;  f := −1
  end
  else
  begin
    z := 1/(1−x);
    u := 0.5 × ln(z) × ln(x↑2×z) − 1.644934066848226;
    f := 1
  end;
  y := 0.008048124718341 × z + 0.008288074835108;
  y := y × z − 0.001481786416153;
  y := y × z − 0.000912777413024;
  y := y × z + 0.005047192127203;
  y := y × z + 0.005300972587634;
  y := y × z + 0.004091615355944;
  y := y × z + 0.004815490327461;
  y := y × z + 0.005966509196748;
  y := y × z + 0.006980881130380;
  y := y × z + 0.008260083434161;
  y := y × z + 0.009997129506220;
  y := y × z + 0.012345919431569;
  y := y × z + 0.015625134938703;
  y := y × z + 0.020408155605916;
  y := y × z + 0.027777774308288;
  y := y × z + 0.040000000124677;
  y := y × z + 0.062500000040762;
  y := y × z + 0.111111111110322;
  y := y × z + 0.249999999999859;
  y = y × z + 1;  dilog = f × y × z + u;
L1:
end dilog;
```

comment The procedure *dilog* was tested on a CDC 3800 computer, using an Algol compiler. It was translated into Fortran and run on a CDC 6600 computer. The tests included the following:

(i) Calculation of $d(x)$ for $x = -1(0.01)1$. A comparison with the 9-figure table given in [1] revealed in few cases a discrepancy of 1 unit in the last figure.
(ii) Calculation of $d(x)$ for $x = \pm 10^i$, $i = 0(10)100$, $x = -3(0.1)15$, $x = \pm 10^i$, $i = -20(1)0$.
(iii) Calculation of $d(x)$ for $x = 1 + i \times 10^{-m}$, $i = -10(1)10$, $m = 10$ in the case of the CDC 3800, $m = 14$ for the CDC 6600.

In all three cases the results have been compared with those obtained by summing the power series directly. Agreement to 13 or 14 significant digits was found, with the exception mentioned in the comment above;

ALGORITHM 328
CHEBYSHEV SOLUTION TO AN OVERDETERMINED LINEAR SYSTEM [F4]

Richard H. Bartels and Gene H. Golub

(Recd. 8 June 1967 and 22 Nov. 1967)

Computer Science Dept., Stanford University, Stanford, Calif. 94305

KEY WORDS AND PHRASES: Chebyshev solutions, overdetermined linear systems, linear equations, exchange algorithm

CR CATEGORIES: 5.13, 5.14, 5.41

```
procedure Chebyshev (A, d, h, m, n, refset, epz, insufficientrank,
  zerolambda);
  value m, n;  integer m, n;  real array A, d, h;
  integer array refset;  real epz;  label insufficientrank, zero-
    lambda;
comment Chebyshev computes a solution in the Chebyshev
  sense to an overdetermined system of linear equations, Ax = d.
  Details and notation are given in a paper by Bartels and Golub
  [Comm. ACM 11 (June 1968), 403–408].
  The parameters to procedure Chebyshev are:
```

identifier	*type*	*comments*
m	integer	Number of equations
n	integer	Number of unknowns
A	real array	Matrix of coefficients Array bounds—[0:*m*-1,0:*n*-1]
d	real array	Right-hand-side vector Array bounds—[0:*m*-1]
h	real array	Solution vector Array bounds—[0:*n*-1]
refset	integer array	Final reference equation numbers Array bounds—[0:*n*]
epz	real	Final reference deviation
zerolambda	label	Exit for condition 1 failure
insufficientrank	label	Exit for condition 2 failure, or in case rank (A) $< n$

```
  The parameters m, n, A and d are not changed by Chebyshev.
  We direct the user's attention to the identifier eta appearing in
  the procedure and to the comment explaining its value and
  purpose.;
begin
  real procedure ip (ii, ll, uu, aa, bb, cc);
   value ll, uu, cc;  real aa, bb, cc;  integer ii, ll, uu;
 comment single-precision inner-product routine;
 begin
   real sum;
   sum := cc;
   for ii := ll step 1 until uu do sum := sum + aa × bb;
   ip := sum
 end ip;
 real procedure ip2 (ii, ll, uu, aa, bb, cc);
 comment ip2 is a version of ip which accumulates the products
   aa × bb in a double-precision sum, whose final value, rounded
   to single-precision, is taken as the value of ip2.;
Boolean finished;  switch decompbranch := return, itr;
switch failures := insufficientrank, zerolambda;
integer m1, n1, np1, i, j, k, l, b, al, a1, lst, kmax, cnt;
real lasteps, preveps, ref, s, t, eps, eta, cnorm, snorm;
real array P[0:n, 0:n], lam, rv, sv, x, w, xr[0:n];
integer array r[0:n], ix[0:m−1];
comment The subsystem of n + 1 equations currently being
  investigated is listed in ix[0], ··· , ix[n]. The other equations
  are listed in the remainder of ix. r contains row indices. Row
  interchanges during the Gauss decomposition of P are carried
  out by permuting the elements of r.;
m1 := m − 1;  n1 := n − 1;  np1 := n + 1;
lasteps := 0;  preveps := − 1;
for i := 0 step 1 until n do r[i] := ix[i] := i;
for i := np1 step 1 until m1 do ix[i] := i;
comment The initial reference subsystem is chosen by making
  a copy of the transpose of A bordered with d and carrying out
  a Gaussian reduction upon it with row and column inter-
  changes used to select the largest possible pivot at each stage.;
begin
  real array TAB[0:n, 0:m1];
  for j := 0 step 1 until m1 do
  begin
    TAB[n, j] := d[j];
    for i := 0 step 1 until n1 do TAB[i, j] := A[j, i]
  end;
  for i := 0 step 1 until n do
  begin
    t := 0;
    for j := i step 1 until n do
    begin
      k := r[j];
      for l := i step 1 until m1 do
      begin
        ref := TAB[k, ix[l]];
        if abs(ref) > t then
        begin
          s := ref;  t := abs(ref);  al := j;  b := l
        end
      end
    end;
    if t = 0 then begin j := 1;  go to singular end;
    k := r[al];  r[al] := r[i];  lst := r[i] := k;
    k := ix[b];  ix[b] := ix[i];  a1 := ix[i] := k;
    for j := i + 1 step 1 until m1 do
      begin
        l := ix[j];
        ref := TAB[lst, l]/s;
        for k := i + 1 step 1 until n do
        begin
          al := r[k];
          TAB[al, l] := TAB[al, l] − TAB[al, a1] × ref
        end
      end
    end
  end;
  b := 0;  a1 := 1;
  comment The following segment of the program performs a
    column-by-column Gaussian reduction of the matrix associ-
    ated with the reference equations, forming an upper and a
    lower triangular matrix into the array P. (Each diagonal
    element of the lower triangular matrix is one.) Interchanges
    of rows take place so that the largest pivot in each column is
    employed. It is assumed that b − 1 columns have already
    been decomposed. If the matrix is not of full rank, the exit
    insufficientrank is taken, and it is left up to the user to deter-
    mine if the given overdetermined system can be solved
    exactly.;
body:
  for i := b step 1 until n do
  begin
    l := ix[i];
    for j := if i = b then 0 else b step 1 until n do
    begin
      kmax := if j < i then j − 1 else i − 1;
      P[i, r[j]] := −ip(k, 0, kmax, P[i, r[k]], P[k, r[j]],
                    −(if r[j] = n then d[l] else A[l, r[j]]))
    end;
    ref := 0;
    for j := i step 1 until n do
    begin
      t := P[i, r[j]];
      if ref < abs(t) then
        begin ref := abs(t);  s := t;  k := j end
    end;
    if ref = 0 then begin j := 1;  go to singular end;
    if i = n then go to decompbranch[a1];
    j := r[k];  r[k] := r[i];  r[i] := j;
    for j := i + 1 step 1 until n do
      P[i, r[j]] := P[i, r[j]]/s
  end;
singular:
  for i := 0 step 1 until n do refset[i] := ix[i];
  go to failures[j];
  comment  Solve for the lambdas.;
return:
  for j := b step 1 until n do
    sv[j] := −ip(k, 0, j − 1, sv[k], P[k, r[j]],
                 −(if r[j] = n then −1 else 0));
  for j := n step −1 until 0 do
    lam[j] := −ip(k, j + 1, n, lam[k], P[k, r[j]], −sv[j])/P[j, r[j]];
  comment  Compute epsilon for the reference subsystem of
    equations.;
  t := 0;
  for i := 0 step 1 until n do t := t + abs(lam[i]);
  eps := 1/t;
  comment  Each new value of eps must be greater than the
    previous one. If this is not so, the solution may have been
    "overshot".;
  if eps < lasteps then go to ed;
  lasteps := eps;
  comment  Solve for the vector x, the Chebyshev solution of the
    reference subsystem of equations.;
  for i := 0 step 1 until n do xr[i] := sign(lam[i]) × eps;
  for i := 0 step 1 until n do
    w[i] := −ip(j, 0, i − 1, w[j], P[i, r[j]], −xr[i])/P[i, r[i]];
  for i := n step −1 until 0 do
    x[r[i]] := −ip(j, i + 1, n, x[r[j]], P[i, r[j]], −w[i]);
  comment  x[n] should be −1. It can be used to purify eps and
    the other components of x.;
  ref: = −x[n];
  for i := 0 step 1 until n1 do x[i] := x[i]/ref;
  eps := eps/ref;
  comment  For each index ix[n+1], ··· , ix[m−1] compute the
    residual A[ix[j], 0] × x[0] + ··· + A[ix[j], n−1] × x[n−1] −
    d[ix[j]]. If the largest of these in magnitude is not greater than
    eps, go to itr to refine the vector x, for it may be the Chebyshev
    solution of the full system.;
  ref := −1;
  for j := np1 step 1 until m1 do
  begin
    i := ix[j];
    t := ip(k, 0, n1, x[k], A[i, k]  −d[i]);
    if abs(t) > ref then
    begin ref := abs(t);  a1 := j;  s := sign(t) end
  end;
  if ref ≤ eps then go to itr;
ovr:
  k := ix[a1];
  comment  The following linear-system solution is computed in
    order to determine which equation is to be dropped from the
    reference set of equations.;
  for i := 0 step 1 until n do
    w[i] := −ip(j, 0, i − 1, w[j], P[j, r[i]],
                −(if r[i] = n then d[k] else A[k, r[i]]));
  for i := n step −1 until 0 do
    w[i] := −ip(j, i + 1, n, w[j], P[j, r[i]], −w[i])/P[i, r[i]];
  comment  s is the sign of the residual with greatest magnitude.
    Find the largest of the ratios (w[k]/lam[k]) × s. If any compo-
    nent of lam is zero, the exit zerolambda is taken.;
  ref := lam[n];  b := n;
  if ref = 0 then begin j := 2;  go to singular end;
  ref := (w[n]/ref) × s;
  for j := 0 step 1 until n1 do
  begin
    t := lam[j];
    if t = 0 then begin j := 2;  go to singular end;
    t := (w[j]/t) × s;
    if t > ref then begin b := j;  ref := t end
  end;
  comment  Form a new reference subsystem by exchanging the
    ix[a1]-th and ix[b]-th equations.;
  ix[a1] := ix[b];  ix[b] := k;  a1 := 1;  go to body;
ed:
  comment  Restore the previous reference substystem.;
  eps := lasteps;  a1 := 2;
  j := ix[a1];  ix[a1] := ix[b];  ix[b] := j;  go to body;
itr:
  lasteps := 0;  cnt := 0;
  comment  Iteratively refine the vector x.;
ilp:
  cnt := cnt + 1;  if cnt > 10 then go to insufficientrank;
  cnorm := snorm := 0;
  for i := 0 step 1 until n do
  begin
    k := ix[i];
    t := abs(x[i]);
    if snorm < t then snorm := t;
    rv[i] := −ip2(j, 0, n, x[j], if j = n then d[k] else
      A[k, j], −xr[i])
  end;
  for i := 0 step 1 until n do
    rv[i] := −ip(j, 0, i − 1, rv[j], P[i, r[j]], −rv[i])/P[i, r[i]];
  for i := n step −1 until 0 do
    w[r[i]] := −ip(j, i+1, n, w[r[j]], P[i, r[j]], −rv[i]);
  for i := 0 step 1 until n do
  begin
    s := w[i];
    x[i] := x[i] + s;
    s := abs(s);
    if cnorm < s then cnorm := s
  end;
  if cnorm/snorm > eta then go to ilp;
  comment  eta is to be preset with a small positive multiple
    of the largest positive single-precision machine number ω
    having the property that 1 + ω = 1 − ω = 1 in a single-precision
    arithmetic. The small multiple will depend upon the
    peculiarities of the machine's rounding process and will have
    to be empirically determined.;
  ref := −x[n];
  for i := 0 step 1 until n1 do x[i] := x[i]/ref;
  eps := eps/ref;
  comment Determine whether a Chebyshev solution has been
    found. If any residual is greater in magnitude than eps while
    eps is smaller than a value produced from an earlier refinement,
    give up, print a warning, and return the best x computed thus
    far.;
  ref := −1;
  for j := np1 step 1 until m1 do
  begin
    i := ix[j];
    t := ip2(k, 0, n1, x[k], A[i, k], −d[i]);
    if abs(t) > ref then
      begin ref := abs(t);  al := j;  s := sign(t) end
  end;
  if ref ≤ eps then finished := true
  else if eps > preveps then finished := false
  else
  begin outstring (1, 'DOUBTFUL SOLUTION');
    go to skip
  end;
  preveps := eps;  refset[n] := ix[n];
  for i := 0 step 1 until n1 do
  begin
    refset[i] := ix[i];
    h[i] := x[i]
  end;
  if ¬ finished then go to ovr;
skip:
  epz := preveps;
end Chebyshev
```

CERTIFICATION OF ALGORITHM 328 [F4] CHEBYSHEV SOLUTION TO AN OVERDETERMINED LINEAR SYSTEM [Richard H. Bartels and Gene H. Golub, *Comm. ACM 11* (June 1968), 428]

Norman L. Schryer (Recd. 14 Nov. 1968, 2 Dec. 1968 and 27 Jan. 1969)
University of Michigan, Ann Arbor, Michigan

KEY WORDS AND PHRASES: Chebyshev solutions, overdetermined linear systems, linear equations, exchange algorithm
CR CATEGORIES: 5.13, 5.14, 5.41

Two modified versions of the procedure *Chebyshev* have been written, one with and one without iterative improvement. The algorithms were compiled in Fortran IV on an IBM System/360 model 67 in double-quadruple and double-precision, respectively. When run on the following test system

$$\begin{bmatrix} 11 & -8 & 6 \\ 0 & -15 & -12 \\ -13 & -3 & 10 \\ 7 & 8 & 2 \\ 10 & -7 & 9 \\ 0 & -5 & 5 \\ 7 & 10 & 9 \\ -15 & 0 & 15 \\ -15 & 3 & -15 \\ 2 & 5 & 14 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -68 \\ -54 \\ 11 \\ 3 \\ -64 \\ -19 \\ 13 \\ 30 \\ 72 \\ -5 \end{bmatrix}$$

both versions gave the correct answer, $x = (-3, 4, -1)$, to full double-precision accuracy (16 digits). The above versions of the procedure *Chebyshev* differ from the published ones in two ways. Without these changes the routines have gone into an "infinite loop" in certain circumstances.

On page 429, first column, the 14th line following the label *return* should be changed from

```
  if eps < lasteps then go to ed;
```

to

```
  if eps ≤ lasteps then go to ed;
```

The above change eliminates the problem of "infinite loops."

When using the version without iterative improvement, one additional change is necessary. Change the code following the label *itr* as indicated on page 405, column 2. Then replace the code between labels *ed* and *itr* on page 429, column 2, by the following:

```
  comment Restore the previous reference subsystem;
  eps := lasteps;
  j := ix[a1];  ix[a1] := ix[b];  ix[b] := j;
  ref := −1;
  for j := np1 step 1 until m1 do
  begin
    i := ix[j];
    t := ip2(k, 0, n1, x[k], A[i, k], −d[i]);
    if abs(t) > ref then ref := abs(t)
  end;
```

This change is necessary in order to give the real variable *ref* the proper value for determining if the vector *x* is a solution or a "doubtful solution." That is, the above value of *ref* will be used in the code following the label *itr* to determine if we have a "doubtful solution."

ALGORITHM 329
DISTRIBUTION OF INDISTINGUISHABLE OBJECTS INTO DISTINGUISHABLE SLOTS [G6]

Robert R. Fenichel

(Recd. 24 Aug. 1967 and 8 Dec. 1967)

Electrical Engineering Department, Massachusetts Institute of Technology, Cambridge, Mass. 02139

KEY WORDS AND PHRASES: object distributions, combinations, distribution numbers
CR CATEGORIES: 5.39

```
procedure dist (k, m, done, q, FirstCall);
  value k, m;  integer k, m;  label done;
  integer array q;  Boolean FirstCall;
```

comment Successive calls to this procedure compute the $\binom{m+k-1}{m}$ distinguishable distributions of *m* indistinguishable objects into *k* distinguishable slots. Upon the first call to *dist*, *FirstCall* must have the value **true**. This value is changed to **false** during the processing of the first call.

Upon return from a call to *dist*, a new distribution has been noted in $q[1:k]$, an integer array. In particular, the number of objects to be distributed to the *i*th slot has been left as the value of $q[i]$.

The call following the $\binom{m+k-1}{m}$-th will cause transfer to the label *done*.

The values of *q* must not be altered between calls to *dist*.

The method is best introduced by means of an example. Suppose that 9 objects must be distributed among 3 slots. Each distribution might be denoted by a three-digit decimal number whose digits sum to 9. By the Rule of Nine, each such "distribution number" is divisible by 9. Conversely, many multiples of 9 are distribution numbers, although some (*e.g.* 189 and 198) are not.

Now the method is as follows:

1. Treat $q[1] \cdots q[k]$ as a *k*-place number in a number system based on $(m+1)$. Usually, return from *dist* after adding *m* to this number.

2. If $q[i-1] \neq q[i] = q[i+1] = \cdots = q[k] = 0$, adding *m* will not result in a distribution number: the sum of the digits will be too large. Find the next distribution number by
   a. Setting $q[k] := q[i-1] - 1$.
   b. Setting $q[i-1] := 0$.
   c. Adding 1 to $q[i-2]$.

The author is indebted to the anonymous referee who, at one point in this algorithm's development, had evidently given it more thought than had the author;

```
begin integer i;  own integer LeftmostZero;
  if FirstCall then
  begin
    for i := 1 step 1 until k - 1 do
      q[i] := 0;
    LeftmostZero := k + 1;
    q[k] := m;
    FirstCall := false
  end
  else if q[1] = m then go to done
  else if LeftmostZero < k + 1 then
  begin
    LeftmostZero := LeftmostZero - 1;
    q[k] := q[LeftmostZero] - 1;
    q[LeftmostZero] := 0;
    q[LeftmostZero - 1] := q[LeftmostZero - 1] + 1
  end skip 99, 189, 198, etc.
  else
  begin
    if q[k] = 1 then LeftmostZero := k;
    q[k] := q[k] - 1;
    q[k-1] := q[k-1] + 1
  end add m to units place
end of dist
```

REMARK ON ALGORITHM 329 [G6]
DISTRIBUTION OF INDISTINGUISHABLE OBJECTS INTO DISTINGUISHABLE SLOTS [Robert R. Fenichel, *Comm. ACM 11* (June 1968), 430]

M. Gray (Recd. 20 Sept. 1968)

Computing Science Department, University of Adelaide, South Australia

As the procedure stands it is incorrect. Preceding

end skip 99,189,198, etc.

the following statement should be inserted:

if $q[k] \neq 0$ **then** *LeftmostZero* := $k + 1$

Thus the compound statement becomes:

```
begin
  LeftmostZero := LeftmostZero -1;
  q[k] := q[LeftmostZero] - 1;
  q[LeftmostZero] := 0;
  q[LeftmostZero-1] := q[LeftmostZero-1] + 1;
  if q[k] ≠ 0 then LeftmostZero := k + 1
end skip 99, 189, 198, etc.
```

ALGORITHM 330
FACTORIAL ANALYSIS OF VARIANCE [G1]

Ian Oliver (Recd. 21 Sept. 1967 and 12 Jan. 1968)
Computer Center, The Ohio State University, 1314 Kinnear Rd., Columbus, Ohio 43212
(Now at Computer Centre, University of Queensland, St. Lucia, Brisbane, Australia 4067)

KEY WORDS AND PHRASES: factorial variance analysis, variance, statistical analysis
CR CATEGORIES: 5.5

```
procedure factorial ANOVA (X, n, levels, T);
  value n;  integer n;  integer array levels;  real array X, T;
```

comment This procedure carries out an analysis of variance on the data from a balanced complete factorial experiment. The experimental observations are assumed to be stored in the array *X*. The elements of the array *levels* are assumed to contain the number of levels in each of the *n* factors. The procedure produces the sum of squares for the analysis of variance table in the array *T*. A method of orthogonal transformations [1] is used.

The levels of the *j*-th factor are numbered 1, 2, ⋯ , *levels*[*j*]. The observations are conveniently stored in a multidimensional array. For example, for $n = 3$, *X*[1, 3, 2] is the observation taken at levels 1, 3, and 2 of the first, second and third factors respectively. *factorial ANOVA* actually uses the procedure *index* to compute the multidimensional subscript and uses *X* as a one dimensional array so that *n* may have any value. Thus, if *factorial ANOVA* is called with a multidimensional array as the first argument, then *index* may have to be rewritten for a given compiler to correctly compute any multiple subscript. As written, *index* assumes that *X* has been declared in a statement such as **real array** *X*[1:*levels*[1],⋯,1:*levels*[*n*]] and that the compiler arranges storage so that the first subscript varies most rapidly.

Alternatively the data may be transmitted in a linear array so that the factor levels associated with each observation are ordered so that the levels of the first factor vary most rapidly. The procedure *index* will then require no modification.

The array *T* may also be considered a linear array, or an *n*-dimensional array declared in a statement of the form **real array** *T*[1:2,1:2,⋯,1:2]. Element *T*[2,1,⋯,1] is the sum of squares for the main effect of the first factor. *T*[1,2,1, ⋯,1] is the main effect for the second factor. *T*[2,2,1,⋯,1] is the interaction between the first two factors, and so on. If *T* is considered as a linear array, an element may be interpreted by examining the bit pattern in the binary value of the subscript minus one. For example, *T*[6] = *T*[5+1] is the interaction between the first and third factors.

On return from *factorial ANOVA* the data array *X* will contain orthogonal components of the sums of squares in the array *T*. As written, the components are the squares of values obtained by performing an Helmert transformation [2] for each factor. The procedure *orthog* may be modified, if the components are required per se, to produce any desired orthogonal contrasts.

The advantages and limitations claimed for *factorial ANOVA* are as follows. The procedure is very conservative of storage provided no factor has a large number of levels. The amount of temporary array storage required is $3n + m(m+2)$ where *m* is the maximum number of levels in any factor. The procedure body is also very short. The routine should therefore be useful for small computers or for inclusion as a subroutine in programs whose primary purpose is not the statistical analysis. No comparison of running time has been made with other methods but this routine requires $\prod_i levels_i(\sum_i levels_i+1)$ floating multiplications and may therefore be comparable in speed with the method described in [3].

This procedure is intended to present an algorithm rather than an optimal program for an algorithm and so the coding can be considerably improved in efficiency which was somewhat sacrificed for clarity.

Acknowledgment. The author wishes to thank the referee and the editor for their valuable comments and suggestions.

References:

1. Oliver, I. Analysis of factorial experiments using generalized matrix operations. *J. ACM 14* (July 1967), 508-519.
2. Kendall, M. G., and Stuart, A. *The Advanced Theory of Statistics*, Vol. 1. Hafner, New York, 1958, pp. 250-251.
3. Hartley, H. O. Analysis of variance. In *Mathematical Methods for Digital Computers*, A. Ralston and H. S. Wilf (eds.), Wiley, New York, 1960, pp. 221-230;

```
begin integer factor, k1, k2, j;  integer array i, Ti, Tlimit [1:n];
  integer procedure index(subscript, limit);
    integer array subscript, limit;
  begin integer j, temp;
    temp := 0;
    for j := n step − 1 until 1 do
      temp := temp × limit[j] + subscript[j] − 1;
    index := temp + 1;
  end index procedure;
  procedure orthog(Q, size);
    value size;  integer size;  real array Q;
  begin integer i, j;
    for i := 1 step 1 until size do Q[i, 1] := 1.0/sqrt(size);
    for j := 2 step 1 until size do
    begin
      for i := 1 step 1 until j − 1 do
      Q[i, j] := − 1.0/sqrt(j×(j−1));
      Q[j, j]:= sqrt(j−1)/j);
      for i := j + 1 step 1 until size do Q[i, j] := 0
    end
  end orthog procedure;
  comment  Carry out orthogonal transformation;
  for factor := 1 step 1 until n do
  begin
    real array A, B[1:levels[factor]], Q[1:levels[factor],
      1:levels[factor]];
    orthog(Q, levels[factor]);
    for j := 1 step 1 until n do i[j] := 1;
loop1:  for i[factor] := 1 step 1 until levels [factor] do
      A[i[factor]] := X[index(i, levels)];
    for k1 := 1 step 1 until levels[factor] do
    begin B[k1] := 0;
      for k2 := 1 step 1 until levels[factor] do
        B[k1] := B [k1] + Q[k2, k1] × A[k2]
    end;
    for i[factor] := 1 step 1 until levels[factor] do
      X[index(i, levels)] := B[i[factor]];
    for j := 1 step 1 until n do
      if j ≠ factor then
      begin
        i [j] := i [j] + 1;
        if i[j] ≤ levels [j] then go to loop1 else i[j] := 1
      end
  end;
  comment  Form mean squares and sums of squares;
  for j := 1 step 1 until n do
    begin Ti[j] := 1; Tlimit[j] := 2 end;
loop2:  for j := 1 step 1 until n do i[j] := Ti[j];
  k1 := index(Ti, Tlimit);  T[k1] := 0;
loop3:  k2 := index(i, levels);
  X[k2] := X[k2] ↑ 2;  T[k1] := T[k1] + X[k2];
  for j := 1 step 1 until n do
    if Ti[j] ≠ 1 then
    begin
      i[j] := i[j] + 1;
      if i[j] ≤ levels[j] then go to loop3 else i[j] := 2
    end;
  for j := 1 step 1 until n do
  begin
    Ti[j] := Ti[j] + 1;
    if Ti[j] ≤ 2 then go to loop2 else Ti[j] := 1
  end
end factorial ANOVA
```

ALGORITHM 331
GAUSSIAN QUADRATURE FORMULAS [D1]

WALTER GAUTSCHI (Recd. 26 Aug. 1967 and 8 Feb. 1968)
Purdue University, Lafayette, Ind., and Argonne National Laboratory,*Argonne, Ill. 60439

*Work performed under the auspices of the US Atomic Energy Commission

KEY WORDS AND PHRASES: quadrature, Gaussian quadrature, numerical integration, weight function, orthogonal polynomials
CR CATEGORIES: 5.16

```
begin
```

comment The procedure *Gauss* below obtains Gaussian quadrature formulas relative to any weight function whose singularities, if any, are monotonic and located at the endpoints of the (finite or infinite) interval of integration. The procedure is most useful for (but not restricted to) "nonclassical" weight functions, i.e. weight functions for which the associated orthogonal polynomials are not known explicitly;

```
real procedure Fourier (c, n);  value c, n;  integer n;   real c;
```

comment This is a subroutine computing

$$1 - 2 \sum_{m=1}^{n} \frac{\cos (2m\theta)}{4m^2 - 1}, \qquad c == \cos \theta,$$

the truncated Fourier series of $(\pi/2) \sin \theta$;

```
begin integer m;  real c0, c1, c2, t, sum;
  c1 := 1;  c0 := 2 × c × c − 1;  t := 2 × c0;
  sum := c0/3;
  for m := 2 step 1 until n do
  begin
    c2 := c1;  c1 := c0;  c0 := t × c1 − c2;
    sum := sum + c0/(4×m×m−1)
  end;
  Fourier := 1 − 2 × sum
end Fourier;
procedure transform (t, phi, phi1);  value t;  real t, phi, phi1 ;
begin real t1;
  t1 := abs(t);
  phi := t/(1−t1);  phi1 := 1/((1−t1)×(1−t1))
end transform;
procedure symm transf (t, phi, phi1);  value t;  real t, phi, phi1;
begin real t2;
  t2 := t × t;
  phi := t/(1−t2);  phi1 := (1+t2)/((1−t2)×(1−t2))
end symm transf;
procedure Gauss (sequential, finite left, finite right, left, right,
  eps, wf, capn, n, results);
  value sequential, finite left, finite right, left, right, eps, capn, n;
  integer capn, n;  real left, right, eps;
  Boolean sequential, finite left, finite right;
  real procedure wf;
  array results;
```

comment This procedure generates approximate values for the abscissas and weights of Gaussian quadrature formulas with weight function *wf*. If the Boolean variable *sequential* has the value **true**, then *k*-point formulas

$$\int_a^b g(x)wf(x)\ dx \cong \sum_{r=1}^{k} w_r^{(k)} g(x_r^{(k)}),$$
$$-\infty \leqq a < x_1^{(k)} < x_2^{(k)} < \cdots < x_k^{(k)} < b \leqq \infty,$$

are generated for $k = 1, 2, \cdots, n$, the abscissa $x_r^{(k)}$ being stored in *results* $[k, r]$, the weight $w_r^{(k)}$ in *results* $[n+1-k, n+2-r]$. The array *results*, in this case, should be declared to have dimensions $[1:n, 1:n+1]$. If the value of *sequential* is **false**, then a single n-point formula is produced with the abscissa $x_r^{(n)}$ being stored in *results* $[1, r]$, the weight $w_r^{(n)}$ in *results* $[2, r]$. In this case, the array *results* need only have dimensions $[1:2, 1:n]$. The Boolean variable *finite left* must be assigned the value **true**, if the lower limit of integration, a, is a finite number, otherwise the value **false**. Similarly for the upper limit b and the associated Boolean variable *finite right*. The parameter *left* is to be set equal to a, if a is finite, and may be assigned an arbitrary value, if $a = -\infty$. Similarly for the parameter *right*, which should be equal to b, if b is finite, and may be arbitrary, if $b = \infty$. The parameter *eps* is a tolerance used to control termination of Newton's iteration for the calculation of the abscissas $x_r^{(k)}$. If d significant digits are desired one may set $eps = .5 \times 10^{-d}$. Some leeway should be allowed to accommodate moderate accumulation of rounding errors.

The method of computation is based on a suitable discretization of the inner product $(f, g) = \int_a^b f(x)g(x)wf(x)\ dx$, the number of points used in the discretization being given by *capn*. The desired abscissas and weights are approximated by the zeros and weight factors of the resulting orthogonal polynomials of a discrete variable. The process converges as $capn \rightarrow \infty$, provided the singularities of the weight function *wf*, if any are present, are located at the endpoints a, b and are monotonic. The traditional approach via moments is deliberately avoided because of its ill-conditioned character (when n is large). Further details of the method are to appear elsewhere [4].

No general rules can be given for the appropriate value of *capn*, the choice depending both on the desired accuracy and the rate of convergence of our process. A reasonable approach is to try, say, $capn = 10 \times n$, and to repeat with a larger value of *capn* (say twice as large). If the results agree to within the desired accuracy, those of the second trial may be accepted as final. Otherwise, *capn* might be further incremented.

The nonsequential version of the procedure is preferable if quadrature formulas for only one, or a few, selected values of n are desired.

The procedure *Gauss* calls on the procedures *transform, symm transf*, and the real procedures *Fourier*, *wf*, all of which (except the last) are declared above. The real procedure *wf* has to be supplied by the user;

```
begin
  integer k, m, r, kmax, count, it;
  real eps1, sum, phi, phi1, t0, t1, pol0, pol1, q, c0, c1, c2, lower
    bound, upper bound;
  array w, x[1:capn], a[0:n−1], b[0:n], p0, p1, p2[−1:capn],
    p[−1:n], list[0:n];
  procedure p and p1(bool, m, n, t, p0, p, p1);
    value m, n, t;  integer m, n;  real t;  Boolean bool;
    array p0, p, p1;
  comment This procedure evaluates the m-times deflated (dis-
    crete) orthonormal polynomials $p_r(x)(r=m, m+1, \cdots, n)$,
    as well as their first derivatives (if bool is true), for given
    argument t. The array p0 is assumed to hold the values of the
    $(m-1)$-times deflated polynomials evaluated at the m-th
    zero of $p_n$. When $m = 0$ these are the values 1, 0, 0, ⋯ , 0;
  begin integer r;
    p[m] := p0[m−1]/b[m];  p[m−1] := 0;
    for r := m step 1 until n − 1 do
      p[r+1] := (p0[r]+(t−a[r])×p[r]−b[r]×p[r−1])/b[r+1];
    if bool then
    begin
      p1[m] := p1[m−1] := 0;
      for r := m step 1 until n − 1 do
        p1[r+1] := (p[r]+(t−a[r])×p1[r]−b[r]×p1[r−1])/b[r+1]
    end
  end p and p1;
  lower bound := left;    upper bound := right;
  comment The piece of program extending from this point to
    the second following comment sets up the abscissas $x_k$ and
    weight factors $w_k$ to be used in the inner product of the dis-
    crete orthogonal polynomials. It is here (and only here)
    where explicit use is made of the given weight function wf;
  kmax := entier (capn/2);
  for k := 1 step 1 until kmax do
  begin
    x[capn+1−k] := cos(1.5707963268×(2×k−1)/capn);
    x[k] := −x[capn+1−k];
    w[k] := w[capn+1−k] := Fourier (x[k], kmax)
  end;
  comment In the preceding for-statement the values of the
    cosine could have been generated recursively with consider-
    able saving of time, but some loss of accuracy, if capn is very
    large. It was decided to sacrifice efficiency in favor of accu-
    racy.
      If the weight function contains a square root singularity,
    typified by $x^{-1/2}$ at $x = 0$, rather improved accuracy may result
    from modifying the last preceding statement to read
    w[k]:=w[capn+1−k]:=1.5707963268 × sqrt(1−x[k]×x[k]),
    and the second following statement to read
                w[kmax+1] := 1.5707963268.
    This is especially so if the square root singularity occurs at
    both endpoints;
  if capn/2 ≠ kmax then
  begin
    x[kmax+1] := 0;  w[kmax+1] := Fourier(0, kmax)
  end;
  if finite left then
  begin
    if finite right then go to L1 else go to L2
  end
  else
  begin
    if finite right then go to L3 else go to L4
  end;
L1: for k := 1 step 1 until capn do
  begin
    x[k] := ((right-left)×x[k]+right+left)/2;
    w[k] := (right-left)×w[k]×wf(x[k])/capn
  end;
  go to continue;
L2: for k := 1 step 1 until capn do
  begin
    transform (.5×(1+x[k]), phi, phi1);
    x[k] := left + phi;
    w[k] := w[k] × wf(x[k]) × phi1/capn
  end;
  go to continue;
L3: for k := 1 step 1 until capn do
  begin
    transform (.5×(−1+x[k]), phi, phi1);
    x[k] := right + phi;
    w[k] := w[k] × wf(x[k]) × phi1/capn
  end;
  go to continue;
L4: for k := 1 step 1 until capn do
  begin
    symm transf (x[k], phi, phi1);
    x[k] := phi;
    w[k] := 2 × w[k] × wf(phi) × phi1/capn
  end;
  comment The piece of program extending from this point to
    the second following comment generates the coefficients $a_r$,
    $b_{r+1}(r=0, 1, \cdots, n-1)$ in the recurrence relation
            $p_{r+1}(x) = ((x-a_r)p_r(x)-b_r p_{r-1}(x))/b_{r+1}$
    for the (discrete) orthononormal polynomials $p_r$ associated
    with the inner product

            $[f, g] = \sum_{k=1}^{capn} w_k f(x_k) g(x_k)$.

    The content of b[0] is set equal to $1/p_0$;
continue: sum := 0;
  for k := 1 step 1 until capn do sum := sum + w[k];
  b[0] := sqrt(sum);
  for k := 1 step 1 until capn do
  begin
    p1[k] := 0;  p2[k] := 1/b[0]
  end;
  for r := 0 step 1 until n−1 do
  begin
    sum := 0;
    comment If a = −∞, or b = ∞, overflow conditions may
      arise in the following two for-statements, which, if ignored,
      should normally be of no consequence;
    for k := 1 step 1 until capn do
    begin
      p0[k] := p1[k];  p1[k] := p2[k];
      sum := sum + w[k] × x[k] × p1[k] × p1[k]
    end;
    a[r] := sum;  sum := 0;
    for k := 1 step 1 until capn do
    begin
      p2[k] := (x[k]−a[r])×p1[k]−b[r]×p0[k];
      sum := sum + w[k] × p2[k] × p2[k]
    end;
    b[r+1] := sqrt(sum);
    for k := 1 step 1 until capn do p2[k] := p2[k]/b[r+1]
  end;
  comment Using the values of $a_r$, $b_{r+1}$ just obtained, the pro-
    cedure now produces upper and lower bounds for the zeros of
    $p_n(x)$ when $b = \infty$, or $a = -\infty$, respectively. The bounds
    are derived by applying the Gershgorin circle theorem to
    the Jacobi matrix associated with the polynomials $p_r$;
  if ¬ finite right then
  begin
    upper bound := a[0] + b[1];
    for r := 1 step 1 until n − 2 do
    begin
      t0 := a[r] + b[r] + b[r+1];
      if t0 > upper bound then upper bound := t0
    end;
    t0 := a[n−1] + b[n−1];
    if t0 > upper bound then upper bound := t0
```

```
end;
if ¬ finite left then
begin
  lower bound := a[0] − b[1];
  for r := 1 step 1 until n − 2 do
  begin
    t0 := a[r] − b[r] − b[r+1];
    if t0 < lower bound then lower bound := t0
  end;
  t0 := a[n−1] − b[n−1];
  if t0 < lower bound then lower bound := t0
end;
```

comment The remaining section of this procedure determines approximations of the desired abscissas and weights. If *sequential* is true, the zeros of the (discrete) orthonormal polynomials $p_r(r=1, 2, \cdots, n)$ are determined sequentially using Newton's method. Suitable initial approximations are found on the basis of the interlacing property of the zeros. Each Newton approximation is checked on whether or not it satisfies this property. If not, the appropriate subinterval is searched more thoroughly for possible zeros. If none is detected the message "search for zeros unsuccessful" is printed out. Otherwise, Newton's iteration is repeated with a revised initial approximation. If again the interlacing property turns out to be violated the message "interlacing property of the zeros is violated" is printed out. The message "Newton iteration diverges" is printed if, for any reason, Newton's iteration fails to converge within 30 iterations. In either of these abortive situations the procedure exits, leaving the current quadrature formula, and all subsequent formulas, uncompleted.

In the nonsequential case, the zeros of p_n are obtained by Newton's method and successive deflation. Each deflation (except the first) is preceded by a refinement of the respective zero using Newton's iteration based on the original (undeflated) polynomial p_n. If this iteration fails to converge within 15 iterations the message "Newton iteration in refinement diverges" is printed out. If Newton's method for the deflated polynomials fails to converge within 30 iterations, it is checked whether this may be due to the tolerance *eps* being too stringent, considering the presence of subtraction errors in the generation of the polynomials and their derivatives. If this is the case, the procedure goes on to refine the particular zero. Otherwise, it prints out the message "Newton iteration diverges." In either of the two abortive situations the procedure exits, leaving the quadrature formula unfinished.

The weights are computed by the formula

$$[w_r^{(k)}]^{-1} = \sum_{s=0}^{k-1} [p_s(x_r^{(k)})]^2;$$

```
p2[−1] := 1;  for k := 0 step 1 until n − 1 do p2[k] := 0;
if sequential then
begin
  list[0] := lower bound;
  results [1, 1] := a[0];  results[n, n+1] := b[0] × b[0];
  for k := 2 step 1 until n do
  begin
    for m := 1 step 1 until k − 1 do
      list[m] := results[k−1, m];
    list[k] := upper bound;
    for m := 1 step 1 until k do
    begin
      t0 := (list[m]+list[m−1])/2;  count := it := 0;
Newton: t1 := t0;  it := it + 1;
      p and p1(true, 0, k, t1, p2, p, p1);
      t0 := t1 − p[k]/p1[k];
      if t0 ≦ list[m−1] ∨ t0 ≧ list[m] then
      begin
        if count = 0 then
        begin
          t0 := list[m−1];
          p and p1 (false, 0, k, t0, p2, p, p1);
          pol0 := p[k];  q := .2 × (list[m]−list[m−1]);
search:   t1 := t0 + q;
          p and p1 (false, 0, k, t1, p2, p, p1);
          pol1 := p[k];
          if pol0 × pol1 > 0 then
          begin
            t0 := t1;
            if t0 < list[m] then go to search else
            begin
              outstring (1, 'search for zeros unsuccessful');
              outinteger (1, k);  outinteger (1, m);
              go to exit
            end
          end
          else
          begin
            t0 := (t0+t1)/2;  count := count + 1;
            go to Newton
          end
        end
        else
        begin
          outstring (1, 'interlacing property of zeros is violated');
          outinteger (1, k);  outinteger (1, m);
          go to exit
        end
      end;
      if it > 30 then
      begin
        outstring (1, 'Newton iteration diverges');
        outinteger (1, k);  outinteger (1, m);
        go to exit
      end;
      if abs(t1−t0) > eps × abs(t0) ∧ abs(t1−t0) > eps
        ∧abs(t0) > eps then
        go to Newton;
      results [k, m] := t0;
      p and p1 (false, 0, k−1, t0, p2, p, p1);
      sum := 0;
      for r := 0 step 1 until k − 1 do
        sum := sum + p[r] × p[r];
      results[n+1−k, n+2−m] := 1/sum
    end
  end
end
else
begin
  p[−1] := 1;
  for k := 0 step 1 until n − 1 do p[k] := 0;
  t0 := lower bound;
  for m := 0 step 1 until n − 2 do
  begin
    for k := m − 1 step 1 until n − 1 do p0[k] := p[k];
    it := 0;
Newton1: t1 := t0;
    it := it + 1;
    p and p1 (true, m, n, t1, p0, p, p1);
    t0 := t1 − p[n]/p1[n];
    if it > 30 then
    begin
```

```
          c0 := abs(p0[n−1]);
          c1 := abs((t1−a[n−1])×p[n−1]);
            c2 := abs(b[n−1]×p[n−2]);
          phi := if c0 ≦ c1 then
            (if c1≦c2 then c2 else c1)
            else
            (if c0≦c2 then c2 else c0);
          phi := phi/b[n];
          c0 := abs(p[n−1]);
          c1 := abs((t1−a[n−1])×p1[n−1]);
            c2 := abs(b[n−1]×p1[n−2]);
          phi1 := if c0 ≦ c1 then
            (if c1≦c2 then c2 else c1)
            else
            (if c0 ≦ c2 then c2 else c0);
          phi1 := abs(phi1/(b[n]×p1[n]));
          phi := if phi < phi1 then phi1 else phi;
          eps1 := if phi > 1 then 10 × phi × eps else 10 × eps;
          if abs(t1−t0) > eps1 × abs(t0) ∧ abs(t0) > eps1  then
          begin
            outstring (1, 'Newton iteration diverges');
            outinteger(1, m+1);
            go to exit
          end
        end
        else
        begin
          if abs(t1−t0) > eps × abs(t0) ∧ abs(t1−t0) > eps
            ∧abs(t0) > eps then
            go to Newton1
        end;
        if m > 0 then
        begin
          it := 0;
refine:  t1 := t0;
          it := it + 1;
          p and p1(true, 0, n, t1, p2, p, p1);
          t0 := t1 − p[n]/p1[n];
          if it > 15 then
          begin
            outstring(1, 'Newton iteration in refinement diverges');
            outinteger(1, m+1);
            go to exit
          end;
          if abs(t1−t0) > eps × abs(t0) ∧ abs(t1−t0) > eps
            ∧abs(t0) > eps then
            go to refine
        end;
        results[1, m+1] := t0;
        p and p1(false, m, n−1, t0, p0, p, p1)
      end;
      results[1, n] := a[n−1] − b[n−1] × p[n−1]/p[n−2];
      for k := 1 step 1 until n do
      begin
        p and p1(false, 0, n−1, results[1, k], p2, p, p1);
        sum := 0;
        for r := 0 step 1 until n − 1 do
          sum := sum + p[r] × p[r];
        results[2, k] := 1/sum
      end
    end;
exit: end Gauss;
```

comment The procedure *Gauss*, in both the sequential and nonsequential form, was tested on the CDC 3600 computer for a number of weight functions. The tolerance *eps* = $.5_{10}$-9 was used throughout. The following surveys the results obtained in a few representative cases.

(i) $wf(x) = x^\alpha \ln(e/x), 0 < x < 1, \alpha = 0(1)3,.5,-.5$, $n = 5$, $capn = 100$. The maximum absolute error (rounded to 3 significant figures) in the abscissas and weights is shown below together with the values of k and r at which the maximum occurs ($1 \leq k \leq n$, $1 \leq r \leq k$). For comparison we used the 7–11S values published by V. I. Krylov and A. A. Pal'cev [5].

α	maximum error in abscissas	k	r	maximum error in weights	k	r
0	$1.10_{10}-6$	2	1	$5.63_{10}-6$	1	1
1	$2.54_{10}-7$	5	4	$3.90_{10}-7$	5	3
2	$7.58_{10}-7$	5	4	$9.53_{10}-7$	5	3
3	$3.88_{10}-7$	4	3	$2.87_{10}-7$	4	3
.5	$6.76_{10}-7$	4	1	$5.46_{10}-7$	5	2
−.5	$1.86_{10}-3$	1	1	$5.97_{10}-2$	1	1

Note the relatively large errors for $\alpha = -\frac{1}{2}$; using the modification mentioned in the sixth comment, these errors are slightly reduced to 6.77_{10}–4 and 2.17_{10}–2 respectively.

(ii) $wf(x) = \ln(e/(1-x)) \ln(e/x)$, $0 < x < 1$, $n = 5$, $capn =$ 100,200,400. Comparing the results with 11S values given by V. I. Krylov and A. A. Pal'cev [5] the following absolute errors were observed.

$capn$	maximum error in abscissas	k	r	maximum error in weights	k	r
100	$9.33_{10}-7$	3	1	$1.13_{10}-5$	1	1
200	$2.32_{10}-7$	3	1	$2.81_{10}-6$	1	1
400	$5.80_{10}-8$	3	3	$7.04_{10}-7$	1	1

(iii) $wf(x) = [(1-x^2)(1-k^2x^2)]^{-\frac{1}{2}}$, $-1 < x < 1$, $k = .1(.2).9$, $.99$, $n = 10$, $capn = 100$. The weight factors (and, indirectly, the abscissas) were checked by comparing the sum $\sum_{r=1}^{n} w_r^{(n)}$ with the zero-order moment

$$m_0 = \int_{-1}^{1} [(1-x^2)(1-k^2x^2)]^{-1/2}\, dx = 2K(k).$$

The moments m_0 , and the observed discrepancies, are shown below, for the versions with and without the modification mentioned in the sixth comment.

k	m_0	error (with mod.)	error (without mod.)
.1	3.1494911230	$1.16_{10}-10$	$6.93_{10}-3$
.3	3.2160972399	$1.75_{10}-10$	$7.23_{10}-3$
.5	3.3715007097	$5.82_{10}-11$	$7.96_{10}-3$
.7	3.6913879968	$4.07_{10}-10$	$9.66_{10}-3$
.9	4.5610982769	$4.66_{10}-10$	$1.58_{10}-2$
.99	6.7132010474	$1.16_{10}-10$	$4.88_{10}-2$

(The elliptic integral $K(k)$ was computed from a 6th-degree polynomial approximation due to W. J. Cody [2].) The rather dramatic improvement due to the modification is well worth noting. The positive abscissas and corresponding weights for $k = .5$, as obtained by the modified procedure, are given below.

r	$x_r^{(10)}$	$w_r^{(10)}$
6	.15746 64996	.31717 65527
7	.45647 98649	.32407 60350
8	.70963 75175	.33617 78803
9	.89237 18385	.34961 83201
10	.98787 25254	.35870 15666

By symmetry, $x_r^{(n)} = -x_{n+1-r}^{(n)}$, $w_r^{(n)} = w_{n+1-r}^{(n)}$ $(r=1,2, \cdots, n)$.

(iv) $wf(x) = 1/((x+\mu^2)\sqrt{x})$, $0 < x \leq 1$, $\mu = 1,.1,.01$, $n = 10$, 20, $capn = 800$. (The abscissas are the squares of the abscissas of the $2n$-point formula corresponding to $wf(x) = 1/(x^2+\mu^2)$, $-1 \leq x \leq 1$, while the weights are twice those of the $2n$-point formula.) The moments m_k satisfy

$$m_0 = \frac{2}{\mu} \text{arc tan} \left(\frac{1}{\mu}\right),$$

$$m_k = \frac{2}{2k-1} - \mu^2 m_{k-1} \quad (k = 1, 2, \cdots, 2n-1).$$

Shown below are the maximum relative errors in the moments m_k, i.e.

$$r_n = \max_{0 \leqq k \leqq 2n-1} \left| \left(\sum_{r=1}^{n} w_r^{(n)} [x_r^{(n)}]^k - m_k \right) \Big/ m_k \right|.$$

Invariably, the maximum was attained for $k = 2n - 1$. Again, the modification mentioned in the sixth comment was used.

μ	n	r_n	μ	n	r_n
1.0	10	$6.11_{10}-6$	1.0	20	$1.25_{10}-5$
.1		$5.95_{10}-6$	.1		$1.24_{10}-5$
.01		$5.94_{10}-6$	.01		$1.24_{10}-5$

(v) $wf(x) = E_1(x) = \int_1^\infty e^{-xt}\, dt/t$, $0 < x < \infty$, $n = 20$, $capn = 160{,}320{,}640$. The moments in this case are given by $m_k = k!/(k+1)$. Shown below are the maximum relative errors r_n in these moments. The maximum invariably occurred at $k = 0$.

$capn$	r_{20}
160	$2.20_{10}-6$
320	$5.50_{10}-7$
640	$1.37_{10}-7$

Because of the intrinsic interest of this quadrature formula in transfer problems [1] we list below the abscissas and weights obtained with $capn$ = 640, but rounded to 8 significant digits.

r	$x_r^{(20}$	$w_r^{(20}$	r	$x_r^{(20}$	$w_r^{(20)}$
1	.041573069	.33006847	11	13.919556	$1.7373646_{10}-7$
2	.27423961	.33501883	12	16.969573	$7.7197014_{10}-9$
3	.73521299	.20272710	13	20.415565	$2.3285653_{10}-10$
4	1.4364648	.090679419	14	24.304884	$4.5495507_{10}-12$
5	2.3868423	.031192649	15	28.701954	$5.4035520_{10}-14$
6	3.5949493	$8.3968173_{10}-3$	16	33.698200	$3.5673003_{10}-16$
7	5.0704204	$1.7051956_{10}-3$	17	39.431367	$1.1447950_{10}-18$
8	6.8247452	$2.6719239_{10}-4$	18	46.128447	$1.4341583_{10}-21$
9	8.8719945	$3.1522527_{10}-5$	19	54.222968	$4.6337407_{10}-25$
10	11.229631	$2.7511645_{10}-6$	20	64.825944	$1.3623986_{10}-29$

(The exponential integral $E_1(x)$ was evaluated by the series expansion $E_1(x) = -\gamma - \ln x - \sum_{n=1}^{\infty} (-1)^n x^n/(nn!)$, if $0 < x < 2$, and from a rational approximation due to Hastings [3, formula 5.1.56], if $x \geq 2$.)

(vi) $wf(x) = |x|^\alpha e^{-x}$, $-\infty < x < \infty$, $\alpha = 1, 2, 3$, $n = 20$, $capn = 200{,}400{,}800$. Shown below are the maximum relative errors of the abscissas and weights as compared with values tabulated by A. H. Stroud and Don Secrest [6].

α	$capn$	maximum errors in abscissas	maximum errors in weights
1	200	$6.31_{10}-3$	$2.11_{10}-1$
	400	$2.73_{10}-6$	$6.89_{10}-5$
	800	$6.59_{10}-7$	$7.40_{10}-7$
2	200	$1.19_{10}-2$	$4.30_{10}-1$
	400	$1.12_{10}-6$	$3.44_{10}-5$
	800	$4.66_{10}-10$	$2.27_{10}-8$
3	200	$1.67_{10}-2$	$5.59_{10}-1$
	400	$2.69_{10}-6$	$7.93_{10}-5$
	800	$4.53_{10}-10$	$2.12_{10}-8$

end

References:

1. Chandrasekhar, S. *Radiative Transfer.* Oxford U. Press, New York, 1950, Ch. 2.
2. Cody, W. J. Chebyshev approximations for the complete elliptic integrals K and E. *Math. Comput. 19* (1965), 105–112.
3. Gautschi, W. and Cahill, W. F. Exponential integral and related functions. In *Handbook of Mathematical Functions* (M. Abramowitz and I. A. Stegun, Eds.), NBS Appl. Math. Ser. 55, 1964, U.S. Govt. Printing Office, Washington D.C., Ch. 5.
4. Gautchi, W. Construction of Gauss-Christoffel quadrature formulas. *Math. Comput. 22* (1968), 251–270.
5. Krylov, V. I., and Pal'cev, A. A. Approximate integration of functions having logarithmic singularities. (Russian) *Vesci Akad. Navuk BSSR*, Ser. Fiz.-Teh. Navuk (1962), No. 1, 13–18.
6. Stroud, A. H., and Secrest, Don. *Gaussian Quadrature Formulas.* Prentice-Hall, Englewood Cliffs, N. J., 1966.

REMARK ON ALGORITHM 331 GAUSSIAN QUADRATURE FORMULAS [D1] [Walter Gautschi, *Comm. ACM 11* (June 1968), 432]

I. D. Hill (Recd. 12 Sept. 1968)

Medical Research Council, Computer Unit (London), London, N.1, England

KEY WORDS AND PHRASES: quadrature, Gaussian quadrature, numerical integration, weight function, orthogonal polynomials

CR CATEGORIES: 5.16

1. On pages 434 and 435 there are five strings, all of which have identical opening and closing string quotes.' and' should be replaced by 'and' in each case.

2. No space symbols appear in these strings. ␣ should be inserted in each space. Otherwise, no spaces will appear in the printed messages.

3. In the second string, the hyphen in the word "violated" should be deleted.

4. In the first column of page 433 there appear:

kmax := *entier*(*capn*/2);

and

if *capn*/2 ≠ *kmax* **then**

Both these are critically dependent upon rounding error in the real division. Presumably,

kmax := *capn* ÷ 2;

and

if *capn* ≠ 2 × *kmax* **then**

are intended.

5. A semicolon is necessary before the final **end** (on page 436). As things stand, this **end** is part of the comment, and the algorithm never finishes.

Alternatively, the semicolon after **end** *Gauss*, two columns earlier, could be deleted (in which case the symbol **comment** could also be deleted if desired, but need not be). If this were done, the final **end** would terminate the comment without the need for a preceding semicolon.

REMARK ON ALGORITHM 331 [D1]
GAUSSIAN QUADRATURE FORMULAS [Walter Gautschi, *Comm. ACM 11* (June 1968), 432–436]

WILLIAM R. WISE, JR.* (Recd. 28 Jan. 1970 and 2 Mar. 1970)

Box 35343, Georgia Institute of Technology, Atlanta, GA 30332

*Work performed at Danish Atomic Energy Commission, Research Establishment Riso, Reactor Physics Department, Computer Group

KEY WORDS AND PHRASES: quadrature, Gaussian quadrature, numerical integration, weight function, orthogonal polynomials, Newton's method, successive deflation

CR CATEGORIES: 5.16

The last Gaussian point calculated in the nonsequential method, being the root of a linear equation, is calculated directly rather than by Newton's method. In doing so it misses out on the refinement process.

If the *m*-loop is extended to include this last point also (and of course the direct calculation deleted), the results agree more closely with those given by Stroud and Secrest [1]. The following corrections will achieve this:

On the 4th line before the line labeled *Newton* 1 replace:

for $m := 0$ **step** 1 **until** $n - 2$ **do**

by:

for $m := 0$ **step** 1 **until** $n - 1$ **do**

Delete the 17th line following the line labeled *refine*, which now reads:

results $[1, n] := a[n-1] - b[n-1] \times p[n-1]/p[n-2]$;

The following change may be made but is not necessary:

After the 14th line following the line labeled *refine* insert the line:

if $m < n - 1$ **then**

Since this only deletes the call for $m = n - 1$, which is almost degenerate, it usually proves to be a bigger waste of time to include the comparisons rather than to have the unnecessary procedure call.

Table I shows test examples indicating the difference between the last Gaussian point computed directly and by including the last Gaussian point in the *m*-loop.

TABLE I. GAUSS POINTS

Weight function	*Limits*		n (*a*)	*Computed with change*	*Stroud and Secrest [1]*	*Computed without change*
	Lower	*Upper*				
$1 + x$	−1	1	7	.955041232	.955041227	.955041260
$(1 + x)^2$	−1	1	7	.959734457	.959734452	.959734490
$(1 - x^2)^{\frac{3}{2}}$	−1	1	7	.876922576	.876922518	.876922615
$(1 - x^2)^{-\frac{1}{2}}$ (*b*)	−1	1	7	.974927912	.974927912	.974927938
1	−1	1	10	.973906536	.973906529	.973906610
$\lvert x \rvert$	−1	1	7	.954679076	.954679025	.954679111
e^{-x}	0	∞	7	19.3958995	19.3957279	19.3958995

(*a*) A *capn* of 70 was used excepted for $n = 10$ in which *capn* = 90 was used, *eps* = $1.00_{10} -9$ throughout.

(*b*) The change for square root singularities suggested in comment 6 was used.

REFERENCE:

1. STROUD, A. H., AND SECREST, DON. *Gaussian Quadrature Formulas*. Prentice-Hall, Englewood Cliffs, N.J., 1966.

ALGORITHM 332
JACOBI POLYNOMIALS [S22]

Bruno F. W. Witte (Recd. 2 Aug. 1967, 11 Oct. 1967, 8 Dec. 1967, 18 Jan. 1968)
U.S. Navy Electronics Laboratory Center, San Diego, California 92152

KEY WORDS AND PHRASES: Jacobi polynomials, orthogonal polynomials, three-term recurrences, special functions
CR CATEGORIES: 5.12

comments *JACOBI* evaluates in double-precision the Jacobi polynomial $F = P_n(x)$, defined by Rodrigues' formula

$$2^n \cdot n! \cdot P_n(x) = (-1)^n(1-x)^{-\alpha}(1+x)^{-\beta} \cdot D^n[(1-x)^{\alpha+n}(1+x)^{\beta+n}],$$

for degrees n from 0 through 25, and for the given values of the double-precision arguments α, β, and x. The subroutine uses the three-term recurrence relation (see, for example, [1, p. 169]):

$$P_j(x) = (U_j + V_j \cdot x) \cdot P_{j-1}(x) - W_j \cdot P_{j-2}(x). \qquad (1)$$

Also calculated are the derivative $FD = dF/dx$ and estimates of the relative errors E and ED of F and FD. U_j, V_j, and W_j are computed only once when *JACOBI* is called repeatedly with the same values of α and β.

To explain the method for finding E and ED, we refer to the two recursions (2) and (3) below:

$$P_j = G \cdot P_{j-1} - W_j \cdot P_{j-2}, \qquad (2)$$

$$Q_j = H \cdot Q_{j-1} - W_j \cdot Q_{j-2} + s. \qquad (3)$$

Relation (2) is an abbreviated form of (1); relation (3) describes a parallel recursion for a sequence of error-perturbed polynomial values Q_j which does two things: (a) it propagates previous errors of the polynomial values P_{j-1} or Q_{j-1}, and P_{j-2} or Q_{j-2} into Q_j; and (b) it includes the effects of two errors generated "locally" at the j^{th} step: the error of G which is included in H, and the error s which arises when forming the difference itself in (2). The error s is estimated from $s = \max(E_1, E_2)$, where E_1 and E_2 are the magnitudes of the errors of the two terms on the right side of (2), i.e. $E_1 = | E_g \cdot P_{j-1} |$ and $E_2 = | E_w \cdot P_{j-2} |$. Here E_g is the error of G, and E_w is the error of W_j. E_g and E_w are estimated from $E_g = \max(| y \cdot U_j |, | y \cdot V_j \cdot x |)$ and $E_w = | y \cdot W_j |$. The value 3E-26 given to y in the DATA statement reflects the accuracy of the CDC-1604. H in (3) is given as $H = G + E_g$. Finally, the relative error E of P_n is obtained from $E = | 1 - Q_n/P_n |$.

One might argue that the use of (3) could have been avoided if the error of P_{j-1} had been taken into account in the evaluation of E_1, and the error of P_{j-2} in the evaluation of E_2. However, in numerical tests this led to serious instability in the vicinity of the zeros of P_n because of correlations between the errors.

Algorithm 332 is the first algorithm written in FORTRAN to be published in the Algorithms department of Communications of the ACM. The department policy was extended to allow for algorithms in FORTRAN in August 1966. (For details see September 1966 issue, page 683.)

References:
1. Bateman Manuscript Project, Calif. Inst. of Tech. *Higher Transcendental Functions*, vol. 2. McGraw-Hill, New York, 1953, pp. 168–174.
2. Szegö, G. *Orthogonal Polynomials.* Colloq. Publ., vol. 23. American Mathematical Society, New York, 1939, pp. 136–138.
3. Stroud, A. H., and Secrest, D. *Gaussian Quadrature Formulas.* Prentice-Hall, Englewood Cliffs, N. J., 1966, pp. 17–31.

```
      SUBROUTINE JACOBI
C     *****************
     *  (DEGREE,ALFA,BETA,X,F,FD,E,ED)

      DOUBLE PRECISION   A,ALF,ALFA,
     *            B,BET,BETA,C,D,F,FD,
     *            G,H,P,PD,Q,QD,
     *            T1,T2,U,V,W,X

      REAL        E,ED,EG,E1,E2,S,Y

      INTEGER     I,J,K,M,N,DEGREE

      DIMENSION   U(25), V(25), W(25),
     *            P(25),PD(25),
     *            Q(25),QD(25)

      DATA        M  /-2    /,
     *            ALF/-2D+00/,
     *            BET/-2D+00/,
     *            Y  /+3E-26/

       IF  (DEGREE.EQ.0)  GO TO 8

       IF ((ALFA.NE.ALF)
     *.OR. (BETA.NE.BET)) GO TO 1

       IF  (DEGREE.LE.M)  GO TO 5

      I     = M
      K     = DEGREE-1
      M     = DEGREE

       IF  (I-2)  2, 3, 3

C     CALCULATE THE U(J),V(J),W(J) IN
C     THE RECURRENCE RELATION   P(J)=
C     P(J-1)*(U(J)+V(J)*X)-P(J-2)*W(J) ..
    1 M     = DEGREE
      ALF   = ALFA
      BET   = BETA
      A     = ALF+BET
      B     = ALF-BET
      U(1)  = B/2.
      V(1)  = 1.+A/2.
      W(1)  = 0D0

       IF  (DEGREE.EQ.1)  GO TO 5

    2 U(2)  = A*B*(A+3.)/(4.*(A+2.)**2)
      V(2)  = (A+3.)*(A+4.)/(4.*(A+2.))
      W(2)  = (1.+ALF)*(1.+BET)*(A+4.)
      W(2)  = W(2)/(2.*(A+2.)**2)
      I     = 2
      K     = DEGREE-1

    3  IF ((DEGREE.EQ.2)
     *.OR. (  I.GT.K   )) GO TO 5

      DO 4   J = I,K
         A      = 2*J+2
         D      = ALF+BET
         A      = A+D
         B      = D*(A-1.)*(ALF-BET)
         C      = J+1
         C      = 2.*C*(A-2.)*(C+D)
         U(J+1)= B/C
         D      = A*(A-1.)*(A-2.)
         V(J+1)= D/C
         D      = J
         A      = 2.*(D+ALF)*(D+BET)*A
         W(J+1)= A/C
    4    CONTINUE
```

```
C     FIND THE STARTING VALUES FOR J=1
C     AND J=2 FOR USE IN THE RECURSION ..
    5 T1    = V(1)*X
      P(1)  = U(1)+T1
      S     = Y*DMAX1(DABS(U(1)),
     *                DABS(T1))
      Q(1)  = P(1)+S
      PD(1)= V(1)
      QD(1)= V(1)

       IF  (DEGREE.EQ.1)  GO TO 7
       T1    = V(2)*X
       G     = U(2)+T1
       EG    = Y*DMAX1(DABS(U(2)),
     *                DABS(T1))
       H     = G+EG
       T1    = G*P(1)
       E1    = DABS(EG*P(1))
       P(2)  = T1-W(2)
       S     = Y*DABS(W(2))
       S     = AMAX1(E1,S)
       Q(2)  = H*Q(1)-W(2)+S
       PD(2)= G*PD(1)+V(2)*P(1)
       QD(2)= H*QD(1)+V(2)*Q(1)

        IF  (DEGREE.EQ.2)  GO TO 7

C       USE THE RECURSION ..
        DO 6   J = 3,DEGREE
            T2     = V(J)*X
            G      = U(J)+T2
            EG     = Y*DMAX1(DABS(U(J)),
     *                       DABS(T2))
            H      = G+EG
            T1     = G*P(J-1)
            T2     = W(J)*P(J-2)
            E1     = DABS(EG*P(J-1))
            E2     = DABS(T2)*Y
            P(J)   = T1-T2
            S      = AMAX1(E1,E2)
            Q(J)   = H*Q(J-1)-W(J)*Q(J-2)+S
            PD(J)  = G*PD(J-1)-W(J)*PD(J-2)
            QD(J)  = H*QD(J-1)-W(J)*QD(J-2)
            PD(J)  = PD(J)+V(J)*P(J-1)
            QD(J)  = QD(J)+V(J)*Q(J-1)
    6       CONTINUE

C       PREPARE THE OUTPUT ..
    7   N     = DEGREE
        F     = P(N)
        E     = Y+DABS(P(N)-Q(N))
     *              /DABS(F)
        FD    = PD(N)
        ED    = Y+DABS(PD(N)-QD(N))
     *              /DABS(FD)
        GO TO 9

    8   F     = 1D0
        E     = 0.
        FD    = 0D0
        ED    = 0.
    9   RETURN
        END
```

REMARKS ON:
ALGORITHM 332 [S22]
JACOBI POLYNOMIALS [Bruno F. W. Witte, *Comm. ACM 11* (June 1968), 436]
ALGORITHM 344 [S14]
STUDENT'S *t*-DISTRIBUTION [David A. Levine, *Comm. ACM 12* (Jan. 1969), 37]
ALGORITHM 351 [D1]
MODIFIED ROMBERG QUADRATURE [Graeme Fairweather, *Comm. 12* (June 1969), 324]
ALGORITHM 359 [G1]
FACTORIAL ANALYSIS OF VARIANCE [John R. Howell, *Comm. ACM 12* (Nov. 1969), 631]

ARTHUR H. J. SALE (Recd. 16 Feb. 1970)
Basser Computing Department, University of Sydney, Sydney, Australia

KEY WORDS AND PHRASES: Fortran standards
CR CATEGORIES: 4.0, 4.22

An unfortunate precedent has been set in several recent algorithms of using an illegal FORTRAN construction. This consists of separating an initial line from its continuation line by a comment line, and is forbidden by the standard (see sections 3.2.1, 3.2.3 and 3.2.4 of [1, 2]). The offending algorithms are to date: 332, 344, 351 and 359.

While this is perhaps a debatable decision by the compilers of the standard, and trivial to correct, it seems a pity to break the rules just for a pretty layout as has been done.

REFERENCES:

1. ANSI Standard FORTRAN (ANSI X3.9-1966), American National Standards Institute, New York, 1966.
2. FORTRAN vs. Basic FORTRAN, *Comm. ACM 7* (Oct. 1964), 591–625.

Remark on Algorithm 332 [S22]
Jacobi Polynomials [Bruno F.W. Witte, *Comm. ACM 11* (June 1968), 436]

Ove Skovgaard (Recd 23 April 1974 and 22 July 1974)
Institute of Hydrodynamics and Hydraulic Engineering, Technical University of Denmark, DK-2800 Lyngby/Denmark

In the last section of Algorithm 332, there are the following statements:

E = Y + DABS(PD(N) − Q(N))/DABS(F)

where *E* should be an estimate of the relative error of the computed value *F* (Jacobi polynomial);

ED = Y + DABS(PD(N) − QD(N))/DABS(FD)

where *ED* should be an estimate of the relative error of the computed value *FD* (derivative of the polynomial).

The value of *F* or *FD* can be zero, but they are not checked in the program. In addition the above statements are not in accordance with the formulas for the relative errors, which are given by Witte in the comments which precede the program.

A reasonable modification of Algorithm 332 is: (i) calculate absolute errors (instead of relative errors) if *F* or *FD* is close to zero (here is used $|F| < y$ or $|FD| < y$); (ii) otherwise assign the relative errors *E* and *ED* in accordance with the formulas $E = |1 - Q_n/P_n|$ and $ED = |1 - QD_n/PD_n|$; and (iii) add two flag variables *FLAGF* and *FLAGFD* indicating what kind of error (absolute, relative, or no error) is estimated. The variable *FLAGF* corresponds to the error *E* of *F*. The variable *FLAGFD* corresponds to the error *ED* of *FD*.

The two flag variables *FLAGF* and *FLAGFD* are assigned the values 0, 1 or 2:

If a relative estimate of the error is used, the flag is assigned the value 0. If an absolute estimate of the error is used, the flag is assigned the value 1. If *DEGREE* ≡ 0, both the errors are equal to zero, and the flags are assigned the value 2.
The following corrections should be made in the program:

The first statement in the subroutine should read:

```
 SUBROUTINE JACOBI
* (DEGREE,ALFA,BETA,X,F,FD,E,ED,FLAGF,FLAGFD)
```

The declaration of the integer variables:

```
INTEGER I,J,K,M,N,DEGREE
```

should read

```
INTEGER I,J,K,M,N,DEGREE,FLAGF,FLAGFD
```

The first *IF* in the program:

```
IF (DEGREE.EQ.0) GO TO 8
```

should read

```
IF (DEGREE.EQ.0) GO TO 10
```

The last section ("Prepare the output") should read:

```
C       PREPARE THE OUTPUT ..
      7 N     = DEGREE
        F     = P(N)
        IF (DABS(F).LT.Y)  GO TO 8
        FLAGF=0
        E     = DABS(1.-Q(N)/F)
        GO TO 9
      8 E     = DABS(F-Q(N))
        FLAGF=1
      9 FD    = PD(N)
        IF (DABS(FD).LT.Y)  GO TO 11
        FLAGFD=0
        ED    = DABS(1.-QD(N)/FD)
        GO TO 12
     10 F     = 1D0
        E     = 0.
        FD    = 0D0
        ED=0.
        FLAGF=2
        FLAGFD=2
        GO TO 12
     11 ED    =DABS(FD-QD(N))
        FLAGFD=1
     12 RETURN
        END
```

The value 3*E*-26 given to *y* in the *DATA* statement reflects, according to the author, the accuracy of the CDC-1604. The author gives no information how one can calculate this constant from the given computer parameters (radix, number of digits in the mantissa and information whether the machine is doing the chopping or rounding). The constant *y* must be some sort of "machine epsilon," e.g. the smallest number (provided by the implementation and the chosen precision) for which

$$1 + y > 1. \tag{1}$$

According to e.g. [4 pp. 7–9], we have

$$y = \begin{cases} \beta^{1-t} \textit{ chopping}, \\ \beta^{1-t}/2 \textit{ rounding}, \end{cases} \tag{2}$$

where β is the radix or base for the floating point numbers and t is the number of digits (with radix β) in the mantissa of the floating numbers. In [2] algorithms and corresponding programs (in Fortran) are published which for any "reasonable" floating point computer compute the radix, number of digits of used floating-point

numbers, and determine whether rounding or chopping is done by the machine, see also [5]. The CDC-1604 has according to e.g. [3] binary base, i.e. $\beta = 2$ with a normal word-length of 48 bit. The word is divided into an exponent with 12 bit and a mantissa or fraction with 36 bit. For the double-precision calculations the rounding CDC-1604 has therefore
$t = 36 + 48 = 84$, i.e. $y = \frac{1}{2} \times 2^{1-84} = 5.2 \times 10^{-26}$.

With these modifications Algorithm 332 ran successfully on an IBM 370/165 with operating system 21.6, and with the IBM Fortran IV G compiler. For double-precision calculations on this chopping computer we have: $\beta = 16, t = 14$, i.e. $y = 16^{-13} = 2.2 \times 10^{-16}$, see [1 p. 163].

References

1. International Business Machines. *IBM System/370 Principles of Operation.* IBM Syst. Order No. GA22-7000-3, IBM, White Plains, N.Y., 1973, xii+318.
2. Malcolm, M.A. Algorithms to reveal properties of floating-point arithmetic. *Comm. ACM 15* (Nov. 1972), 949–951.
3. Stroud, A.H., and Secrest, D. A multiple-precision floating-point interpretive program for the Control Data 1604. *Computer J. 6* (1963), 62–66.
4. Wilkinson, J.H. *Rounding Errors in Algebraic Processes.* Her Majesty's Stationery Office, London, and Prentice-Hall, Englewood Cliffs, N.J., 1963, vi+161.
5. Gentleman, W.M., and Marovich, S.B. More on algorithms that reveal properties of floating point arithmetic units. *Comm. ACM 17*, 5 (May 1974), 276–277.

ALGORITHM 333
MINIT ALGORITHM FOR LINEAR PROGRAMMING [H]

RODOLFO C. SALAZAR AND SUBRATA K. SEN

(Recd. 26 June 1967 and 25 Jan. 1968)

Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, Penna. 15213

KEY WORDS AND PHRASES: linear programming, dual simplex method, primal problem, dual problem
CR CATEGORIES: 5.41

```
real procedure MINIT(m, n, p, e, td);
  integer m, n, p;  array e;  real td;
comment  MINIT(MINimum ITerations) is designed to solve a
```

linear programming problem of n variables and m constraints of which the last p are equality constraints. The problem can be stated as follows:

$$\text{Maximize } z = cX$$
$$\text{subject to } AX \leqq b$$
$$X \geqq 0$$

c is a $(1\times n)$ row vector, X is a $(n\times1)$ column vector, A is a $(m\times n)$ matrix, and b is a $(m\times1)$ column vector. e is a matrix with $(m+1)$ rows and *lcol* columns (where $lcol=m+n-p+1$) and forms the initial tableau of the algorithm:

	1 · · · · · · · · · · · · · · · · · ·		*lcol*	
1	$-c\ (1 \times n)$	$0\ (1 \times m - p)$	0	
·	$A\ (m \times n)$	Identity Matrix $(m - p \times m - p)$	b	$(m - p)$ inequality constraints
·		Zero Matrix $(p \times m - p)$	$(m \times 1)$	p equality constraints
$m + 1$				

td is read into the procedure and should be a very small number, e.g. 10^{-8}. The condition of optimality is the nonnegativity of $e[1, j]$ for $j = 1, \cdots, lcol$-1 and of $e[i, lcol]$ for $i = 2, \cdots, m + 1$. If the $e[i, j]$ values are greater than or equal to $-td$ they are considered to be nonnegative. The value of *td* should reflect the relative magnitude of the coefficient matrix.

It should be noted that when equality constraints are present, the dual solution vector is not complete, i.e. the procedure does not compute the values of the dual variables corresponding to the equality constraints. However, knowing the optimal solution to the primal problem and the values of the dual variables corresponding to the inequality constraints, it is a simple matter to compute the values of the remaining dual variables. In the initial tableau, the elements of the vector b must be nonnegative for the equality constraints.

MINIT is based upon a technique suggested by Llewellyn [1] and is a specialized algorithm based on the principle of the dual simplex method. Llewellyn states that he has found the *MINIT* algorithm to be more efficient than any other method he has used. *MINIT*'s efficiency is based upon the fact that the solution method confines the iterations to those constraints which are defining (equality constraints and those inequality constraints whose slack variables are zero in the optimal solution). The algorithm starts with an infeasible solution as in the dual simplex method. When "greater than or equal to" constraints are involved, it also starts with an incomplete solution since it avoids the use of artificial variables. This feature of the algorithm considerably reduces the number of iterations required to obtain the optimal solution. Both the primal and dual problems are solved simultaneously and the pivotal element at each iteration is so chosen that there is a maximum increase in the functional value of the primal or a maximum decrease in the functional value of the dual. The details of the algorithm and a discussion of the theoretical reasons for its computational efficiency may be obtained from the reference cited below.

The experience of the authors with the *MINIT* algorithm has been very satisfactory. For example, on a CDC G-21 computer, the Simplex code available in the Carnegie Tech. program library took 4 minutes 56 seconds to solve a 51 × 72 linear programming problem (consisting only of inequality constraints) while the same problem was solved in 2 minutes 58 seconds by the *MINIT* algorithm. For problems with mixed constraints, i.e. equality and inequality constraints, the advantage of the *MINIT* algorithm is even more pronounced.

REFERENCE:

1. LLEWELLYN, R. W. *Linear Programming.* Holt, Rinehart and Winston, New York, 1964, pp. 207–220;

```
begin integer i, j, k, L, im, jmin, jm, imax;  real gmin, phimax;
  integer array ind[1:lcol], ind1[1:m+1], chk[2:m+1];
  procedure results;
  comment  prints out the output. The value of the functional
    is given by z. The optimal values of the variables are given by
    x[i] for i = 1, ··· , n and the values of the dual variables are
    given by w[j] for j = 1, ··· , m;
  begin real z;  array x[1:n], w[1:m];
    z :=e[1:lcol];
    for i := 1 step 1 until n do x[i] := 0;
    for j := 1 step 1 until m do w[j] := 0;
    for i := 2 step 1 until m + 1 do
    begin
      if chk[i] > n then chk[i] := 0;
      if chk[i] > 0 then x[chk[i]] := e[i, lcol]
    end;
    for j := n + 1 step 1 until lcol-1 do w[j−n] := e[1, j];
    comment  Insert output statements to print out z, x[i],
      and w[j], for example, the following six statements sepa-
      rated by semicolons: (1) outstring (1, 'value of the func-
      tional'), (2) outreal (1, z), (3) outstring (1, 'optimal values
      of the variables'), (4) outarray (1,x), (5) outstring (1, 'values
      of the dual variables'), (6) outarray (1, w);
    go to LAST
  end results;
  procedure rowtrans(im, jmin);
    integer im, jmin;
  comment  performs the usual tableau transformations in a
    linear programming problem, (im, jmin) being the pivotal
    element;
  begin real dummy;
    if im = 0 then
    begin
      comment Insert an output statement to print "no solu-
        tion", for example, the statement, outstring (1, 'no solu-
        tion');
      go to LAST
    end;
    if jmin = 0 then
    begin
      comment Insert an output statement to print "no solu-
        tion", for example, the statement, outstring (1, 'no solu-
        tion');
      go to LAST
    end;
    dummy := e[im, jmin];
    for j := 1 step 1 until lcol do e[im, j] := e[im, j]/dummy;
    for i := 1 step 1 until m + 1 do
    begin
      if i ≠ im then
      begin
        if e[i, jmin] ≠ 0 then
        begin
          dummy := e[i, jmin];
          for j := 1 step 1 until lcol do
            e[i, j] := e[i, j] − e[im, j] × dummy
        end
      end
    end;
    chk[im] := jmin
  end rowtrans;
  procedure progamma;
  comment performs calculations over columns to determine
    the pivot element;
  begin integer i, L1; real theta, gamma; array thmin[1:lcol];
    integer array imin[1:lcol];
    gmin := $10^6$; jmin := 0;
    comment gmin is set equal to a large number for initializa-
      tion purposes;
    for L1 := 1 step 1 until L − 1 do
    begin
      imin[ind[L1]] := 0; thmin[ind[L1]] := $10^6$;
      for i := 2 step 1 until m + 1 do
      begin
        if e[i, ind[L1]] > td ∧ e[i, lcol] ≥ -td then
        begin
          theta := e[i, lcol]/e[i, ind[L1]];
          if theta < thmin[ind[L1]] then
          begin
            thmin[ind[L1]] := theta; imin[ind[L1]] := i
          end
        end
      end;
      if thmin[ind[L1]] = $10^6$ then gamma := $10^8$
      else gamma := thmin[ind[L1]] × e[1, ind[L1]];
      if gamma < gmin then
      begin
        gmin := gamma; jmin := ind[L1]
      end
    end;
    if jmin > 0 then im := imin[jmin]
  end progamma;
  procedure prophi;
  comment performs calculations over rows to determine the
    pivot element;
  begin integer j, k1; real delta, phi; array delmax[1:m+1];
    integer array jmax[1:m+1];
    phimax := − $10^6$; imax := 0;
    comment phimax is set equal to a small number for initial-
      ization purposes;
```

```
    for k1 := 1 step 1 until k − 1 do
    begin
      jmax[ind1 [k1]] := 0; delmax[ind1 [k1]] := − $10^6$;
      for j := 1 step 1 until lcol − 1 do
      begin
        if e[ind1[k1], j] < −td ∧ e[1, j] ⩾ -td then
        begin
          delta := e[1, j]/e[ind1[k1], j];
          if delta > delmax[ind1[k1]] then
          begin
            delmax[ind1[k1]] := delta; jmax[ind1[k1]] := j
          end
        end
      end;
      if delmax[ind1[k1]] = − $10^6$ then phi := − $10^8$
      else phi := delmax[ind1[k1]] × e[ind1[k1], lcol];
      if phi > phimax then
      begin
        phimax := phi; imax := ind1[k1]
      end
    end;
    if imax > 0 then jm := jmax[imax]
  end prophi;
  procedure phase1;
  comment applied only to equality constraints if any;
  begin integer r; real theta, gamma; array thmin[1:lcol];
    integer array imin[1:lcol];
    for r := 1 step 1 until p do
    begin
      gmin := $10^6$; L := 1;
      comment gmin is set equal to a large number for initial-
        ization purposes;
      for j := 1 step 1 until n do
      begin
        thmin[j] := $10^6$; if e[1, j] < 0 then
          begin
            ind[L] := j; L := L + 1
          end
      end;
      if L = 1 then
      begin
        for j := 1 step 1 until n do ind[j] := j; L := n + 1
      end;
      for k := 1 step 1 until L − 1 do
      begin
        for i := m − p + 2 step 1 until m + 1 do
        begin
          if chk[i] = 0 then
          begin
            if e[i, ind[k]] > 0 then
            begin
              theta := e[i, lcol]/e[i, ind[k]];
              if theta < thmin[ind[k]] then
              begin
                thmin[ind[k]] := theta; imin[ind[k]] := i
              end
            end
          end
        end;
        gamma := thmin[ind[k]] × e[1, ind[k]];
        if gamma < gmin then
        begin
          gmin := gamma; jmin := ind[k]
        end
      end;
      im := imin[jmin]; rowtrans(im, jmin)
    end
  end phase1;
  for i := 2 step 1 until m + 1 do chk[i] := 0;
  if p = 0 then go to RCS else phase1;
  comment If there are any equality constraints in the problem
    the program first goes to phase1, otherwise it goes directly to
    RCS;
RCS: L := 1; k := 1;
  for j := 1 step 1 until lcol − 1 do
  begin
    if e[1, j] < − td then
    begin
      ind[L] := j; L := L + 1;
      comment ind[L] keeps track of the columns in which e[1, j]
        is negative;
    end
  end;
  for i := 2 step 1 until m + 1 do
  begin
    if e[i, lcol] < − td then
    begin
      ind1[k] := i; k := k + 1;
      comment ind1[k] keeps track of the rows in which e[i, lcol]
        is negative;
    end
  end;
  if L = 1 then
  begin
    if k = 1 then results else
    begin
      if k = 2 then
      begin
        for j := 1 step 1 until lcol − 1 do
        begin
          if e[ind1[1], j] < 0 then go to R
        end;
        comment Insert an output statement to print "primal
          problem has no feasible solutions, dual objective func-
          tion is unbounded", for example, the statement out-
          string (1, 'primal problem has no feasible solutions, dual
          objective function is unbounded');
        go to LAST
      end else go to R
    end
end
else
begin
  if L = 2 then
  begin
    if k = 1 then
    begin
      for i := 2 step 1 until m + 1 do
      begin
        if e[i, ind[1]] > 0 then go to C
      end;
      comment Insert an output statement to print "primal
        objective function is unbounded, dual problem has no
        feasible solutions", for example, the statement out-
        string (1, 'primal objective function is unbounded, dual
          problem has no feasible solutions');
        go to LAST
      end else go to S
    end;
    if k = 1 then go to C else go to S
  end;
R: prophi; rowtrans(imax, jm); go to RCS;
C: progamma; rowtrans(im, jmin); go to RCS;
S: progamma; prophi;
  if gmin = $10^6$ then
  begin
    rowtrans(imax, jm); go to RCS
  end;
  if phimax = − $10^6$ then
  begin
    rowtrans(im, jmin); go to RCS;
  end;
  if abs(phimax) > abs(gmin) then rowtrans(imax, jm)
  else rowtrans(im, jmin);
  go to RCS;
LAST: end MINIT
```

REMARK ON ALGORITHM 333 [H] MINIT ALGORITHM FOR LINEAR PROGRAMMING [Rodolfo C. Salazar and Subrata K. Sen, *Comm. ACM 11* (June 1968), 437]

D. K. MESSHAM (Recd. 27 Nov. 1968 and 28 Feb. 1969)
Nelson Research Laboratories, The English Electric Co. Ltd., Stafford, England

KEY WORDS AND PHRASES: linear programming, dual simplex method, primal problem, dual problem
CR CATEGORIES: 5.41

The procedure has been tested with Marconi Myriad Algol, and it ran successfully when the following changes had been made (the first is merely a misprint):

1. The first statement in procedure *results* was changed
   from z := *e*[1 : *lcol*];
   to z := *e*[1, *lcol*];

2. To satisfy an ALGOL 60 restriction that a type procedure should contain an assignment to its procedure identifier, the **real** on the first line of the procedure was removed.

3. It is possible for the published algorithm to give incorrect results when it reaches a state in *phase*1 where there are no possible pivotal elements in one column of the tableau. (For example, maximize $- x_1 - x_2 - x_3$, with $2x_1 + x_2 = 3$ and $x_3 = 1$, reaches this state.) To correct this the line in procedure *phase*1

```
  if gamma < gmin then
```

was changed to

```
  if gamma < gmin ∧ thmin [ind[k]] < $_{10}6$ then
```

All the appearances of 10^6 in this algorithm should be written as $_{10}6$.

The following improvements are also suggested:

4. It is assumed that *lcol* is a global integer with the correct value. This was made unnecessary by adding *lcol* to the list of integers declared on the line immediately following the initial comment; the bounds of the array *ind*, declared on the next line, were changed

```
  from  [1 : lcol]
  to    [1 : m+n−p+1];
  and   lcol := m + n − p + 1;
```

was inserted as the first executable statement of the procedure *MINIT* (after **end** *phase*1;).

5. It is assumed that equality constraints will be given with positive right-hand sides. This restriction was overcome by inserting in the procedure *phase*1 after the line **integer array** *imin* [1 : *lcol*]; the following:

```
  for i := m − p + 2 step 1 until m + 1 do
  if e[i, lcol] < 0 then
  for j := 1 step 1 until lcol do e[i, j] := − e[i, j];
```

REMARK ON ALGORITHM 333 [H] MINIT ALGORITHM FOR LINEAR PROGRAMMING [Rodolfo C. Salazar and Subrata K. Sen, *Comm. ACM 11* (June 1968), 437–440]

Å. Kolm and T. Dahlstrand (Recd. 15 Sept. 1969)
Information Processing Department, ASEA. S-721 83 Västerås, Sweden

KEY WORDS AND PHRASES: linear programming, dual simplex method, primal problem, dual problem
CR CATEGORIES: 5.41

When we tried to run the program on a GE-625 computer it became apparent that the following, rather obvious changes in the procedure *phase*1 of the original program are necessary:

1. The statement after the statement *thmin*[*j*] := 10^6; should begin with

```
if e[1,j] < − td then ···
```

2. The beginning of the statement following the statement **if** *chk*[*i*] = 0 **then** should be replaced by

```
begin
    if e[i,ind[k]] > td then
```

3. The statement *gamma* := *thmin* [*ind*[*k*]] × *e*[1,*ind*[*k*]]; should be preceded by

```
if thmin[ind[k]] = 10^6 then gamma := 10^8 else
```

We also suggest that the parameters *m*, *n*, *p*, *td* of the procedure should be **value**-specified.

After these corrections the procedure has been successfully tested in several problems. For problems of moderate size, which without further modifications can be solved by the procedure, the algorithm turned out to be most efficient. The numerical accuracy was also good.

Remark on Algorithm 333 [H] Minit Algorithm for Linear Programming [Rodolfo C. Salazar and Subrata K. Sen, *Comm. ACM 11* (June 1968), 437–440]

D. Obradović*
Boris Kidrič Institute of Nuclear Sciences,
11001 Beograd, Yugoslavia
* Present address: Institute of Investment Research, 1100 Beograd, Yugoslavia.

The procedure has been tested with CDC 3600 Algol, and it ran successfully when the following changes had been made in the procedure phase 1:

1. After the line

```
comment applied only to equality constraints if any;
```

instead

```
begin integer r;
```

one has to introduce

```
begin integer r, im1, jmin 1;
```

2. After the line

```
integer array imin[1:lcol];
```

one has to introduce a new line

```
im 1 := jmin 1 := 0;
```

3. The line

```
if L = 1 then
```

should read

```
L1: if L = 1 then
```

4. Last nine lines of the procedure phase 1 should be changed to read

```
if thmin[ind[k]] < 10^6 then
begin
gamma := thmin[ind[k]] × e[1, ind[k]];
if gamma < gmin then
begin
gmin := gamma; jmin := ind[k];
end
end;
end;
im := imin[jmin];
if im = im1 ∧ jmin = jmin1 then
begin
L := 1;   go to L1
end;
rowtrans(im, jmin);
im1 := im;  jmin1 := jmin;
end
end phase 1;
```

These changes are necessary to avoid incorrect results in the case if after application of the procedure *rowtrans* all *e*[*i*, *ind*[*k*]] are negative as in the following example

$$z = -0.9\,x_1 - 1.255632\,x_2 + 0.925\,x_3 + 0.375\,x_4$$
$$x_i < 2, \quad i = 1, 2, 3, 4$$
$$2.19069\,x_1 - 0.925\,x_2 - 0.325\,x_3 - 0.1875\,x_4 = 0.76569$$
$$x_1 \qquad - 0.1 \quad x_3 + 0.740896 x_4 = 1.640896$$

when the published algorithm ignores some of the equality constraints.

Remark on Algorithm 333 [H] Minit Algorithm for Linear Programming [Rodolfo C. Salazar and Subrata K. Sen, *Comm. ACM 11* (June 1968), 437–440]

B. Holmgren,* D. Obradović,† and Å. Kolm*
[Recd. 13 May 1971]
* Information Processing Department, ASEA
S-721 83 Västerås Sweden
† Boris Kidrič Institute of Nuclear Sciences,
11001 Beograd, Yugoslavia

In addition to previously given remarks on the algorithm, the following changes in the procedure phase 1 are necessary in order to avoid incorrect results for some types of problems with equality constraints:

1. Introduce into phase 1 the variable *first* by the declaration

```
Boolean first;
```

2. After the statement *L* := 1; one has to set

```
jmin := 0; first := true;
```

3. The statement **if** *L* = 1 **then** . . . should be replaced by

```
L1: if L = 1 then . . .
```

4. The statement *im* := *imin* [*jmin*]; should be preceded by

```
    if jmin = 0 then
begin
    if first then
  begin
    first := false;  L := 1;  go to L1
  end else im := 0
end else
```

After these changes *MINIT* can handle problems, for which equality constraints cause all the current values of *e*[*i*, *ind*[*k*]] to be negative at some stage in phase 1. For such cases the variables *im* and *jmin* in the old version either were left undefined or remained unchanged before entering the procedure *rowtrans*. An example of this is the trivial problem

$max\ x_1$, when
$x_1, x_2 \geq 0$,
$x_1 \leq 1$,
$x_2 = 1$,

where the original procedure completely failed.

ALGORITHM 334
NORMAL RANDOM DEVIATES [G5]

JAMES R. BELL (Recd. 13 Dec. 1965, 29 Nov. 1967, and 23 Jan. 1968)
Stanford Research Institute, Menlo Park, Calif.

KEY WORDS AND PHRASES: normal deviates, normal distribution, random number, random number generator, simulation, probability distribution, frequency distribution, random
CR CATEGORIES: 5.5, 5.13

```
procedure norm (D1, D2);
  real D1, D2;
comment This procedure generates pairs of independent
  normal random deviates with mean zero and standard deviation
  one. The output parameters D1 and D2 are normally distributed
  on the interval (− ∞, + ∞). The method is exact even in the
  tails.
    This algorithm is one of a class of normal deviate generators,
  which we shall call "chi-squared projections" [1, 2]. An al-
  gorithm of this class has two stages. The first stage selects a
  random number L from a $\chi_2^2$-distribution. The second stage
  calculates the sine and cosine of a random angle $\theta$. The generated
  normal deviates are given by L sin ($\theta$) and L cos ($\theta$).
    The two stages can be altered independently. In particular,
  as better $\chi_2^2$ random generators are developed, they can replace
  the first stage. (The negative exponential distribution is the
  same as that of $\chi_2^2$.)
    The fastest exact method previously published is Algorithm
  267 [4], which includes a comparison with earlier algorithms.
  It is a straight chi-squared projection. Our algorithm differs
  from it by using von Neumann rejection to generate sin ($\phi$) and
  cos ($\phi$), [$\phi = 2\theta$], without generating $\phi$ explicitly [3]. This
  significantly enhances speed by eliminating the calls to the
  sin and cos functions.
    The author wishes to express his gratitude to Professor
  George Forsythe for his help in developing the algorithm.
    REFERENCES
1. BOX, G., AND MULLER, M. A note on the generation of normal
   deviates. Ann. Math. Stat. 28, (1958), 610.
2. MULLER, M. E. A comparison of methods for generating
   normal deviates on digital computers. J. ACM, 6 (July
   1959), 376–383.
3. VON NEUMANN, J. Various techniques used in connection with
   random digits. In Nat. Bur. of Standards Appl. Math. Ser.
   12, 1959, p. 36.
4. PIKE, M. C. Algorithm 267, Random Normal Deviate.
   Comm. ACM, 8 (Oct. 1965), 606.;
comment R is any parameterless procedure returning a random
  number uniformly distributed on the interval from zero to one.
  A suitable procedure is given by Algorithm 266, Pseudo-Random
  Numbers [Comm. ACM, 8 (Oct. 1965), 605] if one chooses
  a = 0, b = 1, and initializes y to some large odd number, such
  as y = 13421773.;
begin
  real X, Y, XX, YY, S, L;
  comment von Neumann rejection for choosing a random
    angle $\phi = 2\theta$, $\theta = \tan^{-1} (Y/X)$;
A: X := R; Y := 2 × R − 1;
  XX := X ↑ 2; YY := Y ↑ 2;
  S := XX + YY;
  if S > 1 then go to A;
  comment chooses L randomly from a $\chi_2^2$-distribution and
    normalizes with S;
L := sqrt (−2×ln(R))/S;
  comment computes deviates as L × sin ($\phi$) and L × cos ($\phi$);
D1 := (XX−YY) × L;
D2 := 2 × X × Y × L;
end norm;
```

REMARK ON ALGORITHM 334 [G5]
NORMAL RANDOM DEVIATES [James R. Bell, *Comm. ACM 11* (July 1968), 498]

R. KNOP* (Recd. 5 Aug. 1968 and 8 Nov. 1968)
Physics Dept., University of Maryland, College Park, MD 20742

This work was supported in part by an Atomic Energy Commission contract.

* Present address: Physics Dept., Rutgers University, New Brunswick, NJ 08903

KEY WORDS AND PHRASES: normal deviates, normal distribution, random number, random number generator, simulation, probability distribution, frequency distribution, random
CR CATEGORIES: 5.13, 5.5

Algorithm 334 produces pairs of normally distributed random deviates with zero mean and unit variance by the method of Box and Muller [1]. The sine and cosine required by the Box-Muller method are calculated by the von Neumann rejection technique [2]. This technique allows the calculation of the sine and cosine of an angle uniformly distributed over the interval $(0, 2\pi)$ without referencing the sine, cosine, or square root functions. We note however, that Algorithm 334 require as square root calculation in inverting the distribution function of the radius (equal to $L \times S$ in the notation of the algorithm).

We suggest that since the square root calculation seems unavoidable, it can be used to obtain the required sine and cosine by more conventional means. Thus we propose sampling points from a density uniform over the unit disk in the X, Y-plane and calculating the sine and cosine from their definition in terms of the legs and hypotenuse of a right triangle. The following changes in Algorithm 334 are then necessary:

a. Replace $X := R$ by $X := 2 \times R - 1$
b. Replace $L := sqrt(-2 \times ln(R))/S$ by
$L := sqrt(-2 \times ln(R)/S)$
c. Replace $D1 := (XX - YY) \times L$ by $D1 := X \times L$
d. Replace $D2 := 2 \times X \times Y \times L$ by $D2 := Y \times L$

Acknowledgment. The author thanks B. Kehoe for comments concerning this algorithm.

REFERENCES:

1. BOX, G., AND MULLER, M. A note on the generation of normal deviates. *Ann. Math. Stat. 28* (1958), 610.
2. VON NEUMANN, J. Various techniques used in connection with random digits. In Nat. Bur. Standards Appl. Math. Ser. 12, US Govt. Printing Off., Washington, D. C., 1959, p. 36.

REMARK ON ALGORITHM 334

Normal Random Deviates
[James R. Bell (with modifications due to R. Knop), *Commun. ACM 12*, 5 (May 1969), 281.]

Allen E. Tracht [Received 12 December 1981; revised 16 December 1981; accepted 16 December 1981]
Biomedical Engineering Department, Case Western Reserve University, Cleveland, OH 44106.

As modified by Knop, Algorithm 334 produces pairs of normally distributed random deviates with zero mean and unit variance by a modification of the "polar" method due to Box, Muller, and Marsaglia [2]. The following change converts Algorithm 334, as modified by Knop, to the "polar" method:

Replace: $L := sqrt(-2 \times ln(R)/S)$
by$L := sqrt(-2 \times ln(S)/S)$.

Note that this modification eliminates one invocation of the uniform random number generator R. Using timing information given by Brent [1] in Algorithm 488, the "polar" method would be expected to take $(83 + 1.27U)$ microseconds rather than $(83 + 1.77U)$ microseconds per call. This is faster than the $(91 + 1.38U)$ microseconds given by Brent for Algorithm 488.

REFERENCES

1. Brent, R.P. Algorithm 488. A Gaussian pseudorandom number generator, *Collected Algorithms of the ACM.*, Vol. 2, ACM, New York, 1978.
2. Knuth, D.E. *The Art of Computer Programming*, Vol. 2, Addison-Wesley, Reading, Mass., 1981, pp. 117–118 or 1969, pp. 104–105.

ALGORITHM 334
NORMAL RANDOM DEVIATES [G5]

James R. Bell (Recd. 13 Dec. 1965, 29 Nov. 1967, and 23 Jan. 1968)
Stanford Research Institute, Menlo Park, Calif.

KEY WORDS AND PHRASES: normal deviates, normal distribution, random number, random number generator, simulation, probability distribution, frequency distribution, random
CR CATEGORIES: 5.5, 5.13

```
procedure norm (D1, D2);
  real D1, D2;
```

comment This procedure generates pairs of independent normal random deviates with mean zero and standard deviation one. The output parameters $D1$ and $D2$ are normally distributed on the interval $(-\infty, +\infty)$. The method is exact even in the tails.

This algorithm is one of a class of normal deviate generators, which we shall call "chi-squared projections" [1, 2]. An algorithm of this class has two stages. The first stage selects a random number L from a $\chi_2{}^2$-distribution. The second stage calculates the sine and cosine of a random angle θ. The generated normal deviates are given by $L \sin(\theta)$ and $L \cos(\theta)$.

The two stages can be altered independently. In particular, as better $\chi_2{}^2$ random generators are developed, they can replace the first stage. (The negative exponential distribution is the same as that of $\chi_2{}^2$.)

The fastest exact method previously published is Algorithm 267 [4], which includes a comparison with earlier algorithms. It is a straight chi-squared projection. Our algorithm differs from it by using von Neumann rejection to generate $\sin(\phi)$ and $\cos(\phi)$, $[\phi = 2\theta]$, without generating ϕ explicitly [3]. This significantly enhances speed by eliminating the calls to the sin and cos functions.

The author wishes to express his gratitude to Professor George Forsythe for his help in developing the algorithm.

References

1. Box, G., and Muller, M. A note on the generation of normal deviates. *Ann. Math. Stat. 28*, (1958), 610.
2. Muller, M. E. A comparison of methods for generating normal deviates on digital computers. *J. ACM, 6* (July 1959), 376–383.
3. von Neumann, J. Various techniques used in connection with random digits. In Nat. Bur. of Standards Appl. Math. Ser. 12, 1959, p. 36.
4. Pike, M. C. Algorithm 267, Random Normal Deviate. *Comm. ACM, 8* (Oct. 1965), 606.;

comment R is any parameterless procedure returning a random number uniformly distributed on the interval from zero to one. A suitable procedure is given by Algorithm 266, Pseudo-Random Numbers [*Comm. ACM, 8* (Oct. 1965), 605] if one chooses $a = 0$, $b = 1$, and initializes y to some large odd number, such as $y = 13421773.$;

```
begin
  real X, Y, XX, YY, S, L;
  comment von Neumann rejection for choosing a random
     angle φ = 2θ, θ = tan⁻¹ (Y/X);
A: X := R;  Y := 2 × R − 1;
  XX := X ↑ 2;  YY := Y ↑ 2;
  S := XX + YY;
  if  S > 1 then go to A;
  comment chooses L randomly from a χ₂²-distribution and
     normalizes with S;
L := sqrt (−2×ln(R))/S;
  comment computes deviates as L × sin (φ) and L × cos (φ);
D1 := (XX−YY) × L;
D2 := 2 × X × Y × L;
end norm;
```

REMARK ON ALGORITHM 334 [G5]
NORMAL RANDOM DEVIATES [James R. Bell, *Comm. ACM 11* (July 1968), 498]

R. Knop* (Recd. 5 Aug. 1968 and 8 Nov. 1968)
Physics Dept., University of Maryland, College Park, MD 20742

This work was supported in part by an Atomic Energy Commission contract.

* Present address: Physics Dept., Rutgers University, New Brunswick, NJ 08903

KEY WORDS AND PHRASES: normal deviates, normal distribution, random number, random number generator, simulation, probability distribution, frequency distribution, random
CR CATEGORIES: 5.13, 5.5

Algorithm 334 produces pairs of normally distributed random deviates with zero mean and unit variance by the method of Box and Muller [1]. The sine and cosine required by the Box-Muller method are calculated by the von Neumann rejection technique [2]. This technique allows the calculation of the sine and cosine of an angle uniformly distributed over the interval $(0, 2\pi)$ without referencing the sine, cosine, or square root functions. We note however, that Algorithm 334 require as square root calculation in inverting the distribution function of the radius (equal to $L \times S$ in the notation of the algorithm).

We suggest that since the square root calculation seems unavoidable, it can be used to obtain the required sine and cosine by more conventional means. Thus we propose sampling points from a density uniform over the unit disk in the X, Y-plane and calculating the sine and cosine from their definition in terms of the legs and hypotenuse of a right triangle. The following changes in Algorithm 334 are then necessary:

a. Replace $X := R$ by $X := 2 \times R - 1$
b. Replace $L := sqrt(-2 \times ln(R))/S$ by
   $L := sqrt(-2 \times ln(R)/S)$
c. Replace $D1 := (XX-YY) \times L$ by $D1 := X \times L$
d. Replace $D2 := 2 \times X \times Y \times L$ by $D2 := Y \times L$

Acknowledgment. The author thanks B. Kehoe for comments concerning this algorithm.

References:

1. Box, G., and Muller, M. A note on the generation of normal deviates. *Ann. Math. Stat. 28* (1958), 610.
2. Von Neumann, J. Various techniques used in connection with random digits. In Nat. Bur. Standards Appl. Math. Ser. 12, US Govt. Printing Off., Washington, D. C., 1959, p. 36.

ALGORITHM 335
A SET OF BASIC INPUT-OUTPUT PROCEDURES [I5]

R. De Vogelaere (Recd. 8 Sept. 1966 and 18 Nov. 1966; description revised 2 Nov. 1967)
Department of Mathematics and Computer Center, University of California, Berkeley, CA. 94720

By means of the primitives *insymbol, outsymbol* and *length,* as requested by this journal's Algorithms Policy [Comm. ACM 10 (Nov. 67), 729] a basic set of input-output procedures is defined aiming at quality and flexibility. *outreal,* for instance, is written as a derived procedure; it outputs using the fixed point or the floating point representation, and rounds properly. Variants can easily be written because of the explicit call of the procedures *decompose integer* and *decompose real.* The highly recommended practice of echoing input is made easy with one subset of derived procedures (*ioi, ior, iob, ioa*). The documentation of output in the form of equivalent ALGOL statements is also provided when use is made of the subset *oti, otr, otb, ota.* The Berkeley style of providing information on the form of output using prior calls of procedures such as *real format* is defined. A use of the parameter *outchannel* to provide information for simultaneous output to several channels is suggested. Interrelationship between the declared procedures is furnished in tabular form.

KEY WORDS AND PHRASES: input output, transput, input output procedures, input echo, quality output, decompose integer, decompose real, style, Berkeley style, procedures relationship, output documentation, equivalent ALGOL statements, ALGOL, ALGOL 60, integer format, real format, out integer, read real, input output Boolean, input output array, fixed point representation, floating point representation, output channel interpretation
CR CATEGORIES: 4.0, 4.41

1. Introduction

The reader will find below a set of basic input-output procedures. Let me state first some of the purposes for writing this set and give a general description and specific information about the procedures and their interrelationship.

In the October 1964 issue of the *Communications of the ACM* [1], a report on input-output procedures for Algol 60 was published. This report was prepared by a working group (WG 2.1) of the International Federation for Information Processing (IFIP/TC2) and approved by its Council.

The approved primitives were:

insymbol, outsymbol, length, inreal, outreal, inarray, outarray

In the examples the following derived procedures were defined:

outboolean, outstring, ininteger.

It is stated therein that "one needs, in practice, a fuller set of input-output procedures" and it is observed also that "different scheme of I/O procedures can be defined in it, largely by means of these primitives."

Since then, a few procedures have been published (see for instance [2, 3]) and the Algorithms Policy of this journal has requested [6] the use of the primitives of [1] and the use of *outboolean, outstring, ininteger* and *outinteger* for input-output.

The purpose of this algorithm is to present part of a consistent scheme of input-output procedures. The set uses as primitives, *insymbol, outsymbol,* and *outstring* (or equivalently *length*).

First *in integer, out integer, in real, out real, in Boolean, out Boolean* are derived. *in real* is related to [2]; *out integer* and *out real* call the more basic procedures *decompose integer* and *decompose real. out real* allows not only for floating point representation [3] but also for fixed point representation and for correct rounding.

Several sets of procedures, which point in several directions and which call the more basic ones, are then introduced. One set consists of parameterless input function designators akin to the **procedure** *read* of the Amsterdam Mathematisch Centrum. One set provides for echo of input to insure that the correct numbers have been read in—a practice which I recommend highly; it also provides for easy documentation of the output in the form of equivalent Algol statements. Another set with the same documentation feature is for output only; the last set outputs numbers, but no text.

It is not suggested that the set of procedures of this algorithm be used for quantity output. Its main purpose is for quality output.

2. General Description

2.1. The only primitives used are *insymbol, outsymbol,* and *length* (through *outstring*). *insymbol* and *outsymbol* assume that the value -1 is associated with the symbol carriage return-line feed (or new card), which is not a basic symbol of Algol 60. This is done in accordance with the convention of [1, Sec. 3]. *outstring* could have been avoided with some loss of clarity in the description of the procedures. *insymbol, outsymbol,* and *outstring* are defined in [1].

inreal and *outreal* are defined as in [2, 3] in terms of *insymbol, outsymbol,* and *outstring.* I do not believe that *inreal* and *outreal* should be primitives, firstly, because these procedures can be defined in terms of other primitives, and secondly, because many definitions will satisfy the requirements of [1]. On the other hand, the requirements set forth in [1] are most desirable.

in channel and *out channel* must be declared as integers and assigned a value in accordance with the requirements of *insymbol* and *outsymbol* [1].

I would like to observe in passing that the integer *out channel* cannot only be interpreted as identifying a single channel, but can also be interpreted as identifying a set of channels to all of which the output is to be sent. (If the binary representation of *out channel* is $\sum a[i] \times 2 \uparrow i$, the output is sent to channel i if $a[i] = 1$ and is not sent if $a\,[i] = 0$.) Although this is not yet implemented at Berkeley in this fashion, all output going to a terminal is now also sent to the printer. When time-sharing becomes widespread this interpretation will, I hope, be increasingly popular.

2.2. The more basic input-output procedures are *in integer, in real,* and *in Boolean*; the first two use *in symbol* only through the integer procedure *symbol.*

symbol recognizes only the following basic symbols:

0|1|2|3|4|5|6|7|8|9|·|−|+|$_{10}$|,|␣

and carriage return-line feed (or new card).

in integer associates to the second parameter, which is of type integer, the next integer read from *channel* (the first parameter). Any number of consecutive spaces are ignored before the first digit; after the first digit, termination occurs with two consecutive spaces, a comma, or a carriage return-line feed. A comma before the first digit or sign, a period, ⟨$_{10}$⟩, or any other illegal symbol will call the procedure *error*.

in real associates to the second parameter, which is of type real, the next real number read from *channel* (the first parameter). Any number of consecutive spaces are ignored before the first digit, period, or ⟨$_{10}$⟩; after that, termination occurs with two consecutive spaces, a comma, or a carriage return-line feed. A comma before the first digit, sign, period, or ⟨$_{10}$⟩, or any other illegal symbol will call the procedure *error*. Communication between *in integer*, *in real*, and *in symbol* to take care of separation between integers or reals requires the nonlocals *z8100b* and *z8100bc*.

in Boolean associates to the second parameter, which is of type Boolean the next Boolean read from *channel* (the first parameter); any number of leading spaces or carriage returns-line feed are ignored; any illegal symbol will call the procedure *error*.

The procedure *error* has one parameter of type integer. It can be written according to the wishes of a user or of a group of users. An example with diagnostics in full is given below.

2.3. The more basic output procedures are *out integer*, *out real*, and *out Boolean.* The information on the form of the output can be given in various ways; the style used for these output procedures is what I will call the Berkeley style by contrast with the style used for output procedures at, for instance, the Amsterdam's Mathematisch Centrum or at Copenhagen's Regnecentralen. Call of these output procedures must be preceded by a call of corresponding procedures *integer format*, *real format* and *Boolean format*.

The only parameter of *integer format* determines the field width of any integer sent to the output channel. The parameters of *real format* are a Boolean, which determines when the value is **true** that fixed point representation is desired for the output of real numbers and when the value is **false** that floating point representation is desired. The second parameter determines the field width, the third parameter determines the number of decimal places and affects also the rounding of the number. The only parameter of *Boolean format* determines the field width.

The following decisions were made for *out integer*, *out real*, and *out Boolean:* If the field parameter is less than required, it is replaced by 20. The sign is outputed before the most significant digit if the number is negative. In floating point form, the first significant digit is immediately to the left of the decimal point. The exponent is replaced by four spaces if it is zero; otherwise the sign of the exponent is always outputed and the exponent is restricted to the interval −99 to 99.

If the user wishes to write variants of the Berkeley style, for instance if he wishes always to print the sign, or if he wishes to output it as the first character of the field, or if he wishes to output a space between every third or fifth digit, his task will be greatly eased by the introduction of the procedures *decompose integer* and *decompose real* which provide the basic information about an integer (its sign, the number of significant decimal digits, and the digits) or about a real (its sign, its size, the scale factor such that the scaled number has its first significant digit immediately to the left of the decimal point and the digits).

In *decompose real*, the size information determines if the number is too small; an integer declaration has been chosen instead of a Boolean to provide for the possibility of another test, which would determine if the number is too large. The rounding for reals is taken care of in *decompose real.*

Correct rounding is essential for a set of input-output procedures of quality. Although the point may be argued, I consider incorrect the output of 2 to two decimals as 1.99 unless computer or computations have only that precision. Examples:

real format (**true**, 5, 3);   *out real* (1, 0.99099);
real format (**false**, 10, 2);   *out real* (1, −0.99099);

will output

0.991-9.91$_{10}$ − 1.

2.4. Four more sets of input-output procedures follow; these procedures do not require explicit calls of the format procedures:

read i, *read r*, *read b* are function designators without parameters which can be used to input respectively an integer, a real or a Boolean.

ioi, *ior*, *iob* are function designators and *ioa* is a procedure to input respectively an integer, a real, a Boolean or a real array and to output an equivalent ALGOL statement.

This style, which I have introduced to give the output in the form of parts of an ALGOL program in connection with the generation of the nonlinear equations satisfied by Runge-Kutta type methods (to be published elsewhere), can also be used to describe input and output within the conventions of the ALGOL language.

For *ioi*, *ior*, *iob*, the second parameter gives the string to be outputted; the others give the parameters corresponding to those of the format procedures. For *ioa*, the second and third parameters are the first and last subscript of the element of the one dimensional array to be read and the last parameters give the string to be outputted as well as the format information. Examples:

ior(*r*, '*time*␣*in*␣*minutes*', **true**, 5, 2);
ioa(*a*, 1, 3, '*hippopotamus*', **true**, 4, 1)

would output with appropriate input:

time in minutes := 21.05;
i := 1; **for** *hippopotamus* [*i*] := 15.1, 6.2, 7.0 **do** *i* := *i* + 1;

The next four procedures *oti*, *otr*, *otb*, and *ota* are for output only; the form of output is identical to that of *ioi*, *ior*, *iob*, and *ioa*.

The last four procedures *outi*, *outr*, *outb*, and *outa* are for output only. They output an integer, a real, a Boolean, or a sequence of reals, the format information being provided by the parameters of these procedures.

3. Specific Information About Procedures, Their Relationship, and the Nonlocal Parameters

To ease the local exchange of procedures and nonlocal identifiers of procedures between people at Berkeley, conventions have been introduced which are examplified in the procedures of this algorithm. All appropriate nonlocal identifiers are formed using as first symbols the letter *z* followed by a digit associated to the writer (I use 8) followed by 3 digits corresponding to the number of the procedure in which the nonlocal identifier is first used (my procedure *symbol* is number 100, *in integer* is number 101, etc.) followed by an ordinary identifier.

The following declarations must be made in the same block as that of this algorithm or in an outer block:

integer *in channel*, *out channel*, *z8106n*, *z8107n*, *z8107d*, *z8108n*;
Boolean *z8100b*, *z8100bc*, *z8107B*;
procedure *in symbol* (*channel*, *string*, *destination*); (see *Comm. ACM 7* (Oct. 1964), 628–630)
procedure *out symbol* (*channel*, *string*, *destination*); (Idem)
procedure *out string* (*channel*, *string*); (Idem)

in channel and *out channel* must be assigned an appropriate value before a call of many of the input-output procedures (see Table I).

Table I indicates the relationship between the procedures and the nonlocal variables. Moreover, an explicit call of *out integer*, *out real*, and *out Boolean* requires a preceding call of the corresponding format procedure *integer formal*, *real format*, and *Boolean format.*

TABLE I. Relationship Between Procedures and Nonlocal Variables

File	Procedure	Number	*error*	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	*in channel*	*out channel*	file z8100						
																					z8100b	*z8100bc*	*z8106n*	*z8107n*	*z8107d*	*z8107B*	*z8108n*
z8096	*error*		0																								
	in symbol	97		0																							
	out symbol	98			0																						
	out string	99				0																					
z8100	*symbol*	100	×	×			0												×		×	×					
	in integer	101	×	+			×	0											×		×	×					
	in real	102	×	+			×		0										×		×	×					
	in Boolean	103	×	×						0																	
z8104	*decompose integer*	104									0																
	decompose real	105										0															
z8106	*integer format*	106											0										×				
	real format	107												0										×	×	×	
	Boolean format	108													0												×
z8106	*out integer*	109			×	×					×					0							×				
z8110	*out real*	110			×	×						×					0							×	×	×	
z8110	*out Boolean*	111			×	×												0									×
z8112	*read i*	112	+	+			+	×											×		+	+					
	read r	113	+	+					×										×		+	+					
	read b	114	+	+						×									×								
z8112	*ioi*	115	+	+	+	×	+	×			+		+			×			×	×	+	+	+				
	ior	116	+	+	+	×	+		×			+		+			×		×	×	+	+		+	+	+	
	iob	117	+	+	+	×				×					+			×	×	×							+
	ioa	118	+	+	+	×	+		×	×		+		+			×		×	×	+	+		+	+	+	
z8119	*oti*	119			+	×					+		×			×				×			+				
	otr	120			+	×						+		×						×				+	+	+	
	otb	121			+	×									×					×							+
	ota	122			+	×						+		×						×				+	+	+	
z8119	*outi*	123			+	+					+		×			×				×			+				
	outr	124			+	+						+		×			×			×				+	+	+	
	outb	125			+	+									×			×		×							+
	outa	126			+	+						+		×			×			×				+	+	+	

In Table I, each of the procedures is identified by a number. An × indicates that the procedure corresponding to the number in the same column or the nonlocal identifier on top of the same column is used explicitly (and perhaps also implicitly); + indicates that the corresponding procedure or identifier is used implicitly; 0 is placed in the column corresponding to the number of the procedure. Related procedures are grouped together in a file whose name appears in the first column. This information will be used in further publications.

The following declaration can be used for the **procedure** *error*:

```
procedure error (i);  value i;  integer i;
begin procedure nlcr;  outsymbol (channel, ‘ ’, −1);
      nlcr;
      if i = 8100 then out string (1,‘a␣symbol␣is␣read␣which␣is␣not␣a␣digit␣·␣,␣
        −␣+␣₁₀␣(space)␣carriage␣return-line␣feed’) else
      if i = 810100 then out string (1,‘while␣reading␣an␣integer,␣an␣illegal␣symbol␣
        is␣read␣before␣the␣first␣digit’) else
      if i = 810101 then out string (1,‘while␣reading␣an␣integer,␣an␣illegal␣symbol␣
        is␣read␣after␣the␣first␣digit’) else
      if i = 810200 then out string (1,‘while␣reading␣a␣real,␣an␣illegal␣symbol␣
        is␣read␣while␣reading␣the␣decimal␣fraction’) else
      if i = 810201 then out string (1,‘while␣reading␣a␣real,␣an␣illegal␣symbol␣is␣
        read␣before␣the␣first␣digit␣period␣or␣10’) else
      if i = 810202 then out string (1,‘while␣reading␣a␣real,␣an␣illegal␣symbol␣is␣
        read␣while␣reading␣the␣exponent␣part’) else
      if i = 810203 then out string (1,‘a␣real␣number␣is␣improperly␣terminated’)
        else
      out string (1,‘while␣reading␣a␣Boolean␣a␣symbol␣which␣is␣not␣true␣or␣false,
        is␣read␣before␣termination’);
      nlcr
end error
```

Acknowledgment. The implementation of the procedures in this paper has been made possible by the existence of an ALGOL interpreter, which is the responsibility of many (see [4]). The editor, Q.E.D., used to prepare the program on the SDS 930, has been planned and implemented by Peter Deutsch and Butler Lampson. I especially thank Mr. Deutsch for the inclusion of requested features to copy part of a line until a given character noninclusive and to delete part of a line until a given character noninclusive. I thank my colleague R. S. Lehman for the use of his syntax checker and transliterator to BC-ALGOL.

Machine time for the preparation and implementation of the procedures and their tests was furnished by Project Genie of the Computer Center operating under Contract SD-185 with the Advanced Research Project Agency and by the Berkeley Campus Committee on Research.

REFERENCES

1. Report on input-output procedures for ALGOL 60. *Comm. ACM 7* (Oct. 1964), 628–630.
2. McKEEMAN, W. M. Algorithm 239, Free Field Read. *Comm. ACM 7* (Aug. 1964), 481.
3. WIRTH, N. E. Algorithm 249, Outreal *n*. *Comm. ACM 8* (Feb. 1965), 104.
4. BC ALGOL Manual. U. of California, Computer Center, Berkeley, Oct. 1966 (Third Ed.).
5. ANGLUIN, D. C., DEUTSCH, L. P. Reference manual, Q.E.D., time-sharing editor. Doc. 30.60.30, Jan. 26, 1967, Contract SD-185, Office of the Secretary of Defense, ARPA, Washington, D. C.
6. Revised Algorithms Policy. *Comm. ACM 7* (Oct. 1964), 586.

```
integer procedure symbol(s);  integer s;
comment symbol := s := the integer representation of the
    next symbol read, 0 to 9 for the integers, 10 for ‘·’, 11 for
    ‘−’, 12 for ‘+’, 13 for ‘₁₀’, and 14 for ‘,’ or for carriage
    return (or new card) represented by −1 when processed by
    in symbol or for two consecutive spaces when the nonlocal
    Boolean z8100b is false. When z8100b is true any number of
    consecutive spaces are ignored. Any other symbol will call a
    nonlocal procedure error with parameter equal to 8100;
begin
read: in symbol(in channel, ‘0123456789.−+₁₀␣,’, s);
  if s = −1 ∧ z8100bc then go to read;
  if s = 15 then
  begin
    if z8100b then go to read
    else in symbol(in channel, ‘0123456789.−+₁₀␣,’, s)
  end;
  if s = −1 ∨ s = 16 then symbol := s := 14
  else
  begin if s ≦ 0 then error(8100);  symbol := s := s − 1 end
end symbol;
procedure in integer(channel, i);  value channel;
  integer channel, i;
comment i := the next integer read from channel, any number of
  consecutive spaces are ignored before the first digit, after the
  digit termination occurs with two consecutive spaces, a comma
  or a carriage return, any illegal symbol will call a nonlocal
  procedure error with parameter equal to 8100 or 810100 or
  810101;
begin
  integer s;  Boolean negative;
  negative := false;  z8100b := z8100bc := true;
    in channel := channel;
  symbol(i);  z8100bc := false;
  if i = 12 then symbol(i)
  else if i = 11 then begin negative := true;  symbol(i) end;
  if i ≧ 10 then error(810100);
  z8100b := false;
L1:  if symbol (s) < 10 then begin i := 1^ × i + s;  go to L1 end;
  if   s ≠ 14 then error(810101);
  if negative then i := −i
end in integer;
procedure in real(channel, r);  value channel;
  integer channel;  real r;
comment r := the next real number read from channel, any num-
  ber of consecutive spaces are ignored before the first digit.
  After the first digit termination occurs with two consecutive
  spaces, a comma or a carriage return. Any illegal symbol will
  call a non local procedure error with paramater equal to 8100
  or 810200 or 810201 or 810202 or 810203. The main differences
  with ALGORITHM 239 of W. M. McKeeman [2] are the substi-
  tution of his integer procedure CHAR by symbol, the introduc-
  tion of the Boolean z8100b, the introduction of a parameter in
  the nonlocal procedure error and the change of type of a few
  declarations;
begin
  real sig, fp, d, ep, ip;  integer esig, ch;
  real procedure unsigned integer;
  begin
    real u;
    u := ch;
K:  if symbol(ch) < 10 then begin u := u × 10 + ch;  go to K end;
      unsigned integer := u
  end unsigned integer;
  sig := 1.0;  ep := fp := 0;  z8100b := z8100bc := true;
  in channel := channel;
  symbol(ch);  z8100bc := false;
  if ch = 12 then symbol(ch)
  else if ch = 11 then begin sig := −1.0;  symbol(ch) end;
  z8100b := false;
  if ch ≦ 10 then
  begin
    ip := if ch < 10 then unsigned integer else 0;
    if ch = 10 then
    begin
      if symbol(ch) ≧ 10 then error(810200);
      fp := 0;  d := 0.1;
```

```
M:   fp := fp + ch × d;  d := d × 0.1;
        if symbol(ch) < 10 then go to M
      end decimal fraction
    end decimal number
    else if ch = 13 then ip := 1
    else begin error(810201);  ip := 1 end;
    if ch = 13 then
    begin esig := 1;
      if symbol(ch) = 12 then symbol(ch)
      else if ch = 11 then begin esig := −1;  symbol(ch) end;
      if ch < 10 then ep := unsigned integer × esig
      else begin error(810202);  ep := 0 end
    end exponent part;
    if ch ≠ 14 then error(810203);
    r := sig × (ip+fp) × 10.0 ↑ ep
  end in real;
  procedure in Boolean(channel, b);  value channel;
    integer channel;  Boolean b;
  comment b := the next Boolean read from channel, any number
    of spaces or carriage returns are ignored, any other symbol will
    call a nonlocal procedure error with parameter equal to 8103;
  begin
    integer i;
  L:in symbol(channel, 'true false␣', i);
    if i = 3 ∨ i = −1 then go to L;
    if i ≦ 0 then error(8103);
    b := i = 1
  end in Boolean;
  procedure decompose integer(i, negative, n of digits, digit);
    value i;  integer i, n of digits;  Boolean negative;
    integer array digit;
  comment negative := i < 0, n of digits := the number of decimal
    digits of i (if i = 0 then n of digits := 0), digit [0: n of digits − 1]
    := the decimal digits of i starting from the right;
  begin
    integer j;
    if i < 0 then begin negative := true;  i := −i end
    else negative := false;
    n of digits := 0;
  L:
    if i > 0 then
    begin
      j := i ÷ 10;  digit[n of digits] := i − j × 10;
      n of digits := n of digits + 1;  i := j;  go to L
    end
  end decompose integer;
  procedure decompose real(r, max n of digits, negative, size, exponent,
    digit);
    value r;  integer max n of digits, size, exponent;  real r;
    Boolean negative;  integer array digit;
  comment negative := r < 0, size := −1 if r is too small, i.e. is
    such that when abs(r) is multiplied repeatedly by 10 it does
    not become eventually larger than one, size := 0 otherwise,
    exponent := the power of 10 by which r is to be divided to ob-
    tain a number whose first significant digit is immediately to
    the left of the decimal point, digit [0: max n of digits − 1] :=
    the decimal digits of r starting with the first significant digit
    to the left;
  begin
    integer i, k, m;
    Boolean procedure too small(r);  real r;
      too small := abs(r) < 2 ↑ (−127);
    comment this procedure should be replaced appropriately;
    negative := false;
    if too small (r) then
    begin size := 1;  go to end decompose end
    else size := 0;
```

```
    if r < 0 then begin negative := true;  r := −r end;
    if r < 1 then
    begin
      exponent := −1;
  scale up: r := r × 10;
      if r < 1 then
      begin exponent := exponent − 1;  go to scale up end
    end
    else
    begin
      exponent := 0;
  test:
    if r ≧ 10 then
      begin exponent := exponent + 1;  r := r × 0.1;
        go to test end
    end;
    m := max n of digits;
    r := r + 5 × 0.1 ↑ m;
    i := entier(r);
    if i = 10 then
    begin
      i := 1;  exponent := exponent + 1;  m := m + 1;  r := r/10
    end
    else if i = 0 then i := 1;
    digit[0] := i;
    for k := 1 step 1 until m − 1 do
    begin
      r := (r−i) × 10;  i := entier(r);
      i := digit[k] := if i ≦ 0 then 0 else if i = 10 then 9 else i
    end;
  end decompose:
  end decompose real;
  procedure integer format(n);  integer n;  z8106n := n;
  procedure real format(B, n, d);  integer n, d;  Boolean B;
  begin
    z8107B := B;  z8107n := n;  z8107d := d
  end real format;
  procedure Boolean format(n);  integer n;  z8108n := n;
  procedure out integer(channel, i);  value channel, i;
    integer channel, i;
  comment the style of this procedure and of the out real and out
    Boolean procedures given below is what I will call the Berkeley
    style by contrast with that used for output procedures at the
    Amsterdam Mathematisch Centrum or at the Copenhagen
    Regnecentralen, for instance. It is characterized by the use of
    a field width parameter n and for real numbers, by the use of a
    parameter B which decides if the fixed point (value true)
    or the floating point representation (value false) is requested
    and by the number of digits d after the decimal point. The
    sign is outputed just before the most significant digit, if the
    number is negative. In floating point form the first significant
    digit is immediately to the left of the decimal point. If the
    field parameter is less than required, it is replaced by 20. These
    procedures pair with the corresponding input procedures if the
    field width is at least two units greater than required;
  begin
    integer n of digits, j, k;  Boolean negative;
    integer array digit[0: 19];
    decompose integer(i, negative, n of digits, digit);
    if n of digits = 0 then
    begin n of digits := 1;  digit[0] := 0 end;
    j := n of digits + (if negative then 1 else 0);
    for k := (if j>z8106n then 19 else z8106n−1)
      step −1 until j do out string(channel, '␣');
    if negative then out string(channel, '−');
    for k := n of digits −1 step −1 until 0 do
      out symbol(channel, '0123456789', digit[k]+1)
  end out integer;
```

```
procedure out real(channel, r);  value channel, r;
  integer channel;  real r;
comment this procedure outputs r properly rounded to channel
  using the Berkeley style. In this variant, the exponent part
  in the floating point form is replaced by 4 spaces if the exponent
  is zero. The sign of the exponent is always outputed, for com-
  patibility with in real. The exponent is restricted to the interval
  −99 to 99;
begin
  integer j, k, size, exponent;  Boolean negative;
  integer array digit[0: z8107d+1+(if z8107B then
    entier(ln(abs(r)+1)×0.4343) else 0)];
  procedure out digit(d);  integer d;
  begin
    out symbol(channel, '0123456789', d+1)
  end out digit;
  if z8107B then
  begin
    decompose real(r, if z8107d+exponent≦0 then 1 else 1+
    z8107d+ exponent, negative, size, exponent, digit);
    if size = −1 then
    begin
      exponent := if z8107d = 0 then 0 else −z8107d − 1;
      digit[0] := 0
    end
    else if z8107d = 0 ∧ exponent < 0 then
    begin exponent := 0;  digit[0] := end;
    j := (if negative then 3 else 2) +
      (if z8107d = 0 then −1 else z8107d) +
      (if exponent ≧ 0 then exponent else −1);
    for k := (if j>z8107n then 19 else z8107n--1) step −1
      until j do out string(channel, '␣');
    if negative then out string (channel, '−');
    for k := 0 step 1 until exponent do
      out digit(digit[k]);
    if z8107d > 0 then
    begin
      out string(channel, '·');
      for k := exponent + 1 step 1 until exponent + z8107d do
      if k < 0 then out string(channel, '0') else out digit(digit[k])
    end
  end fixed point representation
  else
  begin
    decompose real(r, z8107d+1, negative, size, exponent, digit);
    if size = −1 then
    begin
      exponent := 0;
      for k := 0 step 1 until z8107d do digit[k] := 0
    end;
    j := 6 + (if z8107d=0 then −1 else z8107d)+
      (if negative then 1 else 0);
    for k := (if j>z8107n then 19 else z8107n−1)
      step −1 until j do
      out string(channel, '␣');
    if negative then out string(channel, '-');
    out digit (digit [0]);
    if z8107d ≠ 0 then out string(channel, '·');
    for k := 1 step 1 until z8107d do out digit(digit[k]);
    if exponent = 0 then out string(channel, '␣␣␣␣')
    else
    begin
      out string(channel, '₁₀');
      comment This procedure assumes that ₁₀ takes one space,
        if not, the preceding statement should be modified;
      if exponent ≧ 0 then out string(channel, '+')
      else
      begin out string(channel, '−');
        exponent := −exponent
      end;
      j := exponent ÷ 10;
      if j = 0 then out string(channel, '␣')
      else out digit(j);
      out digit(exponent−j×10)
    end
  end floating point representation
end out real;
procedure out Boolean(channel, b);  value channel;
  integer channel;  Boolean b;
begin
  integer k, j;
  j := if b then 4 else 5;
  comment this procedure assumes that true and false take
    respectively 4 and 5 spaces, if not the preceding statement
     should be modified;
  for k := (if j>z8108n then 19 else z8108n−1) step −1 until
    j do out string(channel, '␣');
  out symbol(channel, 'true false', j−3)
end out Boolean;
integer procedure read i;
begin
  integer i;
  in integer(in channel, i);  read i := i
end read i;
real procedure read r;
begin
  real r;
  in real(in channel, r);  read r := r
end read r;
Boolean procedure read b;
begin
  Boolean b;
  in Boolean(in channel, b);  read b := b
end read b;
integer procedure ioi(i,s,n);  string s;  integer i, n;
comment this and the next 3 procedures input respectively an
  integer, a real number, a Boolean or a one dimensional array,
  they output an equivalent Algol statement;
begin
  out string(out channel, s);  out string(out channel, '␣ := ␣');
  in integer(in channel, i);  ioi := i;
  integer format(n);  out integer(out channel, i);
  out string(out channel, ';␣')
end ioi;
real procedure ior(r, s, B, n, d);
  real r;  string s;  Boolean B;  integer n, d;
begin
  out string(out channel, s);
  out string(out channel, '␣ := ␣');
  in real(in channel, r);  ior := r;
  real format(B, n, d);  out real(out channel, r);
  out string(out channel, ';␣')
end ior;
Boolean procedure iob(B, s, n);  Boolean b;  string s;
  integer n;
begin
  out string(out channel, s);
  out string(out channel, '␣ := ␣');
  in Boolean(in channel, B);  iob := B;
  Boolean format(n);  out Boolean(out channel, B);
  out string(out channel, ';␣')
end iob;
procedure ioa(a, l, u, s, B, n, d);
  integer l, u, n, d;  array a;  string s;  Boolean B;
```

```
begin
  integer i;
  if l > u then go to end ioa;
  real format(B, n, d);  oti(l, 'i', 3);
  out string(out channel, '␣for␣');
  out string(out channel, s);
  out string(out channel, '[i]␣ := ␣');
  for i := l step 1 until u do
  begin
    in real(in channel, a[i]);  out real(out channel, a[i]);
    if i < u then out string(out channel, ',␣')
    else out string (out channel, '␣do␣i␣ := ␣i␣+␣1;␣')
  end;
end ioa:
end ioa;
procedure oti(i, s, n);  value i, n;  integer i, n;  string s;
comment this and the following 3 procedures output Algol
  statements compatible with those of the input output procedures
  ioi, ior, iob, ioa;
begin
  out string(out channel, s);
  out string(out channel, '␣ := ␣');
  integer format(n); out integer(out channel, i);
  out string(out channel, ';␣')
end oti;
procedure otr(r, s, B, n, d);
  real r;  string s;  Boolean B;  integer n, d;
begin
  out string(out channel, s);
  out string(out channel, '␣ := ␣');
  real format(B, n, d);  out real(out channel, r);
  out string(out channel, ';␣')
end otr;
procedure otb(B, s, n);  Boolean B;  string s;  integer n;
begin
  out string(out channel, s);
  out string(out channel, '␣ := ␣');
  Boolean format(n);  out Boolean(out channel, B);
  out string (out channel, '; ␣')
end otb;
procedure ota(a, l, u, s, B, n, d);
  integer l, u, n, d;  array a;  string s;  Boolean B;
begin
  integer i;
  if l > u then go to end ota;
  real format(B, n, d);  oti(l, 'i', 3);
  out string(out channel, '␣for␣');
  out string(out channel, s);
  out string(out channel, '[i]␣ := ␣');
  for i := l step 1 until u do
  begin
    out real(out channel, a[i]);
    if i < u then out string(out channel, ',␣')
    else out string (out channel, '␣do␣i␣:=␣i␣+␣1;␣')
  end;
end ota:
end ota;
procedure outi(i, n);  integer i, n;
comment this and the following 3 procedures output integers,
  real numbers, Booleans or one dimensional arrays using format
  as indicated in out integer;
begin
  integer format(n);
  out integer(out channel, i)
end outi;
procedure outr(r, B, n, d);  real r;  Boolean B;  integer n, d;
```

```
begin
  real format(B, n, d);
  out real(out channel, r)
end outr;
procedure outb(B, n);  Boolean b;  integer n;
begin
  Boolean format(n);
  out Boolean(out channel, B)
end outb;
procedure outa(a, l, u, B, n, d);  integer l, u, n, d;  array a;
  Boolean B;
begin
  integer i;
  if l > u then go to end outa;
  real format(B, n, d);
  for i := l step 1 until u do out real(out channel, a[i]);
end outa:
end outa
```

ALGORITHM 336
NETFLOW [H]

T. A. Bray and C. Witzgall

(Recd. 2 Oct. 1967 and 20 May 1968)

Boeing Scientific Research Laboratories, Seattle, WA 98124

KEY WORDS AND PHRASES: capacitated network, linear programming, minimum-cost flow, network flow, out-of-kilter
CR CATEGORIES: 5.32, 5.41

```
procedure NETFLOW (nodes, arcs, I, J, cost, hi, lo, flow, pi,
  INFEAS);
  value nodes, arcs; integer nodes, arcs;
  integer array I, J, cost, hi, lo, flow, pi; label INFEAS;
comment This procedure determines the least-cost flow over an upper and lower bound capacitated flow network.
  Each directed network arc a is defined by nodes I[a] and J[a], has upper and lower flow bounds hi[a] and lo[a], and cost per unit of flow cost[a]. Costs and flow bounds may be any positive or negative integers. An upper flow bound must be greater than or equal to its corresponding lower flow bound for a feasible solution to exist. There may be any number of parallel arcs connecting any two nodes.
  The procedure returns vectors flow and pi. flow[a] is the computed optimal flow over network arc a. pi[n] is a number—the dual variable—which represents the relative value of injecting one unit of flow into the network of node n. NETFLOW may be entered with any values in vectors flow and pi (such as those from a previous or a guessed solution) feasible or not. If the initial contents of flow do not conserve flow at any node, the solution values will also not conserve flow at that node, by the same amount.
  This procedure is a revision (see remark by T. A. Bray and C. Witzgall [1]) of Algorithm 248 [2]. Like the original, it follows the out-of-kilter algorithm described by D. R. Fulkerson [3] and elsewhere. It follows the RAND code by R. J. Clasen (Fortran) in three instances, using a single set of labels na, which correspond to the nb of Algorithm 248, avoiding superfluous tests in the part following BACK (for instance, c > 0 ∧ flow[a] < lo[a] is equivalent to c > 0 at this point of the program), and taking advantage of the fact that arcs remain in kilter and need not be rechecked again. In addition, the convention inf = −1 is adopted in order to permit costs and bounds of value around 99999999 without their interfering with the initiation of minimum search.
  References:
  1. Bray, T. A., and Witzgall, C. Remark on Algorithm 248, NETFLOW. Comm. ACM 11 (Sept. 1968), 633.
  2. Briggs, William A. Algorithm 248, NETFLOW. Comm. ACM 8 (Feb. 1965), 103.
  3. Fulkerson, D. R. An out-of-kilte method for minimal-cost flow problems. J. Soc. Ind. Appl. Math. 9 (Mar. 1961), 18-27;
begin
  integer a, aok, c, cok, del, eps, inf, lab, m, n, src, snk;
  integer array na[1: nodes];
  integer procedure minp(x,y); value x,y; integer x,y;
  begin
    if x < y ∧ x ≥ 0 then minp := x else minp := y
  end minp;
  comment check feasibility of formulation;
  for a := 1 step 1 until arcs do
    if lo[a] > hi[a] then go to INFEAS;
  inf := −1;
  comment find out-of-kilter arc;
  for aok := 1 step 1 until arcs do
  begin
    cok := cost[aok] + pi[I[aok]] − pi[J[aok]];
TEST: if flow[aok] < lo[aok] ∨ (cok<0∧flow[aok]<hi[aok]) then
    begin
      src := J[aok]; snk := I[aok]; na[src] := + aok;
      go to LABL
    end;
    if flow[aok] > hi[aok] ∨ (cok>0∧flow[aok]>lo[aok]) then
    begin
      src := I[aok]; snk := J[aok]; na[src] := −aok;
      go to LABL
    end;
    comment arc aok is in kilter;
    go to NEXT;
    comment arc aok is out-of-kilter, clear all labels but source
      label, start new labeling;
LABL: for n := 1 step 1 until src − 1, src + 1 step 1 until
  nodes do na[n] := 0;
LOOP: lab := 0;
    comment switch set for determining whether a pass thru
      the list of arcs yields a new label;
    for a := 1 step 1 until arcs do
    begin
      if (na[I[a]]=0∧na[J[a]]=0) ∨ (na[I[a]]≠0∧na[J[a]]≠0) then
        go to XC;
      c := cost[a] + pi[I[a]] − pi[J[a]];
      if na[I[a]] = 0 then go to XA;
      if flow[a] ≥ hi[a] ∨ (flow[a]≥ lo[a]∧c>0) then
        go to XC;
      na[J[a]] := +a; go to XB;
XA:   if flow[a] ≤ lo[a] ∨ (flow[a] ≤hi[a]∧c<0) then
        go to XC;
      na[I[a]] := −a;
XB:   lab := 1;
      comment node labeled, test for breakthru;
      if na[snk] ≠ 0 then go to INCR;
XC:  end no breakthru;
    if lab ≠ 0 then go to LOOP;
    comment nonbreakthru, determine change to pi vector;
    del := inf;
    for a := 1 step 1 until arcs do
    begin
      if (na[I[a]]=0∧na[J[a]]=0) ∨ (na[I[a]]≠0∧na[J[a]]≠0) then
        go to XD;
      c := cost[a] + pi[I[a]] − pi[J[a]];
      if na[J[a]] = 0 ∧ flow[a] < hi[a] then
        del := minp(del,c);
      if na[J[a]] ≠ 0 ∧ flow[a] > lo[a] then
        del := minp(del,−c);
XD:  end;
    if del = inf then
    begin
      if flow[aok] = hi[aok] ∨ flow[aok] = lo[aok] then
```

```
            del := abs(cok)
          else go to INFEAS
      end exit, no feasible flow;
      comment change pi vector by computed del;
      for n := 1 step 1 until nodes do
        if na[n] = 0 then pi[n] := pi[n] + del;
      comment test whether aok is now in kilter;
      if del = abs(cok) ∧ flow[aok] ≥ lo[aok] ∧ flow[aok]
        ≤ hi[aok] then
        go to NEXT;
      cok := cost[aok] + pi[I[aok]] − pi[J[aok]];
      go to LOOP;
      comment breakthru, compute incremental flow;
INCR: eps := inf; n := src;
BACK: a := na[n];
      if a > 0 then
      begin
        m := I[a];
        if cost[a] + pi[m] − pi[n] > 0 then
          eps := minp(eps, lo[a]−flow[a])
        else eps := minp(eps, hi[a]−flow[a])
      end
      else
      begin
        m := J[−a];
        if cost[−a] + pi [n] − pi [m] < 0 then
          eps := minp(eps,flow[−a]−hi[−a])
        else eps := minp(eps,flow[−a]−lo[−a])
      end;
      n := m; if n ≠ src then go to BACK;
      comment change flow by eps;
BACK2: a := na[n];
      if a > 0 then
      begin
        m := I[a]; flow[a] := flow[a] + eps
      end
      else
      begin
        m := J[−a]; flow[−a] := flow[−a] − eps
      end;
      n := m; if n ≠ src then go to BACK2;
      comment test whether aok is now in kilter;
      go to TEST;
NEXT:
   end find next out-of-kilter arc
end NETFLOW with a feasible, optimal flow
```

REMARK ON ALGORITHM 336 [H]

NETFLOW [T. A. Bray and C. Witzgall, *Comm. ACM 11* (Sept. 1968), 631–632]

T. A. Bray and C. Witzgall (Recd. 20 Oct. 1969)
Boeing Scientific Research Laboratories, Seattle, WA 98124

KEY WORDS AND PHRASES: capacitated network, linear programming, minimum-cost flow, network flow, out-of-kilter
CR CATEGORIES: 5.32, 5.41

The algorithm as published contains an error on the 11th line following the line labled *XD*, which reads:

if $del = abs(cok) \wedge \ldots$

This line should read

if $del \geq abs(cok) \wedge \ldots$

Fortunately, this error does not invalidate the algorithm but may in some cases lead to additional operations.

ALGORITHM 337
CALCULATION OF A POLYNOMIAL AND ITS DERIVATIVE VALUES BY HORNER SCHEME [C1]

W. PANKIEWICZ (Recd. 28 Mar. 1968 and 16 May 1968)
Warszawa - 1, Al. 3-go Maja 2/68, Poland

KEY WORDS AND PHRASES: function evaluation, polynomial evaluation, Algol procedure, Horner's scheme
CR CATEGORIES: 5.12, 4.22

```
procedure horner(n,a,k,r,x0,b);  value n,k,x0,b;
  integer n,k;  real x0;  Boolean b;  array a,r;
```

comment If b is **true** the procedure calculates and stores in $r[i]$ the value of

$$d^i(\sum_{j=0}^{n} a[j]\times x\uparrow j)/dx^i$$

and $x = x0$ for $i = 0, 1, \cdots, k$. If b is **false** it calculates and stores in the array r the values of the first $k+1$ coefficients of the expansion of the polynomial in a power series in the neighborhood of $x0$, i.e.

$$\sum_{j=0}^{n} a[j]\times x\uparrow j = \sum_{i=0}^{n} r[i]\times(x-x0)\uparrow i.$$

Here n is the degree of the polynomial whose coefficients are given by $a[0{:}n]$. It is assumed that $0 \le k \le n$. If $k = 0$ only the value of the polynomial is calculated. If b is **false** the choice $k = n$ would be most useful.

This algorithm is essentially equivalent to Algorithm 29 [*Comm. ACM 3* (Nov. 1960), 604] in terms of quantities computed, but the application of Horner's scheme significantly reduces the number of operations.

Example 1. For the polynomial of degree $n = 5$: $w(x) = x\uparrow 5 + 2\times x\uparrow 4 - 3\times x\uparrow 3 + 8\times x\uparrow 2 - 7\times x + 11$, $k = 2$, $x0 = 2$ and b = **true**, the following was obtained: $r[0] = 69$, $r[1] = 133$, $r[2] = 236$, i.e. $w(2) = 69$, $w'(2) = 133$ and $w''(2) = 236$.

Example 2. For the polynomial of degree $n = 7$: $w(x) = x\uparrow 7 - 7\times x\uparrow 5 + 6\times x\uparrow 4 + 4\times x\uparrow 3 - x\uparrow 2 - 2\times x - 9$, $k = 7$, $x0 = 2$ and b = **false** the following vector r was obtained: 15, 122, 279, 332, 216, 77, 14, 1, i.e., the given polynomial can be expressed in the form: $w(x) = 15 + 122\times(x-2) + 279\times(x-2)\uparrow 2 + 332\times(x-2)\uparrow 3 + 216\times(x-2)\uparrow 4 + 77\times(x-2)\uparrow 5 + 14\times(x-2)\uparrow 6 + (x-2)\uparrow 7$;

```
begin
  integer i, j, l;  real rr;
  rr := a[0];
  for i := 0 step 1 until k do
    r[i] := rr;
  for j := 1 step 1 until n do
  begin
    r[0] := r[0] × x0 + a[j];
    l := if n − j > k then k else n − j;
    for i := 1 step 1 until l do
      r[i] := r[i] × x0 + r[i−1]
  end;
  if b then
  begin
    l := 1;
    for i := 2 step 1 until k do
    begin
      l := l × i;
      r[i] := r[i] × l
    end
  end
end horner
```

REMARK ON ALGORITHM 337 [C1]
CALCULATION OF A POLYNOMIAL AND ITS DERIVATIVE VALUES BY HORNER SCHEME

[W. Pankiewicz, *Comm. ACM 11* (Sept. 1968), 633]

OLIVER K. SMITH (Recd. 27 Sept. 1968)
Applied Mathematics Dept., Systems Group of TRW, Inc., 1 Space Park, Redondo Beach, CA 90278

KEY WORDS AND PHRASES: function evaluation, polynomial evaluation, ALGOL procedure, Horner's scheme
CR CATEGORIES: 4.22, 5.12

The definition of the given polynomial is incorrect in the comment. In both the third line and the eighth line of the comment, $a[j]$ should be replaced by $a[n-j]$. Also the first word "and" of the fourth line of the comment should be changed to "at".

ALGORITHM 338
ALGOL PROCEDURES FOR THE FAST FOURIER TRANSFORM [C6]

RICHARD C. SINGLETON*

(Recd. 21 Nov. 1966, 2 Aug. 1967 and 18 July 1968)

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, virtual core memory, permutation

CR CATEGORIES: 3.15, 3.83, 5.12, 5.14

The following procedures are based on the Cooley-Tukey algorithm [1] for computing the finite Fourier transform of a complex data vector; the dimension of the data vector is assumed here to be a power of two. Procedure *COMPLEXTRANSFORM* computes either the complex Fourier transform or its inverse. Procedure *REALTRANSFORM* computes either the Fourier coefficients of a sequence of real data points or evaluates a Fourier series with given cosine and sine coefficients. The number of arithmetic operations for either procedure is proportional to $n \log_2 n$, where n is the number of data points.

Procedures *FFT*2, *REVFFT*2, *REORDER*, and *REALTRAN* are building blocks, and are used in the two complete procedures mentioned above. The fast transform can be computed in a number of different ways, and these building block procedures were written so as to make practical the computing of large transforms on a system with virtual memory. Using a method proposed by Singleton [2], data is accessed in sub-sequences of consecutive array elements, and as much computing as possible is done in one section of the data before moving on to another. Procedure *FFT*2 computes the Fourier transform of data in normal order, giving a result in reverse binary order. Procedure *REVFFT*2 computes the Fourier transform of data in reverse binary order and leaves the result in normal binary order. Procedure *REORDER* permutes a complex vector from binary to reverse binary order or from reverse binary to binary order; this procedure also permutes real data in preparation for efficient use of the complex Fourier transform. Procedures *FFT*2, *REVFFT*2, and *REORDER* may also be used to compute multivariate Fourier transforms. The procedure *REALTRAN* is used to unscramble and combine the complex transforms of the even and odd numbered elements of a sequence of real data points. This procedure is not restricted to powers of two and can be used whenever the number of data points is even.

REFERENCES:

1. COOLEY, J. W., and TUKEY, J. W. An algorithm for the machine calculation of complex Fourier series. *Math. Comput. 19*, 90, (Apr. 1965), 297-301.
2. SINGLETON, R. C. On computing the fast Fourier transform. *Comm. ACM 10* (Oct. 1967), 647-654;

```
procedure COMPLEXTRANSFORM (A, B, m, inverse);
  value m, inverse; integer m;
  Boolean inverse; array A, B;
comment Computes the Fourier transform of $2^m$ complex data
  values. The arrays $A[0: n-1]$ and $B[0: n-1]$, where $n = 2^m$,
  initially contain the real and imaginary components of the data,
  and on exit contain the corresponding Fourier coefficient values.
  If inverse is false, the Fourier transform

      $$\frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} (a_k + ib_k) \exp(i2\pi jk/n)$$

  is computed. The transform followed by the inverse transform
  (or the inverse transform followed by the transform) gives an
  identity transformation. Procedures FFT2 and REORDER are
  used by this procedure and must also be declared;
begin integer n, j; real p, q;
  n := 2 ↑ m; p := q := 1.0/sqrt(n);
  if inverse then
  begin
    q := -q;
    for j := n - 1 step -1 until 0 do B[j] := -B[j]
  end;
  FFT2(A, B, n, m, n); REORDER(A, B, n, m, n, false);
  for j := n - 1 step -1 until 0 do
    begin A[j] := A[j] × p; B[j] := B[j] × q end
end COMPLEXTRANSFORM;

procedure REALTRANSFORM(A, B, m, inverse);
  value m, inverse; integer m;
  Boolean inverse; array A, B;
comment Computes the finite Fourier transform of $2^{m+1} \geqq 4$
  real data points. If inverse is false, the arrays $A[0: n]$ and
  $B[0: n]$, where $n = 2^m$, initially contain the first $2^m$ real data
  points $x_0, x_1, \cdots, x_{n-1}$ as $A[0], \cdots, A[n-1]$ and the remaining
  $2^m$ real data points $x_n, x_{n+1}, \cdots, x_{2n-1}$ as $B[0], B[1], \cdots, B[n-1]$.
  On completion of the transform the arrays A and B contain
  respectively the Fourier cosine and sine coefficients $a_k$ and $b_k$,
  computed according to the relations

      $$a_k = \frac{1}{n} \sum_{j=0}^{2n-1} x_j \cos(\pi jk/n) \quad \text{for} \quad k = 0, 1, \cdots, n,$$

  and

      $$b_k = \frac{1}{n} \sum_{k=0}^{2n-1} x_j \sin(\pi jk/n) \quad \text{for} \quad k = 0, 1, \cdots, n.$$

  If inverse is true, the arrays A and B initially contain $n + 1$
  cosine coefficients $a_0, a_1, \cdots, a_n$ and $n + 1$ sine coefficients
  $b_0, b_1, \cdots, b_n$, where $b_0 = b_n = 0$. The procedure evaluates the
  corresponding time series $x_0, x_1, \cdots, x_{2n-1}$, where

      $$x_j = \frac{a_0}{2} + \sum_{k=1}^{n-1} [a_k \cos(\pi jk/n) + b_k \sin(\pi jk/n)] + \frac{a_n}{2} \cos(\pi j),$$

  and leaves the first n values as $A[0], A[1], \cdots, A[n-1]$ and the
  remaining n values as $B[0], B[1], \cdots, B[n-1]$. The procedures
  FFT2, REVFFT2, REORDER, and REALTRAN are used by
  this procedure, and must also be declared;
```

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

```
begin integer n, j;  real p;
  n := 2 ↑ m;
  if inverse then
  begin
    REALTRAN(A, B, n, true);
    for j := n − 1 step −1 until 0 do B[j] := −B[j];
    FFT2(A, B, n, m, n);
    for j := n − 1 step −1 until 0 do
      begin A[j] := 0.5 × A[j];  B[j] := −0.5 × B[j] end;
    REORDER(A, B, n, m, n, true)
  end
  else
  begin
    REORDER(A, B, n, m, n, true);
    REVFFT2(A, B, n, m, 1);  p := 0.5/n;
    for j := n − 1 step −1 until 0 do
      begin A[j] := p × A[j];  B[j] := p × B[j] end;
    REALTRAN(A, B, n, false)
  end
end REALTRANSFORM;
procedure FFT2(A, B, n, m, ks);  value n, m, ks;
  integer n, m, ks;  array A, B;
```

comment Computes the fast Fourier transform for one variable of dimension 2^m in a multivariate transform. n is the number of data points, i.e., $n = n_1 \times n_2 \times \cdots \times n_p$ for a p-variate transform, and $ks = n_k \times n_{k+1} \times \cdots \times n_p$, where $n_k = 2^m$ is the dimension of the current variable. Arrays $A[0:n-1]$ and $B[0:n-1]$ originally contain the real and imaginary components of the data in normal order. Multivariate data is stored according to the usual convention, e.g., a_{jkl} is in $A[j \times n_2 \times n_3 + k \times n_3 + l]$ for $j = 0, 1, \cdots, n_1 - 1$, $k = 0, 1, \cdots, n_2 - 1$, and $l = 0, 1, \cdots, n_3 - 1$. On exit, the real and imaginary components of the resulting Fourier coefficients for the current variable are in reverse binary order. Continuing the above example, if the "column" variable n_2 is the current one, column

$$k = k_{m-1}2^{m-1} + k_{m-2}2^{m-2} + \cdots + k_1 2 + k_0$$

is permuted to position

$$k_0 2^{m-1} + k_1 2^{m-2} + \cdots + k_{m-2}2 + k_{m-1} .$$

A separate procedure may be used to permute the results to normal order between transform steps or all at once at the end. If $n = ks = 2^m$, the single-variate transform

$$(x_j + iy_j) = \sum_{k=0}^{n-1} (a_k + ib_k) \exp\,(i2\pi jk/n)$$

for $j = 0, \cdots, n-1$ is computed, where $(a + ib)$ represent the initial values and $(x + iy)$ represent the transformed values;

```
begin integer k0, k1, k2, k3, span, j, jj, k, kb, kn, mm, mk;
  real rad, c1, c2, c3, s1, s2, s3, ck, sk, sq;
  real A0, A1, A2, A3, B0, B1, B2, B3;
  integer array C[0: m];
  sq := 0.707106781187;
  sk := 0.382683432366;
  ck := 0.92387953251;
  C[m] := ks;  mm := (m÷2) × 2;  kn := 0;
  for k := m − 1 step −1 until 0 do C[k] := C[k+1] ÷ 2;
  rad := 6.28318530718/(C[0]×ks);  mk := m − 5;
L:  kb := kn;  kn := kn + ks;
  if mm ≠ m then
  begin
    k2 := kn;  k0 := C[mm] + kb;
L2:  k2 := k2 − 1;  k0 := k0 − 1;
    A0 := A[k2];  B0 := B[k2];
    A[k2] := A[k0] − A0;  A[k0] := A[k0] + A0;
    B[k2] := B[k0] − B0;  B[k0] := B[k0] + B0;
    if k0 > kb then go to L2
  end;
  c1 := 1.0;  s1 := 0;
  jj := 0;  k := mm −2;  j := 3;
  if k ≧ 0 then go to L4 else go to L6;
L3:  if C[j] ≦ jj then
  begin
    jj := jj − C[j];  j := j − 1;
    if C[j] ≦ jj then
    begin
      jj := jj − C[j];  j := j − 1;  k := k + 2;
      go to L3
    end
  end;
  jj := C[j] + jj;  j := 3;
L4:  span := C[k];
  if jj ≠ 0 then
  begin
    c2 := jj × span × rad;  c1 := cos(c2);  s1 := sin(c2);
L5:  c2 := c1 ↑ 2 − s1 ↑ 2;  s2 := 2.0 × c1 × s1;
    c3 := c2 × c1 − s2 × s1;  s3 := c2 × s1 + s2 × c1
  end;
  for k0 := kb + span − 1 step −1 until kb do
  begin
    k1 := k0 + span;  k2 := k1 + span;  k3 := k2 + span;
    A0 := A[k0];  B0 := B[k0];
    if s1 = 0 then
    begin
      A1 := A[k1];  B1 := B[k1];
      A2 := A[k2];  B2 := B[k2];
      A3 := A[k3];  B3 := B[k3]
    end
    else
    begin
      A1 := A[k1] × c1 − B[k1] × s1;
      B1 := A[k1] × s1 + B[k1] × c1;
      A2 := A[k2] × c2 − B[k2] × s2;
      B2 := A[k2] × s2 + B[k2] × c2;
      A3 := A[k3] × c3 − B[k3] × s3;
      B3 := A[k3] × s3 + B[k3] × c3
    end;
    A[k0] := A0 + A2 + A1 + A3;  B[k0] := B0 + B2 + B1 + B3;
    A[k1] := A0 + A2 − A1 − A3;  B[k1] := B0 + B2 − B1 − B3;
    A[k2] := A0 − A2 − B1 + B3;  B[k2] := B0 − B2 + A1 − A3;
    A[k3] := A0 − A2 + B1 − B3;  B[k3] := B0 − B2 − A1 + A3
  end;
  if k > 0 then begin k := k − 2;  go to L4 end;
  kb := k3 + span;
  if kb < kn then
  begin
    if j = 0 then begin k := 2;  j := mk;  go to L3 end;
    j := j − 1;  c2 := c1;
    if j = 1 then
      begin c1 := c1 × ck + s1 × sk;  s1 := s1 × ck − c2 × sk end
    else begin c1 := (c1−s1) × sq;  s1 := (c2 + s1) × sq end;
    go to L5
  end;
L6:  if kn < n then go to L
end FFT2;

procedure REVFFT2(A, B, n, m, ks);  value n, m, ks;
  integer n, m, ks;  array A, B;
```

comment Computes the fast Fourier transform for one variable of dimension 2^m in a multivariate transform. n is the number of data points, i.e., $n = n_1 \times n_2 \times \cdots \times n_p$ for a p-variate transform, and $ks = n_{k+1} \times n_{k+2} \times \cdots \times n_p$, where $n_k = 2^m$ is the dimension of the current variable. Arrays $A[0:n-1]$ and $B[0:n-1]$ originally contain the real and imaginary components of the data with the indices of each variable in reverse binary order, e.g., a_{jkl} is in $A[j' \times n_2 \times n_3 + k' \times n_3 + l']$ for $j = 0, 1, \cdots,$

$n_1 - 1$, $k = 0, 1, \cdots, n_2 - 1$, and $l = 0, \cdots, n_3 - 1$, where j', k', and l' are the bit-reversed values of j, k, and l. On completion of the multivariate transform, the real and imaginary components of the resulting Fourier coefficients are in A and B in normal order. If $n = 2^m$ and $ks = 1$, a single-variate transform is computed;

```
begin
  integer k0, k1, k2, k3, k4, span, nn, j, jj, k, kb, nt, kn, mk;
  real rad, c1, c2, c3, s1, s2, s3, ck, sk, sq;
  real A0, A1, A2, A3, B0, B1, B2, B3, re, im;
  integer array C[0: m];
  sq := 0.707106781187;
  sk := 0.382683432366;
  ck := 0.92387953251;
  C[0] := ks;  kn := 0;  k4 := 4 × ks;  mk := m − 4;
  for k := 1 step 1 until m do C[k] := ks := ks + ks·
  rad := 3.14159265359/(C[0]×ks);
L:  kb := kn + k4;  kn := kn + ks;
  if m = 1 then go to L5;
  k := jj := 0;  j := mk;  nt := 3;
  c1 := 1.0;  s1 := 0;
L2:  span := C[k];
  if jj ≠ 0 then
  begin
    c2 := jj × span × rad;  c1 := cos(c2);  s1 := sin(c2);
L3:  c2 := c1 ↑ 2 − s1 ↑ 2;  s2 := 2.0 × c1 × s1;
    c3 := c2 × c1 − s2 × s1;  s3 := c2 × s1 + s2 × c1
  end else s1 := 0;
  k3 := kb − span;
L4:  k2 := k3 − span;  k1 := k2 − span;  k0 := k1 − span;
  A0 := A[k0];  B0 := B[k0];
  A1 := A[k1];  B1 := B[k1];
  A2 := A[k2];  B2 := B[k2];
  A3 := A[k3];  B3 := B[k3];
  A[k0] := A0 + A1 + A2 + A3;  B[k0] := B0 + B1 + B2 + B3;
  if s1 = 0 then
  begin
    A[k1] := A0 − A1 − B2 + B3;  B[k1] := B0 − B1 + A2 − A3;
    A[k2] := A0 + A1 − A2 − A3;  B[k2] := B0 + B1 − B2 − B3;
    A[k3] := A0 − A1 + B2 − B3;  B[k3] := B0 − B1 − A2 + A3
  end
  else
  begin
    re := A0 − A1 − B2 + B3;  im := B0 − B1 + A2 − A3;
    A[k1] := re × c1 − im × s1;  B[k1] := re × s1 + im × c1;
    re := A0 + A1 − A2 − A3;  im := B0 + B1 − B2 − B3;
    A[k2] := re × c2 − im × s2;  B[k2] := re × s2 + im × c2;
    re := A0 − A1 + B2 − B3;  im := B0 − B1 − A2 + A3;
    A[k3] := re × c3 − im × s3;  B[k3] := re × s3 + im × c3
  end;
  k3 := k3 + 1;  if k3 < kb then go to L4;
  nt := nt − 1;
  if nt ≧ 0 then
  begin
    c2 := c1;
    if nt = 1 then
      begin c1 := c1 × ck + s1 × sk;  s1 := s1 × ck − c2 × sk end
    else begin c1 := (c1−s1) × sq;  s1 := (c2+s1) × sq end;
    kb := kb + k4;  if kb ≦ kn then go to L3 else go to L5
  end;
  if nt = −1 then begin k := 2;  go to L2 end;
  if C[j] ≦ jj then
  begin
    jj := jj − C[j];  j := j − 1;
    if C[j] ≦ jj then
      begin jj := jj − C[j];  j := j − 1;  k := k + 2 end
    else begin jj := C[j] + jj;  j := mk end
  end
  else begin jj := C[j] + jj;  j := mk end;
  if j < mk then go to L2;  k := 0;  nt := 3;
  kb := kb + k4;  if kb ≦ kn then go to L2;
L5:  k := (m÷2) × 2;
  if k ≠ m then
  begin
    k2 := kn;  k0 := j := kn − C[k];
L6:  k2 := k2 − 1;  k0 := k0 − 1;
    A0 := A[k2];  B0 := B[k2];
    A[k2] := A[k0] − A0;  A[k0] := A[k0] + A0;
    B[k2] := B[k0] − B0;  B[k0] := B[k0] + B0;
    if k2 > j then go to L6
  end;
  if kn < n then go to L
end REVFFT2;
```

```
procedure REORDER(A, B, n, m, ks, reel);
  value n, m, ks, reel;  integer n, m, ks;
  Boolean reel; array A, B;
```

comment Permutes data from normal to reverse binary order or from reverse binary to normal order. If *reel* is **false**, data for one variate of dimension 2^m in a multivariate data set of size n is permuted. In a p-variate transform with $n = n_1 \times n_2 \times \cdots \times n_p$, ks has the value $ks = n_k \times n_{k+1} \times \cdots \times n_p$, where $n_k = 2^m$ is the dimension of the current variable. For a single-variate transform, $n = ks = 2^m$. If *reel* is **true**, $A[2\times j+1]$ and $B[2\times j]$ are exchanged for $j = 0, 1, \cdots, (n-2)/2$, then adjacent pairs of entries in A and B are permuted to reverse-binary order. This option is used when transforming $2n$ real data values, with the first n stored in A and the second n in B. After permutation, the even-numbered entries are in A and the odd-numberd entries are in B, each in reverse-binary order.

Calling *REORDER* twice with the same parameter values gives an identity transformation;

```
begin integer i, j, jj, k, kk, kb, k2, ku, lim, p;
  real t;
  integer array C, LST[0: m];
  C[m] := ks;
  for k := m step −1 until 1 do C[k − 1] := C[k] ÷ 2:
  p := j := m − 1;  i := kb := 0;
  if reel then
  begin
    ku := n − 2;
    for k := 0 step 2 until ku do
      begin t := A[k + 1];  A[k + 1] := B[k];  B[k] := t end
  end else m := m − 1;
  lim := (m + 2) ÷ 2;  if p ≦ 0 then go to L4;
L:  ku := k2 := C[j] + kb;  jj := C[m − j];  kk := kb + jj;
L2:  k := kk + jj;
L3:  t := A[kk];  A[kk] := A[k2];  A[k2] := t;
  t := B[kk];  B[kk] := B[k2];  B[k2] := t;
  kk := kk + 1;  k2 := k2 + 1;
  if kk < k then go to L3;
  kk := kk + jj;  k2 := k2 + jj;
  if kk < ku then go to L2;
  if j > lim then
  begin
    j := j − 1;  i := i + 1;
    LST[i] := j;  go to L
  end;
  kb := k2;
  if i > 0 then
    begin j := LST[i];  i := i − 1;  go to L end;
  if kb < n then begin j := p; go to L end;
L4:
end REORDER;
```

```
procedure REALTRAN(A, B, n, evaluate);
  value n, evaluate; integer n;
  Boolean evaluate; array A, B;
```

```
comment If evaluate is false, this procedure unscrambles the
  single-variate complex transform of the n even-numbered and
  n-odd-numbered elements of a real sequence of length 2n, where
  the even-numbered elements were originally in A and the odd-
  numbered elements in B. Then it combines the two real trans-
  forms to give the Fourier cosine coefficients A[0], A[1], ··· , A[n]
  and sine coefficients B[0], B[1], ··· , B[n] for the full sequence
  of 2n elements. If evaluate is true, the process is reversed, and a
  set of Fourier cosine and sine coefficients is made ready for
  evaluation of the corresponding Fourier series by means of
  the inverse complex transform. Going in either direction,
  REALTRAN scales by a factor of two, which should be taken
  into account in determining the appropriate overall scaling;
begin integer k, nk, nh;
  real aa, ab, ba, bb, re, im, ck, sk, dc, ds, r;
  nh := n ÷ 2;  r := 3.14159265359/n;
  ds := sin(r);  r := −(2×sin(0.5×r)) ↑ 2;
  dc := −0.5 × r;  ck := 1.0;  sk := 0;
  if evaluate then
    begin ck := −1.0;  dc := −dc end
  else begin A[n] := A[0];  B[n] := B[0] end;
  for k := 0 step 1 until nh do
  begin
    nk := n − k;
    aa := A[k] + A[nk];  ab := A[k] − A[nk];
    ba := B[k] + B[nk];  bb := B[k] − B[nk];
    re := ck × ba + sk × ab;  im := sk × ba − ck × ab;
    B[nk] := im − bb;  B[k] := im + bb;
    A[nk] := aa − re;  A[k] := aa + re;
    dc := r × ck + dc;  ck := ck + dc;
    ds := r × sk + ds;  sk := sk + ds
  end
end REALTRAN
```

ALGORITHM 339
AN ALGOL PROCEDURE FOR THE FAST FOURIER TRANSFORM WITH ARBITRARY FACTORS [C6]

Richard C. Singleton*

(Recd. 2 Dec. 1966, 19 July 1967, 2 Aug. 1967 and 18 July 1968)

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, virtual core memory, permutation
CR CATEGORIES: 3.15, 3.83, 5.12, 5.14

procedure *FFT*(*A*, *B*, *n*, *nv*, *ks*); **value** *n*, *nv*, *ks*;
  integer *n*, *nv*, *ks*; **array** *A*, *B*;
comment This procedure computes the finite Fourier transform for one variate of dimension *nv* within a multivariate transform of *n* complex data values. The real and imaginary components of the data are stored in arrays $A[0:n-1]$ and $B[0:n-1]$, following the usual arrangement for indexing multivariate data in a single-dimensional array, e.g., a_{jkl} is stored in location $A[j\times n_2\times n_3+k\times n_3+l]$ for $j = 0, 1, \cdots, n_1 - 1$, $k = 0, 1, \cdots, n_2 - 1$, and $l = 0, 1, \cdots, n_3 - 1$. The value of *ks* for the *k*th variate of a *p*-variate transform is

$$ks = n_k \times n_{k+1} \times \cdots \times n_p$$

where $nv = n_k$ and $n = n_1 \times n_2 \times \cdots \times n_p$. On completion of the transform, the real and imaginary components of the resulting Fourier coefficients are in *A* and *B* respectively. For a single variable, $n = nv = ks$, and the transform

$$\sum_{k=0}^{n-1} (a_k + ib_k) \exp (i2\pi jk/n)$$

is computed for $j = 0, 1, \cdots, n - 1$.

For a single-variate transform of $2n$ real-valued points, the amount of computing can be reduced by approximately one-half by using procedure *REALTRAN* [3] together with *FFT*. The even-numbered data points are stored initially in array *A*, the odd-numbered data points in array *B*, the transform is computed with

$$FFT(A, B, n, n, n),$$

and the result is unscrambled with

$$REALTRAN(A, B, n, \textbf{false})$$

and then scaled by $1/2n$ to give the cosine coefficients as $A[0]$, $A[1]$, $\cdots$, $A[n]$ and the sine coefficients as $B[1]$, $B[2]$, $\cdots$, $B[n-1]$, with $B[0] = B[n] = 0$. The inverse operation, evaluating the Fourier series with cosine coefficients *A* and sine coefficients *B*, is computed by

$$REALTRAN(A, B, n, \textbf{true})$$

followed by

$$FFT(A, B, n, n, n),$$

then scaling by 1/2, yielding the even-numbered time domain values in array *A* and the odd-numbered values in array *B*. Note that the upper bounds of array *A* and *B* must be increased to *n* when procedure *REALTRAN* is used.

The method is based on an algorithm due to Cooley and Tukey [1], with modifications proposed by Singleton [2], to allow computing of large transforms on a system with virtual memory. The dimension *nv* is first decomposed into its prime factors nv_1, nv_2, $\cdots$, nv_m, and then nv/nv_i transforms of dimension nv_i are computed for $i = 1, 2, \cdots, m$. The resulting transformed values are then permuted to normal order in a final step. Computing times, to a first approximation, should be proportional to $n(nv_1+nv_2+\cdots+nv_m)$. The dimension of array *FACTOR* must be increased if *nv* has more than 20 factors.

In factoring *nv* at the beginning of the procedure, factors that are squares of primes are first removed, then the square-free portion is factored. The two factors of each square are placed symmetrically about the square-free factors. For example, $nv = 72$ is factored as $2 \times 3 \times 2 \times 3 \times 2$. This arrangement is used to simplify the final reordering in place. One symmetric permutation step is done for each square factor, and the reordering is completed by following the permutation cycles of the square-free portion.

In the transform phase of the procedure, special coding for factors of 2 and 3 is included for efficiency. Adjacent factors of 2 are also paired, and the results stored as for factors of 2 rather than 4. The remaining factors are handled by an odd-factor routine, using trigonometric function symmetries and smaller real transforms to reduce the number of multiplications by one-half as compared with a straightforward complex transform of an odd factor. The approximate number of complex multiplications is $n/2$ for a factor of 2, $3n/4$ for a factor of 4, and $(p-1)(p+3)n/4p$ for an odd factor *p*.

In both the transform and reordering phases, data is accessed in subsequences of consecutive array elements, and as much computing as possible is done in one section of the data before moving on to another. This is done to reduce the number of memory overlay operations in a system with virtual memory. After the first transform or symmetric permutation step, the remaining steps can be performed independently on each of nv_1 spans of data. We complete all remaining steps on the first span before beginning with the second. Similarly, after the second step the first span is subdivided in nv_2 independent spans. This subdivision process is continued through the remaining steps.

A number of working storage arrays are declared within this procedure. For large *n*, the total working storage is small in comparison with the $2n$ locations for data arrays *A* and *B*, except in a couple of cases. In the transform phase, approximately $6q$ working storage locations are used, where *q* is the largest prime factor in the transform. This requirement is minor except in a single-variate transform with *n* a prime number. During the reordering phase, the worst case occurs when doing a single-variate transform with *n* a product of two or more primes with no square factors. In this case, approximately *n* working storage locations are required.

This program was tested on the Burroughs B5500 computer and compared with another program computing a single *n*-by-*n*

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

complex Fourier transform. Whenever *n* had two or more prime factors, procedure *FFT* was much faster. The B5500 ALGOL system limits single-dimension arrays to 1023 words, but larger transforms can be computed by declaring

array *A*, *B*[0: (*n*−1) ÷ 512, 0: 511],

storing the data 512 entries per row, and using partial word indexing *A*[*J*.[30:9], *J*.[39:9]] instead of *A*[*J*] wherever *A* and *B* appear in procedure *FFT*.

References:

1. Cooley, J. W., and Tukey, J. W. An algorithm for the machine calculation of complex Fourier series. *Math. Comput. 19*, 90 (Apr. 1965), 297–301.
2. Singleton, R. C. On computing the fast Fourier transform. *Comm. ACM 10* (Oct. 1967), 647–654.
3. Singleton, R. C. Algorithm 338: ALGOL procedures for the fast Fourier transform. *Comm. ACM 11* (Nov. 1968), 771–774;

```
begin integer array FACTOR[0: 20]; Boolean zero;
  real A0, A1, A2, A3, B0, B1, B2, B3, cm, sm,
    c1, c2, c3, s1, s2, s3, c30, rad;
  integer k0, k1, k2, k3, jk, kf, kh, jf, mm,
    i, j, jj, k, kb, m, span, kt, kn;
  comment Determine the square factors of nv;
  k := nv;  m := 0;  j := 2;  jj := 4;  jf := 0;
  FACTOR [0] : =1;
L:  for i := k ÷ jj while i × jj = k do
    begin m := m + 1;  FACTOR[m] := j;  k := i end;
  if j = 2 then j := 3 else j := j + 2;
  jj := j × j;  if jj ≦ k then go to L;  kt := m;
  comment Determine the remaining factors of nv;
  for j := 2, 3 step 2 until k do
  for i := k ÷ j while i × j = k do
    begin m := m + 1;  FACTOR[m] := j;  k := i end;
  if FACTOR[kt] > FACTOR[m] then k := FACTOR[kt]
  else k := FACTOR[m];
  for j := kt step −1 until 1 do
    begin m := m+1;  FACTOR[m] := FACTOR[j] end;
  begin integer array C,D[0: m];
    begin array CK, SK, CF, SF[0:k−1];
      array AP, BP, AM, BM[0:(k−1)÷2];
      array RD, CC, SS[0:m];
      Boolean array BB[0:m+1];
      rad := 6.28318530718;  c30 := 0.866025403784;
      for j := m step −1 until 2 do
      begin
        BB[j] := (FACTOR[j−1]+FACTOR[j]) = 4;
        if BB [j] then
          begin j := j − 1;  BB[j] := false end
      end;
      BB[m+1] := BB[1] := false;
      C[0] := ks ÷ nv;  kn := 0;  D[0] := ks;
      for j := 1 step 1 until m do
      begin
        k := FACTOR[j];  C[j] := C[j−1] × k;
        D[j] := D[j−1] ÷ k;  RD[j] := rad/C[j];
        c1 := rad/k;
        if k > 2 then
          begin CC[j] := cos(c1);  SS[j] := sin(c1) end
      end;
      mm := if BB[m] then m−1 else m;
      if mm > 1 then
      begin
        sm := C[mm−2] × RD[m];
        cm := cos(sm);  sm := sin(sm)
      end;
L1:   kb := kn;  kn := kn + ks;  jj := 0;  i := 1;
      c1 := 1.0;  s1 := 0;  zero := true;
L2:   if BB[i+1] then
        begin kf := 4;  i := i + 1 end
      else kf := FACTOR[i];
      span := D[i];
      if ¬ zero then
      begin
        s1 := jj × RD[i];  c1 := cos(s1);  s1 := sin(s1)
      end;
      comment Factors of 2, 3, and 4 are handled
        separately to gain efficiency;
L3:   if kf = 4 then
      begin
        if ¬ zero then
        begin
          c2 := c1 ↑ 2 − s1 ↑ 2;  s2 := 2.0 × c1  × s1;
          c3 := c2 × c1 − s2 × s1;  s3 := c2 × s1 + s2 × c1
        end;
        for k0 := kb + span −1 step −1 until kb do
        begin
          k1 := k0 + span;  k2 := k1 + span;  k3 := k2 + span;
          A0 := A[k0];  B0 := B[k0];
          if zero then
          begin
            A1 := A[k1];  B1 := B[k1];
            A2 := A[k2];  B2 := B[k2];
            A3 := A[k3];  B3 := B[k3]
          end
          else
          begin
            A1 := A[k1] × c1 − B[k1] × s1;
            B1 := A[k1] × s1 + B[k1] × c1;
            A2 := A[k2] × c2 − B[k2] × s2;
            B2 := A[k2] × s2 + B[k2] × c2;
            A3 := A[k3] × c3 − B[k3] × s3;
            B3 := A[k3] × s3 + B[k3] × c3
          end;
          A[k0] := A0 + A2 + A1 + A3;  B[k0] := B0 + B2 +
            B1 + B3;
          A[k1] := A0 + A2 − A1 − A3;  B[k1] := B0 + B2 −
            B1 − B3;
          A[k2] := A0 − A2 − B1 + B3;  B[k2] := B0 − B2 +
            A1 − A3;
          A[k3] := A0 − A2 + B1 − B3;  B[k3] := B0 − B2 −
            A1 + A3
        end
      end
      else if kf = 3 then
      begin
        if ¬ zero then
          begin c2 := c1 ↑ 2 − s1 ↑ 2;  s2 := 2.0 × c1 × s1 end;
        for k0 := kb + span − 1 step −1 until kb do
        begin
          k1 := k0 + span;  k2 := k1 + span;
          A0 := A[k0];  B0 := B[k0];
          if zero then
          begin
            A1 := A[k1];  B1 := B[k1];
            A2 := A[k2];  B2 := B[k2]
          end
          else
          begin
            A1 := A[k1] × c1 − B[k1] × s1;
            B1 := A[k1] × s1 + B[k1] × c1;
            A2 := A[k2] × c2 − B[k2] × s2;
            B2 := A[k2] × s2 + B[k2] × c2
          end;
          A[k0] := A0 + A1 + A2;  B[k0] := B0 + B1 + B2;
          A0 := − 0.5 × (A1+A2) + A0;  A1 := (A1−A2) ×
            c30;
        B0 := − 0.5 × (B1+B2) + B0;  B1 := (B1−B2) ×
          c30;
        A[k1] := A0 − B1;  B[k1] := B0 + A1;
        A[k2] := A0 + B1;  B[k2] := B0 − A1
      end
    end
    else if kf = 2 then
    begin
      k0 := kb + span;  k2 := k0 + span;
      if zero then
      begin
        for k0 := k0 − 1 while k0 ≧ kb do
        begin
          k2 := k2 − 1;  A0 := A[k2];  B0 := B[k2];
          A[k2] := A[k0] − A0;  A[k0] := A[k0] + A0;
          B[k2] := B[k0] − B0;  B[k0] := B[k0] + B0
        end
      end
      else
      for k0 := k0 − 1 while k0 ≧ kb do
      begin
        k2 := k2 − 1;
        A0 := A[k2] × c1 − B[k2] × s1;
        B0 := A[k2] × s1 + B[k2] × c1;
        A[k2] := A[k0] − A0;  A[k0] := A[k0] + A0;
        B[k2] := B[k0] − B0;  B[k0] := B[k0] + B0
      end
    end
    else
    begin
      jk := kf − 1;  kh := jk ÷ 2;  k3 := D [i−1];
      k0 := kb + span;
      if ¬ zero then
      begin
        k := jk − 1;  CF[1] := c1;  SF[1] := s1;
        for j := 1 step 1 until k do
        begin
          CF[j+1] := CF[j] × c1 − SF[j] × s1;
          SF[j+1] := CF[j] × s1 + SF[j] × c1
        end
      end;
      if kf ≠ jf then
      begin
        CK[jk] := CK[1] := c2 := CC[i];\
        SK[1] := s2 := SS[i];  SK[jk] := −s2;
        for j := 1 step 1 until kh do
        begin
          k := jk − j;
          CK[k] := CK[j+1] := CK[j] × c2 − SK[j] × s2;
          SK[j+1] := CK[j] × s2 + SK[j] × c2;
          SK[k] := −SK[j+1]
        end
      end;
L4:   k1 := k0 := k0 − 1;  k2 := k0 + k3;
      A3 := A0 := A[k0];  B3 := B0 := B[k0];
      for j := 1 step 1 until kh do
      begin
        k1 := k1 + span;  k2 := k2 − span;
        if zero then
        begin
          A1 := A[k1];  B1 := B[k1];
          A2 := A[k2];  B2 := B[k2]
        end
        else
        begin
          k := kf − j;
          A1 := A[k1] × CF[j] − B[k1] × SF[j];
          B1 := A[k1] × SF[j] + B[k1] × CF[j];
          A2 := A[k2] × CF[k] − B[k2] × SF[k];
          B2 := A[k2] × SF[k] + B[k2] × CF[k]
        end;
        AP[j] := A1 + A2;  AM[j] := A1 − A2;
        BP[j] := B1 + B2;  BM[j] := B1 − B2;
        A3 := AP[j] + A3;  B3 := BP[j] + B3
      end;
      A[k0] := A3;  B[k0] := B3;
      k1 := k0;  k2 := k0 + k3;
      for j := 1 step 1 until kh do
      begin
        k1 := k1 + span;  k2 := k2 − span;  jk := j;
        A1 := A0;  B1 := B0;  A2 := B2 := 0;
        for k := 1 step 1 until kh do
        begin
          A1 := AP[k] × CK[jk] + A1;
          A2 := AM[k] × SK[jk] + A2;
          B1 := BP[k] × CK[jk] + B1;
          B2 := BM[k] × SK[jk] + B2;
          jk := jk + j;  if jk ≧ kf then jk := jk − kf
        end;
        A[k1] := A1 − B2;  A[k2] := A1 + B2;
        B[k1] := B1 + A2;  B[k2] := B1 − A2
      end;
      if k0 > kb then go to L4;  jf := kf
    end;
    if i < mm then
      begin i := i + 1;  go to L2 end;
    i := mm;  zero := false;
    kb := D[i − 1] + kb;
    if kb < kn then
    begin
      for jj := C[i−2] + jj while jj ≧ C[i−1] do
        begin i := i − 1;  jj := jj − C[i] end;
      if i = mm then
      begin
        c2 := c1;  c1 := cm × c1 − sm × s1;
        s1 := sm × c2 + cm × s1;  go to L3
      end;
      if BB[i] then i := i + 1;  go to L2
    end;
    if kn < n then go to L1
  end;
  i := 1;
  for j := kt − 1 step −1 until 1 do
  begin
    FACTOR[j] := FACTOR[j] − 1;  i := FACTOR[j] + i
  end;
  comment  We now permute the result to normal order;
  comment  The following if statement does the complete re-
    ordering if the square-free portion of n has at most one
    prime factor. Otherwise it does a partial reordering, leaving
    each entry in its correct section of length n ÷ c[kt],
    where c[kt] ↑ 2 is the product of the square factors;
  if kt > 0 then
  begin integer array S[0:i];
    j := 1;  i := kb := 0;
L5: k3 := k2 := D[j] + kb;  jk := jj := C[j−1];
    k0 := kb + jj;  span := C[j] − jj;
L6: k := k0 + jj;
L7: A0 := A[k0];  A[k0] := A[k2];  A[k2] := A0;
    B0 := B[k0];  B[k0] := B[k2];  B[k2] := B0;
    k0 := k0 + 1;  k2 := k2 + 1;
    if k0 < k then go to L7;
    k0 := k0 + span;  k2 := k2 + span;
    if k0 < k3 then go to L6;
    if k0 < (k3+span) then
```

```
        begin k0 := k0 − D[j] + jj;  go to L6 end;
      k3 := D[j] + k3;
      if (k3−kb) < D[j−1] then
      begin
        k2 := k3 + jk;  jk := jk + jj;
        k0 := k3 − D[j] + jk;  go to L6
      end;
      if j < kt then
      begin
        k := FACTOR[j] + i;  j := j + 1;
L8:     i := i + 1;  S[i] := j;  if i < k then go to L8;
        go to L5
      end;
      kb := k3;
      if i > 0 then
        begin j := S[i];  i := i − 1;  go to L5 end;
      if kb < n then begin j := 1;  go to L5 end
    end;
    jk := C[kt];  span := D[kt];  m := m − kt;
    kb := span ÷ jk −2;
    comment  The following if statement completes the reorder-
      ing if the square-free portion of n has two or more prime
      factors;
    if kt < m − 1 then
    begin integer array R[0:kb];
      array TA, TB[0:jk−1];
      for j := kt step 1 until m do D[j] := D[j] ÷ jk;
      jj := 0;
      for j := 1 step 1 until kb do
        begin
        k := kt;
        for jj := D[k+1] + jj while jj ≧ D[k] do
          begin jj := jj − D[k];  k := k + 1 end;
        if jj = j then R[j] := − j else R[j] := jj
      end;
      comment  Determine the permutation cycles of length
        ≧2;
      for j := 1 step 1 until kb do if R[j] > 0 then
      begin
        k2 := j;
        for k2 := abs(R[k2]) while k2 ≠ j do R[k2] := −R[k2]
      end;
      comment  Reorder A and B following the permutation
        cycles;
      kn := i := j := 0;
LA:   kb := kn;  kn := kn + ks;
LB:   j := j + 1;  if R[j] < 0 then go to LB;
      k := R[j];  k0 := jk × k + kb;
LC:   TA[i] := A[k0+i];  TB[i] := B[k0+i];
      i := i + 1;  if i < jk then go to LC;  i := 0;
LD:   k := −R[k];  jj := k0;  k0 := jk × k + kb;
LE:   A[jj+i] := A[k0+i];  B[jj+i] := B[k0+i];
      i := i + 1;  if i < jk then go to LE;  i := 0;
      if k ≠ j then go to LD;
LF:   A[k0+i] := TA[i];  B[k0+i] := TB[i];
      i := i + 1;  if i < jk then go to LF;  i := 0;
      if j < k2 then go to LB;  j := 0;
      kb := kb + span;  if kb < kn then go to LB;
      if kn < n then go to LA
    end
  end
end FFT
```

REMARK ON ALGORITH 339 [C6]
AN ALGOL PROCEDURE FOR THE FAST FOURIER TRANSFORM WITH ARBITRARY FACTORS

[Richard C. Singleton, *Comm. ACM 11* (Nov. 1968), 776]

Richard C. Singleton (Recd. 27 Nov. 1968)
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, virtual core memory, permutation

CR CATEGORIES. 3.15, 3.83, 5.12, 5.14

On page 778, column 2, the 7th and 6th lines from the bottom should be corrected to read:

```
LJ:  jj := C[i−2] + jj;  if jj ≧ C[i−1] then
       begin i := i − 1;  jj := jj − C[i];  go to LJ end;
```

On page 779, column 1, the 9th and 8th lines from the bottom should be corrected to read:

```
LX:  jj := D[k+1] + jj;  if jj ≧ D[k] then
       begin jj := jj − D[k];  k := k + 1;  go to LX end;
```

In both cases *jj* was originally used as the controlled variable of a for clause and thus was undefined after exit; the corrections preserve the value of *jj* for later use.

If the user prefers to compute constants with library functions, line 5 in column 2 on page 777 may be replaced by:

```
rad := 8.0 × arctan(1.0);  c30 := sqrt(0.75);
```

Algorithms 338 [*Comm. ACM 11* (Nov. 1968), 773] and 339 were punched from the printed page and tested on the CDC 6400 Algol compiler. After changing a colon to a semicolon at the end of line 37 in column 2 on page 775, the test results agreed with those obtained earlier with this compiler.

When computing a single-variate Fourier transform of real data, procedure *REALTRAN* may be used with procedure *FFT* (Algorithm 339) to reduce computing time. Two versions of *REALTRAN* have been given (Algorithms 338 and 345 [*Comm. ACM 12* (Mar. 1969), 179–184]); the first version is the faster of the two, but the second should be used if arithmetic results for real quantities are truncated rather than rounded.

In describing the evaluation of a real Fourier series, in the middle of column 2 on page 776, the necessary steps of reversing the signs of the *B* array values both before and after calling *FFT* were omitted. The correct steps, including scaling, are as follows:

```
REALTRAN(A B, n, true);
for j := n − 1 step −1 until 0 do B[j] := −B[j];
FFT(A, B, n, n, n);
for j := n − 1 step −1 until 0 do
  begin A[j] := 0.5 × A[j];  B[j] := −0.5 × B[j] end;
```

ALGORITHM 340
ROOTS OF POLYNOMIALS BY A ROOT-SQUARING AND RESULTANT ROUTINE [C2]

ALBERT NOLTEMEIER

(Recd. 2 Nov. 1967, 25 Jan. 1968 and 16 July 1968)

Technische Universität Hannover, Rechenzentrum, Hannover, Germany

KEY WORDS AND PHRASES: rootfinders, roots of polynomial equations, polynomial zeros, root-squaring operations, Graeffe method, resultant procedure, subresultant procedure, testing of roots, acceptance criteria

CR CATEGORIES: 5.15

```
procedure AG4(n, c, mm, delta, epsilon, range) Result: (re, im,
    mu, rt, gc, m, i, t) Exit: (fail);
  value n, mm, delta, epsilon, range;
  integer n, m, i, mm;  real delta, epsilon, range;
  integer array mu;
  array c, re, im, rt, gc, t;
  label fail;
```

comment *AG*4 finds simultaneously zeros of a polynomial of degree *n* with real coefficients by a root-squaring and resultant routine.

This procedure supersedes Algorithm 59 [2]. The following changes were made:

(a) In the procedure heading, the meaning of the old formal parameter *alpha* is shared by the three new parameters *mm*, *delta*, and *epsilon*, and *range*, *m*, *i*, *t*, *fail* are added to the formal parameter list.

(b) In the beginning of the procedure body the polynomial is tested for 0 as a zero (label *ZROTEST*). Although the modulus $\rho = 0$ can be found by squaring operations, the procedure usually will not find the root 0 without that test.

(c) In the program section labeled *SQUARING OPERATION* the iteratively squared coefficient is tested whether it will remain in the allowed range of numbers (formal parameter *range*) for a particular machine after another squaring operation.

(d) If there is a complex zero with a real part of 0, the resultant $R(p)$ is a polynomial of degree n with the coefficients $r_{n-1} = r_n = 0$. Computing the moduli of the zeros of this polynomial in the program section labeled *SQUARING OPERATION* and testing for pivotal coefficients, one would have to divide by 0. This case has been excluded by testing the divisor.

(e) If the acceptance criteria *epsilon* and *delta* are chosen too large, the sum of the multiplicities of the already found zeros may be greater than the degree *n* of the polynomial. In the program sections labeled *IT* and *D*, the test for the degree of the residual polynomial, the number of zeros, and the sum of the multiplicities of zeros in order to end the procedure has been improved.

Tests: The procedure *AG*4 has been tested on the CDC 1604-A computer at the Rechenzentrum, Technische Universität Hannover. The following results were obtained in a few representative cases. The parameters of acceptance criteria are *delta* = 0.2, *epsilon* = 10^{-7}, and *mm* = 10.

(i) $P_1(x) = x^8 - 30x^6 + 273x^4 - 820x^2 + 576$

$x_1 = 4.000\ 000\ 0010$ $x_2 = -4.000\ 000\ 0010$
$x_3 = 2.999\ 999\ 9990$ $x_4 = -2.999\ 999\ 9990$
$x_5 = 2.000\ 000\ 0000$ $x_6 = -2.000\ 000\ 0000$
$x_7 = 1.000\ 000\ 0000$ $x_8 = -1.000\ 000\ 0000$

(ii) $P_2(x) = x^5 + 7x^4 + 5x^3 + 6x^2 + 3x + 2$

$x_1 = -6.3509936102$
$x_{2,3} = 1.3506884657 \times 10^{-1} \pm i \times 7.7014185283 \times 10^{-1}$
$x_{4,5} = -4.5957204142 \times 10^{-1} \pm i \times 5.5126354891 \times 10^{-1}$

(iii) $P_3(x) = x^6 - 2x^5 + 2x^4 + x^3 + 6x^2 - 6x + 8$

$x_{1,2} = -9.9999999974 \times 10^{-1} \pm i \times 1.0000000002$
$x_{3,4} = 4.9999999999 \times 10^{-1} \pm i \times 8.6602540377 \times 10^{-1}$
$x_{5,6} = 1.4999999997 \pm i \times 1.3228756548$

(iv) $P_4(x) = x^2 - 4.01x + 4.02$

The procedure fails to compute any zero in this case (parameter $m = 0$). After changing the parameter *epsilon* to 10^{-5}, *AG*4 evaluates the zero $x = 2.0049937655$ with multiplicity 2 and remainder term 2.5×10^{-5};

Parameters:

n degree of the polynomial
c real coefficients of the polynomial
  $c[j](j=0,\cdots,n)$, where $c[n]$ is the constant term
delta, *epsilon* parameters for acceptence criteria
  practical input *delta* = 0.2, *epsilon* = 10 ↑ (−7)
range upper bound of the range of real constants
  (for the cDc 1604 -A *range* = 10 ↑ 307)
mm number of root-squaring iterations
  practical input *mm* = 10
re real part of each zero $re[j](j=1,\cdots,m)$
im imaginary part of each zero $im[j](j=1,\cdots,m)$
mu corresponding multiplicity $mu[j](j=1,\cdots,m)$
rt remainder term $rt[j](j=1,\cdots,m)$
gc coefficients of the polynomial generated from these zeros
  $gc[j](j=0,\cdots,n-i)$
m number of distinct zeros found by the routine
i degree of the residual polynomial
t coefficients of the residual polynomial
  $t[j](j=0,\cdots,i)$, where $t[i]$ is the constant term
fail a zero with multiplicity greater than *n* found, change parameters for acceptance criteria.

REFERENCES:

1. BAREISS, E. H. Resultant procedure and the mechanization of the Graeffe process, *J. ACM 7* (Oct, 1960), 346–386.
2. BAREISS, E. H. AND FISHERKELLER, M. A. Algorithm 59, Zeros of a real polynomial by resultant procedure, *Comm. ACM 4* (May 1961), 236–237.
3. THACHER, H. C. Certification of algorithm 3, *Comm. ACM 3* (June 1960), 354.
4. GRAU, A. A. Algorithm 256, Modified Graeffe method, *Comm. ACM 8* (June 1965), 379;

```
begin
  integer d,numzro;
  Boolean zero;
  numzro := 0;  zero := false;  d := n;
ZROTEST:
  if c[d] = 0 then
  begin
    zero := true;  d := d - 1;  numzro := numzro + 1;
    go to ZROTEST
  end;
  begin
    integer ct, nu, nuc, beta, j, jc, k, p, em, l, mmc, ll, me, sm;
    Boolean root;
    real x, y, gx, rp, h;
    array a, ac[0: d, 0: mm], rr, rc[0: d], s[−1: d],
      ag[0: d+1, −1: d+1], rh, q, g, f[1: 2 × d];
    switch ss := S1, S2;
    switch tt := T1, T2;
    switch vv := V1, V2;
    integer procedure min(u, v);  integer u, v;
      min := if u ≦ v then u else v;
    real procedure synd(ww, qq, ii, tt);
      integer ii;  real ww, qq;  array tt;
SYNTHETICDIV:
    begin
      s[−1] := 0;  s[0] := tt[0];
      for em := 1 step 1 until ii do
        s[em] := tt[em] − ww × s[em−1] − qq × s[em−2];
      if qq = 0 then synd := abs(s[ii])
      else synd := abs(s[ii−1] × sqrt(abs(qq))) + abs(s[ii])
    end synd;
    ct := beta := 1;
SQUARING OPERATION:
    me := mm;
    begin
      for m := 1 step 1 until mm do
      begin
        for j := 0 step 1 until d do
        begin
          h := 0;
          for ll := 1 step 1 until min(d−j,j) do
            h := h + (−1) ↑ ll × a[j−ll, m−1] × a[j+ll, m−1];
          a[j, m] := (−1) ↑ j × (a[j, m−1] ↑ 2 +2 × h)
        end;
        for l := 0 step 1 until d do
        begin
          if abs(a[l, m]) ≧ sqrt(range) then
            begin me := m;  go to W1 end
        end
      end
    end;
W1:
    for j := 0 step 1 until d do
    rr[j] :=if a[j, me] = 0 then 0 else
      (−1) ↑ j × a[j, me−1] ↑ 2/a[j, me];
    ll := 0;
    for j := d step −1 until 0 do
    begin
      if a[j, me] = 0 then
        begin ll := ll + 1;  rr[j] := ll end
      else go to W2
    end;
W2:
    j := 1;  nu := 1;
RD:
    if (1−delta≦rr[j]) ∧ (rr[j]≦1+delta) then
    begin
      rp := abs(a[j, me]/a[j−nu, me]) ↑ (1/(2↑me×nu));
      go to tt[beta]
    end;
M1:
    nu := nu + 1;
M2:
    j := j + 1;
    if j = d + 1 then go to ss[beta] else go to RD;
M3:
    nu := 1;  go to M2;
```

```
T1: rh[ct] := rp;  x := rp + epsilon × rp;
    y := x + epsilon × rp;
    for k := 0 step 1 until d do t[k] := abs(c[k]);
    f[ct] := synd(−y, 0.0, d, t) − synd(−x, 0.0, d, t);
    g[ct] := synd(−rh[ct], 0.0, d, c);
    if abs(f[ct]) > g[ct] then
    begin
      root := true;  q[ct] := 0;
      ct := ct + 1;  f[ct] := f[ct−1]
    end;
    rh[ct] := −rp;
    g[ct] := synd(−rh[ct], 0.0, d, c);
    if abs(f[ct]) > g[ct] then
    begin
      root := true;  q[ct] := 0;
      ct := ct + 1;  f[ct] := f[ct−1]
    end;
    if nu = 1 then go to M2;
    q[ct] := rp ↑ 2;  nuc := nu;  jc := j;
    mmc := me;
    for j := 0 step 1 until d do
    begin
      rc[j] := rr[j];  ac[j, me] := a[j, me]
    end;
RESULTANT:
    begin
      array b[−1:d+1, −1:d+1], aa[0:d],
        r[0:d, 0:d], cb[−1:d+1];
      cb[−1] := cb[d+ 1] := 0;
      for j := 0 step 1 until d do
        cb[j] := c[j];
      b[0, 0] := 1;
      for k := 0 step 1 until d do
      begin
        b[k, −1] := 0;  b[k−1, k] := 0;
        for j := 0 step 1 until k do
          b[k+1, j] := b[k, j−1] − q[ct] × b[k−1, j];
        b[k+1, k+1] := 1;  h := 0;
        for j := d − k step −1 until 0 do
          h :=h + (cb[j] × cb[k+j] − cb[j−1]
          × cb[k+j+1]) × q[ct] ↑ (d−k−j);
        aa[k] := (−1) ↑ k × h;
        for j := 0 step 1 until k − 1 do
          r[k, j] := r[k−1, j] + aa[k] × b[k, j];
        r[k, k] := aa[k]
      end;
      beta := 2;
      for j := 0 step 1 until d do
        a[j, 0] := r[d, d−j]/r[d, d]
    end;
    go to SQUARING OPERATION;
T2:
    if (rp/2) ↑ 2 > q[ct] then go to M3;
    rh[ct] := rp;
    g[ct] := synd(−rh[ct], q[ct], d, c);
    if abs(f[ct]) > g[ct] then
    begin
      ct := ct + 1;  f[ct] := f[ct−1];
      q[ct] := q[ct−1]
    end;
    rh[ct] := −rp;
    g[ct] := synd(−rh[ct], q[ct], d, c);
    if abs(f[ct]) > g[ct] then
    begin
      ct := ct + 1;  f[ct] := f[ct−1];
      q[ct] := q[ct−1]
    end;
    go to M3;
```

```
S2:
    me := mmc;
    for j := 0 step 1 until d do
    begin
      a[j, me] := ac[j, me];  rr[j] := rc[j]
    end;
    j := jc;  beta := 1;
    if root then go to M3 else nu := nuc;
    go to M1;
S1:
    for j := 0 step 1 until d do ag[j, 0] := 1;
    for j := -1, 1 step 1 until d do
    for m := 0 step 1 until d do
      ag[m, j] := 0;
    k := 1;  i := d;  m := 1;  ll := 0;
    for j := 0 step 1 until d do t[j] := c[j];
MULT:
    mu[m] := 0;
    p := if q[k] = 0 then 1 else 2;
IT:
    gx := synd(-rh[k], q[k], i, t);
    if abs(f[k]) > gx then
    begin
      ll := ll + p;
      for j := 1 step 1 until ll do
        ag[ll, j] := ag[ll-p, j] - rh[k] × ag[ll-p, j-1] + q[k] ×
          ag[ll-p, j-2];
      mu[m] := mu[m] + p;  i := i - p;
      if i < 0 then go to fail;
      if i = 0 then go to E1;
      for j := 0 step 1 until i do t[j] := s[j];
      go to IT
    end
    else if mu[m] ≠ 0 then
E1:
    begin
      rt[m] := g[k];  go to vv[p];
    end
    else go to D1;
V1:
    re[m] := rh[k];  im[m] := 0;  go to E;
V2:
    re[m] := rh[k]/2;
    im[m] := sqrt(q[k] - re[m] ↑ 2);
E:
    m := m + 1;
D1:
    k := k + 1;
    sm := 0;
    if m ≠ 1 then
    for j := 1 step 1 until m - 1 do sm := sm + mu[j];
    if k ≤ ct ∧ sm ≤ d ∧ i > 0 then go to MULT;
    for j := 0 step 1 until d do gc[j] := ag[ll, j];
    m := m - 1;
    if zero then
    begin
      for j := d + 1 step 1 until d + numzro do gc[j] := 0;
      m := m + 1;
      re[m] := 0;  im[m] := 0;  mu[m] := numzro;  rt[m] := 0
    end
  end
end AG4
```

REMARK ON ALGORITHM 340 [C2]
ROOTS OF POLYNOMIALS BY A ROOT-SQUARING AND RESULTANT ROUTINE [Albert Noltemeier, *Comm. ACM 11* (Nov. 1968), 779]

Albert Noltemeier (Recd. 6 Jan. 1969)
Technische Universität Hannover, Rechenzentrum, Hannover, Germany

KEY WORDS AND PHRASES: rootfinders, roots of polynomial equations, polynomial zeros, root-squaring operations, Graeffe method, resultant procedure, subresultant procedure, testing of roots, acceptance criteria
CR CATEGORIES: 5.15

The following misprints were found in the algorithm and should be corrected as indicated:

1. In the comment, in the first column on page 780, the last line before the paragraph beginning with the word "Parameters" ends with a semicolon; it should end with a period.
2. In the seventh line following the word "Parameters" the abbreviation CDC should appear in capital letters.
3. In the procedure body, in the second column on page 780, the line before the label *SQUARING OPERATION* is missing. It should read as follows:

```
for j := 0 step 1 until d do a[j, 0] := c[j];
```

ALGORITHM 341
SOLUTION OF LINEAR PROGRAMS IN 0-1 VARIABLES BY IMPLICIT ENUMERATION [H]

J. L. Byrne and L. G. Proll

(Recd. 8 Nov. 1967 and 17 June 1968)

Department of Mathematics, University of Southampton, Hampshire, England

KEY WORDS AND PHRASES: linear programming, zero-one variables, partial enumeration
CR CATEGORIES: 5.41

```
procedure IMPLEN (m, n, A, x, api, nosoln, count, inf);
  value m, n, inf;  integer m, n, count;  real inf;
  Boolean api, nosoln;  real array A; integer array x;
comment  This procedure solves the integer linear program,
  minimize    A[0, 1] × x[1] + ··· + A[0, n] × x[n]
  subject to  A[i, 1] × x[1] + ··· + A[i, n] × x[n]
                                  + A[i, 0] ≧ 0  (i=1, 2, ··· , m)
  and         x[j] = 0 or 1  (j=1, 2, ··· , n).
  It is assumed that A[0, j] ≧ 0  (j=1, 2, ··· , n). The algorithm
  used is that of Geoffrion (SIAM Rev. 9, No. 2). On entry, inf
  is the largest positive real number available and api is set to
  true if a priori information concerning the solution is supplied
  in the form of a binary vector x[1: n] and its associated cost
  A[0, 0]. On exit nosoln is true if no feasible solution to the con-
  straints has been found, otherwise it is false and x contains the
  optimal solution, A[0, 0] contains the optimal value of the ob-
  jective function and A[i, 0] contains the values of the slack
  variables. In either case count contains the number of iterations
  performed;
begin
  integer i, j, k, ia, e, d;  real z, q, max, r;  Boolean null;
  integer array s, v[1: n];
  comment  s holds the current partial solution in order of as-
    signment, v is a state vector associated with s;
  if api then
  begin
    for j := 1 step 1 until n do
    if x[j] = 0 then begin s[j] := -j;  v[j] := 2 end
    else
    begin
      s[j] := j;  v[j] := 3;
      for i := 1 step 1 until m do
        A[i, 0] := A[i, 0] + A[i, j]
    end;
    e := n;  z := A[0, 0];  go to L0
  end;
  for j := 1 step 1 until n do s[j] := v[j] := 0;
  z := 0.0;  e := 0;
L0:  nosoln := true;  count := 0;  A[0, 0] := inf;
  comment  all relevant variables are now initialized;
START:  count := count + 1;
  for i := 1 step 1 until m do
    if A[i, 0] < 0.0 then go to FORMT;
  comment  best completion of s is feasible;
  go to INCUMBENT;
FORMT:  null := true;
  comment  form set T of free variables to which 1 may be profit-
    ably assigned;
  for j := 1 step 1 until n do
  begin
    if ¬ (v[j] = 0 ∧ A[0, j] + z < A[0, 0]) then go to L1;
    for k := i step 1 until m do
    if A[k, 0] < 0.0 ∧ A[k, j] > 0.0 then
      begin null := false;  v[j] := 1;  go to L1 end;
L1:  end;
  if null then go to NEWS;
  comment  if T is empty then s is fathomed;
  for k := i step 1 until m do
  begin
    if A[k, 0] ≧ 0.0 then go to L2;
    q := A[k, 0];
    for j := 1 step 1 until n do
      if v[j] = 1 ∧ A[k, j] > 0.0 then q := q + A[k, j];
    if q < 0.0 then go to NEWS;
    comment  if q is negative s is fathomed;
L2:  end;
  max := -inf;
  for j := 1 step 1 until n do
  begin
    if v[j] ≠ 1 then go to L3;  q := 0.0;
    for i := 1 step 1 until m do
    begin
      r := A[i, 0] + A[i, j];
      if r < 0.0 then q := q + r
    end;
    if max ≦ q then
      begin max := q;  d := j end;
L3:  end;
  e := e + 1;  s[e] := d;  v[d] := 3;  ia := 1;
  comment  Augment s by assigning 1 to x[d];
RESET:  for j := 1 step 1 until n do
    if v[j] = 1 then v[j] := 0;
  comment  clear T;
  for i := 1 step 1 until m do
    A[i, 0] := A[i, 0] + ia × A[i, d];
  z := z + ia × A[0, d];
  comment  Recalculate slacks and objective function;
  go to START;
INCUMBENT:  nosoln := false;
  if z ≧ A[0, 0] then go to NEWS;
  A[0, 0] := z;
  if api then begin api := false; go to L4 end;
  for j := 1 step 1 until n do
    x[j] := if v[j] = 3 then 1 else 0;
NEWS:  if e = 0 then go to RESULT;
L4:  d := s[e];
  if d > 0 then go to UNDERLINE;
  v[-d] := 0;  e := e - 1;  comment  backtrack;
  go to NEWS;
UNDERLINE:  s[e] := -d;  v[d] := 2;  ia := -1;
  comment  Assign 0 to x[d];
  go to RESET;
RESULT:
end
```

REMARK ON ALGORITHM 341 [H]
SOLUTION OF LINEAR PROGRAMS IN 0-1 VARIABLES BY IMPLICIT ENUMERATION

[J. L. Byrne and L. G. Proll, *Comm. ACM 11* (Nov. 1968), 782]

L. G. Proll (Recd. 5 Dec. 1968 and 18 Aug. 1969)
University of Southampton, Department of Mathematics, Hampshire, England

KEY WORDS AND PHRASES: linear programming, zero-one variables, partial enumeration
CR CATEGORIES: 5.41

The published algorithm contains an error in the assembly of the initial partial solution, *s*, if a priori information is given. In certain cases this can cause premature termination of the algorithm. The error may be corrected by replacing the following lines of the procedure body, from

```
begin
  for j := 1 step 1 until n do
```

to

```
  e := n;  z := A[0, 0];  go to L0;
```

by

```
begin
  e := 0;
  for j := 1 step 1 until n do
    if x[j] = 0 then v[j] := 0
  else
  begin
    e := e + 1;     s[e] := j;  v[j] := 3;
    for i := 1 step 1 until m do
      A[i, 0] := A[i, 0] + A[i, j];
  end;
  z := A[0, 0];  go to L0;
```

and by deleting the line

```
if api then begin api := false;  go to L4 end;
```

REMARK ON ALGORITHM 341 [H]
SOLUTION OF LINEAR PROGRAMS IN 0–1 VARIABLES BY IMPLICIT ENUMERATION

[J. L. Bryne and L. G. Proll *Comm. ACM 11* (Nov. 1968), 782]

M. M. Guignard (Recd. 21 Mar. 1969 and 17 Nov. 1969)
Laboratoire de Calcul, 13 Place Philippe Lebon, Lille, France

KEY WORDS AND PHRASES: linear programming, zero-one variables, partial enumeration
CR CATEGORIES: 5.41

There is an error in the procedure; the slack variables are destroyed during computation. It is necessary then to declare an array *slacks* local to the procedure, and to return the final slacks in

$$A[i, 0], \qquad i = 1, 2, \cdots, m.$$

One could correct the program as follows. Add before second **comment:**

```
real array slacks [1:m];
```

add before *NEWS*:

```
for i := 1 step 1 until m do
slacks [i] := A[i, 0];
```

add after *RESULT*:

```
if ¬ nosoln then
for i := 1 step 1 until m do
A[i, 0] := slacks [i];
```

ALGORITHM 342
GENERATOR OF RANDOM NUMBERS SATISFYING THE POISSON DISTRIBUTION [G5]

Richard H. Snow (Recd. 20 Dec. 1966, 24 Aug. 1967, 5 Feb. 1968, 26 Mar. 1968, 5 June 1968 and 9 Sept. 1968)
IIT Research Institute, Chicago, Ill. 60616

KEY WORDS AND PHRASES: Poisson distribution, random number generator, Monte Carlo
CR CATEGORIES: 5.12, 5.5

```
integer procedure poisson carlo (npx, npx1, random); value npx,
  random;  real npx, npx1, random;
```

comment The Poisson distribution gives the probability that *px* events will occur in a certain interval or volume, where the expected or mean value of events is *npx*. Applications are described by B. W. Lindgren and G. W. McElrath [1]. For a Monte Carlo calculation we wish to generate numbers *px* that satisfy the Poisson distribution, that is to find the inverse of the Poisson function. To do this we generate a pseudo-random number in the interval 0, 1 and find the number *px* such that *random* ≤ (probability that the number is *px* or less) and *random* > (the probability that the number is *px* − 1 or less).

poisson carlo returns the value −1 to signal that the procedure was called with a value of *npx* < 0 or too large for the precision of the computer. It is the responsibility of the user to test the calculated value if there is any possibility of the occurrence of the error condition.

In order to save computing time, values of the Poisson distribution computed at a previous entry for the same value of *npx* are stored in the **own array** *pson*. The previous value of *npx* is *npx*1. The actual parameter corresponding to *npx*1 must be a real identifier, not a constant or an expression. Before the first call of *poisson carlo* the calling program must set *npx*1 to a value ≠ *npx*. The number of *pson* elements that were previously computed and stored is computed. If it is desired to save storage space at the expense of computing time, the upper bound 84 of *pson* may be reduced, but then the limit of *computed* near the end of the procedure must also be decreased accordingly.

The procedure which generates *random* is preferably algorithm 266 [3] or 294 [2]. It can be called as the actual parameter in the procedure call of *poisson carlo*.

The author thanks Mr. I. D. Hill for numerous suggestions and corrections which greatly improved the algorithm.

References:

1. Lindgren, B. W., and McElrath, G. W. *Introduction to Probability and Statistics*, 2 ed. Macmillan, New York, 1966, pp. 64–68.
2. Pike, M. C., and Hill, I. D. Algorithm 266, pseudo-random numbers. *Comm. ACM 8* (Oct. 1965), 605.
3. Strome, W. M. Algorithm 294, uniform random. *Comm. ACM 10* (Jan. 1967), 40;

```
begin
  own integer computed;  own real pnc;
    own real array pson [0:84];
   integer n;  real ps;
   if npx < 0 then go to error;
   if npx ≠ npx1 then
   begin
    computed := 0;
    pnc := pson [0] := exp (−npx);
    if pnc = 0 then go to error;
    comment  pson [0] is the probability that poisson carlo = 0.
      It cannot be zero unless −npx underflows the argument
      range of procedure exp. For most computers this sets an
      upper limit of 85 for npx;
    npx1 := npx
  end new npx;
  ps := pson [computed];
  if random ≦ ps then
  begin
    integer nmin, nmax;
    comment  The probability term can be found by searching
      the stored values;
    nmin := 0;  nmax := computed + 1;
    for n := (nmax+nmin−1) ÷ 2 while nmax − nmin > 1 do
      if random > pson[n] then nmin := n + 1 else nmax := n + 1;
    poisson carlo := nmin
  end search
  else
  begin
    real psc, pn;  pn := pnc;
    comment  Additional probability terms must be computed;
    for n := computed + 1, n + 1 while random > ps do
    begin
      pn := pn × npx/n;
      psc := ps;  ps := ps + pn;
      comment  ps = cumulative probability of terms up to n,
        and pn = probability of nth term;
      if ps = psc then go to error;
      if n ≦ 84 then begin pson[n] := ps;
      pnc := pn;  computed := n end;
      poisson carlo := n
    end
  end more;
  go to fin;
error:  poisson carlo := −1;
fin:
end  poisson carlo;
comment  The following is an example of a calling program for
  the case where poisson carlo is compiled within the calling
  program rather than separately. Instead of own variables,
  non-local variables may then be used. The program is within
  the IFIP subset if this change is made, and if the expression
  (nmax+nmin−1) ÷ 2 is replaced by the less efficient expression
  .501× (nmax+nmin−2);
begin
  integer x, computed;  real array pson [0:84];
  real pnc, npx, npx1;
  real procedure random (x);
   comment  Procedure body random is inserted here;
   integer procedure poisson carlo (npx, npx1, random);
   comment  Procedure body of poisson carlo is inserted here
     after deleting declarations of own variables;
  ininteger (2, x);  npx1 := −1;
in1:  inreal (2, npx);
  outinteger (1, poisson carlo (npx, npx1, random (x)));
  go to in1
end
```

ALGORITHM 343
EIGENVALUES AND EIGENVECTORS OF A REAL GENERAL MATRIX [F2]

J. Grad and M. A. Brebner

(Recd. 12 Oct. 1967, 1 July 1968 and 8 July 1968)

Computer Services, University of Birmingham, Birmingham 15, England

KEY WORDS AND PHRASES: eigenvalues, eigenvectors, latent roots, latent vectors, Householder's method, QR algorithm, inverse iteration

CR CATEGORIES: 5.14

Abstract:

Purpose. This subroutine finds all the eigenvalues and eigenvectors of a real general matrix. The eigenvalues are computed by the QR double-step method and the eigenvectors by inverse iteration.

Method. Firstly the following preliminary modifications are carried out to improve the accuracy of the computed results. (i) The matrix is scaled by a sequence of similarity transformations so that the absolute sums of corresponding rows and columns are roughly equal. (ii) The scaled matrix is normalized so that the value of the Euclidean norm is equal to one.

The main part of the process commences with the reduction of the matrix to an upper-Hessenberg form by means of similarity transformations (Householder's method). Then the QR double-step iterative process is performed on the Hessenberg matrix until all elements of the subdiagonal that converge to zero are in modulus less than $2^{-t} \| H \|_E$, where t is the number of significant digits in the mantissa of a binary floating-point number. The eigenvalues are then extracted from this reduced form.

Inverse iteration is performed on the upper-Hessenberg matrix until the absolute value of the largest component of the right-hand side vector is greater than the bound $2^t/(100\,N)$, where N is the order of the matrix. Normally after this bound is achieved, one step more is performed to obtain the computed eigenvector, but at each step the residuals are computed, and if the residuals of one particular step are greater in absolute value than the residuals of the previous step, then the vector of the previous step is accepted as the computed eigenvector.

Program. The subroutine EIGENP is completely self-contained (composed of five subroutines

EIGENP, SCALE, HESQR, REALVE, and COMPVE)

and communication to it is solely through the argument list. The entrance to the subroutine is achieved by

CALL EIGENP (N, NM, A, T, EVR, EVI, VECR, VECI, INDIC)

The meaning of the parameters is described in the comments at the beginning of the subroutine EIGENP.

References:

1. Wilkinson, J. H. *The Algebraic Eigenvalue Problem.* Clarendon Press, Oxford, 1965, pp. 347–353, 485–567, 619–633.

Test results. All tests have been performed on a KDF9 computer (t = 39). No breakdown of the method has occurred and in general very accurate computed eigenvalues and eigenvectors have been obtained.

Some examples:

(i) The matrix

$$\begin{bmatrix} -.5 & -1 & -1 & -.5 & -1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

has all eigenvalues with modulus equal to one. The computed eigenvalues are
$-1.00000\,0000$, $-.25000\,00000 \pm i.96824\,58366$, $.50000\,00000 \pm i.86602\,54038$.

The computed eigenvectors are

x_1	x_2 , x_3	x_4 , x_5
.447213595	1.000000000	$-.500000000 \mp i.866025404$
$-.447213595$	$-.250000000 \mp i.968245837$	$-1.000000000 \mp i.16E\text{-}10$
.447213595	$-.875000000 \pm i.484122918$	$-.500000000 \pm i.866025404$
$-.447213595$	$.687500000 \pm i.726184377$	$.500000000 \pm i.866025404$
.447213595	$.531250000 \mp i.847215107$	1.000000000

and the computed residuals are in modulus less than $.3E - 10$.

(ii) The matrix

$$\begin{bmatrix} -2 & 1 & 1 & 1 \\ -7 & -5 & -2 & -4 \\ 0 & -1 & -3 & -2 \\ -1 & 0 & -1 & 0 \end{bmatrix}$$

has the eigenvalues
$-4 \pm i2$ and $-1 \pm \sqrt{2}$.

The computed eigenvalues are
$-4.000000000 \pm i2.000000000$, -2.414213562, $.4142135624$.

The computed eigenvectors are

x_1 , x_2	x_3	x_4
$-.2000000000 \mp i.4000000000$	$.60E\text{-}12$	$-.12E\text{-}11$
1.000000000	$-.7941044878$	.4759631495
$.2000000000 \pm i.4000000000$	.5615166683	.3365567706
$.14E\text{-}10 \quad \pm i.63E\text{-}11$	.2325878195	$-.8125199201$

and the computed residuals are in modulus less than $.7E - 10$.

(iii) The matrix A

$$A = \begin{bmatrix} 1 & 0 & 0.01 \\ 0.1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

is transformed by the process of scaling into the form B

$$B = \begin{bmatrix} .574423 & 0 & .066333 \\ .053454 & .574423 & 0 \\ 0 & .053454 & .574423 \end{bmatrix}$$

with the elements given to six decimal places. The obtained matrix B is essentially invariant under the QR double-step process. This kind of trouble was overcome by introducing the statements

```
R = DABS(X) + DABS(Y)
IF(R.EQ.0.0)SHIFT = A(M,M-1)
IF(R.EQ.0.0)GO TO 21
```

in the subroutine HESQR.

The exact eigenvalues of A are
1.1, $0.95 \pm i0.5\sqrt{0.03}$.

The computed eigenvalues are
1.100000000, 0.9500000000 ± i0.0866025404.

Acknowledgments. The authors wish to thank Dr. K. A. Redish, the former director of Computer Services at the University of Birmingham, and Dr. S. H. Hollingdale, the present director of Computer Services, for their encouragement. Finally, the authors are indebted to Dr. J. H. Wilkinson, National Physical Laboratory, Teddington, for useful consultations and suggestions.

```
      SUBROUTINE EIGENP(N,NM,A,T,EVR,EVI,VECR,VECI,INDIC)
      DOUBLE PRECISION D1,D2,D3,PRFACT
      INTEGER I,IVEC,J,K,K1,KON,L,L1,M,N,NM
      REAL ENORM,EPS,EX,R,R1,T
      DIMENSION A(NM,1),VECR(NM,1),VECI(NM,1),
     1EVR(NM),EVI(NM),INDIC(NM)
      DIMENSION IWORK(100),LOCAL(100),PRFACT(100)
     1,SUBDIA(100),WORK1(100),WORK2(100),WORK(100)
C
C THIS SUBROUTINE FINDS ALL THE EIGENVALUES AND THE
C EIGENVECTORS OF A REAL GENERAL MATRIX OF ORDER N.
C
C FIRST IN THE SUBROUTINE SCALE THE MATRIX IS SCALED SO THAT
C THE CORRESPONDING ROWS AND COLUMNS ARE APPROXIMATELY
C BALANCED AND THEN THE MATRIX IS NORMALISED SO THAT THE
C VALUE OF THE EUCLIDIAN NORM OF THE MATRIX IS EQUAL TO ONE.
C
C THE EIGENVALUES ARE COMPUTED BY THE QR DOUBLE-STEP METHOD
C IN THE SUBROUTINE HESQR.
C THE EIGENVECTORS ARE COMPUTED BY INVERSE ITERATION IN
C THE SUBROUTINE REALVE,FOR THE REAL EIGENVALUES,OR IN THE
C SUBROUTINE COMPVE,FOR THE COMPLEX EIGENVALUES.
C
C THE ELEMENTS OF THE MATRIX ARE TO BE STORED IN THE FIRST N
C ROWS AND COLUMNS OF THE TWO DIMENSIONAL ARRAY A. THE
C ORIGINAL MATRIX IS DESTROYED BY THE SUBROUTINE.
C N IS THE ORDER OF THE MATRIX.
C NM DEFINES THE FIRST DIMENSION OF THE TWO DIMENSIONAL
C ARRAYS A,VECR,VECI AND THE DIMENSION OF THE ONE
C DIMENSIONAL ARRAYS EVR,EVI AND INDIC. THEREFORE THE
C CALLING PROGRAM SHOULD CONTAIN THE FOLLOWING DECLARATION
C      DIMENSION A(NM,NN),VECR(NM,NN),VECI(NM,NN),
C     1EVR(NM),EVI(NM),INDIC(NM)
C WHERE NM AND NN ARE ANY NUMBERS EQUAL TO OR GREATER THAN N
C THE UPPER LIMIT FOR NM IS EQUAL TO 100 BUT MAY BE
C INCREASED TO THE VALUE MAX BY REPLACING THE DIMENSION
C STATEMENT
C      DIMENSION IWORK(100),LOCAL(100), ... ,WORK(100)
C IN THE SUBROUTINE EIGENP WITH
C      DIMENSION IWORK(MAX),LOCAL(MAX), ... ,WORK(MAX)
C NM AND NN ARE OF COURSE BOUNDED BY THE SIZE OF THE STORE.
C
C THE REAL PARAMETER T MUST BE SET EQUAL TO THE NUMBER OF
C BINARY DIGITS IN THE MANTISSA OF A SINGLE PRECISION
C FLOATING-POINT NUMBER.
C
C THE REAL PARTS OF THE N COMPUTED EIGENVALUES WILL BE FOUND
C IN THE FIRST N PLACES OF THE ARRAY EVR AND THE IMAGINARY
C PARTS IN THE FIRST N PLACES OF THE ARRAY EVI.
C THE REAL COMPONENTS OF THE NORMALISED EIGENVECTOR I
C (I=1,2,...,N) CORRESPONDING TO THE EIGENVALUE STORED IN
C EVR(I) AND EVI(I) WILL BE FOUND IN THE FIRST N PLACES OF
C THE COLUMN I OF THE TWO DIMENSIONAL ARRAY VECR AND THE
C IMAGINARY COMPONENTS IN THE FIRST N PLACES OF THE COLUMN I
C OF THE TWO DIMENSIONAL ARRAY VECI.
C
C THE REAL EIGENVECTOR IS NORMALISED SO THAT THE SUM OF THE
C SQUARES OF THE COMPONENTS IS EQUAL TO ONE.
C THE COMPLEX EIGENVECTOR IS NORMALISED SO THAT THE
C COMPONENT WITH THE LARGEST VALUE IN MODULUS HAS ITS REAL
C PART EQUAL TO ONE AND THE IMAGINARY PART EQUAL TO ZERO.
C
C THE ARRAY INDIC INDICATES THE SUCCESS OF THE SUBROUTINE
C EIGENP AS FOLLOWS
C      VALUE OF INDIC(I)      EIGENVALUE I     EIGENVECTOR I
C             0                 NOT FOUND        NOT FOUND
C             1                 FOUND            NOT FOUND
C             2                 FOUND            FOUND
C
C
      IF(N.NE.1)GO TO 1
      EVR(1) = A(1,1)
      EVI(1) = 0.0
      VECR(1,1) = 1.0
      VECI(1,1) = 0.0
      INDIC(1) = 2
      GO TO 25
C
    1 CALL SCALE(N,NM,A,VECI,PRFACT,ENORM)
C THE COMPUTATION OF THE EIGENVALUES OF THE NORMALISED
C MATRIX.
      EX = EXP(-T*ALOG(2.0))
      CALL HESQR(N,NM,A,VECI,EVR,EVI,SUBDIA,INDIC,EPS,EX)
C
C THE POSSIBLE DECOMPOSITION OF THE UPPER-HESSENBERG MATRIX
C INTO THE SUBMATRICES OF LOWER ORDER IS INDICATED IN THE
C ARRAY LOCAL. THE DECOMPOSITION OCCURS WHEN SOME
C SUBDIAGONAL ELEMENTS ARE IN MODULUS LESS THAN A SMALL
C POSITIVE NUMBER EPS DEFINED IN THE SUBROUTINE HESQR . THE
C AMOUNT OF WORK IN THE EIGENVECTOR PROBLEM MAY BE
C DIMINISHED IN THIS WAY.
      J = N
      I = 1
      LOCAL(1) = 1
      IF(J.EQ.1)GO TO 4
    2 IF(ABS(SUBDIA(J-1)).GT.EPS)GO TO 3
      I = I+1
      LOCAL(I)=0
    3 J = J-1
      LOCAL(I)=LOCAL(I)+1
      IF(J.NE.1)GO TO 2
C
C THE EIGENVECTOR PROBLEM.
    4 K = 1
      KON = 0
      L = LOCAL(1)
      M = N
      DO 10 I=1,N
        IVEC = N-I+1
        IF(I.LE.L)GO TO 5
        K = K+1
        M = N-L
        L = L+LOCAL(K)
    5   IF(INDIC(IVEC).EQ.0)GO TO 10
        IF(EVI(IVEC).NE.0.0)GO TO 8
C
C TRANSFER OF AN UPPER-HESSENBERG MATRIX OF THE ORDER M FROM
C THE ARRAYS VECI AND SUBDIA INTO THE ARRAY A.
        DO 7 K1=1,M
          DO 6 L1=K1,M
    6       A(K1,L1) = VECI(K1,L1)
          IF(K1.EQ.1)GO TO 7
          A(K1,K1-1) = SUBDIA(K1-1)
    7     CONTINUE
C
C THE COMPUTATION OF THE REAL EIGENVECTOR IVEC OF THE UPPER-
C HESSENBERG MATRIX CORRESPONDING TO THE REAL EIGENVALUE
C EVR(IVEC).
        CALL REALVE(N,NM,M,IVEC,A,VECR,EVR,EVI,IWORK,
     1  WORK,INDIC,EPS,EX)
        GO TO 10
C
C THE COMPUTATION OF THE COMPLEX EIGENVECTOR IVEC OF THE
C UPPER-HESSENBERG MATRIX CORRESPONDING TO THE COMPLEX
C EIGENVALUE EVR(IVEC) + I*EVI(IVEC). IF THE VALUE OF KON IS
C NOT EQUAL TO ZERO THEN THIS COMPLEX EIGENVECTOR HAS
C ALREADY BEEN FOUND FROM ITS CONJUGATE.
    8   IF(KON.NE.0)GO TO 9
        KON = 1
        CALL COMPVE(N,NM,M,IVEC,A,VECR,VECI,EVR,EVI,INDIC,
     1  IWORK,SUBDIA,WORK1,WORK2,WORK,EPS,EX)
        GO TO 10
    9   KON = 0
   10   CONTINUE
C
C THE RECONSTRUCTION OF THE MATRIX USED IN THE REDUCTION OF
C MATRIX A TO AN UPPER-HESSENBERG FORM BY HOUSEHOLDER METHOD
        DO 12 I=1,N
          DO 11 J=I,N
            A(I,J) = 0.0
   11       A(J,I) = 0.0
   12     A(I,I) = 1.0
        IF(N.LE.2)GO TO 15
        M = N-2
        DO 14 K=1,M
          L = K+1
          DO 14 J=2,N
            D1 = 0.0
            DO 13 I=L,N
              D2 = VECI(I,K)
   13         D1 = D1+ D2*A(J,I)
            DO 14 I=L,N
   14         A(J,I) = A(J,I)-VECI(I,K)*D1
C
C THE COMPUTATION OF THE EIGENVECTORS OF THE ORIGINAL NON-
C SCALED MATRIX.
   15 KON = 1
        DO 24 I=1,N
          L = 0
          IF(EVI(I).EQ.0.0)GO TO 16
          L = 1
          IF(KON.EQ.0)GO TO 16
          KON = 0
          GO TO 24
   16     DO 18 J=1,N
            D1 = 0.0
            D2 = 0.0
            DO 17 K=1,N
              D3 = A(J,K)
              D1 = D1+D3*VECR(K,I)
              IF(L.EQ.0)GO TO 17
          D2 = D2+D3*VECR(K,I-1)
   17     CONTINUE
        WORK(J) = D1/PRFACT(J)
        IF(L.EQ.0)GO TO 18
        SUBDIA(J)=D2/PRFACT(J)
   18   CONTINUE
C
C THE NORMALISATION OF THE EIGENVECTORS AND THE COMPUTATION
C OF THE EIGENVALUES OF THE ORIGINAL NON-NORMALISED MATRIX.
        IF(L.EQ.1)GO TO 21
        D1 = 0.0
        DO 19 M=1,N
   19     D1 = D1+WORK(M)**2
        D1 = DSQRT(D1)
        DO 20 M=1,N
          VECI(M,I) = 0.0
   20     VECR(M,I) = WORK(M)/D1
        EVR(I) = EVR(I)*ENORM
        GO TO 24
C
   21   KON = 1
        EVR(I) = EVR(I)*ENORM
        EVR(I-1) = EVR(I)
        EVI(I) = EVI(I)*ENORM
        EVI(I-1) =-EVI(I)
        R = 0.0
        DO 22 J=1,N
          R1 = WORK(J)**2 + SUBDIA(J)**2
          IF(R.GE.R1)GO TO 22
          R = R1
          L = J
   22     CONTINUE
        D3 = WORK(L)
        R1 = SUBDIA(L)
        DO 23 J=1,N
          D1 = WORK(J)
          D2 = SUBDIA(J)
          VECR(J,I) = (D1*D3+D2*R1)/R
          VECI(J,I) = (D2*D3-D1*R1)/R
          VECR(J,I-1) = VECR(J,I)
   23     VECI(J,I-1) =-VECI(J,I)
   24   CONTINUE
C
   25 RETURN
      END


      SUBROUTINE SCALE(N,NM,A,H,PRFACT,ENORM)
      DOUBLE PRECISION COLUMN,FACTOR,FNORM,PRFACT,Q,ROW
      INTEGER I,J,ITER,N,NCOUNT,NM
      REAL BOUND1,BOUND2,ENORM
      DIMENSION A(NM,1),H(NM,1),PRFACT(NM)
C
C THIS SUBROUTINE STORES THE MATRIX OF THE ORDER N FROM THE
C ARRAY A INTO THE ARRAY H. AFTERWARD THE MATRIX IN THE
C ARRAY A IS SCALED SO THAT THE QUOTIENT OF THE ABSOLUTE SUM
C OF THE OFF-DIAGONAL ELEMENTS OF COLUMN I AND THE ABSOLUTE
C SUM OF THE OFF-DIAGONAL ELEMENTS OF ROW I LIES WITHIN THE
C VALUES OF 9OUND1 AND BOUND2.
C THE COMPONENT I OF THE EIGENVECTOR OBTAINED BY USING THE
C SCALED MATRIX MUST BE DIVIDED BY THE VALUE FOUND IN THE
C PRFACT(I) OF THE ARRAY PRFACT. IN THIS WAY THE EIGENVECTOR
C OF THE NON-SCALED MATRIX IS OBTAINED.
C
C AFTER THE MATRIX IS SCALED IT IS NORMALISED SO THAT THE
C VALUE OF THE EUCLIDIAN NORM IS EQUAL TO ONE.
C IF THE PROCESS OF SCALING WAS NOT SUCCESSFUL THE ORIGINAL
C MATRIX FROM THE ARRAY H WOULD BE STORED BACK INTO A AND
C THE EIGENPROBLEM WOULD BE SOLVED BY USING THIS MATRIX.
C NM DEFINES THE FIRST DIMENSION OF THE ARRAYS A AND H. NM
C MUST BE GREATER OR EQUAL TO N.
C THE EIGENVALUES OF THE NORMALISED MATRIX MUST BE
C MULTIPLIED BY THE SCALAR ENORM IN ORDER THAT THEY BECOME
C THE EIGENVALUES OF THE NON-NORMALISED MATRIX.
C
      DO 2 I=1,N
        DO 1 J=1,N
    1     H(I,J) = A(I,J)
    2   PRFACT(I)= 1.0
      BOUND1 = 0.75
      BOUND2 = 1.33
      ITER = 0
    3 NCOUNT = 0
      DO 8 I=1,N
        COLUMN = 0.0
        ROW = 0.0
        DO 4 J=1,N
          IF(I.EQ.J)GO TO 4
          COLUMN = COLUMN+ ABS(A(J,I))
          ROW = ROW +  ABS(A(I,J))
    4     CONTINUE
        IF(COLUMN.EQ.0.0)GO TO 5
        IF(ROW.EQ.0.0)GO TO 5
        Q = COLUMN/ROW
        IF(Q.LT.BOUND1)GO TO 6
        IF(Q.GT.BOUND2)GO TO 6
```

```
    5   NCOUNT = NCOUNT + 1
        GO TO 8
    6   FACTOR = DSQRT(Q)
        DO 7 J=1,N
          IF(I.EQ.J)GO TO 7
          A(I,J) = A(I,J)*FACTOR
          A(J,I) = A(J,I)/FACTOR
    7     CONTINUE
        PRFACT(I) = PRFACT(I)*FACTOR
    8   CONTINUE
      ITER = ITER+1
      IF(ITER.GT.30)GO TO 11
      IF(NCOUNT.LT.N)GO TO 3
C
      FNORM = 0.0
      DO 9 I=1,N
        DO 9 J=1,N
          Q = A(I,J)
    9     FNORM = FNORM+Q*Q
      FNORM = DSQRT(FNORM)
      DO 10 I=1,N
        DO 10 J=1,N
   10     A(I,J)=A(I,J)/FNORM
      ENORM = FNORM
      GO TO 13
C
   11 DO 12 I=1,N
        DO 12 J=1,N
   12     A(I,J) = H(I,J)
      ENORM = 1.0
C
   13 RETURN
      END


      SUBROUTINE HESQR(N,NM,A,H,EVR,EVI,SUBDIA,INDIC,EPS,EX)
      DOUBLE PRECISION S,SR,SR2,X,Y,Z
      INTEGER I,J,K,L,M,MAXST,M1,N,NM,NS
      REAL EPS,EX,R,SHIFT,T
      DIMENSION A(NM,1),H(NM,1),EVR(NM),EVI(NM),SUBDIA(NM)
      DIMENSION INDIC(NM)
C
C THIS SUBROUTINE FINDS ALL THE EIGENVALUES OF A REAL
C GENERAL MATRIX. THE ORIGINAL MATRIX A OF ORDER N IS
C REDUCED TO THE UPPER-HESSENBERG FORM H BY MEANS OF
C SIMILARITY TRANSFORMATIONS(HOUSEHOLDER METHOD). THE MATRIX
C H IS PRESERVED IN THE UPPER HALF OF THE ARRAY H AND IN THE
C ARRAY SUBDIA. THE SPECIAL VECTORS USED IN THE DEFINITION
C OF THE HOUSEHOLDER TRANSFORMATION MATRICES ARE STORED IN
C THE LOWER PART OF THE ARRAY H.
C NM IS THE FIRST DIMENSION OF THE ARRAYS A AND H. NM MUST
C BE EQUAL TO OR GREATER THAN N.
C THE REAL PARTS OF THE N EIGENVALUES WILL BE FOUND IN THE
C FIRST N PLACES OF THE ARRAY EVR,AND
C THE IMAGINARY PARTS IN THE FIRST N PLACES OF THE ARRAY EVI
C THE ARRAY INDIC INDICATES THE SUCCESS OF THE ROUTINE AS
C FOLLOWS
C     VALUE OF INDIC(I)      EIGENVALUE I
C           0                  NOT FOUND
C           1                    FOUND
C EPS IS A SMALL POSITIVE NUMBER THAT NUMERICALLY REPRESENTS
C ZERO IN THE PROGRAM. EPS = (EUCLIDIAN NORM OF H)*EX ,WHERE
C EX = 2**(-T). T IS THE NUMBER OF BINARY DIGITS IN THE
C MANTISSA OF A FLOATING POINT NUMBER.
C
C
C
C REDUCTION OF THE MATRIX A TO AN UPPER-HESSENBERG FORM H.
C THERE ARE N-2 STEPS.
      IF(N-2)14,1,2
    1 SUBDIA(1) = A(2,1)
      GO TO 14
    2 M = N-2
      DO 12 K=1,M
        L = K+1
        S = 0.0
        DO 3 I=L,N
          H(I,K) = A(I,K)
    3     S = S+ABS(A(I,K))
        IF(S.NE.ABS(A(K+1,K)))GO TO 4
        SUBDIA(K) = A(K+1,K)
        H(K+1,K) = 0.0
        GO TO 12
    4   SR2 = 0.0
        DO 5 I=L,N
          SR = A(I,K)
          SR = SR/S
          A(I,K) = SR
    5     SR2 = SR2+SR*SR
        SR = DSQRT(SR2)
        IF(A(L,K).LT.0.0)GO TO 6
        SR = -SR
    6   SR2 = SR2-SR*A(L,K)
        A(L,K) = A(L,K)-SR
        H(L,K) = H(L,K)-SR*S
        SUBDIA(K) = SR*S
        X = S*DSQRT(SR2)
        DO 7 I=L,N
          H(I,K) = H(I,K)/X
    7     SUBDIA(I) = A(I,K)/SR2
C PREMULTIPLICATION BY THE MATRIX PR.
        DO 9 J=L,N
          SR = 0.0
          DO 8 I=L,N
    8       SR = SR+A(I,K)*A(I,J)
          DO 9 I=L,N
    9       A(I,J) = A(I,J)-SUBDIA(I)*SR
C POSTMULTIPLICATION BY THE MATRIX PR.
        DO 11 J=1,N
          SR=0.0
          DO 10 I=L,N
   10       SR = SR+A(J,I)*A(I,K)
          DO 11 I=L,N
   11       A(J,I) = A(J,I)-SUBDIA(I)*SR
   12   CONTINUE
      DO 13 K=1,M
   13   A(K+1,K) = SUBDIA(K)
C TRANSFER OF THE UPPER HALF OF THE MATRIX A INTO THE
C ARRAY H AND THE CALCULATION OF THE SMALL POSITIVE NUMBER
C EPS.
      SUBDIA(N-1) = A(N,N-1)
   14 EPS = 0.0
      DO 15 K=1,N
        INDIC(K) = 0
       IF(K.NE.N)EPS = EPS+SUBDIA(K)**2
       DO 15 I=K,N
         H(K,I) = A(K,I)
   15    EPS = EPS + A(K,I)**2
     EPS = EX*SQRT(EPS)
C
C THE QR ITERATIVE PROCESS. THE UPPER-HESSENBERG MATRIX H IS
C REDUCED TO THE UPPER-MODIFIED TRIANGULAR FORM.
C
C DETERMINATION OF THE SHIFT OF ORIGIN FOR THE FIRST STEP OF
C THE QR ITERATIVE PROCESS.
      SHIFT = A(N,N-1)
      IF(N.LE.2)SHIFT = 0.0
      IF(A(N,N).NE.0.0)SHIFT = 0.0
      IF(A(N-1,N).NE.0.0)SHIFT = 0.0
      IF(A(N-1,N-1).NE.0.0)SHIFT = 0.0
      M = N
      NS= 0
      MAXST = N*10
C
C TESTING IF THE UPPER HALF OF THE MATRIX IS EQUAL TO ZEkO.
C IF IT IS EQUAL TO ZERO THE QR PROCESS IS NOT NECESSARY.
      DO 16 I=2,N
        DO 16 K=I,N
          IF(A(I-1,K).NE.0.0)GO TO 18
   16     CONTINUE
      DO 17 I=1,N
        INDIC(I)=1
        EVR(I) = A(I,I)
   17   EVI(I) = 0.0
      GO TO 37
C
C START THE MAIN LOOP OF THE QR PROCESS.
   18 K=M-1
      M1=K
      I = K
C FIND ANY DECOMPOSITIONS OF THE MATRIX.
C JUMP TO 34 IF THE LAST SUBMATRIX OF THE DECOMPOSITION IS
C OF THE ORDER ONE.
C JUMP TO 35 IF THE LAST SUBMATRIX OF THE DECOMPOSITION IS
C OF THE ORDER TWO.
      IF(K)37,34,19
   19 IF(ABS(A(M,K)).LE.EPS)GO TO 34
      IF(M-2.EQ.0)GO TO 35
   20   I = I-1
        IF(ABS(A(K,I)).LE.EPS)GO TO 21
        K = I
        IF(K.GT.1)GO TO 20
   21 IF(K.EQ.M1)GO TO 35
C TRANSFORMATION OF THE MATRIX OF THE ORDER GREATER THAN TWO
      S = A(M,M)+A(M1,M1)+SHIFT
      SR= A(M,M)*A(M1,M1)-A(M,M1)*A(M1,M)+0.25*SHIFT**2
      A(K+2,K) = 0.0
C CALCULATE X1,Y1,Z1,FOR THE SUBMATRIX OBTAINED BY THE
C DECOMPOSITION.
      X = A(K,K)*(A(K,K)-S)+A(K,K+1)*A(K+1,K)+SR
      Y = A(K+1,K)*(A(K,K)+A(K+1,K+1)-S)
      R = DABS(X)+DABS(Y)
      IF(R.EQ.0.0)SHIFT = A(M,M-1)
      IF(R.EQ.0.0)GO TO 21
      Z = A(K+2,K+1)*A(K+1,K)
      SHIFT = 0.0
      NS = NS+1
C
C THE LOOP FOR ONE STEP OF THE QR PROCESS.
      DO 33 I=K,M1
        IF(I.EQ.K)GO TO 22
C CALCULATE XR,YR,ZR.
        X = A(I,I-1)
        Y = A(I+1,I-1)
        Z = 0.0
        IF(I+2.GT.M)GO TO 22
        Z = A(I+2,I-1)
   22   SR2 = DABS(X)+DABS(Y)+DABS(Z)
        IF(SR2.EQ.0.0)GO TO 23
        X = X/SR2
        Y = Y/SR2
        Z = Z/SR2
   23   S = DSQRT(X*X + Y*Y + Z*Z)
        IF(X.LT.0.0)GO TO 24
        S = -S
   24   IF(I.EQ.K)GO TO 25
        A(I,I-1) = S*SR2
   25   IF(SR2.NE.0.0)GO TO 26
        IF(I+3.GT.M)GO TO 33
        GO TO 32
   26   SR = 1.0-X/S
        S = X-S
        X = Y/S
        Y = Z/S
C PREMULTIPLICATION BY THE MATRIX PR.
        DO 28 J=I,M
          S = A(I,J)+A(I+1,J)*X
          IF(I+2.GT.M)GO TO 27
          S = S+A(I+2,J)*Y
   27     S = S*SR
          A(I,J) = A(I,J)-S
          A(I+1,J) = A(I+1,J)-S*X
          IF(I+2.GT.M)GO TO 28
          A(I+2,J) = A(I+2,J)-S*Y
   28     CONTINUE
C POSTMULTIPLICATION BY THE MATRIX PR.
        L = I+2
        IF(I.LT.M1)GO TO 29
        L = M
   29   DO 31 J=K,L
          S = A(J,I)+A(J,I+1)*X
          IF(I+2.GT.M)GO TO 30
          S = S + A(J,I+2)*Y
   30     S = S*SR
          A(J,I) = A(J,I)-S
          A(J,I+1)=A(J,I+1)-S*X
          IF(I+2.GT.M)GO TO 31
          A(J,I+2)=A(J,I+2)-S*Y
   31     CONTINUE
        IF(I+3.GT.M)GO TO 33
        S = -A(I+3,I+2)*Y*SR
   32   A(I+3,I) = S
        A(I+3,I+1) = S*X
        A(I+3,I+2) = S*Y + A(I+3,I+2)
   33   CONTINUE
C
      IF(NS.GT.MAXST)GO TO 37
      GO TO 18
C
C COMPUTE THE LAST EIGENVALUE.
   34 EVR(M) = A(M,M)
      EVI(M) = 0.0
      INDIC(M) = 1
      M = K
      GO TO 18
C
C COMPUTE THE EIGENVALUES OF THE LAST 2X2 MATRIX OBTAINED BY
C THE DECOMPOSITION.
   35 R = 0.5*(A(K,K)+A(M,M))
      S = 0.5*(A(M,M)-A(K,K))
      S = S*S + A(K,M)*A(M,K)
      INDIC(K) = 1
      INDIC(M) = 1
      IF(S.LT.0.0)GO TO 36
      T = DSQRT(S)
      EVR(K) = R-T
      EVR(M) = R+T
      EVI(K) = 0.0
      EVI(M) = 0.0
      M = M-2
      GO TO 18
   36 T = DSQRT(-S)
      EVR(K) = R
      EVI(K) = T
      EVR(M) = R
      EVI(M) = -T
      M = M-2
      GO TO 18
C
   37 RETURN
      END

      SUBROUTINE REALVE(N,NM,M,IVEC,A,VECR,EVR,EVI,
     1IWORK,WORK,INDIC,EPS,EX)
      DOUBLE PRECISION S,SR
      INTEGER I,IVEC,ITER,J,K,L,M,N,NM,NS
      REAL BOUND,EPS,EVALUE,EX,PREVIS,R,R1,T
      DIMENSION A(NM,1),VECR(NM,1),EVR(NM)
      DIMENSION EVI(NM),IWORK(NM),WORK(NM),INDIC(NM)
C
```

```
C THIS SUBROUTINE FINDS THE REAL EIGENVECTOR OF THE REAL
C UPPER-HESSENBERG MATRIX IN THE ARRAY A,CORRESPONDING TO
C THE REAL EIGENVALUE STORED IN EVR(IVEC). THE INVERSE
C ITERATION METHOD IS USED.
C NOTE THE MATRIX IN A IS DESTROYED BY THE SUBROUTINE.
C N IS THE ORDER OF THE UPPER-HESSENBERG MATRIX.
C NM DEFINES THE FIRST DIMENSION OF THE TWO DIMENSIONAL
C ARRAYS A AND VECR. NM MUST BE EQUAL TO OR GREATER THAN N.
C M IS THE ORDER OF THE SUBMATRIX OBTAINED BY A SUITABLE
C DECOMPOSITION OF THE UPPER-HESSENBERG MATRIX IF SOME
C SUBDIAGONAL ELEMENTS ARE EQUAL TO ZERO. THE VALUE OF M IS
C CHOSEN SO THAT THE LAST N-M COMPONENTS OF THE EIGENVECTOR
C ARE ZERO.
C IVEC GIVES THE POSITION OF THE EIGENVALUE IN THE ARRAY EVR
C FOR WHICH THE CORRESPONDING EIGENVECTOR IS COMPUTED.
C THE ARRAY EVI WOULD CONTAIN THE IMAGINARY PARTS OF THE N
C EIGENVALUES IF THEY EXISTED.
C
C THE M COMPONENTS OF THE COMPUTED REAL EIGENVECTOR WILL BE
C FOUND IN THE FIRST M PLACES OF THE COLUMN IVEC OF THE TWO
C DIMENSIONAL ARRAY VECR.
C
C IWORK AND WORK ARE THE WORKING STORES USED DURING THE
C GAUSSIAN ELIMINATION AND BACKSUBSTITUTION PROCESS.
C THE ARRAY INDIC INDICATES THE SUCCESS OF THE ROUTINE AS
C FOLLOWS
C     VALUE OF INDIC(I)      EIGENVECTOR I
C             1                 NOT FOUND
C             2                   FOUND
C EPS IS A SMALL POSITIVE NUMBER THAT NUMERICALLY REPRESENTS
C ZERO IN THE PROGRAM. EPS = (EUCLIDIAN NORM OF A)*EX,WHERE
C EX = 2**(-T). T IS THE NUMBER OF BINARY DIGITS IN THE
C MANTISSA OF A FLOATING POINT NUMBER.
      VECR(1,IVEC) = 1.0
      IF(M.EQ.1)GO TO 24
C SMALL PERTURBATION OF EQUAL EIGENVALUES TO OBTAIN A FULL
C SET OF EIGENVECTORS.
      EVALUE = EVR(IVEC)
      IF(IVEC.EQ.M)GO TO 2
      K = IVEC+1
      R = 0.0
      DO 1 I=K,M
        IF(EVALUE.NE.EVR(I))GO TO 1
        IF(EVI(I).NE.0.0)GO TO 1
        R = R+3.0
    1   CONTINUE
      EVALUE = EVALUE+R*EX
    2 DO 3 K=1,M
    3   A(K,K) = A(K,K)-EVALUE
C
C GAUSSIAN ELIMINATION OF THE UPPER-HESSENBERG MATRIX A. ALL
C ROW INTERCHANGES ARE INDICATED IN THE ARRAY IWORK.ALL THE
C MULTIPLIERS ARE STORED AS THE SUBDIAGONAL ELEMENTS OF A.
      K = M-1
      DO 8 I=1,K
        L = I+1
        IWORK(I) = 0
        IF(A(I+1,I).NE.0.0)GO TO 4
        IF(A(I,I).NE.0.0)GO TO 8
        A(I,I) = EPS
        GO TO 8
    4   IF(ABS(A(I,I)).GE.ABS(A(I+1,I)))GO TO 6
        IWORK(I) = 1
        DO 5 J=I,M
          R = A(I,J)
          A(I,J) = A(I+1,J)
    5     A(I+1,J) = R
    6   R = -A(I+1,I)/A(I,I)
        A(I+1,I) = R
        DO 7 J=L,M
    7     A(I+1,J) = A(I+1,J)+R*A(I,J)
    8   CONTINUE
      IF(A(M,M).NE.0.0)GO TO 9
      A(M,M) = EPS
C
C THE VECTOR (1,1,...,1) IS STORED IN THE PLACE OF THE RIGHT
C HAND SIDE COLUMN VECTOR.
    9 DO 11 I=1,N
        IF(I.GT.M)GO TO 10
        WORK(I) = 1.U
        GO TO 11
   10   WORK(I) = 0.0
   11   CONTINUE
C
C THE INVERSE ITERATION IS PERFORMED ON THE MATRIX UNTIL THE
C INFINITE NORM OF THE RIGHT-HAND SIDE VECTOR IS GREATER
C THAN THE BOUND DEFINED AS  0.01/(N*EX).
      BOUND = 0.01/(EX * FLOAT(N))
      NS = 0
      ITER = 1
C
C THE BACKSUBSTITUTION.
   12 R = 0.0
      DO 15 I=1,M
        J = M-I+1
        S = WORK(J)
        IF(J.EQ.M)GO TO 14
        L = J+1
        DO 13 K=L,M
          SR = WORK(K)
   13     S  = S - SR*A(J,K)
   14   WORK(J) = S/A(J,J)
        T = ABS(WORK(J))
        IF(R.GE.T)GO TO 15
        R = T
   15   CONTINUE
C
C THE COMPUTATION OF THE RIGHT-HAND SIDE VECTOR FOR THE NEW
C ITERATION STEP.
      DO 16 I=1,M
   16   WORK(I) = WORK(I)/R
C
C THE COMPUTATION OF THE RESIDUALS AND COMPARISON OF THE
C RESIDUALS OF THE TWO SUCCESSIVE STEPS OF THE INVERSE
C ITERATION.IF THE INFINITE NORM OF THE RESIDUAL VECTOR IS
C GREATER THAN THE INFINITE NORM OF THE PREVIOUS RESIDUAL
C VECTOR THE COMPUTED EIGENVECTOR OF THE PREVIOUS STEP IS
C TAKEN AS THE FINAL EIGENVECTOR.
      R1 = 0.0
      DO 18 I=1,M
        T = 0.0
        DO 17 J=I,M
   17     T = T+A(I,J)*WORK(J)
        T = ABS(T)
        IF(R1.GE.T)GO TO 18
        R1= T
   18   CONTINUE
      IF(ITER.EQ.1)GO TO 19
      IF(PREVIS.LE.R1)GO TO 24
   19 DO 20 I=1,M
   20   VECR(I,IVEC) = WORK(I)
      PREVIS = R1
      IF(NS.EQ.1)GO TO 24
      IF(ITER.GT.6)GO TO 25
      ITER = ITER+1
      IF(R.LT.BOUND)GO TO 21
      NS = 1
C
C GAUSSIAN ELIMINATION OF THE RIGHT-HAND SIDE VECTOR.
   21 K = M-1
      DO 23 I=1,K
        R = WORK(I+1)
        IF(IWORK(I).EQ.0)GO TO 22
        WORK(I+1)=WORK(I)+WORK(I+1)*A(I+1,I)
        WORK(I) = R
        GO TO 23
   22   WORK(I+1)=WORK(I+1)+WORK(I)*A(I+1,I)
   23   CONTINUE
      GO TO 12
C
   24 INDIC(IVEC) = 2
   25 IF(M.EQ.N)GO TO 27
      J = M+1
      DO 26 I=J,N
   26   VECR(I,IVEC) = 0.0
   27 RETURN
      END
```

```
      SUBROUTINE COMPVE(N,NM,M,IVEC,A,VECR,H,EVR,EVI,INDIC,
     1IWORK,SUBDIA,WORK1,WORK2,WORK,EPS,EX)
      DOUBLE PRECISION D,D1
      INTEGER I,I1,I2,ITER,IVEC,J,K,L,M,N,NM,NS
      REAL B,BOUND,EPS,ETA,EX,FKSI,PREVIS,R,S,U,V
      DIMENSION A(NM,1),VECR(NM,1),H(NM,1),EVR(NM),EVI(NM),
     1INDIC(NM),IWORK(NM),SUBDIA(NM),WORK1(NM),WORK2(NM),
     2WORK(NM)
C
C THIS SUBROUTINE FINDS THE COMPLEX EIGENVECTOR OF THE REAL
C UPPER-HESSENBERG MATRIX OF ORDER N CORRESPONDING TO THE
C COMPLEX EIGENVALUE WITH THE REAL PART IN EVR(IVEC) AND THE
C CORRESPONDING IMAGINARY PART IN EVI(IVEC). THE INVERSE
C ITERATION METHOD IS USED MODIFIED TO AVOID THE USE OF
C COMPLEX ARITHMETIC.
C THE MATRIX ON WHICH THE INVERSE ITERATION IS PERFORMED IS
C BUILT UP IN THE ARRAY A BY USING THE UPPER-HESSENBERG
C MATRIX PRESERVED IN THE UPPER HALF OF THE ARRAY H AND IN
C THE ARRAY SUBDIA.
C NM DEFINES THE FIRST DIMENSION OF THE TWO DIMENSIONAL
C ARRAYS A,VECR AND H. NM MUST BE EQUAL TO OR GREATER
C THAN N.
C M IS THE ORDER OF THE SUBMATRIX OBTAINED BY A SUITABLE
C DECOMPOSITION OF THE UPPER-HESSENBERG MATRIX IF SOME
C SUBDIAGONAL ELEMENTS ARE EQUAL TO ZERO. THE VALUE OF M IS
C CHOSEN SO THAT THE LAST N-M COMPONENTS OF THE COMPLEX
C EIGENVECTOR ARE ZERO.
C
C THE REAL PARTS OF THE FIRST M COMPONENTS OF THE COMPUTED
C COMPLEX EIGENVECTOR WILL BE FOUND IN THE FIRST M PLACES OF
C THE COLUMN WHOSE TOP ELEMENT IS VECR(1,IVEC) AND THE
C CORRESPONDING IMAGINARY PARTS OF THE FIRST M COMPONENTS OF
C THE COMPLEX EIGENVECTOR WILL BE FOUND IN THE FIRST M
C PLACES OF THE COLUMN WHOSE TOP ELEMENT IS VECR(1,IVEC-1).
C
```

```
C THE ARRAY INDIC INDICATES THE SUCCESS OF THE ROUTINE AS
C FOLLOWS
C     VALUE OF INDIC(I)      EIGENVECTOR I
C            1                  NOT FOUND
C            2                    FOUND
C THE ARRAYS IWORK,WORK1,WORK2 AND WORK ARE THE WORKING
C STORES USED DURING THE INVERSE ITERATION PROCESS.
C EPS IS A SMALL POSITIVE NUMBER THAT NUMERICALLY REPRESENTS
C ZERO IN THE PROGRAM. EPS = (EUCLIDIAN NORM OF H)*EX, WHERE
C EX = 2**(-T). T IS THE NUMBER OF BINARY DIGITS IN THE
C MANTISSA OF A FLOATING POINT NUMBER.
C
      FKSI = EVR(IVEC)
      ETA  = EVI(IVEC)
C THE MODIFICATION OF THE EIGENVALUE (FKSI + I*ETA) IF MORE
C EIGENVALUES ARE EQUAL.
      IF(IVEC.EQ.M)GO TO 2
      K = IVEC+1
      R = 0.0
      DO 1 I=K,M
        IF(FKSI.NE.EVR(I))GO TO 1
        IF(ABS(ETA).NE.ABS(EVI(I)))GO TO 1
        R = R + 3.0
    1   CONTINUE
      R = R*EX
      FKSI = FKSI+R
      ETA  = ETA +R
C
C THE MATRIX  ((H-FKSI*I)*(H-FKSI*I) + (ETA*ETA)*I)  IS
C STORED INTO THE ARRAY A.
    2 R = FKSI*FKSI + ETA*ETA
      S = 2.0*FKSI
      L = M-1
      DO 5 I=1,M
        DO 4 J=I,M
          D = 0.0
          A(J,I) = 0.0
          DO 3 K = I,J
    3       D = D+H(I,K)*H(K,J)
    4     A(I,J) = D-S*H(I,J)
    5   A(I,I) = A(I,I)+R
     DO 9 I=1,L
        R = SUBDIA(I)
        A(I+1,I) = -S*R
        I1 = I+1
        DO 6 J=1,I1
    6     A(J,I) = A(J,I)+R*H(J,I+1)
        IF(I.EQ.1)GO TO 7
        A(I+1,I-1) = R*SUBDIA(I-1)
    7   DO 8 J=I,M
    8     A(I+1,J) = A(I+1,J)+R*H(I,J)
    9   CONTINUE
C
C THE GAUSSIAN ELIMINATION OF THE MATRIX
C ((H-FKSI*I)*(H-FKSI*I) + (ETA*ETA)*I) IN THE ARRAY A. THE
C ROW INTERCHANGES THAT OCCUR ARE INDICATED IN THE ARRAY
C IWORK. ALL THE MULTIPLIERS ARE STORED IN THE FIRST AND IN
C THE SECOND SUBDIAGONAL OF THE ARRAY A.
      K = M-1
      DO 18 I=1,K
        I1 = I+1
        I2 = I+2
        IWORK(I) = 0
        IF(I.EQ.K)GO TO 10
        IF(A(I+2,I).NE.0.0)GO TO 11
   10   IF(A(I+1,I).NE.0.0)GO TO 11
        IF(A(I,I).NE.0.0)GO TO 18
        A(I,I) = EPS
        GO TO 18
C
   11   IF(I.EQ.K)GO TO 12
        IF(ABS(A(I+1,I)).GE.ABS(A(I+2,I)))GO TO 12
        IF(ABS(A(I,I)).GE.ABS(A(I+2,I)))GO TO 16
        L = I+2
        IWORK(I) = 2
        GO TO 13
   12   IF(ABS(A(I,I)).GE.ABS(A(I+1,I)))GO TO 15
        L = I+1
        IWORK(I) = 1
C
   13   DO 14 J=I,M
          R = A(I,J)
          A(I,J) = A(L,J)
   14     A(L,J) = R
   15   IF(I.NE.K)GO TO 16
        I2 = I1
   16   DO 17 L=I1,I2
          R = -A(L,I)/A(I,I)
          A(L,I) = R
          DO 17 J=I1,M
   17       A(L,J) = A(L,J)+R*A(I,J)
   18   CONTINUE
      IF(A(M,M).NE.0.0)GO TO 19
      A(M,M) = EPS
C
C THE VECTOR (1,1,...,1) IS STORED INTO THE RIGHT-HAND SIDE
C VECTORS VECR( ,IVEC) AND VECR( ,IVEC-1) REPRESENTING THE
C COMPLEX RIGHT-HAND SIDE VECTOR.
   19 DO 21 I=1,N
        IF(I.GT.M)GO TO 20
        VECR(I,IVEC) = 1.0
        VECR(I,IVEC-1) = 1.0
        GO TO 21
   20   VECR(I,IVEC) = 0.0
        VECR(I,IVEC-1) = 0.0
   21   CONTINUE
C
C THE INVERSE ITERATION IS PERFORMED ON THE MATRIX UNTIL THE
C INFINITE NORM OF THE RIGHT-HAND SIDE VECTOR IS GREATER
C THAN THE BOUND DEFINED AS 0.01/(N*EX).
      BOUND = 0.01/(EX*FLOAT(N))
      NS = 0
      ITER = 1
      DO 22 I=1,M
   22   WORK(I) = H(I,I)-FKSI
C
C THE SEQUENCE OF THE COMPLEX VECTORS Z(S) = P(S)+I*Q(S) AND
C W(S+1)= U(S+1)+I*V(S+1) IS GIVEN BY THE RELATIONS
C (A - (FKSI-I*ETA)*I)*W(S+1) = Z(S)  AND
C Z(S+1) = W(S+1)/MAX(W(S+1)).
C THE FINAL W(S) IS TAKEN AS THE COMPUTED EIGENVECTOR.
C
C THE COMPUTATION OF THE RIGHT-HAND SIDE VECTOR
C (A-FKSI*I)*P(S)-ETA*Q(S). A IS AN UPPER-HESSENBERG MATRIX.
   23 DO 27 I=1,M
        D = WORK(I)*VECR(I,IVEC)
        IF(I.EQ.1)GO TO 24
        D = D+SUBDIA(I-1)*VECR(I-1,IVEC)
   24   L = I+1
        IF(L.GT.M)GO TO 26
        DO 25 K=L,M
   25     D = D+H(I,K)*VECR(K,IVEC)
   26   VECR(I,IVEC-1) = D-ETA*VECR(I,IVEC-1)
   27   CONTINUE
C
C GAUSSIAN ELIMINATION OF THE RIGHT-HAND SIDE VECTOR.
      K = M-1
      DO 28 I=1,K
        L = I+IWORK(I)
        R = VECR(L,IVEC-1)
        VECR(L,IVEC-1) = VECR(I,IVEC-1)
        VECR(I,IVEC-1) = R
        VECR(I+1,IVEC-1) = VECR(I+1,IVEC-1)+A(I+1,I)*R
        IF(I.EQ.K)GO TO 28
        VECR(I+2,IVEC-1) = VECR(I+2,IVEC-1)+A(I+2,I)*R
   28   CONTINUE
C
C THE COMPUTATION OF THE REAL PART U(S+1) OF THE COMPLEX
C VECTOR W(S+1). THE VECTOR U(S+1) IS OBTAINED AFTER THE
C BACKSUBSTITUTION.
      DO 31 I=1,M
        J = M-I+1
        D = VECR(J,IVEC-1)
        IF(J.EQ.M)GO TO 30
        L = J+1
        DO 29 K=L,M
          D1 = A(J,K)
   29     D = D-D1*VECR(K,IVEC-1)
   30   VECR(J,IVEC-1) = D/A(J,J)
   31   CONTINUE
C
C THE COMPUTATION OF THE IMAGINARY PART V(S+1) OF THE VECTOR
C W(S+1), WHERE  V(S+1) = (P(S)-(A-FKSI*I)*U(S+1))/ETA.
      DO 35 I=1,M
        D = WORK(I)*VECR(I,IVEC-1)
        IF(I.EQ.1)GO TO 32
        D = D+SUBDIA(I-1)*VECR(I-1,IVEC-1)
   32   L = I+1
        IF(L.GT.M)GO TO 34
        DO 33 K=L,M
   33     D = D+H(I,K)*VECR(K,IVEC-1)
   34   VECR(I,IVEC) = (VECR(I,IVEC)-D)/ETA
   35   CONTINUE
C
C THE COMPUTATION OF (INFIN. NORM OF W(S+1))**2 .
      L = 1
      S = 0.0
      DO 36 I=1,M
        R = VECR(I,IVEC)**2 + VECR(I,IVEC-1)**2
        IF(R.LE.S)GO TO 36
        S = R
        L = I
   36   CONTINUE
C THE COMPUTATION OF THE VECTOR Z(S+1),WHERE Z(S+1)= W(S+1)/
C (COMPONENT OF W(S+1) WITH THE LARGEST ABSOLUTE VALUE) .
      U = VECR(L,IVEC-1)
      V = VECR(L,IVEC)
      DO 37 I=1,M
        B = VECR(I,IVEC)
        R = VECR(I,IVEC-1)
        VECR(I,IVEC) = (R*U + B*V)/S
   37   VECR(I,IVEC-1) = (B*U-R*V)/S
```

```
C THE COMPUTATION OF THE RESIDUALS AND COMPARISON OF THE
C RESIDUALS OF THE TWO SUCCESSIVE STEPS OF THE INVERSE
C ITERATION. IF THE INFINITE NORM OF THE RESIDUAL VECTOR IS
C GREATER THAN THE INFINITE NORM OF THE PREVIOUS RESIDUAL
C VECTOR THE COMPUTED VECTOR OF THE PREVIOUS STEP IS TAKEN
C AS THE COMPUTED APPROXIMATION TO THE EIGENVECTOR.
      B = 0.0
      DO 41 I=1,M
        R = WORK(I)*VECR(I,IVEC-1) - ETA*VECR(I,IVEC)
        U = WORK(I)*VECR(I,IVEC) + ETA*VECR(I,IVEC-1)
        IF(I.EQ.1)GO TO 38
        R = R+SUBDIA(I-1)*VECR(I-1,IVEC-1)
        U = U+SUBDIA(I-1)*VECR(I-1,IVEC)
   38   L = I+1
        IF(L.GT.M)GO TO 40
        DO 39 J=L,M
          R = R+H(I,J)*VECR(J,IVEC-1)
   39     U = U+H(I,J)*VECR(J,IVEC)
   40   U = R*R + U*U
        IF(B.GE.U)GO TO 41
        B = U
   41   CONTINUE
      IF(ITER.EQ.1)GO TO 42
      IF(PREVIS.LE.B)GO TO 44
   42 DO 43 I=1,N
        WORK1(I) = VECR(I,IVEC)
   43   WORK2(I) = VECR(I,IVEC-1)
      PREVIS = B
      IF(NS.EQ.1)GO TO 46
      IF(ITER.GT.6)GO TO 47
      ITER = ITER+1
      IF(BOUND.GT.SQRT(S))GO TO 23
      NS = 1
      GO TO 23
C
   44 DO 45 I=1,N
        VECR(I,IVEC) = WORK1(I)
   45   VECR(I,IVEC-1)=WORK2(I)
   46 INDIC(IVEC-1) = 2
      INDIC(IVEC)   = 2
   47 RETURN
      END
```

Added in Proof. A small alteration to the program is desirable. The four statements in the subroutine SCALE, page 822, lines 3–6, should be replaced by the four statements below. The alteration is necessary so that the program will also give correct eigenvectors for the case when no convergence of the process of scaling occurs.

```
        PRFACT (I)  =  1.0
        DO  12  J = 1, N
12        A (I, J)  =  H (I, J)
      ENORM  =  1.0
```

CERTIFICATION OF ALGORITHM 343 [F1] EIGENVALUES AND EIGENVECTORS OF A REAL GENERAL MATRIX [J. Grad and M. A. Brebner, *Comm. ACM 11* (Dec. 1968), 820–826]

H. D. Knoble (Recd. 2 July 1969 and 18 Sept. 1969)
The Pennsylvania State University, Computation Center, University Park, PA 16802

KEY WORDS AND PHRASES: norm, characteristic equation, degenerate eigensystem, diagonalizable matrix, defective matrix
CR CATEGORIES: 5.14

The program used for this certification was copied directly from the printed Fortran algorithm [1]. In addition to incorporating the suggested modification, the algorithm as used here was modified to operate completely in double precision arithmetic. The tests were run on an IBM System/360 model 67 using Fortran IV, double precision arithmetic (15 significant decimal digits; $t = 53$). One criterion for measuring numerical precision of the results was a norm of a residual matrix. That is, given a coefficient matrix, A of order n, pose the characteristic equation as $AX_k = y_kX_k$ and define the norm of M as $\| M \|_1 = \max_j(\sum_i \mid m_{i,j} \mid)$, where $M = (M_k) = (AX_k - y_kX_k)$, X_k is the kth right-hand eigenvector of A, and y_k is the kth eigenvalue for $k = 1, 2, \cdots, n$.

The norm $\| M \|_1$ essentially measures the worst eigenvalue-eigenvector pair associated with the characteristic equation. In order to gain information concerning the other extreme, as well as an average measure of precision, the notations $\mid M \mid_{\min} = \min_j(\sum_i \mid m_{i,j} \mid)$ and $\| M \|_{\text{ave}} = \sum_j(\sum_i \mid m_{i,j} \mid)/n$ will be used respectively, the former simply indicating the quantity is not a matrix norm.

The algorithm's performance was also analyzed by generating test matrices with certain known properties thereby permitting comparisons to be made between computational and theoretical results.

The objective was to study the algorithm's sensitivity to ill-conditioning and degeneracy by observing its behavior relative to the speed and precision, and accuracy where possible, with which a variety of eigensystems could be solved. Testing was carried out by entertaining four sets of matrices as follows:

Case 1. Small Matrices with Known Solutions. Several matrices from each of [1, 2, 3] varying in order from 3 to 8 yielded eigenvalues, and eigenvectors where documented, accurate to at least 7 decimal places. The largest 1-norm was $\| M \|_1 < 10^{-13}$; $\| M \|_{\text{ave}}$ averaged 10^{-14}; and the largest value of $\mid M \mid_{\min}$, was less than 10^{-14}. Maximum computation time for any of these matrices was less than a second.

Included in this test was a matrix, A, belonging to a large class of test matrices discovered by Gear [3]. This matrix, $A = (a_{i,j})$ of order 8 is defined as:

$$A = \begin{cases} a_{i,i+1} = a_{i+1,i} = 1, & \text{for} \quad i = 1, 2, \cdots, 7, \\ a_{1,6} = a_{8,3} = 1, & \\ a_{i,j} = 0, & \text{otherwise.} \end{cases}$$

This nonsymmetric matrix has a zero trace and eigenvalue pairs: ±2, ±1, ±1, ±1. The algorithm yielded four of the eigenvalues accurate to 15 decimal places and four values accurate to 7 places.

Deserving special note here is example (*iii*) presented with the original algorithm. As the authors [1] stated, although this matrix when transformed by scaling becomes invariant under the *QR* process, the original, single precision algorithm yielded correct results. However, the double precision version failed completely regardless of the value of the hardware parameter t. In addition, the algorithm may erroneously indicate success for this case; however, with the machine configuration noted earlier, failure was correctly indicated.

Case 2. Degenerate and Defective Matrices. Using an algorithm suggested by the work of Hall and Porsching [4], a degenerate, nonsymmetric matrix of order 30 with known positive eigenvalues was generated with eigenvalues: $y_1 = 30$; $y_i = 25$ for $i = 2, 3, \cdots, 10$; $y_i = 31 - i$ for $i = 11, 12, \cdots, 20$; and $y_i = 1$ for $i = 21, 22, \cdots, 30$. All eigenvalues were returned accurate to at least 14 decimal places; $\| M \|_1 < 10^{-11}$, $\| M \|_{\text{ave}} < 10^{-12}$, and $\mid M \mid_{\min} < 10^{-13}$. Computation time was about 4 sec.

Gear [3] defines a class of matrices including a matrix B of order 25 such that

$$B = \begin{pmatrix} A & I & 0 & I & 0 \\ I & A & I & 0 & 0 \\ 0 & I & A & I & 0 \\ 0 & 0 & I & A & I \\ 0 & I & 0 & I & A \end{pmatrix} \quad \text{where} \quad A = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix},$$

and I is the identity matrix. Using the theory developed by Gear [3], it is easy to show the matrix B has 11 zeros, six pairs of eigenvalues equal to ±2, and one pair of eigenvalues equal to ±4. The algorithm yielded 14 eigenvalues accurate to 7 decimal places and 11 eigenvalues (not all the zero values) with at least 14 place accuracy; $\| M \|_1 < 10^{-12}$, $\| M \|_{\text{ave}} < 10^{-14}$, and $\mid M \mid_{\min} < 10^{-14}$. Computation time was less than 3 seconds.

To gain a measure of the algorithm's ability to separate eigenvectors corresponding to the same eigenvalue, a degenerate symmetric matrix was generated using an algorithm of Ortega [5]. Briefly, a similarity transformation was used to generate a matrix

A of order 6. That is, using Ortega's notation, $A = CDC$ where $D = diag(1,2,3,1,2,3)$, $C = (I-2vv')$, and v is a column vector with each element in this case equal to $1/\sqrt{6}$. For this case $\| M \|_1 < 10^{-13}$, $\| M \|_{ave} < 10^{-13}$, and $| M |_{min} < 10^{-14}$. The eigenvectors corresponding to each eigenvalue pair are listed below to three decimal places.

Eigenvalue	*Transposed eigenvectors*
1	(+.480, −.438, −.438, −.042, −.438, −.438)
	(+.449, −.447, −.447, −.002, −.447, −.447)
2	(−.254, +.723, −.254, −.254, −.469, −.254)
	(−.151, +.745, −.151, −.151, −.595, −.151)
3	(−.328, −.328, +.672, −.328, −.328, −.344)
	(−.328, −.328, +.671, −.329, −.329, −.342)

Even though the matrix A is obviously not defective, by inspection it can be seen that the algorithm did not yield well-separated eigenvectors. This fact is also evident by noting that if the algorithm extracts independent eigenvectors they will be returned orthogonal (in fact orthonormal), yet the determinant of the eigenvector matrix for this case is less then 10^{-4} in absolute value.

Case 3. Ill-Conditioning. Two ill-conditioned matrices suggested by Wilkinson [6] were solved. One of these is a matrix A of the form:

$$A = \left\{ \begin{array}{r} a_{i,i} = 21 - i \\ a_{i,i+1} = 20 \\ a_{i,j} = 0 \\ a_{20,1} = \epsilon \end{array} \right\}, \quad \text{for} \quad i, j = 1, 2, \cdots, 20; j \neq i, i+1$$

whose eigenvalues are very sensitive to perturbations of ϵ. With $\epsilon = 0$, the matrix is triangular and the eigenvalues were returned accurate to 15 places with $\| M \|_1 < 10^{-14}$, $\| M \|_{ave} < 10^{-14}$, and $| M |_{min} < 10^{-15}$. As Wilkinson [6] points out, with $\epsilon = 10^{-10}$ the eigenvalues change drastically, having been computed in this case in complete agreement with this reference. For the perturbed case $\| M \|_1 > 10^{-10}$, $\| M \|_{ave} > 10^{-11}$ and $| M |_{min}$ remained less than 10^{-14}.

The algorithm was tested under a combination of ill-conditioning and degeneracy by generating nonsymmetric matrices as in Case 2, but of order 20, conditioned such that $max \mid eigenvalue \mid = 10^j \times min \mid eigenvalue \mid$ for $j = 2, 3, \cdots, 20$; degeneracy was introduced by generating the matrices with only 10 distinct eigenvalues. The values of $\| M \|_1$ for the matrices tested in this class were such that $\| M \|_1 \simeq 10^{j-11}$ for $j = 2, 3, \cdots, 18$. $\| M \|_{ave}$ followed a similar curve; $| M |_{min} < 10^{-11}$ for $j < 14$ and never exceeded 10^{-5}. Although the algorithm indicated success, severe computational breakdown was evident during this test for values of j greater than 18. However, the largest eigenvalue in every case was returned accurate to 15 decimal places. Computation time for matrices of order 20 was consistently less than 2 seconds.

Case 4. Large Matrices. Several nonsymmetric matrices of order 50 with elements uniform on the interval (0, 50) were solved yielding the following average figures: $\| M \|_1 < 10^{-9}$, $\| M \|_{ave} < 10^{-10}$, and $| M |_{min} < 10^{-11}$. Computation time averaged 31 seconds.

A diagonal matrix A of order 50 with elements:

$$A = \left\{ \begin{array}{ll} a_{i,i} = 1, & \text{for} \quad i = 1, 10, 20, 30, 40, 50, \\ a_{i,j} = 0, & \text{otherwise}, \end{array} \right\}$$

was solved yielding $\| M \|_1 < 10^{-16}$, $\| M \|_{ave} < 10^{-17}$, and $| M |_{min} < 10^{-31}$.

Computation time was about 5 seconds and all eigenvalues were returned correct to 15 decimal places.

Conclusions. The algorithm is capable of successfully computing eigenvalues and eigenvectors of real general matrices even under conditions considered unstable. It has the advantage of being documented in ANSI (USASI) Fortran, being computationally fast, and has the capability of yielding results with as much precision as the hardware will permit. The algorithm does not break down when presented with a matrix which is not diagonalizable; that is, a set of eigenvectors satisfying the eigenequation is computed regardless of the existence of linearly independent eigenvectors. However, when a matrix is diagonalizable and degenerate, the algorithm does not yield well separated eigenvectors corresponding to non-distinct eigenvalues. Another apparent disadvantage is the possible indication of completely successful computation (INDIC), even in clearly ill-conditioned situations where computational difficulties are inevitable. This latter property, however, is a common fault of other algorithms as well.

Acknowledgments. This author wishes to thank the editor and referee for their valuable critique and useful suggestions.

References:

1. Grad, J., and Brebner, M. A. Algorithm 343, Eigenvalues and eigenvectors of a real general matrix. *Comm. ACM, 11* (Dec. 1968), 820–826.
2. Barlow, C. A. Jr., and Jones, E. L. A method for the solution of roots of a nonlinear equation and for solution of the general eigenvalue problem. *J. ACM 13*, 1 (Jan. 1966), 135–142.
3. Gear, C. W. A simple set of test matrices for eigenvalue programs. *Math. Comput. 23*, 1 (Jan. 1969), 119–125.
4. Hall, C. A., and Porsching, T. A. Generation of positive test matrices with known positive spectra. *Comm. ACM 11*, 8 (Aug. 1968), 559–560.
5. Ortega, J. M. Generation of test matrices of similarity transformations. *Comm. ACM 7*, 6 (June 1964), 377–378.
6. Wilkinson, J. H. *The Algebraic Eigenvalue Problem.* Clarendon Press, Oxford, 1965, pp. 86–93.

REMARK ON ALGORITHM 343 [F1] EIGENVALUES AND EIGENVECTORS OF A REAL GENERAL MATRIX [J. Grad and M. A. Brebner. *Comm. ACM 11* (Dec. 1968), 820-826]

William Knight and William Mersereau (Recd. 7 Apr. 1970)
Computing Center, University of New Brunswick, Fredericton, New Brunswick, Canada

KEY WORDS AND PHRASES: eigenvalues, eigenvectors, latent roots, Householder's method, QR algorithm, inverse iteration
CR CATEGORIES: 5.14

This remark reports certain failures of Algorithm 343 when applied to pathological matrices. The smallest example is a 4 × 4 matrix for which 16 guard bits (5+ digits) proved insufficient; all computed eigenvalues were incorrect in the most significant digit.

The algorithm was implemented on an IBM System/360 model 50 using Fortran IV-G. The program was not modified to operate completely in double precision as was done for Knoble's certification [2]. Satisfactory agreement was obtained for the three sample matrices given with the algorithm.

Example A

$$\begin{array}{rrrr} -50 & 53 & 52 & 51 \\ -52 & 1 & 53 & 52 \\ -53 & 0 & 1 & 53 \\ -51 & 53 & 52 & 52 \end{array}$$

The exact eigenvalues are all 1. The computed eigenvalues follow. (Computed eigenvalues are reported rounded to 2 places after the decimal point, any further figures being, rather obviously, pointless.)

2.35
1.03 ± 1.38 i
−0.41

The maximum error in a computed eigenvalue exceeds 2 percent of the largest element of the matrix.

Example B

$$\begin{matrix} -41 & 55 & 4 & 3 & 2 & 51 \\ -2 & 10 & 55 & 4 & 3 & 2 \\ -3 & 0 & 10 & 55 & 4 & 3 \\ -4 & 0 & 0 & 10 & 55 & 4 \\ -55 & 0 & 0 & 0 & 10 & 55 \\ -51 & 55 & 4 & 3 & 2 & 61 \end{matrix}$$

The exact eigenvalues are all 10. The computed eigenvalues:

14.76 ± 2.92 i
9.70 ± 5.33 i
5.54 ± 2.39 i

The maximum error in a computed eigenvalue exceeds 9% of the largest element in the matrix.

Example C

$$\begin{matrix} -91 & -94 & 0 & 0 & 0 & 0 & 0 & 0 \\ 95 & 98 & 0 & 0 & 0 & 0 & 0 & 0 \\ 90 & 99 & 5 & 0 & 0 & 0 & 0 & 0 \\ 90 & 0 & 99 & 6 & 0 & 0 & 0 & 0 \\ 90 & 0 & 0 & 99 & 7 & 0 & 0 & 0 \\ 90 & 0 & 0 & 0 & 99 & 8 & 0 & 0 \\ 90 & 0 & 0 & 0 & 0 & 99 & 9 & 0 \\ 99 & 99 & 0 & 0 & 0 & 0 & 99 & 10 \end{matrix}$$

The exact eigenvalues are 3, 4, 5, 6, 7, 8, 9, 10. The computed eigenvalues are:

12.68
10.96 ± 3.73 i
6.47 ± 5.38 i
2.09 ± 3.73 i
0.27

Although all eigenvalues are real, the imaginary part of one pair of computed eigenvalues exceeds 5 percent of the largest element of the matrix. This matrix, like the other two, was maliciously devised to take advantage of the program; it is indicative of this that the transpose, being already in lower Hessenberg form, fares much better, all computed eigenvalues being correct to within ±0.05.

Although, in view of the known sensitivity of multiple eigenvalues to small changes of certain elements of certain matrices, such counter examples are to be expected, it is probably worth putting a few examples on record as the casual and unsophisticated user is more apt to take warning of the dangers of eigenvalue computations in single precision from a concrete case.

References:

[1] Grad, J., and M. A. Brebner. Algorithm 343, Eigenvalues and eigenvectors of a real general matrix. *Comm. ACM 11* (Dec. 1968), 820-826.

[2] Knoble, H. D. Certification of Algorithm 343. Eigenvalues and eigenvectors of a real general matrix. *Comm. ACM 13* (Feb. 1970), 122-124.

Remark on Algorithm 343 [F2]

Eigenvalues and Eigenvectors of a Real General Matrix [J. Grad and M. A. Brebner, *Comm. ACM 11* (Dec. 1968), 820-826]

Herbert Niessner (Recd. 26 Oct. 1970 and 18 Jan. 1971)
Brown, Boveri and Company, Baden, Switzerland

Key Words and Phrases: eigenvalues, eigenvectors, QR-algorithm, nonsymmetric matrices, general matrices
CR Categories: 5.14

We had at our disposal a double precision version (all real variables are declared to be of type double precision) for the IBM 360/50 of the algorithm 343 [1] with logical *IF* statements converted to arithmetical ones. In the following three modifications which we found to be of practical value are to be discussed.

a. Modification of the test of smallness of R in *HESQR*: 10 and 11 lines after statement 21, a test is made on R whether it is zero or not. Because R is not of type integer such a test is almost inefficient. Let us call α some value representing the order of the elements of the matrix A (for example the Euclidean norm of A), ϵ_A a small positive number numerically representing zero elements of A and ϵ_m the relative machine accuracy. In *HESQR* ϵ_A is chosen to be $\epsilon_A \sim \alpha\epsilon_m$. By inspection of the formulas it is seen that R is of the order of α^2; therefore R should be considered to be small if $R < \alpha^2\epsilon_m = \epsilon_A{}^2/\epsilon_m$. This is equivalent to $R/\epsilon_A < \epsilon_A/\epsilon_m$, which does not have the risk of underflow.

Following these ideas we changed the statements

```
      IF(R.EQ.0.0) SHIFT=A(M,M-1)
      IF(R.EQ.0.0) GO TO 21
      Z=A(K+2,K+1)*A(K+1,K)
```

10, 11, and 12 lines after statement 21 to

```
      IF(R/EPS-EPS/EX) 215,215,217
215   IF(SHIFT-A(M,M-1)) 216,217,216
216   SHIFT=A(M,M-1)
      GO TO 21
217   Z=A(K+2,K+1)*A(K+1,K)
```

(keeping in mind that $\epsilon_A = EPS$ and $\epsilon_m = EX$), and we were able to solve example (i) and (ii) as well as example (iii) of [1].

b. Modifications in *EIGENP*: In order to suppress unnecessary and possibly impermissible computations in case of failure, the subroutine *EIGENP* was modified as follows. We changed the statement $L = 0$, two lines after statement 15, to

```
      ISW=INDIC(I)-1
      IF(ISW) 24,152,152
152   L=0
```

statement

```
16    DO 18 J=1,N
```

to

```
16    IF(ISW) 24,161,162
161   IF(L) 232,202,232
162   DO 18 J=1,N
```

and statement

```
      EVR(I)=EVR(I)*ENORM
```

one line after statement 20, to

```
202  EVR(I) = EVR(I)*ENORM
```

Statements

```
 21  KON=1
     ⋮
     EVI(I-1) = -EVI(I)
```

have been removed and reinserted as

```
232  KON=1
     ⋮
     EVI(I-1) = -EVI(I)
```

between statement 23 and 24. Finally statement

```
     R=0.0,
```

five lines after statement 21, has been changed to

```
 21  R=0.0
```

c. Modifications in *SCALE*: It seems to be reasonable to change statement

```
     Q=A(I,J)
```

preceding statement 9 to

```
     IF(I-J) 88,89,88
 88  A(I,J)=H(I,J)*PRFACT(I)/PRFACT(J)
 89  Q=A(I,J)
```

so that even in case of many iterations being necessary to calculate *PRFACT*, the relation of similarity of the result matrix to the input matrix will almost not be changed by rounding errors.

References

1. Grad, J., and Brebner, M.A. Algorithm 343, Eigenvalues and eigenvectors of a real general matrix. *Comm. ACM 11* (Dec. 1968), 820–826.

ALGORITHM 344
STUDENT'S *t*-DISTRIBUTION [S14]

David A. Levine (Recd. 26 Mar. 1968 and 2 Aug. 1968)
State University of New York at Stony Brook, Stony Brook, NY 11790

KEY WORDS AND PHRASES: Student's *t*-Distribution, *t*-test, small-sample statistics, distribution function
CR CATEGORIES: 5.12, 5.5

Comment *t*-Test evaluates in single-precision the value of Student's [2] *t*-distribution for argument *T* and degrees of freedom *DF*. The two-tailed Student's *t*-distribution, *A*, is obtained as the indefinite integral:

$$A(T, DF) = C \int_T^\infty \left(1 + \frac{x^2}{DF}\right)^{-\frac{DF+1}{2}} dx$$

where C is chosen so that $A\ (0, DF) = 1$.

The integration of A can be accomplished exactly by integrating by parts successively, obtaining:
for DF an odd integer,

$$A(T, DF) = 1 - \frac{2}{\pi}\left\{\arctan a + ab\left[1 + b\left(\frac{2}{3}\right) + b^2\left(\frac{2}{3}\cdot\frac{4}{5}\right) + \cdots + b^{\frac{DF-3}{2}}\left(\frac{2}{3}\cdot\frac{4}{5}\cdots\frac{DF-3}{DF-2}\right)\right]\right\},$$

and for DF an even integer,

$$A(T, DF) = 1 - a\sqrt{b}\left[1 + b\cdot\left(\frac{1}{2}\right) + b^2\left(\frac{1}{2}\cdot\frac{3}{4}\right) + \cdots + b^{\frac{DF-2}{2}}\left(\frac{1}{2}\cdot\frac{3}{4}\cdot\cdots\cdot\frac{DF-3}{DF-2}\right)\right],$$

where $a = \dfrac{T}{\sqrt{DF}}$, $b = (1 + a^2)^{-1}$.

A Fortran program evaluating these series is given below, giving at least six correct significant figures after the decimal—more than enough accuracy for most statistical applications. The *t*-Test is usually applied in small-sample statistics [1] where $DF \leq 30$. The algorithm presented here is faster and simpler, with accuracy equal to previous algorithms for $DF \leq 30$. In the range $30 \leq DF \leq 100$, this algorithm is competitive in speed and accuracy with previous algorithms. For the range $DF > 100$, small-sample assumptions may be altered by replacing the integrand of the distribution by a Gaussian (normal) curve; hence much greater speed is obtained in this range by employing, for example, Algorithm 209 [3]. Instructive comments and bibliography are obtainable from Algorithm 321 [4], where an algorithm competitive for the range $30 \leq DF \leq 100$ is presented and the use of Algorithm 209 is discussed.

Thanks to the referee for many helpful suggestions, which have been incorporated, and to Joan Warner, who has aided in the programming and testing of this algorithm.

References:

1. Alder, H. L., and Roessler, E. B. Introduction to probability and statistics, 3rd ed. W. H. Freeman and Co., San Francisco, 1964, p. 125
2. Gosset, W. S. (Student). The probable error of a mean. BIOMETRIKA *6* (1908), 1.
3. Ibbetson, D. Algorithm 209, Gauss. *Comm. ACM*, *6* (Oct. 1963), 616.
4. Morris, J. Algorithm 321, *t*-test probabilities. *Comm. ACM* *11* (Feb. 1968), 115.

```
      SUBROUTINE TTEST
C     ****************
     *  (T,DF,ANS,KERR)
C
      REAL      ANS,D1,D2,F1,F2,T,T1,T2
C
      INTEGER   DF,I,KERR,N
C
      DATA      D1/.63661977/
C
C     0.636619772367581 34...= 2/ PI
C
      KERR = 0
C
      IF(DF.GT.0) GO TO 1
C
C     ERROR RETURN IF DF NOT POSITIVE
C
      KERR = 1
      ANS  = 0.
      RETURN
C
C     BEGIN COMPUTATION OF SERIES
C
    1 T    = ABS(T)
      T1   = T/SQRT(FLOAT(DF))
      T2   = 1./(1.+T1*T1)
C
      IF((DF/2)*2.EQ.DF) GO TO 5
C
C     DF IS AN ODD INTEGER
C
      ANS = 1.-D1*ATAN(T1)
C
      IF(DF.EQ.1) GO TO 4
C
      D2  = D1*T1*T2
      ANS = ANS-D2
C
      IF(DF.EQ.3) GO TO 4
C
      F1  = 0.
    2 N   = (DF-2)/2
      DO 3 I=1,N
      F2  = 2.*FLOAT(I)-F1
      D2  = D2*T2*F2/(F2+1.)
    3 ANS = ANS-D2
C
C     COMMON RETURN AFTER COMPUTATION
C
    4 IF(ANS.LT.0.) ANS = 0.
      RETURN
C
```

```
C       DF IS AN EVEN INTEGER
C
    5   D2  = T1*SQRT(T2)
        ANS = 1.-D2
C
        IF(DF.EQ.2) GO TO 4
C
        F1  = 1.
        GO TO 2
        END
```

REMARKS ON
ALGORITHM 321 [S14] *t*-TEST PROBABILITIES [John Morris, *Comm. ACM 11* (Feb. 1968), 115-6]
ALGORITHM 344, STUDENT'S *t*-DISTRIBUTION [David Levine, *Comm. ACM 12* (Jan. 1969), 37-8]

G. W. Hill, and Mary Loughhead* (Recd. 16 Apr. 1969 and 29 Sept. 1969)
Commonwealth Scientific and Industrial Research Organization, Division of Mathematical Statistics, Glen Osmond, South Australia
* Present address: Monash University, Clayton, Victoria, Australia

KEY WORDS AND PHRASES: *t*-test, Student's *t*-statistic, distribution function, approximation
CR CATEGORIES: 5.12, 5.5

Algorithm 321, as published, was coded in CSIRO 3200 Algol and run on a CDC 3200 with programmed floating point operations. A Fortran equivalent of Algorithm 321 was run for comparison with the Fortran Algorithm 344, which uses the same recurrence relation based on Student's cosine formula as that used in Algorithm 321 for *df* degrees of freedom less than *maxn*. Numerical results agreed with 6-digit tabulated values [1] and double precision calculations indicate that accuracy is limited by truncation of intermediate results to the precision of the processor, with error in the final result increasing as the square root of *df*. Timing tests rated Algorithm 344 at approximately ($\frac{3}{4}$ *df*+$1\frac{1}{2}$) msec; slightly faster than Algorithm 321, which required approximately ($\frac{3}{4}$ *df*+$2\frac{1}{2}$)msec for *df* < *maxn*.

For *df* ≥ *maxn* Algorithm 321 uses Fisher's [2] fifth order approximation, whose accuracy is summarized in the diagram for *df* = 10(10)50 (see Figure 1). The shaded regions indicate values of *t* for which the claimed accuracy of 3×10^{-7} for *maxn* = 30 is not attained. For $t > 6.0$ this algorithm returns zero values, giving errors up to 1.39×10^{-6}. The following alterations avoid this error and, by "nesting" Fisher's polynomial approximation, reduced the time from about 25msec to 20msec and reduced the store requirement by 27%.

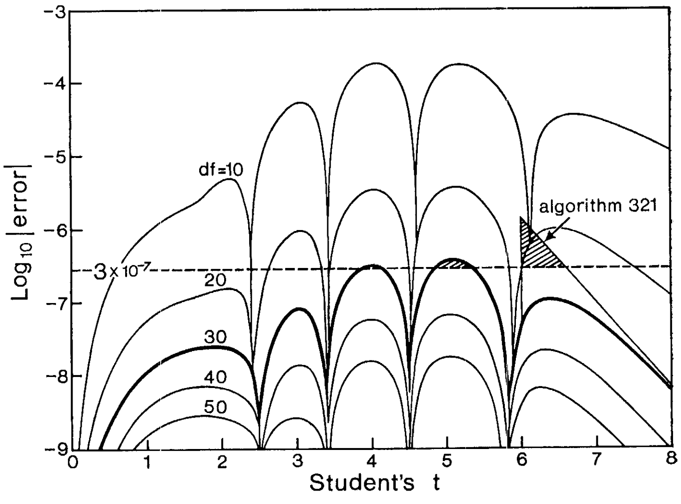


Fig. 1

Replace the 19 lines beginning "*g*: *t* := 1.0 − *t*" by

```
g: x := 1.0 − t
end else
begin x := 2.0 × gauss (−t);
if df < ₁₀6 then
begin real f, t2;
f := 0.25/df;  t2 := t × t;
x := ((((((((((((3.0×t2−133.0)×t2
   +1764.0)×t2−7516.0)×t2+5994.0)×t2+2490.0)×t2
   +1140.0)×t2+180.0)×t2+5355.0)×t2+17955.0) × f
   + ((((((15.0×t2−375.0)×t2+2225.0)×t2−2141.0)×t2
   −939.0)×t2−213.0)×t2−915.0)×t2+945.0) × f/60.0
   + ((((t2−11.0)×t2+14.0)×t2+6.0)×t2−3.0)×t2−15.0) × f
   +((3.0×t2−7.0)×t2−5.0)×t2−3.0)×f/6.0
   +(t2+1.0)) × f × t × 0.7978845608 × exp (−0.5×t2) + x
end;
ttest := if x < 0.0 then 0.0 else x
```

The last statement, recommended by the referee, avoids negative results due to rounding errors when the answer is small.

In Algorithm 344 the three statements beginning "1 T = ABS(T)" were replaced by:

```
1  T2 = T*T/FLOAT(DF)
   T1 = SQRT(T2)
   T2 = 1./(1.+T2)
```

to avoid changing the calling parameter T.

Although Algorithm 321 occupies about twice the store space needed for Algorithm 344, and is slightly slower for *df* < *maxn* = 30, it is about three times faster for *df* = 100.

References:
1. Smirnov, N. V. *Tables for the Distribution and Density Functions of t-distribution.* Pergamon Press, New York, 1961.
2. Fisher, R. A. Expansion of "Student's" integral in powers of n^{-1}. *Metron. 5*, 3 (1926), 109–112.

REMARKS ON:
ALGORITHM 332 [S22] JACOBI POLYNOMIALS [Bruno F. W. Witte, *Comm. ACM 11* (June 1968), 436]
ALGORITHM 344 [S14] STUDENT'S *t*-DISTRIBUTION [David A. Levine, *Comm. ACM 12* (Jan. 1969), 37]
ALGORITHM 351 [D1] MODIFIED ROMBERG QUADRATURE [Graeme Fairweather, *Comm. 12* (June 1969), 324]
ALGORITHM 359 [G1] FACTORIAL ANALYSIS OF VARIANCE [John R. Howell, *Comm. ACM 12* (Nov. 1969), 631]

Arthur H. J. Sale (Recd. 16 Feb. 1970)
Basser Computing Department, University of Sydney, Sydney, Australia

KEY WORDS AND PHRASES: Fortran standards
CR CATEGORIES: 4.0, 4.22

An unfortunate precedent has been set in several recent algorithms of using an illegal FORTRAN construction. This consists of separating an initial line from its continuation line by a comment line, and is forbidden by the standard (see sections 3.2.1, 3.2.3 and 3.2.4 of [1, 2]). The offending algorithms are to date: 332, 344, 351 and 359.

While this is perhaps a debatable decision by the compilers of the standard, and trivial to correct, it seems a pity to break the rules just for a pretty layout as has been done.

References:

1. ANSI Standard FORTRAN (ANSI X3.9-1966), American National Standards Institute, New York, 1966.
2. FORTRAN vs. Basic FORTRAN, *Comm. ACM 7* (Oct. 1964), 591-625.

ALGORITHM 345
AN ALGOL CONVOLUTION PROCEDURE BASED ON THE FAST FOURIER TRANSFORM [C6]

Richard C. Singleton* (Recd. 30 Dec. 1966, 26 July 1967, 19 July 1968, and 8 Nov. 1968)
Stanford Research Institute, Menlo Park, CA 94025

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

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

Stockham [6] and Gentleman and Sande [3] have shown the practical advantages of computing the circular convolution

$$C_k = \sum_{j=0}^{n-1} A_j B_{(j+k) \bmod n}\,, \qquad k = 0, 1, \cdots, n-1,$$

of two real vectors A and B of period n by the fast Fourier transform [2, 3, 4]. The Fourier transforms

$$\alpha_j = \sum_{p=0}^{n-1} A_p \exp(i2\pi pj/n)$$

and

$$\beta_j = \sum_{q=0}^{n-1} B_q \exp(i2\pi qj/n)$$

are first computed, then the convolution

$$C_k = \frac{1}{n} \sum_{j=0}^{n-1} \alpha_j \beta_j^* \exp(i2\pi jk/n)$$

where β_j^* is the complex conjugate of β_j. By this method the number of arithmetic operations increases by a factor slightly more than 2 when n is doubled, as compared with a factor of 4 for the direct method. Tests show a 16 to 1 time advantage for the transform method at $n = 256$.

The operation of convolution is used in computing autocorrelation and cross-correlation functions, in digital filtering of time series, and many other applications.

Procedure *CONVOLUTION* computes the convolution of two real vectors of dimension $n = 2^m$. The special features of this procedure are: (1) the usual reordering of the fast Fourier transform results is avoided, and (2) the return from frequency to time is made with a transform of dimension $n/2$ instead of n. The two vectors A and B are first transformed with a single complex Fourier transform of dimension n. The complex product $\alpha\beta^*$ is then formed, leaving the result in reverse binary order. Since the convolution is real-valued, the real part x of the complex product is an even function and the imaginary part y is an odd function; thus the Fourier transform of x is real and that of y is imaginary. These properties lead to the identity

$$\begin{aligned} T(x+iy) &= \mathrm{Re}(Tx) - \mathrm{Im}(Ty) \\ &= \mathrm{Re}(T(x-y)) + \mathrm{Im}(T(x-y)) \end{aligned}$$

where T represents the Fourier transform and $T(x+iy)$ is the desired convolution. We subtract y from x, yielding a real vector of dimension n, then transform using a complex transform of dimension $n/2$ and add the resulting cosine and sine coefficients to give the convolution. Thus with procedure *CONVOLUTION* we make maximum use of the complex Fourier transform in each direction and avoid any reverse binary to binary permutation. The Fourier transform

$$T(A + iB) = \alpha + i\beta$$

of the two original vectors is available in reverse binary order on exit from the procedure. We can permute this transform to normal order with procedure *REVERSEBINARY* and readily compute the power spectra and cross spectrum of the two data vectors.

Procedure *CONVOLUTION* uses procedure *REALTRAN*, given in Algorithm 338 [5], but repeated here with revisions to improve accuracy on computers using truncated floating-point arithmetic. Procedures *FFT*4 and *REVFFT*4 are also used and perform the same computation as procedures *FFT*2 and *REVFFT*2 given in Algorithm 338 for use on a system with virtual memory. The transform procedures given here are organized without regard to the problem of memory overlay. This change yields a 10 percent reduction in computing time on the Burroughs B5500 for transforms of dimension $n = 512$ or smaller. Procedure *FFT*4 is based on an organization of the fast Fourier transform due to Sande [3], and procedure *REVFFT*4 is similar to the method proposed by Cooley and Tukey [2], except that the data is in reverse binary order. In both cases, trigonometric functions are used in normal sequence, rather than reverse binary sequence, thus eliminating the need for a reverse binary counter. Another gain in efficiency comes from reducing the time for computing trigonometric function values. The following difference-equation method is used:

$$\cos((k+1)\theta) = \cos(k\theta) - (C \times \cos(k\theta) + S \times \sin(k\theta))$$

and

$$\sin((k+1)\theta) = \sin(k\theta) + (S \times \cos(k\theta) - C \times \sin(k\theta)),$$

where the constant multipliers are $C = 2\sin^2(\theta/2)$ and $S = \sin(\theta)$, and the initial values are $\cos(0) = 1$ and $\sin(0) = 0$.

These initial values should be computed to full machine precision; if necessary, a stored table of $\sin(\theta)$ for $\theta = \pi/2, \pi/4, \pi/8, \cdots, \pi/n$ can be added to procedures *FFT*4 and *REVFFT*4. Using the standard sine function to compute initial values, the ratio of rms error to rms data is about 2×10^{-11} for the transform-inverse pair at $n = 512$ on the Burroughs B5500 computer; this error is about the same as that obtained when the sine and cosine functions are used for all trigonometric function values. On a computer using truncated, rather than rounded, arithmetic operations, the sequence of values for $\cos(k\theta) + i\sin(k\theta)$ tends to spiral inward from the unit circle. Since the error is primarily one of magnitude, rather than angle, rescaling to the unit circle at each step gives a satisfactory correction. This correction is included in procedures *FFT*4 and *REVFFT*4 but may be removed to improve running speed if rounded arithmetic is used.

Procedures *FFT*8 and *REVFFT*8 are included as possible substitutes for *FFT*4 and *REVFFT*4. These procedures use radix 8 arithmetic [1], rather than radix 4, and run about 20 percent faster on the Burroughs B5500 computer; however, the compiled code is twice as long. The code could be shortened by use of subscripted variables and FOR statements, but this change would probably eliminate most of the time-saving.

The permutation procedure *REVERSEBINARY* is based on a modified dual counter, one in normal sequence and the other in reverse binary sequence. In permuting a vector of dimension n, the normal sequence counter goes from 1 to $n/2 - 1$, and the elements indexed $1, 3, \cdots, n/2 - 1$ are exchanged with their reverse-binary counterparts (indexed greater than or equal to $n/2$) without need of a test. The reverse binary counter is incremented only $n/4$ times, and exchanges of pairs of elements below $n/2$ are done jointly with pair exchanges in the upper half of the array; i.e. if x_j and x_k are exchanged, where $j, k < n/2$, then x_{n-1-j} and x_{n-1-k} are also exchanged. This procedure is twice as fast on the Burroughs B5500 as *REORDER* given in Algorithm 338 [5] and is the better choice when the additional features of *REORDER* are not needed. For a single-variate, complex Fourier transform of dimension $n = 2^m$,

$$REVERSEBINARY(A, B, m);$$

$$REVFFT8(A, B, n, m, 1)$$

was found to be the best combination for $n \leqq 512$ on the B5500 computer, giving a time of 0.79 sec. for $n = 512$.

References:

1. Bergland, G. D. A fast Fourier transform algorithm using base 8 iterations. *Math. Comput. 22*, 102 (Apr. 1968), 275–279.
2. Cooley, J. W., and Tukey, J. W. An algorithm for the machine calculation of complex Fourier series. *Math. Comput. 19*, 90 (Apr. 1965), 297–301.
3. Gentleman, W. G., and Sande, G. Fast Fourier transforms—for fun and profit. Proc. AFIPS 1966 Fall Joint Comput. Conf., Vol. 29, Spartan Books, New York, 1966, pp. 563–578.
4. Singleton, R. C. On computing the fast Fourier transform. *Comm. ACM 10* (Oct. 1967), 647–654.
5. Singleton, R. C. Algorithm 338, Algol procedures for the fast Fourier transform. *Comm. ACM 11* (Nov. 1968), 773–776.
6. Stockham, T. G. High-speed convolution and correlation. Proc. AFIPS 1966 Spring Joint Comput. Conf., Vol. 28, Spartan Books, New York, 1966, pp. 229–233;

procedure *CONVOLUTION* $(A, B, C, D, m, scale)$;
  value $m, scale$; **integer** m; **real** $scale$; **array** A, B, C, D;

comment This procedure computes the circular convolution

$$C_k = scale \sum_{j=0}^{n-1} A_j B_{(j+k) \bmod n}, \qquad k = 0, 1, \cdots, n-1,$$

where $n = 2^m$ and p mod n represents the remainder after division of p by n. (It is assumed that $m \geqq 1$.) Arrays $A, B[0:n-1]$ originally contain the two data vectors to be convoluted, and on exit, contain the Fourier transform of $A + iB$ arranged in reverse binary order. A and B must not be the same array. On exit, array $C[0:n-1]$ contains the convolution multiplied by the factor *scale*. Array D is a scratch storage array with lower bound zero and upper bound at least $n \div 2$. If the Fourier transform of the data is not needed, the procedure can be called with arrays A and B used for C and D in either order, for example, *CONVOLUTION* $(A, B, A, B, m, scale)$. If the Fourier transform is used, it should first be permuted to normal order by the call $REVERSEBINARY(A, B, m)$. After doing this, the Fourier cosine coefficients of the A vector are

$$(A_k + A_{n-k})/n, \qquad k = 1, 2, \cdots, n/2,$$
$$(2A_0)/n, \qquad k = 0,$$

and the sine coefficients are

$$(B_k - B_{n-k})/n, \qquad k = 1, 2, \cdots, n/2 - 1.$$

The Fourier cosine coefficients of the B vector are

$$(B_k + B_{n-k})/n, \qquad k = 1, 2, \cdots, n/2,$$
$$(2B_0)/n, \qquad k = 0,$$

and the sine coefficients are

$$(A_{n-k} - A_k)/n, \qquad k = 1, 2, \cdots, n/2 - 1.$$

The procedures *FFT*4, *REVFFT*4, and *REALTRAN* are used by this procedure and must also be declared. If convolutions of large dimension are to be computed on a system with virtual memory, procedures *FFT*2 and *REVFFT*2 (Algorithm 338) [5] should be substituted for procedures *FFT*4 and *REVFFT*4;

```
begin integer j, kk, ks, n;  real aa, ab, ba, bb, im;
  n := 2↑m;  j := 1;
  FFT4(A, B, n, m, n);
  C[0] := 4 × (A[0]×B[0]);
L:   kk := j;  ks := j := j + j;
L2:  ks := ks − 1;
  aa := A[kk] + A[ks];  ab := A[kk] − A[ks];
  ba := B[kk] + B[ks];  bb := B[kk] − B[ks];
  im := ba × bb + aa × ab;  aa := aa × ba − ab × bb;
  C[kk] := aa − im;  C[ks] := aa + im;
  kk := kk + 1;  if kk < ks then go to L2;
  if j < n then go to L;
  kk := n ÷ 2;  ks := kk − 1;  scale := scale/(8×n);
  for j := 0 step 1 until ks do D[j] := C[j+kk];
  REVFFT4(C, D, kk, m−1, 1);
  REALTRAN(C, D, kk, false);
  C[0] := scale × C[0];  C[kk] := scale × C[kk];
  for j := 1 step 1 until ks do
  begin C[n−j] := scale × (C[j]−D[j]);
  end
  C[j] := scale × (C[j]+D[j])
end CONVOLUTION;
procedure FFT4(A, B, n, m, ks);  value n, m, ks;
  integer n, m, ks;  array A, B;
```

comment This procedure computes the fast Fourier transform for one variable of dimension 2^m in a multivariate transform. n is the number of data points, i.e. $n = n_1 \times n_2 \times \cdots \times n_p$ for a p-variate transform, and $ks = n_k \times n_{k+1} \times \cdots \times n_p$, where $n_k = 2^m$ is the dimension of the current variable. Arrays $A[0:n-1]$ and $B[0:n-1]$ originally contain the real and imaginary components of the data in normal order. Multivariate data is stored according to the usual convention, e.g. a_{jkl} is in $A[j \times n_2 \times n_3 + k \times n_3 + l]$ for $j = 0, 1, \cdots, n_1 - 1$, $k = 0, 1, \cdots, n_2 - 1$, and $l = 0, 1, \cdots, n_3 - 1$. On exit, the Fourier coefficients for the current variable are in reverse binary order. Continuing the above example, if the "column" variable n_2 is the current one, column

$$k = k_{m-1}2^{m-1} + k_{m-2}2^{m-2} + \cdots + k_1 2 + k_0$$

is permuted to position

$$k_0 2^{m-1} + k_1 2^{m-2} + \cdots + k_{m-2} 2 + k_{m-1}.$$

A separate procedure may be used to permute the results to normal order between transform steps or all at once at the end. If $n = ks = 2^m$, the single-variate transform

$$(x_j + iy_j) = \sum_{k=0}^{n-1} (a_k + ib_k) \exp(i2\pi jk/n)$$

for $j = 0, 1, \cdots, n-1$ is computed, where $(a+ib)$ represent the initial values and $(x+iy)$ represent the transformed values;

```
begin integer k0, k1, k2, k3, k, span;
  real A0, A1, A2, A3, B0, B1, B2, B3, re, im;
  real rad, dc, ds, c1, c2, c3, s1, s2, s3;
  span := ks;  ks := 2↑m;  rad := 4.0 × arctan(1.0)/ks;
  ks := span ÷ ks;  n := n − 1;  k := m;
  for m := m − 2 while m ≧ 0 do
  begin
    c1 := 1.0;  s1 := 0;  k0 := 0;  k := ks;
    dc := 2.0 × sin(rad)↑2;  rad := rad + rad;
    ds := sin(rad);  rad := rad + rad;
    span := span ÷ 4;
La: k1 := k0 + span;  k2 := k1 + span;  k3 := k2 + span;
    A0 := A[k0];  B0 := B[k0];
    A1 := A[k1];  B1 := B[k1];
    A2 := A[k2];  B2 := B[k2];
    A3 := A[k3];  B3 := B[k3];
    A[k0] := A0 + A2 + A1 + A3;
    B[k0] := B0 + B2 + B1 + B3;
    if s1 = 0 then
    begin
      A[k1] := A0 + A2 − A1 − A3;
      B[k1] := B0 + B2 − B1 − B3;
      A[k2] := A0 − A2 − B1 + B3;
      B[k2] := B0 − B2 + A1 − A3;
      A[k3] := A0 − A2 + B1 − B3;
      B[k3] := B0 − B2 − A1 + A3
    end
    else
    begin
      re := A0 + A2 − A1 − A3;  im := B0 + B2 − B1 − B3;
      A[k1] := re × c2 − im × s2;
      B[k1] := re × s2 + im × c2;
      re := A0 − A2 − B1 + B3;  im := B0 − B2 + A1 − A3;
      A[k2] := re × c1 − im × s1;
      B[k2] := re × s1 + im × c1;
      re := A0 − A2 + B1 − B3;  im := B0 − B2 − A1 + A3;
      A[k3] := re × c3 − im × s3;
      B[k3] := re × s3 + im × c3
    end;
    k0 := k3 + span;  if k0 < n then go to La;
    k0 := k0 − n;  if k0 ≠ k then go to La;
    comment If computing for the current factor of 4 is not
      finished then increment the sine and cosine values;
    if k0 ≠ span then
    begin
      c2 := c1 − (dc×c1+ds×s1);
      s1 := (ds×c1−dc×s1) + s1;
      comment The following three statements compensate
        for truncation error. If rounded arithmetic is used, sub-
        stitute c1 := c2;
      c1 := 1.5−0.5 × (c2↑2+s1↑2);
      s1 := c1 × s1;  c1 := c1 × c2;
      c2 := c1↑2 − s1↑2;  s2 := 2.0 × c1 × s1;
      c3 := c2 × c1 − s2 × s1;  s3 := c2 × s1 + s2 × c1;
      k := k + ks;  go to La
    end;
    k := m
  end,
  comment If m is odd then compute for one factor of 2;
  if k ≠ 0 then
  begin
    span := span ÷ 2;  k0 := 0;
Lb: k2 := k0 + span;  A0 := A[k2];  B0 := B[k2];
    A[k2] := A[k0] − A0;  A[k0] := A[k0] + A0;
    B[k2] := B[k0] − B0;  B[k0] := B[k0] + B0;
    k0 := k2 + span;  if k0 < n then go to Lb;
    k0 := k0 − n;  if k0 ≠ span then go to Lb
  end
end FFT4;
procedure REVFFT4(A, B, n, m, ks);  value n, m, ks;
  integer n, m, ks;  array A, B;
comment This procedure computes the fast Fourier transform
  for one variable of dimension 2^m in a multivariate transform.
  n is the number of data points, i.e. n = n1 × n2 × ··· × np
  for a p-variate transform, and ks = nk+1 × nk+2 × ··· × np ,
  where nk = 2^m is the dimension of the current variable. Arrays
  A[0:n−1] and B[0:n−1] originally contain the real and imagi-
  nary components of the data with the indices of each variable
  in reverse binary order, e.g. ajkl is in A[j'×n2×n3+k'×n3+l']
  for j = 0, 1, ··· , n1 − 1,  k = 0, 1, ··· n2 − 1,  and  l = 0,
  1, ··· n3 − 1, where j', k', and l' are the bit-reversed values of
  j, k, and l. On completion of the multivariate transform, the
  real and imaginary components of the resulting Fourier coeffi-
  cients are in A and B in normal order. If n = 2^m and ks = 1,
  a single-variate transform is computed;
begin integer k0, k1, k2, k3, k, span;
  real A0, A1, A2, A3, B0, B1, B2, B3;
  real rad, dc, ds, c1, c2, c3, s1, s2, s3;
  rad := 4.0 × arctan(1.0);  n := n − 1;
  k0 := 0;  span := ks;
  comment If m is odd then compute for one factor of 2;
  if (m÷2) × 2 ≠ m then
  begin
La: k2 := k0 + span;  A0 := A[k2];  B0 := B[k2];
    A[k2] := A[k0] − A0;  A[k0] := A[k0] + A0;
    B[k2] := B[k0] − B0;  B[k0] := B[k0] + B0;
    k0 := k2 + span;  if k0 < n then go to La;
    k0 := k0 − n;  if k0 ≠ span then go to La;
    span := span + span;  rad := 0.5 × rad
  end;
  for m := m − 2 while m ≧ 0 do
  begin
    c1 := 1.0;  s1 := 0;  k0 := 0;  rad := 0.25 × rad;
    dc := 2.0 × sin(rad)↑2;
    ds := sin(rad+rad);  k := ks;
Lb: k1 := k0 + span;  k2 := k1 + span;  k3 := k2 + span;
    A0 := A[k0];  B0 := B[k0];
    if s1 = 0 then
    begin
      A2 := A[k1];  B2 := B[k1];
      A1 := A[k2];  B1 := B[k2];
      A3 := A[k3];  B3 := B[k3]
    end
    else
    begin
      A2 := A[k1] × c2 − B[k1] × s2;
      B2 := A[k1] × s2 + B[k1] × c2;
      A1 := A[k2] × c1 − B[k2] × s1;
      B1 := A[k2] × s1 + B[k2] × c1;
      A3 := A[k3] × c3 − B[k3] × s3;
      B3 := A[k3] × s3 + B[k3] × c3
    end;
    A[k0] := A0 + A2 + A1 + A3;
    B[k0] := B0 + B2 + B1 + B3;
    A[k1] := A0 − A2 − B1 + B3;
    B[k1] := B0 − B2 + A1 − A3;
    A[k2] := A0 + A2 − A1 − A3;
    B[k2] := B0 + B2 − B1 − B3;
    A[k3] := A0 − A2 + B1 − B3;
    B[k3] := B0 − B2 − A1 + A3;
    k0 := k3 + span;  if k0 < n then go to Lb;
    k0 := k0 − n;  if k0 ≠ k then go to Lb;
    comment If computing for the current factor of 4 is not
      finished then increment the sine and cosine values;
    if k0 ≠ span then
    begin
      c2 := c1 − (dc×c1+ds×s1);
      s1 := (ds×c1−dc×s1) + s1;
      comment The following three statements compensate
        for truncation error. If rounded arithmetic is used, sub-
        stitute c1 := c2;
      c1 := 1.5−0.5 × (c2↑2+s1↑2);
      s1 := c1 × s1;  c1 := c1 × c2;
      c2 := c1↑ 2 − s1↑2;  s2 := 2.0 × c1 × s1;
      c3 := c2 × c1 − s2 × s1;  s3 := c2 × s1 + s2 × c1;
      k := k + ks;  go to Lb
    end;
    span := 4 × span
  end
end REVFFT4;
procedure REALTRAN(A, B, n, evaluate);
  value n, evaluate;  integer n;
  Boolean evaluate;  array A, B;
comment If evaluate is false, this procedure unscrambles the
  single-variate complex transform of the n even-numbered and
  n odd-numbered elements of a real sequence of length 2n, where
  the even-numbered elements were originally in A and the odd-
  numbered elements in B. Then it combines the two real trans-
  forms to give the Fourier cosine coefficients A[0], A[1], ··· ,
  A[n] and sine coefficients B[0], B[1], ··· , B[n] for the full
  sequence of 2n elements. If evaluate is true, the process is
  reversed, and a set of Fourier cosine and sine coefficients is
  made ready for evaluation of the corresponding Fourier series
  by means of the inverse complex transform. Going in either
  direction, REALTRAN scales by a factor of two, which should
  be taken into account in determining the appropriate overall
  scaling;
begin integer k, nk, nh;
  real aa, ab, ba, bb, re, im, ck, sk, dc, ds;
  nh := n ÷ 2;  ds := 2.0 × arctan(1.0)/n;
  dc := 2.0 × sin(ds) ↑ 2;  ds := sin(ds+ds);
  sk := 0;
  if evaluate then
    begin ck := −1.0;  ds := −ds end
  else begin ck := 1.0;  A[n] := A[0];  B[n] := B[0]  end;
  for k := 0 step 1 until nh do
  begin
    nk := n − k;
    aa := A[k] + A[nk];  ab := A[k] − A[nk];
    ba := B[k] + B[nk];  bb := B[k] − B[nk];
    re := ck × ba + sk × ab;  im := sk × ba − ck × ab;
    B[nk] := im − bb;  B[k] := im + bb;
    A[nk] := aa − re;  A[k] := aa + re;
    aa := ck − (dc×ck+ds×sk);
    sk := (ds×ck−dc×sk) + sk;
    comment The following three statements compensate for
      truncation error. If rounded arithmetic is used, substitute
      ck := aa;
    ck := 1.5−0.5 × (aa↑2+sk↑2);
    sk := ck × sk;  ck := ck × aa
  end
end REALTRAN;
procedure REVERSEBINARY(A, B, m);  value m;
  integer m; array A, B;
 comment This procedure permutes the elements A[j] and B[j]
   of arrays A and B, for j = 0, 1, ··· , 2↑m − 1, according to
   the reverse binary transformation. Element
```

$$k = k_{m-1}2^{m-1} + k_{m-2}2^{m-2} + \cdots + k_1 2 + k_0$$

```
   is moved to location
```

$$k_0 2^{m-1} + k_1 2^{m-2} + \cdots + k_{m-2}2 + k_{m-1} .$$

```
  Two successive calls of this procedure give an identity trans-
  formation;
  begin integer j, jj, k, lim, jk, n2, n4, n8, nn;
  real t;
  integer array C[0:m];
  C[0] := nn := 1;  jj := 0;
  for j := 1 step 1 until m do C[j] := nn := nn + nn;
  if m > 1 then n4 := C[m−2];  if m > 2 then n8 := C[m−3];
  n2 := C[m−1];  lim := n2 − 1;  nn := nn − 1;  m := m − 4;
  for j := 1 step 1 until lim do
  begin
    jk := jj + n2;
    t := A[j];  A[j] := A[jk];  A[jk] := t;
    t := B[j];  B[j] := B[jk];  B[jk] := t;
    j := j + 1;
    if jj ≧ n4 then
    begin
    jj := jj − n4;
      if jj ≧ n8 then
      begin
        jj := jj − n8;  k := m;
L:  if C[k] ≦ jj then
  begin jj := jj − C[k];  k := k − 1;  go to L end;
    jj := C[k] + jj
      end
      else jj := jj + n8
    end
    else jj := jj + n4;
    if jj > j then
    begin
      k := nn − j;  jk := nn − jj;
      t := A[j];  A[j] := A[jj];  A[jj] := t;
      t := B[j];  B[j] := B[jj];  B[jj] := t;
      t := A[k];  A[k] := A[jk];  A[jk] := t;
      t := B[k];  B[k] := B[jk];  B[jk] := t
    end
  end
end REVERSEBINARY;
procedure FFT8(A, B, n, m, ks);  value n, m, ks;
  integer n, m, ks;  array A, B;
comment This procedure computes the fast Fourier transform
  for one variable of dimension 2^m in a multivariate transform.
  n is the number of data points, i.e. n = n₁ × n₂ × ··· × nₚ
  for a p-variate transform, ks = n_k × n_{k+1} × ··· × n_p , where
  n_k = 2^m is the dimension of the current variable. Arrays A[0:n−
  1] and B[0:n−1] originally contain the real and imaginary com-
  ponents of the data in normal order. Multivariate data is stored
  according to the usual convention, e.g. a_{jkl} is in A[j×n₂×n₃+
  k×n₃+l] for j = 0, 1, ··· , n₁ − 1,  k = 0, 1, ··· , n₂ − 1, and
  l = 0, 1, ··· , n₃ − 1. On exit, the Fourier coefficients for the
  current variable are in reverse binary order. Continuing the
  above example, if the "column" variable n₂ is the current one,
  column
```

$$k = k_{m-1}2^{m-1} + k_{m-2}2^{m-2} + \cdots + k_1 2 + k_0$$

```
  is permuted to position
```

$$k_0 2^{m-1} + k_1 2^{m-2} + \cdots + k_{m-2}2 + k_{m-1} .$$

```
  A separate procedure may be used to permute the results to
  normal order between transform steps or all at once at the end.
  If n = ks = 2^m, the single variate transform
```

$$(x_j+iy_j) = \sum_{k=0}^{n-1} (a_k+ib_k) \exp\ (i2\pi jk/n)$$

```
  for j = 0, 1, ··· , n − 1 is computed, where (a+ib) represent
  the initial values and (x+iy) represent the transformed values;
begin integer k0, k1, k2, k3, k4, k5, k6, k7, k, span;
```

```
  real A0, A1, A2, A3, A4, A5, A6, A7, B0, B1, B2, B3, B4, B5,
    B6, B7, x0, x1, x2, x3, x4, x5, x6, x7, y0, y1, y2, y3, y4, y5, y6, y7,
    c1, c2, c3, c4, c5, c6, c7, s1, s2, s3, s4, s5, s6, s7, c45, dc, ds, rad;
  span := ks;  ks := 2↑m;  rad := 4.0 × arctan(1.0)/ks;
  ks := span ÷ ks;  n := n − 1;  c45 := sqrt(0.5);  k := m;
  comment  Radix 8 transform;
  for m := m − 3 while m ≧ 0 do
  begin
    c1 := 1.0;  s1 := 0;  k0 := 0;  k := ks;
    dc := 2.0 × sin(rad)↑2;  rad := rad + rad;
    ds := sin(rad);  rad := 4 × rad;
    span := span ÷ 8;
La: k1 := k0 + span;  k2 := k1 + span;  k3 := k2 + span;
    k4 := k3 + span;  k5 := k4 + span;  k6 := k5 + span;
    k7 := k6 + span;  A0 := A[k0];  B0 := B[k0];
    A1 := A[k1];  B1 := B[k1];
    A2 := A[k2];  B2 := B[k2];
    A3 := A[k3];  B3 := B[k3];
    A4 := A[k4];  B4 := B[k4];
    A5 := A[k5];  B5 := B[k5];
    A6 := A[k6];  B6 := B[k6];
    A7 := A[k7];  B7 := B[k7];
    x0 := A0 + A4;  y0 := B0 + B4;
    x4 := A0 − A4;  y4 := B0 − B4;
    x1 := A1 + A5;  y1 := B1 + B5;
    x5 := (A1 − A5 − B1 + B5) × c45;
    y5 := (A1 − A5 + B1 − B5) × c45;
    x2 := A2 + A6;  y2 := B2 + B6;
    x6 := B6 − B2;  y6 := A2 − A6;
    x3 := A3 + A7;  y3 := B3 + B7;
    x7 := (A7−A3−B3+B7) × c45;
    y7 := (A3−A7−B3+B7) × c45;
    A1 := x0 + x2 − x1 − x3;  B1 := y0 + y2 − y1 − y3;
    A2 := x0 − x2 − y1 + y3;  B2 := y0 − y2 + x1 − x3;
    A3 := x0 − x2 + y1 − y3;  B3 := y0 − y2 − x1 + x3;
    A4 := x4 + x6 + x5 + x7;  B4 := y4 + y6 + y5 + y7;
    A5 := x4 + x6 − x5 − x7;  B5 := y4 + y6 − y5 − y7;
    A6 := x4 − x6 − y5 + y7;  B6 := y4 − y6 + x5 − x7;
    A7 := x4 − x6 + y5 − y7;  B7 := y4 − y6 − x5 + x7;
    A[k0] := x0 + x2 + x1 + x3;  B[k0] := y0 + y2 + y1 + y3;
    if s1 = 0 then
    begin
      A[k1] := A1;  B[k1] := B1;
      A[k2] := A2;  B[k2] := B2;
      A[k3] := A3;  B[k3] := B3;
      A[k4] := A4;  B[k4] := B4;
      A[k5] := A5;  B[k5] := B5;
      A[k6] := A6;  B[k6] := B6;
      A[k7] := A7;  B[k7] := B7
    end
    else
    begin
      A[k1] := c4 × A1 − s4 × B1;
      B[k1] := s4 × A1 + c4 × B1;
      A[k2] := c2 × A2 − s2 × B2;
      B[k2] := s2 × A2 + c2 × B2;
      A[k3] := c6 × A3 − s6 × B3;
      B[k3] := s6 × A3 + c6 × B3;
      A[k4] := c1 × A4 − s1 × B4;
      B[k4] := s1 × A4 + c1 × B4;
      A[k5] := c5 × A5 − s5 × B5;
      B[k5] := s5 × A5 + c5 × B5;
      A[k6] := c3 × A6 − s3 × B6;
      B[k6] := s3 × A6 + c3 × B6;
      A[k7] := c7 × A7 − s7 × B7;
      B[k7] := s7 × A7 + c7 × B7
    end;
    k0 := k7 + span;  if k0 < n then go to La;
    k0 := k0 − n;  if k0 ≠ k then go to La;
    comment Increment sine and cosine values;
    if k0 ≠ span then
    begin
      c2 := c1 − (dc×c1+ds×s1);
      s1 := (ds×c1−dc×s1) + s1;
      comment  The following three statements compensate
        for truncation error. If rounded arithmetic is used,
        substitute c1 := c2;
      c1 := 1.5–0.5 × (c2↑2+s1↑2);
      s1 := c1 × s1;  c1 := c1 × c2;
      c2 := c1↑2 − s1↑2;  s2 := 2.0 × c1 × s1;
      c3 := c2 × c1−s2 × s1;  s3 := c2 × s1 + s2 × c1;
      c4 := c2↑2 − s2↑2;  s4 := 2.0 × c2 × s2;
      c5 := c1 × c4 − s1 × s4;  s5 := s1 × c4 + c1 × s4;
      c6 := c3↑2 − s3↑2;  s6 := 2.0 × c3 × s3;
      c7 := c1 × c6 − s1 × s6;  s7 := s1 × c6 + c1 × s6;
      k := k + ks;  go to La
    end;
    k3 := m
  end;
  comment  If m is not a multiple of 3, then complete the trans-
    form with radix 2 steps;
  for k3 := k3 − 1 while k3 ≧ 0 do
  begin
    k0 := 0;  span := span ÷ 2;
Lb: k2 := k0 + span;
    A2 := A[k2];  B2 := B[k2];
    A[k2] := A[k0] − A2;  B[k2] := B[k0] − B2;
    A[k0] := A[k0] + A2;  B[k0] := B[k0] + B2;
    k0 := k2 + span;  if k0 < n then go to Lb;
    k0 := k0 − n;  if k0 < ks then go to Lb;
    if ks = span then go to Ld;
Lc: k2 := k0 + span;
    A2 := A[k0] − A[k2];  B2 := B[k0] − B[k2];
    A[k0] := A[k0] + A[k2];  B[k0] := B[k0] + B[k2];
    A[k2] := −B2;  B[k2] := A2;
    k0 := k2 + span;  if k0 < n then go to Lc;
    k0 := k0 − n;  if k0 < span then go to Lc;
Ld: end
end FFT8;
procedure REVFFT8(A, B, n, m, ks);  value n, m, ks;
  integer n, m, ks;  array A, B;
comment  This procedure computes the fast Fourier transform
  for one variable of dimension 2^m in a multivariate transform.
  n is the number of data points, i.e., n = n_1 × n_2 × ··· × n_p
  for a p-variate transform, and ks = n_{k+1} × n_{k+2} × ··· × n_p ,
  where n_k = 2^m is the dimension of the current variable. Arrays
  A[0:n−1] and B[0:n−1] originally contain the real and imagi-
  nary components of the data with the indices of each variable
  in reverse binary order, e.g. a_{jkl} is in A[j'×n_2×n_3+k'×n_3+l']
  for j = 0, 1, ··· , n_1 − 1,  k = 0, 1, ··· , n_2 − 1, and l =
  0, 1, ··· , n_3 − 1, where j', k', and l' are the bit-reversed values
  of j, k, and l. On completion of the multivariate transform, the
  real and imaginary components of the resulting Fourier coeffi-
  cients are in A and B in normal order. If n = 2^m and ks = 1,
  a single-variate transform is computed;
begin integer k0, k1, k2, k3, k4, k5, k6, k7, k, span;
  real A0, A1, A2, A3, A4, A5, A6, A7, B0, B1, B2, B3, B4, B5,
    B6, B7, x0, x1, x2, x3, x4, x5, x6, x7, y0, y1, y2, y3, y4, y5, y6, y7,
    c1, c2, c3, c4, c5, c6, c7, s1, s2, s3, s4, s5, s6, s7, c45, dc, ds, rad;
  rad := 4.0 × arctan(1.0);  n := n − 1;
  c45 := sqrt(0.5);  span := ks;
  comment  Compute radix 2 steps if m is not a multiple of 3;
  k3 := (m÷3) × 3;
  for k3 := k3 + 1 while k3 ≦ m do
  begin
```

```
      k0 := 0;
La:   k2 := k0 + span;
      A2 := A[k2];  B2 := B[k2];
      A[k2] := A[k0] − A2;  B[k2] := B[k0] − B2;
      A[k0] := A[k0] + A2;  B[k0] := B[k0] + B2;
      k0 := k2 + span;  if k0 < n then go to La;
      k0 := k0 − n;  if k0 < ks then go to La;
      if ks = span then go to Lc;
Lb:   k2 := k0 + span;
      A2 := A[k2];  B2 := B[k2];
      A[k2] := A[k0] + B2;  B[k2] := B[k0] − A2;
      A[k0] := A[k0] − B2;  B[k0] := B[k0] + A2;
      k0 := k2 + span;  if k0 < n then go to Lb;
      k0 := k0 − n;  if k0 < span then go to Lb;
Lc:   span := span + span;  rad := 0.5 × rad
    end;
    comment  Radix 8 transform;
    for m := m − 3 while m ≧ 0 do
    begin
      c1 := 1.0;  s1 := 0;  k0 := 0;  k := ks;
      rad := 0.125 × rad;  dc := 2.0 × sin(rad) ↑ 2;
      ds := sin(rad+rad);
Ld:   k1 := k0 + span;  k2 := k1 + span;  k3 := k2 + span;
      k4 := k3 + span;  k5 := k4 + span;  k6 := k5 + span;
      k7 := k6 + span;  A0 := A[k0];  B0 := B[k0];
      if s1 = 0 then
      begin
        A1 := A[k1];  B1 := B[k1];
        A2 := A[k2];  B2 := B[k2];
        A3 := A[k3];  B3 := B[k3];
        A4 := A[k4];  B4 := B[k4];
        A5 := A[k5];  B5 := B[k5];
        A6 := A[k6];  B6 := B[k6];
        A7 := A[k7];  B7 := B[k7]
      end
      else
      begin
        A1 := A[k1] × c4 − B[k1] × s4;
        B1 := A[k1] × s4 + B[k1] × c4;
        A2 := A[k2] × c2 − B[k2] × s2;
        B2 := A[k2] × s2 + B[k2] × c2;
        A3 := A[k3] × c6 − B[k3] × s6;
        B3 := A[k3] × s6 + B[k3] × c6;
        A4 := A[k4] × c1 − B[k4] × s1;
        B4 := A[k4] × s1 + B[k4] × c1;
        A5 := A[k5] × c5 − B[k5] × s5;
        B5 := A[k5] × s5 + B[k5] × c5;
        A6 := A[k6] × c3 − B[k6] × s3;
        B6 := A[k6] × s3 + B[k6] × c3;
        A7 := A[k7] × c7 − B[k7] × s7;
        B7 := A[k7] × s7 + B[k7] × c7
      end;
      x0 := A0 + A1 + A2 + A3;  y0 := B0 + B1 + B2 + B3;
      x1 := A0 − A1 − B2 + B3;  y1 := B0 − B1 + A2 − A3;
      x2 := A0 + A1 − A2 − A3;  y2 := B0 + B1 − B2 − B3;
      x3 := A0 − A1 + B2 − B3;  y3 := B0 − B1 − A2 + A3;
      x4 := A4 + A5 + A6 + A7;  y4 := B4 + B5 + B6 + B7;
      x5 := (A4−A5−B6+B7) × c45;
      y5 := (B4−B5+A6−A7) × c45;
      x6 := A4 + A5 − A6 − A7;  y6 := B4 + B5 − B6 − B7;
      x7 := (A4−A5+B6−B7) × c45;
      y7 := (B4−B5−A6+A7) × c45;
      A[k0] := x0 + x4;  B[k0] := y0 + y4;
      A[k1] := x1 + x5 − y5;  B[k1] := y1 + x5 + y5;
      A[k2] := x2 − y6;  B[k2] := y2 + x6;
      A[k3] := x3 − x7 − y7;  B[k3] := y3 + x7 − y7;
      A[k4] := x0 − x4;  B[k4] := y0 − y4;
      A[k5] := x1 − x5 + y5;  B[k5] := y1 − x5 − y5;
      A[k6] := x2 + y6;  B[k6] := y2 − x6;
      A[k7] := x3 + x7 + y7;  B[k7] := y3 ∸ x7 + y7;
      k0 := k7 + span;  if k0 < n then go to Ld;
      k0 := k0 − n;  if k0 < k then go to Ld;
      comment  Increment the sine and cosine values;
      if k0 ≠ span then
      begin
        c2 := c1 − (dc×c1+ds×s1);
        s1 := (ds×c1−dc×s1) + s1;
        comment  The following three statements compensate
          for truncation error. If rounded arithmetic is used,
          substitute c1 := c2;
        c1 := 1.5−0.5 × (c2 ↑ 2+s1 ↑ 2);
        s1 := c1 × s1;  c1 := c1 × c2;
        c2 := c1 ↑ 2 − s1 ↑ 2;  s2 := 2.0 × c1 × s1;
        c3 := c1 × c2 − s1 × s2;  s3 := s1 × c2 + c1 × s2;
        c4 := c2 ↑ 2 − s2 ↑ 2;  s4 := 2.0 × c2 × s2;
        c5 := c1 × c4 − s1 × s4;  s5 := s1 × c4 + c1 × s4;
        c6 := c3 ↑ 2 − s3 ↑ 2;  s6 := 2.0 × c3 × s3;
        c7 := c1 × c6 − s1 × s6;  s7 := s1 × c6 + c1 × s6;
        k := k + ks;  go to Ld
      end;
      span := 8 × span
    end
  end REVFFT8
```

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.

ALGORITHM 346
F-TEST PROBABILITIES [S14]

John Morris (Recd. 10 Apr. 1968, 12 Sept. 1968, and 6 Nov. 1968)
Computer Institute for Social Science Research, Michigan State University, East Lansing, MI 48823

KEY WORDS AND PHRASES: F-test, Snedecor F-statistic, Fisher test, distribution function
CR CATEGORIES: 5.5

```
procedure Ftest (f, df1, df2, maxn, prob, gauss, error);
  value f, df1, df2, maxn;  real f, prob;  integer df1, df2, maxn:
  real procedure gauss;  label error;
```

comment This procedure gives the probability that F will be greater than the value of f where

$$f = \sigma_1^2/\sigma_2^2,$$

σ_1^2 is the variance of the sample with size N_1, σ_2^2 is the variance of the sample with size N_2, $df1 = N_1 - 1$, $df2 = N_2 - 1$, and F is the Snedecor-Fisher statistic as defined and tabled by Snedecor [4].

The present algorithm computes a value which is directly related to that of Algorithm 322, such that $prob = 1 - Fisher$. A number of test runs on various computers suggest that *Ftest* may be considerably faster than *Fisher*.

An approximation is included to limit execution time when sample size is large. It should be used when register overflow would otherwise result, and the appropriate value for *maxn* will therefore depend upon the specific implementation. When $maxn = 500$ the approximation appears to give three-digit accuracy. The **real procedure** *gauss* computes the area under the left-hand portion of the normal curve. Algorithm 209 [3] may be used for this purpose. If $f < 0$ or if $df1 < 1$ or if $df2 < 1$ then exit to the label *error* occurs.

National Bureau of Standards formulas 26.6.4, 26.6.5, and 26.6.8 are used for computation of the statistic, and 26.6.15 is used for the approximation [2].

Thanks to Mary E. Rafter for extensive testing of this procedure and to the referee for a number of suggestions.

References:

1. Dorrer, Egon. Algorithm 322, F-Distribution. *Comm. ACM 11* (Feb. 1968), 116–117.
2. *Handbook of Mathematical Functions.* National Bureau of Standards, Appl. Math. Ser. Vol., 55, Washington, D.C., 1965, pp. 946–947.
3. Ibbetson, D. Algorithm 209, Gauss. *Comm. ACM 6* (Oct. 1963), 616.
4. Snedecor, George W. *Statistical Methods.* Iowa State U. Press, Ames, Iowa, 1956, pp. 244–250;

```
begin
  if df1 < 1 ∨ df2 < 1 ∨ f < 0.0 then go to error;
  if f = 0.0 then prob := 1.0
  else
  begin
    real f1, f2, x, ft, vp;
    f1 := df1;  f2 := df2;  ft := 0.0;
    x := f2/(f2+f1×f);  vp := f1 + f2 − 2.0;
    if 2 × (df1÷2) = df1 ∧ df1 ≦ maxn then
    begin
      real xx;  xx := 1.0 − x;
      for f1 := f1 − 2.0 step − 2.0 until 1.0 do
      begin
        vp := vp − 2.0;
        ft := xx × vp/f1 × (1.0+ft)
      end;
      ft := x ↑ (0.5×f2) × (1.0+ft)
    end
    else if 2 × (df2 ÷ 2) = df2 ∧ df2 ≦ maxn then
    begin
      for f2 := f2 − 2.0 step − 2.0 until 1.0 do
      begin
        vp := vp − 2.0;
        ft := x × vp/f2 × (1.0+ft)
      end;
      ft := 1.0 − (1.0−x) ↑ (0.5×f1) × (1.0+ft)
    end
    else if df1 + df2 ≦ maxn then
    begin
      real theta, sth, cth, sts, cts, a, b, xi, gamma;
      theta := arctan(sqrt(f1×f/f2));
      sth := sin(theta);  cth := cos(theta);
      sts := sth ↑ 2;  cts: = cth ↑ 2;
      a := b := 0.0;
      if df2 > 1 then
      begin
        for f2 := f2 − 2.0 step − 2.0 until 2.0 do
          a := cts × (f2−1.0)/f2 × (1.0+a);
        a := sth × cth × (1.0+a)
      end;
      a := theta + a;
      if df1 > 1 then
      begin
        for f1 := f1 − 2.0 step − 2.0 until 2.0 do
        begin
          vp := vp − 2.0;
          b := sts × vp/f1 × (1.0+b)
        end;
        gamma := 1.0;  f2 := 0.5 × df2;
        for xi := 1.0 step 1.0 until f2 do
          gamma := xi × gamma/(xi−0.5);
        b := gamma × sth × cth ↑ df2 × (1.0+b)
      end;
      ft := 1.0 + 0.636619772368 × (b−a);
      comment 0.63661977236758134307553 51 ··· = 2.0/π;
    end
    else
    begin
      real cbrf;
      f1 := 2.0/(9.0 × f1);  f2 := 2.0/(9.0×f2);
      cbrf := f ↑ 0.333333333333;
      ft := gauss(−((1.0−f2)×cbrf+f1−1.0)/
        sqrt(f2×cbrf ↑ 2+f1))
    end;
    prob := if ft < 0.0 then 0.0 else ft
  end
end Ftest
```

ALGORITHM 347
AN EFFICIENT ALGORITHM FOR SORTING WITH MINIMAL STORAGE [M1]

RICHARD C. SINGLETON* (Recd. 17 Sept. 1968)
Mathematical Statistics and Operations Research Department, Stanford Research Institute, Menlo Park, CA 94025

KEY WORDS AND PHRASES: sorting, minimal storage sorting, digital computer sorting
CR CATEGORIES: 5.31

```
procedure SORT(A, i, j);
  value i, j;  integer i, j;
  array A;
```

comment This procedure sorts the elements of array A into ascending order, so that

$$A[k] \leq A[k+1], \qquad k = i, i+1, \cdots, j-1.$$

The method used is similar to *QUICKERSORT* by R. S. Scowen [5], which in turn is similar to an algorithm given by Hibbard [2, 3] and to Hoare's *QUICKSORT* [4]. *QUICKERSORT* is used as a standard, as it was shown in a recent comparison to be the fastest among four ACM algorithms tested [1]. On the Burroughs B5500 computer, the present algorithm is about 25 percent faster than *QUICKERSORT* when tested on random uniform numbers (see Table I) and about 40 percent faster on numbers in natural order $(1, 2, \cdots, n)$, in reverse order $(n, n-1, \cdots, 1)$, and sorted by halves $(2, 4, \cdots, n, 1, 3, \cdots, n-1)$. *QUICKERSORT* is slow in sorting data with numerous "tied" observations, a problem that can be corrected by changing the code to exchange elements $a[k] \geq t$ in the lower segment with elements $a[q] \leq t$ in the upper segment. This change gives a better split of the original segment, which more than compensates for the additional interchanges.

In the earlier algorithms, an element with value t was selected from the array. Then the array was split into a lower segment with all values less than or equal to t and an upper segment with all values greater than or equal to t, separated by a third segment of length one and value t. The method was then applied recursively to the lower and upper segments, continuing until all segments were of length one and the data were sorted. The present method differs slightly—the middle segment is usually missing—since the comparison element with value t is not removed from the array while splitting. A more important difference is that the median of the values of $A[i]$, $A[(i+j)\div 2]$, and $A[j]$ is used for t, yielding a better estimate of the median value for the segment than the single element used in the earlier algorithms. Then while searching for a pair of elements to exchange, the previously sorted data (initially, $A[i] \leq t \leq A[j]$) are used to bound the search, and the index values are compared only when an exchange is about to be made. This leads to a small amount of overshoot in the search, adding to the fixed cost of splitting a segment but lowering the variable cost. The longest segment remaining after splitting a segment o n has length less than or equal to $n-2$, rather than $n-1$ as in *QUICKERSORT*.

TABLE I. SORTING TIMES IN SECONDS FOR *SORT* AND *QUICKERSORT*, ON THE BURROUGHS B5500 COMPUTER—AVERAGE OF FIVE TRIALS

	Algorithm	
Original order and number of items	*SORT*	*QUICKERSORT*
Random uniform:		
500	0.48	0.63
1000	1.02	1.40
Natural order:		
500	0.29	0.48
1000	0.62	1.00
Reverse order:		
500	0.30	0.51
1000	0.63	1.08
Sorted by halves:		
500	0.73	1.15
1000	1.72	2.89
Constant value:		
500	0.43	10.60
1000	0.97	41.65

For efficiency, the upper and lower segments after splitting should be of nearly equal length. Thus t should be close to the median of the data in the segment to be split. For good statistical properties, the median estimate should be based on an odd number of observations. Three gives an improvement over one and the extra effort involved in using five or more observations may be worthwhile on long segments, particularly in the early stages of a sort.

Hibbard [3] suggests using an alternative method, such as Shell's [6], to complete the sort on short sequences. An experimental investigation of this idea using the splitting algorithm adopted here showed no improvement in going beyond the final stage of Shell's algorithm, i.e. the familiar "sinking" method of sorting by interchange of adjacent pairs. The minimum time was obtained by sorting sequences of 11 or fewer items by this method. Again the number of comparisons is reduced by using the data themselves to bound the downward search. This requires

$$A[i-1] \leq A[k], \qquad i \leq k \leq j.$$

* This work was supported by Stanford Research Institute with Research and Development funds.

Thus the initial segment cannot be sorted in this way. The initial segment is treated as a special case and sorted by the splitting algorithm. Because of this feature, the present algorithm lacks the pure recursive structure of the earlier algorithms.

For n elements to be sorted, where $2^k \leq n < 2^{k+1}$, a maximum of k elements each are needed in arrays IL and IU. On the B5500 computer, single-dimensional arrays have a maximum length of 1023. Thus the array bounds [0:8] suffice.

This algorithm was developed as a FORTRAN subroutine, then translated to ALGOL. The original FORTRAN version follows:

```
      SUBROUTINE SORT(A,II,JJ)
C   SORTS ARRAY A INTO INCREASING ORDER, FROM A(II) TO A(JJ)
C   ORDERING IS BY INTEGER SUBTRACTION, THUS FLOATING POINT
C     NUMBERS MUST BE IN NORMALIZED FORM.
C   ARRAYS IU(K) AND IL(K) PERMIT SORTING UP TO 2**(K+1)-1 ELEMENTS
      DIMENSION A(1),IU(16),IL(16)
      INTEGER A,T,TT
      M=1
      I=II
      J=JJ
    5 IF(I .GE. J) GO TO 70
   10 K=I
      IJ=(J+I)/2
      T=A(IJ)
      IF(A(I) .LE. T) GO TO 20
      A(IJ)=A(I)
      A(I)=T
      T=A(IJ)
   20 L=J
      IF(A(J) .GE. T) GO TO 40
      A(IJ)=A(J)
      A(J)=T
      T=A(IJ)
      IF(A(I) .LE. T) GO TO 40
      A(IJ)=A(I)
      A(I)=T
      T=A(IJ)
      GO TO 40
   30 A(L)=A(K)
      A(K)=TT
   40 L=L-1
      IF(A(L) .GT. T) GO TO 40
      TT=A(L)
   50 K=K+1
      IF(A(K) .LT. T) GO TO 50
      IF(K .LE. L) GO TO 30
      IF(L-I .LE. J-K) GO TO 60
      IL(M)=I
      IU(M)=L
      I=K
      M=M+1
      GO TO 80
   60 IL(M)=K
      IU(M)=J
      J=L
      M=M+1
      GO TO 80
   70 M=M-1
      IF(M .EQ. 0) RETURN
      I=IL(M)
      J=IU(M)
   80 IF(J-I .GE. 11) GO TO 10
      IF(I .EQ. II) GO TO 5
      I=I-1
   90 I=I+1
      IF(I .EQ. J) GO TO 70
      T=A(I+1)
      IF(A(I) .LE. T) GO TO 90
      K=I
  100 A(K+1)=A(K)
      K=K-1
      IF(T .LT. A(K)) GO TO 100
      A(K+1)=T
      GO TO 90
      END
```

This FORTRAN subroutine was tested on a CDC 6400 computer. For random uniform numbers, sorting times divided by $n \log_2 n$ were nearly constant at 20.2×10^{-6} for $100 \leq n \leq 10{,}000$, with a time of 0.202 seconds for 1000 items. This subroutine was also hand-compiled for the same computer to produce a more efficient machine code. In this version the constant of proportionality was 5.2×10^{-6}, with a time of 0.052 seconds for 1000 items. In both cases, integer comparisons were used to order normalized floating-point numbe.:s.

REFERENCES:

1. BLAIR, CHARLES R. Certification of algorithm 271. *Comm. ACM 9* (May 1966), 354.
2. HIBBARD, THOMAS N. Some combinatorial properties of certain trees with applications to searching and sorting. *J. ACM 9* (Jan. 1962), 13–28.
3. HIBBARD, THOMAS N. An empirical study of minimal storage sorting. *Comm. ACM 6* (May 1963), 206–213.
4. HOARE, C. A. R. Algorithms 63, Partition, and 64, Quicksort. *Comm. ACM 4* (July 1961), 321.
5. SCOWEN, R. S. Algorithm 271, Quickersort. *Comm. ACM 8* (Nov. 1965), 669.
6. SHELL, D. L. A high speed sorting procedure. *Comm. ACM 2* (July 1959), 30–32;

```
begin
  real t, tt;
  integer ii, ij, k, L, m;
  integer array IL, IU[0:8];
  m := 0;  ii := i;  go to L4;
L1:  ij := (i+j) ÷ 2;  t := A[ij];  k := i;  L := j;
  if A[i] > t then
     begin A[ij] := A[i];  A[i] := t;  t := A[ij] end;
  if A[j] < t then
  begin
     A[ij] := A[j];  A[j] := t;  t := A[ij];
     if A[i] > t then
        begin A[ij] := A[i];  A[i] := t;  t := A[ij] end
  end;
L2:  L := L − 1;
  if A[L] > t then go to L2;
     tt := A[L];
L3:  k := k + 1;
  if A[k] < t then go to L3;
  if k ≤ L then
     begin A[L] := A[k];  A[k] := tt;  go to L2 end;
  if L − i > j − k then
     begin IL[m] := i;  IU[m] := L;  i := k end
  else
     begin IL[m] := k;  IU[m] := j;  j := L end;
  m := m + 1;
L4:  if j − i > 10 then go to L1;
     if i = ii then
     begin if i < j then go to L1 end;
     for i := i + 1 step 1 until j do
     begin
     t := A[i];  k := i − 1;
     if A[k] > t then
     begin
L5:  A[k+1] := A[k];  k := k − 1;
     if A[k] > t then go to L5;
     A[k+1] := t
     end
  end;
  m := m − 1;  if m ≥ 0 then
     begin i := IL[m];  j := IU[m];  go to L4 end
  end SORT
```

REMARK ON ALGORITHM 347 [M1] AN EFFICIENT ALGORITHM FOR SORTING WITH MINIMAL STORAGE

[Richard C. Singleton, *Comm. ACM 12* (Mar. 1969), 185]

Robin Griffin and K. A. Redish (Recd. 14 Apr. 1969 and 11 Aug. 1969)
McMaster University, Hamilton, Ontario, Canada

KEY WORDS AND PHRASES: sorting, minimal storage sorting, digital computer sorting
CR CATEGORIES: 5.31

The algorithm was tested on the CDC 6400 Algol compiler (version 1.1, running under the SCOPE operating system, version 3.1.4). One trial was made using an array of 5000 pseudorandom numbers; the results were correct.

The central processor time was about 6.9 seconds corresponding to a value for K (defined below) of about 110 microseconds.

It would be more in the spirit of Algol to follow QUICKERSORT [1] and give arrays *IL* and *IU* dynamic bounds. This involves changing line 4 on page 187 from

integer array *IL*, *IU*[0:8];

to

integer array *IL*, *IU*[0:*ln*(*j*−*i*+1)/*ln*(2)−0.9];

The Fortran subroutine given in the comments to the algorithm was tested on a CDC Fortran compiler (the RUN compiler version 2.3, running under the SCOPE operating system, version 3.1.4). Tests were made with each of the five initial orderings described with the algorithm for a variety of array lengths from 500 to 40,000. For integer arrays, the results were correct; but when the actual argument corresponding to the dummy argument A was a real array containing large positive and negative numbers, errors occurred. This does not invalidate the subroutine, but the comments should be changed to

```
C  SORTS INTEGER ARRAY A INTO INCREASING OR-
     DER, FROM A(II) TO A(IJ)
C  ARRAYS IU(K) AND IL(K) PERMIT SORTING UP TO
     2**(K+1) - 1 ELEMENTS
C  THE USER SHOULD CONSIDER THE POSSIBILITY OF
     INTEGER OVERFLOW
C  THE ONLY ARITHMETIC OPERATION ON THE ARRAY
     ELEMENTS IS SUBTRACTION
```

This gives enough information (and a hint) but leaves the responsibility for any abuse of American National Standards Institute (formerly USASI) Fortran where it belongs—with the user.

The subroutine was also tested on the IBM 7040 Fortran compiler (the IBFTC compiler running under the IBSYS operating system, version 9 level 10). The results were correct. The statement

```
INTEGER A, T, TT
```

was removed and the amended subroutine tested using similar, but real, arrays. The results were again correct; running times increased by up to 5 percent on the CDC 6400 and were unchanged on the IBM 7040.

Tables I and II summarize the information on running times in terms of *K*, where

$$\text{time} = Kn \log_2 n$$

(runs of other lengths are omitted for brevity).

TABLE I. Sorting Times
K in microseconds where time $= Kn \log_2 n$

Test	Method			
Original order and number of items	Burroughs 5500 ALGOL*	CDC 6400 FORTRAN (REAL)	CDC 6400 FORTRAN (INTEGER ARRAY)	IBM 7040 FORTRAN
Random uniform				
500	107	21.2	20.5	
1000	102	21.7	20.5	
5000		21.1	20.2	269
10000		21.1	20.1	263
40000		21.2	20.1	
Natural order				
500	65	12.9	12.5	
1000	62	13.1	12.4	146
5000		12.6	11.9	148
10000		12.7	12.0	
40000		12.9	12.1	
Reverse order				
500	67	14.3	13.4	
1000	63	13.9	13.4	
5000		13.4	12.7	158
10000		13.4	12.7	158
40000		13.5	12.8	
Sorted by halves				
500	163	34.8	32.6	
1000	173	37.1	35.1	
5000		39.5	37.2	465
10000		41.8	39.3	491
40000		46.6	44.1	
Constant value				
500	96	19.2	18.5	
1000	97	19.4	18.7	
5000		19.4	18.7	237
10000		19.9	19.0	241
40000		20.2	19.5	

* Calculated from Singleton's results

TABLE II. Values of $n \log_2 n$

n	500	1000	5000	10000	40000
$10^{-6}\, n \log_2 n$	0.00448	0.00996	0.0614	0.1329	0.6115

For use as a library routine one slight change is recommended: JJ−II should be tested on entry and a suitable error message produced if negative. It would be possible to transfer "work" arrays to replace IU and IL thus allowing the user more control of storage allocation, but the additional instructions needed to handle the extra arguments reduce the saving and this is hardly worthwhile.

The authors would like to thank the referee for his helpful comments.

Reference:

1. Scowen, R. S. Algorithm 271, Quickersort. *Comm. ACM 8* (Nov. 1965), 669–670.

REMARK ON ALGORITHM 347 [M1]
AN EFFICIENT ALGORITHM FOR SORTING WITH MINIMAL STORAGE [Richard C. Singleton, *Comm. ACM 12* (Mar. 1969), 185]

Richard Peto (Recd. 18 Feb. 1970)
Medical Research Council, 115 Gower Street, London
W. C. 1

KEY WORDS AND PHRASES: sorting, ranking, minimal storage sorting, digital computer sorting
CR CATEGORIES: 5.31

If the values of *ij*, instead of always being $(i+j) \div 2$, are at varying positions between *i* and *j*, then there is less likelihood of peculiar initial structure causing failure of the algorithm to perform rapidly. The position of *ij* can be made to vary by replacing the statements

```
m := 0;  ii := i;  go to L4;  L1: ij := (i+j) ÷ 2;
```

by

```
real r;  r := 0.375;  m := 0;  ii := i;  go to L4;
L1: r := if r > 0.58984375 then r − 0.21875 else r + 0.0390625;
ij := i + (j−i) × r;
```

comment These four decimal constants, which are respectively 48/128, 75.5/128, 28/128, and 5/128, are rather arbitrary. On most compilers their binary representations will be exact, and the use of them in the statement *L*1 causes *r* to vary cyclically over the 33 values 48/128 ⋯ 80/128. Therefore *ij* takes a variable position somewhere within the middle quarter of the segment to be sorted. Wider variation of *ij* would be undesirable in the special case of a partially presorted array;

In sorting an array of *N* elements which are initially in random order this will waste (on ICL Atlas) less than $N/10^5$ seconds, but if the array is, for example, composed initially of two equal presorted halves, then the use of the original rather than the modified version would more than double the sorting time required if $N > 10^4$.

As the author points out, the published version could fail if used to sort arrays of 1024 or more elements because the upper bounds of *IU* and *IL* might be inadequate. For a standard procedure the declaration *IL*, *IU* [0:8] should be replaced by the declaration *IL*, *IU* [0:20]. This permits the sorting of arrays of up to 4 million elements, which is, with present core store sizes, sufficient.

The statement *tt* := *a*[*L*] which precedes *L3*: will be executed less frequently if it is transferred into the next conditional statement, which then reads

```
if k ≤ L then begin tt := a[L];  a[L] := a[k];  a[k] := tt;
  go to L2 end
```

ALGORITHM 348
MATRIX SCALING BY INTEGER PROGRAMMING [F1]

R. R. Klimpel (Recd. 4 Mar. 1968, 13 June 1968, 16 Oct. 1968 and 21 Nov. 1968)
Computation Research Laboratory, The Dow Chemical Co., Midland, MI 48640

KEY WORDS AND PHRASES: integer programming, linear algebra, mathematical programming, matrix condition, matrix scaling
CR CATEGORIES: 5.14, 5.41

```
procedure scale (a, m, n, g, u, v);
  value m, n, g;  integer m, n;  real g;
  real array a;  integer array u, v;
comment The use of scaling to precondition matrices so as to
  improve subsequent computational characteristics is of considerable
  importance. To measure the scaling condition of a
  matrix, $a_{ij}$ ($i=1, \cdots, m$ and $j=1, \cdots, n$), Fulkerson and Wolfe
  [1] suggested the ratio of the matrix entry of largest absolute
  value to that of the smallest nonzero absolute value. This
  procedure implements the method of [1], i.e. finding multiplicative
  row factors, $r_i$, and column factors, $s_j$, which, when applied,
  minimize the above condition number. The minimization
  problem can be expressed as an equivalent additive discrete
  problem by taking logarithms and defining:

        $r_i = g^{u_i}$,    $s_j = g^{v_j}$,    $b_{ij} = log_g (abs(a_{ij}))$

  and taking $c_{ij}$ to be the least integer greater than or equal to
  $b_{ij}$. Thus the formulation becomes: minimize an integer $w$
  subject to the constraints $0 \le u_i + v_j + c_{ij} \le w$ where $u_i$ and
  $v_j$ are unrestricted and integral in value. The effect of decreasing
  the value of the base $g$ would be to more accurately approximate
  the continuous scaling problem by the discrete form.
  REFERENCE:
1. FULKERSON, D. R., AND WOLFE, P. An algorithm for scaling
  matrices. SIAM Rev. 4 (1962), 142-146;
begin
  integer array c[1:m, 1:n], ri[1:m], si[1:n];
  real val;
  integer max, store, markr, markc, num, nopt, i, j;
  nopt := 0;
  comment Create initial integer matrix c. Due to machine
    round-off errors, it may be desirable for some problems to
    insert a tolerance when checking for zero values of the input
    matrix and for matrix entries which are exact integral powers
    of the base g;
  for i := 1 step 1 until m do
  for j := 1 step 1 until n do
  begin
    if (a[i, j]=0) then
    begin
      c[i, j] := 0;
      go to intf
    end;
    val := ln(abs(a[i, j]))/ln(g);
    c[i, j] := entier(val) + 1;
    if ((c[i, j]-1)=val) then c[i, j] := c[i, j] - 1;
intf:
  end;
  comment Select initial values of $u_i$ and $v_j$ that satisfy constraints
    of discrete formulation;
  for i := 1 step 1 until m do
  begin
    u[i] := c[i, 1];
    for j := 2 step 1 until n do
      if (c[i, j]<u[i]) then u[i] := c[i, j];
    u[i] := -u[i]
  end;
  for j := 1 step 1 until n do
  begin
    v[j] := c[1, j] + u[1];
    for i := 2 step 1 until m do
    begin
      store := c[i, j] + u[i];
      if (store<v[j]) then v[j] := store;
    end;
    v[j] := -v[j];
  end;
  comment Step one. Initialize row and column markers with
    unmarked rows and columns denoted by a 1 in ri[i] and si[j],
    respectively. Locate and mark maximum entry of current
    working array;
rcmax: max := 0;
  for i := 1 step 1 until m do
  begin
    ri[i] := 1;
    for j := 1 step 1 until n do
    begin
      if (i = 1) then si[j] := 1;
      if (nopt=0) then c[i, j] := u[i] + v[j] + c[i, j];
      if (c[i, j]≥max) then
      begin
        markr := i;
        markc := j;
        max := c[i, j]
      end
    end
  end;
  nopt := 1;
  ri[markr] := -1;
  comment Repeat steps two and three in succession until
    there are either no freshly marked rows or no freshly marked
    columns. Any row or column marked in the immediately preceding
    application of step one, two, or three is called freshly
    marked and denoted by -1 in the appropriate indicator
    vector. Previously marked rows and columns that are not
    freshly marked are denoted by zero values;
  comment Step two;
rmarks: num := 0;
  for i := 1 step 1 until m do
  begin
    if (ri[i]>-1) then go to rmarkf;
```

```
      ri[i] := 0;
      num := num + 1;
      for j := 1 step 1 until n do
        if (si[j]=1) ∧ (c[i, j]=0) then si[j] := −1;
rmarkf:
   end;
   if (num=0) then go to change;
   comment Step three;
   num := 0;
   for j := 1 step 1 until n do
   begin
      if (si[j]>−1) then go to cmarkf;
      si[j] := 0;
      num := num + 1;
      for i := 1 step 1 until m do
        if (ri[i]=1) ∧
          ((c[i, j]=max) ∨(c[i, j]=(max−1))) then
          ri[i] := −1;
   cmarkf:
   end;
   if (num≠0) then go to rmarks;
   comment Step four. Modify integer scaling factors u and v
      and adjust current working matrix ($c_{ij}+u_i+v_j$);
   change: if (si[markc]<1) then go to finis;
   for i := 1 step 1 until m do
   if (ri[i]<1) then
   begin
      u[i] := u[i] − 1;
      for j := 1 step 1 until n do
        c[i, j] := c[i, j] − 1
   end;
   for j := 1 step 1 until n do
   if (si[j]<1) then
   begin
      v[j] := v[j] + 1;
      for i := 1 step 1 until m do
        c[i, j] := c[i, j] + 1
   end;
   go to rcmax;
finis:
end
```

ALGORITHM 349
POLYGAMMA FUNCTIONS WITH ARBITRARY PRECISION* [S14]

Adilson Tadeu de Medeiros and
Georges Schwachheim (Recd. 15 Mar. 1968, 1 July 1968, 28 Oct. 1968 and 3 Dec. 1968)
Centro Brasileiro de Pesquisas Físicas, Rio de Janeiro, ZC 82, Brasil

* This work was supported by the Conselho Nacional de Pesquisas and the Banco Nacional do Desenvolvimento Economico of Brasil.

KEY WORDS AND PHRASES: polygamma function, psi function, digamma function, trigamma function, tetragamma function, pentagamma function, special functions
CR CATEGORIES: 5.12

```
procedure polygamma (n, z, nd, polygam, error);
  value n, z, nd; real z, polygam; integer n, nd; label error;
comment This procedure assigns to polygam the value of the
  polygamma function of order n for any real argument z. For
  n = 0, we have the psi or digamma function, for n = 1 the tri-
  gamma function, for n = 2 the tetragamma function, and so on.
  For arguments that are poles of the function (nonpositive
  integer values), an exit is made through the label error. The
  parameter nd gives the requested relative precision expressed
  in number of decimal digits.
```

It computes the polygamma function through the asymptotic series

$$\psi^{(n)}(z) \sim (-1)^{n-1}\left[\frac{(n-1)!}{z^n}+\frac{n!}{2z^{n+1}}+\sum_{k=1}^{\infty} B_{2k}\frac{(2k+n-1)!}{(2k)!\,z^{2k+n}}\right]$$

except for $n = 0$, when the first term is $-\ln(z)$.

If the simple empirical relationship

$$2z > n + nd$$

is true, as well as $z > n$, one enters directly into the asymptotic series with the original argument. Otherwise, the computation of small arguments is reduced to that of sufficiently large arguments, applying repeatedly the recurrence relation:

$$\psi^{(n)}(z+1) = \psi^{(n)}(z) + (-1)^n n! z^{-n-1}$$

To save computation time, the argument, once larger than n, is increased just to the point when the minimum term of the asymptotic expansion is sufficiently small so as not to alter the value of the result within the chosen precision.

The order of the minimum term is estimated by the first order approximation

$$\pi z - n/2,$$

and the corresponding absolute value by the approximation formula

$$(2\pi)^n \exp(-2\pi z).$$

Negative arguments are related to positive ones through the reflection formula:

$$(-1)^n \psi^{(n)}(1-z) = \psi^{(n)}(z) + \pi\frac{d^n}{dz^n}\cot \pi z$$

The nth-order derivative of the cotangent is computed by term by term differentiation of the tangent or cotangent series after the convenient trigonometric reductions of the argument's value.

This procedure is not recursive and uses no own variable;

```
begin
  real pi, pf, soma, zq, t1, fac, prec, w, sab, pv;
  integer pr, n1, k1, m1;
  real procedure fat (n);
    value n; integer n;
  begin
    real f; integer i;
    f := 1;
    for i := n step −1 until 2 do f := f × i;
    fat := f
  end of fat;
  procedure inc (s, x1, L);
    real s, x1; label L;
  begin
    real sant;
    sant := s; s := s + x1;
    if abs (s−sant) ≤ abs (prec × s) then go to L
  end of inc;
  comment The procedure polygamma uses a table of coeffi-
    cients sb for its series with the value
```

$$sb(i) = \frac{|B_{2i}|}{(2i)!} = \frac{\sum_{k=1}^{\infty}(--1)^{k-1}/k^{2i}}{\pi^{2i}(2^{2i-1}-1)} \cong \frac{2}{(2\pi)^{2i}},$$

the last being an asymptotic value for large i. The computation of these coefficients need not to be repeated at each procedure call; so it is convenient to transfer the declaration and block below to the main program and execute it just once.

One should replace *flund* by the smallest positive real number within the machine representation, and *ms* by the number of decimal digits of the mantissa;

```
  array sb [1 : entier (.272 × ln(2/flund))];
  begin
    real piq, sm, pipo, ptwo, dpi, sa;
    integer sg, in, k2, imax;
    array tr, q[2 : entier (10 ↑ (ms/22))+1];
    imax := entier (.272 × ln(2/flund));
    piq := 9.86960440108935861883449099987615113531369940724079;
    pipo := piq ↑ 11; ptwo := 2097152; dpi := 4 × piq;
    sb [1] := 1/12;
    sb [2] := 1/720;
    sb [3] := 1/30240;
    sb [4] := 1/1209600;
    sb [5] := 1/47900160;
    sb [6] := 691/1307674368₁₀3;
    sb [7] := 1/74724249600;
    sb [8] := 3617/10670622842 88₁₀4;
    sb [9] := 43867/5109094217170944₁₀3;
    sb [10] := 174611/802857662698291 2₁₀5;
    sm := 1; sg := −1;
    for in := 2, in + 1 while sm ≠ sa do
    begin
```

```
        q[in] := 1/(in × in);
        tr[in] := sg × q[in] ↑ 11; sa := sm;
        sm := sm + tr[in];  sg := −sg
      end;
      sb[11] := sm/(pipo × (ptwo−1));
      for k2 := 12 step 1 until imax do
      begin
        sm := 1;  in := 1;
B:      in := in + 1;  tr[in] := tr[in] × q[in];  sa := sm;
        sm := sm + tr[in];  if sa ≠ sm then go to B;
        pipo := pipo × piq;  ptwo := ptwo × 4;
        sb[k2] := sm/(pipo × (ptwo−1));
        if in = 2 then go to L
      end;
      go to A;
L:    for k2 := k2 + 1 step 1 until imax do
      sb[k2] := sb[k2−1]/dpi;
A:  end of sb coefficients computation;
    pi := 3.1415926535897932384626433832795028841971693993751O;
    prec := 10 ↑ (−nd);  fac := fat (n);
    pr := if n ÷ 2 × 2 = n then 1 else − 1;
    pf := pr × fac;  n1 := n + 1;
    if z ≤ 0 then
   begin
    if z = entier(z) then go to error
    else
    begin
      real x, y;  integer d, l;  Boolean C;
      k1 := pr;  d := z;  x := d − z;
      if x > 0 then l := 1
      else
      begin x := −x;  l := −pr end;
      C := x > .25;  y := pi × (if C then (.5−x) else x);
      if n = 0 then
        soma := l × pi × (if C then sin(y)/cos(y) else cos(y)/
          sin(y))
      else
      begin
        integer m, np, j, i;  integer array ft [1:4];
        real y2, p, f, t, s, v;
        m := n ÷ 2;  np := m × 2;
        ft[1] := np + 1;  ft[2] := np;  ft[3] := pr;
        ft[4] := 0;  y2 := y × y;  j := m + 1;
        f := fat(np+1);  p := 4 ↑ (m+1);
        t := if pr = −1 then 1 else y;
        s := if C then 0 else pf/y ↑ n1;
E:      v := if C then p × (1−p) else p;
        inc(s, -sb[j] × f × t × v, D);
        for i := 1 step 1 until 4 do
          ft[i] := ft[i] + 2;
        f := f × ft[1] × ft[2] × y2/(ft[3] × ft[4]);
        p := 4 × p;  j := j + 1;
        go to E;
D:      soma := l × pi ↑ n1 × (if C then s × pr else s)
      end
    end;
    z := 1 − z;  w := z ↑ n;
    pv := if n = 0 then ln(z) else fac/(n × w);
    sab := abs(soma);
    if pv < sab then nd := nd − .434 × ln(sab/pv)
  end
  else
  begin soma := 0;  k1 := 1;  w := z ↑ n end;
  if nd ≤ 0 then go to L;
  if 2 × z < n + nd ∨ z < n then
 begin
   real term, cond;
   term := −pf/(z × w);
   inc(soma, term, L);
   cond := (n × 1.8378−ln(abs(term)) + 2.3025 × nd) × .1591;
   if cond < n then cond := n;
   if cond ≤ z then z := z + 1
   else
   begin
     integer ip, k;
     ip := cond − z + 1;
     if ip < 1 then go to L;
     for k := 1 step 1 until ip do
       inc(soma, −pf/(z+k) ↑ n1, L);
     z := z + ip + 1
   end
   w := z ↑ n
 end;
 inc(soma, if n=0 then ln(z) else −pf/(n × w), L);
 inc(soma, −pf × .5/(z × w), L);
 zq := z × z;  t1 := pf × n1/(w × zq);
 for m1 := 2 step 2 until 6.283 × z + n do
 begin
   inc(soma, −t1 × sb[m1÷2], L);
   t1 := −t1 × (n1+m1) × (n+m1)/zq
 end;
L: polygam := soma × k1
end of polygamma
```

ACM Transactions on Mathematical Software, Vol. 1, No. 4, December 1975, Pages 380-381

CERTIFICATION OF ALGORITHM 349

Polygamma Functions with Arbitrary Precision [S14]
[Adilson Tadeu de Medeiros and Georges Schwachheim, *Comm. ACM 12*, 4 (April 1969), 213–214]

John Gregg Lewis [Recd 30 May 1974]
Computer Science Department, Stanford University, Stanford, CA 94305

This work was supported by a fellowship from the IBM Corp. and by the Stanford Center for Information Processing.

A casual user should not be misled by the title of this algorithm. Algorithm 349 does not offer arbitrarily precise values of the polygamma functions. It does offer results with precision *adjustable* downward from something somewhat less than the

Table I. Consistency Checks

Order of Magnitude of Relative Error, machine precision ≅ 10^{-16}

Requested precision (decimal digits)	Positive arguments (digamma-pentagamma)	Negative arguments				
		Digamma	Trigamma	Tetragamma	Pentagamma	Hexagamma
		−10(+.005)0		−10(+.01)0		−10(.1)0
6	(−6)	(−7)	(−5)	(−5)	(−5)	(--5)
9	(−9)	(−10)	(−10)	(−8)	(−8)	(--8)
10	(−11)	(−11)	(−11)	(−11)	(−9)	(--9)
11	(−12)	(−12)	(−12)	(−12)	(−10)	(−11)
12	(−13)	(−13)	(−13)	(−12)	(−11)	(−11)
15	(−15)	(−13)	(−13)	(−13)	(−12)	(−11)
17	(−15)	(−13)	(−13)	(−13)	(−12)	(−11)

floating-point precision of the computer on which it is run. Further, unlike the highly tuned functions to which we have become accustomed, this routine is not accurate to the last bit. In general, the last several decimal digits of the results of this procedure are in doubt. This procedure does not use rational function approximations. Instead, it computes the polygamma functions as limits of asymptotic series. Hence it is relatively slow. It is on numerically shaky grounds since some values are the result of three separate summation processes where no efforts are made to rearrange the terms to preserve accuracy. Despite this, if used carefully within its limitations, the procedure performs as advertised.

Algorithm 349 was translated into Fortran and tested in long precision on Stanford University's IBM 360/67 computer using both the Waterloo WATFIV compiler and IBM's Fortran compilers. Since no other software to compute these functions is available at Stanford, the routine was checked by comparison with published tables of values and by several crude, but revealing consistency checks. For the digamma, trigamma, tetragamma, and pentagamma functions we checked directly against the tables in Abramowitz and Stegun [1], which give at least 10 and at most 11 significant digits in the range 1(.005)2.[1] These were checked, requesting in turn 6, 9, 10, 11, 12, 15, and 17 decimal digits of precision. In this range the procedure either provided the number of digits requested or agreed completely with the published tables, except that for the trigamma function, even with full machine precision requested, the numerical results (correctly rounded or truncated) for most arguments of the form 1.xx5 disagreed with the last digit of the published value, an error on the order of 1×10^{-10}. The trigamma, tetragamma, pentagamma, and hexagamma functions were also compared with tables provided by the authors [2] for negative arguments −9.9(.1)(−.1). The results of these tests are recorded in Table I.

The following internal checks were made. For positive arguments in the range (0,1) and (2,11), we checked the translation properties of the procedure by computing the shifts in reverse order (to full machine precision) and compared results. For negative arguments the procedure computes derivatives of the cotangent function as limits of a series. We computed the needed low order derivatives analytically and evaluated them using standard trigonometric functions instead. For the functions in the first test we compared results in the range (−10(.005)0), skipping the poles at the negative integers. All of the values in the second test were checked similarly. In the latter case, where published tables for negative arguments were available, this internal check proved sharp—whenever the internal check indicated an error larger than the precision of the tables, the error was found to be of the expected order.

Note on translation. In the Fortran program, the first block of the Algol procedure was made a separate initializing subroutine. The unnecessary procedures

[1] 1(.01)2 for tetragamma and pentagamma.

FAC and INC were replaced by in-line code. To enhance portability, all constants are computed at run time. (The dimension of the arrays SB, TR, and Q are machine dependent.) The routine is available from the Numerical Analysis Program Librarian, Stanford Center for Information Processing, Stanford, CA 94305. It should not be implemented in single precision on short word-length machines.

REFERENCES

1. Abramowitz, M., and Stegun, I.A., Eds. *Handbook of Mathematical Functions.* Nat. Bur. Standards Appl. Math. Series 55, U.S. Govt. Printing Office, Washington, D.C., 1964, pp. 267–273.
2. Davis, H.T. *Tables of Mathematical Functions, Vol. II,* revised. Principia Press, Trinity U., San Antonio, Tex., 1963.

ALGORITHM 350
SIMPLEX METHOD PROCEDURE EMPLOYING LU DECOMPOSITION* [H]

RICHARD H. BARTELS AND GENE H. GOLUB (Recd. 2 Aug. 1967 and 5 June 1968)

Computer Science Department, Stanford University, Stanford, CA 94305

* This project was supported in part by contracts NSF GP948 and ONR NR 044 211.

KEY WORDS AND PHRASES: simplex method, linear programming, LU decomposition, round-off errors, computational stability

CR CATEGORIES: 5.41

```
procedure linprog (m, n, kappa, G, b, d, x, z, ind, infeasible, un-
    bounded, singular);
  value m, n;  integer m, n, kappa;  real z;
  array G, b, d, x;  integer array ind;  label infeasible, un-
    bounded, singular;
comment  linprog attacks the linear programming problem:
              maximize d^T x
              subject to Gx = b and x ≥ 0
  Details about the methods used are given in a paper by Bartels
  and Golub [Comm. ACM 12 (May 1969), 266-268].
    The array G[0:m-1, 0:n-1] contains the constraint coeffi-
  cients. Array b[0:m-1] contains the constraint vector, and
  d[0:n-1] contains the objective function coefficients (cost
  vector). The computed solution will be stored in x[0:n-1], and
  z will have the maximum value of the objective function if
  linprog terminates successfully. Error exit singular will be taken
  if a singular basis matrix is encountered. Error exit infeasible
  will be taken if the given problem has no basic feasible solution,
  and exit unbounded will be taken if the objective function is
  unbounded. If kappa = 0, problem (2) of the referenced paper
  will be set up and phase 1 entered. If 1 ≤ kappa ≤ m − 1, prob-
  lem (4) of the paper will be set up and phase 1 entered. The last
  kappa columns of G will be preceded by the first m − kappa
  columns of the identity matrix to form the initial basis matrix.
  If kappa = m, phase 2 computation will begin on problem (1)
  with variables numbered ind[0], ··· , ind[m-1] as the initial
  basic variables and variables numbered ind[m], ··· , ind[n-1] as
  the initial nonbasic variables. Hence each component of ind must
  hold an integer between 0 and n − 1 specified by the user. Fi-
  nally, if kappa > m, problem (3) will be set up, and phase 2
  computation will begin with variables numbered ind[0], ··· ,
  ind[m] as the initial basic variables and variables numbered
  ind[m+1], ··· , ind[n+kappa-m-1] as the initial nonbasic
  variables. This option is of interest only because linprog, upon
  successful termination, leaves all variable numbers recorded in
  ind in their final order and provides kappa with an appropriate
  value. This permits linprog to be reentered at the phase 2 point
  after modifications have been made to G, b, or d. An understand-
  ing of the simplex method and the accompanying paper by Bar-
  tels and Golub will make clear what modifications can be per-
  mitted. If phase 1 is to be executed, ind must have array bounds
  [0:m+n-kappa] to allow for artificial variables. Otherwise, ind
  must have bounds [0:n+kappa-m-1]. The values in array b
  must be nonnegative if phase 1 is to be executed. The contents
  of m, n, G, b, and d are left unchanged by linprog;
begin
  real procedure ip2(ii, ll, uu, aa, bb, cc);
    value uu;  integer ii, ll, uu;  real aa, bb, cc;
  begin
  comment  ip2 must produce a double-precision, accumulated
    inner product. Jensen's device is used. The main statement in
    ip2 is
      for ii := ll step 1 until uu do sum := sum + aa × bb
    where the local variable sum has been initialized by cc. How-
    ever, the multiplication aa × bb must produce a double-pre-
    cision result, so sum represents a double-precision accumu-
    lated sum. After all products have been summed together, sum
    is to be rounded to single-precision and used as the value of
    ip2;
  end ip2;
  procedure trisolv(fis, fid, fie, sis, sie, fi, si, sol, rhs, mat, piv);
    value fid, fie;  integer fis, fid, fie, sis, sie, fi, si;  real sol, rhs,
    mat, piv;
  comment  trisolv solves a triangular system of linear equa-
    tions. The off-diagonal part of the system's coefficient matrix
    is given by mat, the diagonal part by piv, and the right-hand
    side of the system by rhs. The solution is developed in sol.
    By appropriately setting the first five parameters, either an
    upper or a lower triangular system can be treated. Column by
    column LU decomposition of a matrix can be compactly ex-
    pressed using trisolv;
  begin real tt, pv;
    for fi := fis step fid until fie do
    begin tt := -ip2(si, sis, sie, sol, mat, -rhs);
      si := fi;  pv := piv;
      sol := if pv = 1.0 then tt else tt/pv
    end
  end trisolv;
  array q, h, w, y, v[0:m], P[0:m, 0:m];
  integer array ix[0:m+n], ro[0:m];
  integer mu, nu, alpha, beta, gamma, gm1, im1, i, j, k, l;
  real t1, t2, infinity, prevz, eta;
  real procedure Gmat(ri, ci);
    value ri, ci;  integer ri, ci;
      Gmat := if ri = m then (if ci < n then 0 else 1.0)
              else if ci < n then G[ri, ci]
              else if ci − n = ri then 1.0 else 0;
  real procedure dvec(ii);  value ii;  integer ii;
    dvec := if ii < n then d[ii] else 0;
  procedure decompose (mat, bottom, top);
    value bottom, top;  integer bottom, top;  real mat;
  comment  This procedure performs a column-by-column re-
    duction of the matrix given by mat, forming an upper and a
    lower triangular matrix into the array P. (Each diagonal ele-
    ment of the lower triangular matrix is 1.) Interchanges of rows
    take place so that the largest pivot in each column is em-
    ployed. If P already contains the LU decomposition of a
    matrix differing from mat in only the (beta)-th column, ad-
    vantage is taken of this. The parameters bottom and top enable
    decompose to concentrate on a lower right-hand submatrix of
    mat. This feature saves computation during phase 1. If mat
    is singular, exit singular is taken;
  begin
```

```
      for i := beta step 1 until mu do
      begin
        im1 := i − 1;  l := ix[i];
        trisolv(if i=beta then bottom else top, 1, im1, bottom, j − 1,
          j, k, P[ro[k], i], mat, P[ro[j], k], 1.0);
        trisolv(i, 1, mu, bottom, im1, j, k, P[ro[k], i], mat,
          P[ro[j], k], 1.0);
        t1 := 0;
        for j := i step 1 until mu do
        begin
          t2 := P[ro[j], i];
          if abs(t1) < abs(t2) then begin t1 := t2;  k := j end
        end;
        if t1 = 0 then go to singular;
        if i = mu then go to decompover;
        j := ro[i];  ro[i] := ro[k];  ro[k] := j;
        for j := i + 1 step 1 until mu do P[ro[j], i] :=
          P[ro[j], i]/t1
      end;
decompover:
  end decompose;
  procedure findbeta;
  comment This procedure determines which of the basic
    variables is to become nonbasic;
  begin
    t1 := infinity;
    for i := 0 step 1 until mu do
    begin
      if y[i] > 0 then
      begin
        t2 := h[i]/y[i];
        if t2 < t1 then begin t1 := t2;  beta := i end
      end
    end
  end findbeta;
  procedure findalpha(mat, vec);  real mat, vec;
  comment This procedure determines which of the nonbasic
    variables is to be made basic;
  begin
    t1 := infinity;
    for i := mu + 1 step 1 until nu do
    begin
      k := ix[i];
      t2 := ip2(j, 0, mu, mat, w[j], vec);
      if t2 < t1 then begin alpha := i;  t1 := t2 end
    end
  end findalpha;
  procedure refine(mat, rhs, od, lp, up, vec, fi, si, ord, ill);  value
    ord;  integer ord, fi, si;  real mat, rhs, od, lp, up, vec;  label
    ill;
  comment This procedure makes an iterative refinement of
    vec, which is the solution of the matrix equation mat × vec =
    rhs. The matrix mat has order ord. The LU decomposition of
    mat is specified by od, lp, and up. Exit ill is taken if mat is too
    ill-conditioned for the refinement process to be successful.
    Note the global identifier eta, whose value and purpose are
    given in the next comment;
  begin
    array cor[0:ord];  real cnorm, snorm, eps, tt;  integer cnt;
    cnt := 0;  eps := 5 × eta;
loop:
    cnorm := snorm := 0;  cnt := cnt + 1;
    for fi := 0 step 1 until ord do
    begin
      cor[fi] := −ip2(si, 0, ord, mat, vec, −rhs);
      si := fi;  tt := abs(vec);
      if tt > snorm then snorm := tt
    end;
trisolv(0, 1, ord, 0, fi−1, fi, si, cor[si], cor[fi], od, lp);
    trisolv(ord, −1, 0, fi+1, ord, fi, si, cor[si], cor[fi], od, up);
    for si := 0 step 1 until ord do
    begin
      tt := cor[si];
      vec := vec + tt;
      if abs(tt) > cnorm then cnorm := abs(tt)
    end;
    if cnt > 15 then go to ill;
    if snorm ≠ 0 then
      begin if cnorm/snorm > eps then go to loop end
  end refine;
  comment At this point, infinity and eta are set to special
    values. Set infinity to the largest positive single-precision
    floating-point number. Set eta to the largest positive floating-
    point number such that 1.0 + eta = 1.0 − eta = 1.0 in single-
    precision arithmetic. The convergence of the iterative re-
    finement process which is applied in refine is determined using
    eta;
  prevz := −infinity;
  for i := 0 step 1 until m do ro[i] := i;
  comment Determine from kappa whether phase 1 is to be
    skipped;
  if kappa ≥ m then
  begin
    nu := n + kappa − m−1;  l := 0;
    for i := 0 step 1 until nu do
    begin
      j := ind[i];  if j ≥ n then l := 1;  ix[i] := j
    end;
    mu := if l = 0 then m − 1 else m;
    go to phase 2
  end;
  mu := m − 1;  gamma := m − kappa;  gm1 := gamma − 1;
  nu := n + gm1;  l := n − m;
  comment Set up the appropriate phase 1 problem;
  for i := 0 step 1 until gm1 do
  begin
    ix[i] := n + i;
    P[i, i] := 1.0;
    for j := i + 1 step 1 until gm1 do P[i, j] := P[j, i] := 0;
    for j := gamma + 1 step 1 until mu do P[i, j] := G[i, l+j]
  end;
  for i := gamma step 1 until mu do
  begin
    ix[i] := l + i;
    for j := 0 step 1 until gm1 do P[i,j] := 0
  end;
  for i := m step 1 until nu do ix[i] := i − m;
  beta ·= gamma;
  go to no removal;
new phase 1 cycle:;
  comment Begin a new simplex step on the phase 1 problem.
    Check the phase 1 problem objective function;
  if ip2(i, 0, mu, w[i], b[i], 0) = 0 then go to phase 2;
  comment Determine which nonbasic variable is to become
    basic;
  findalpha(G[j,k],0);
  if t1 ≥ 0 then go to infeasible;
  j := ix[alpha];
  comment Solve a linear system for a vector y;
  trisolv(gamma, 1, mu, gamma, l − 1, l, k, v[k], G[ro[l],j],
    P[ro[l],k], 1.0);
  trisolv(mu, −1, gamma, l + 1, mu, l, k, y[k], v[l],
    P[ro[l],k], P[ro[l],l]);
  for i := 0 step 1 until gm1 do
  begin
    l := ro[i];
    y[i] := −ip2(k, gamma, mu, y[k], P[l,k], −G[l,j])
  end;
  comment Use the vector y to determine which basic variable
    becomes nonbasic. If the variable which has become non-
    basic is an artificial variable, remove it entirely from the
    problem and make an appropriate row and column inter-
    change upon the basis matrix P;
  findbeta;
  if beta ≥ gamma then
  begin
    k := ix[alpha]; ix[alpha] := ix[beta]; ix[beta] := k;
    go to no removal
  end;
  k := ro[gm1]; i := ro[gm1] := ro[beta]; ro[beta] := k;
  P[k, beta] := 1.0; P[i, beta] := 0;
  ix[beta] := ix[gm1]; ix[gm1] := ix[alpha]; beta := gm1;
  for i := alpha + 1 step 1 until nu do ix[i−1] := ix[i];
  gamma := gm1; gm1 := gm1 − 1; nu := nu − 1;
no removal:;
  comment Produce the LU decomposition of the new basis
    matrix;
  k := ix[beta];
  for i := 0 step 1 until gm1 do P[ro[i],beta] := G[ro[i],k];
  decompose(G[ro[j],l], gamma, gamma);
  comment Find the basic solution h;
  trisolv(gamma, 1, mu, gamma, j − 1, j, k, v[k],
    b[ro[j]], P[ro[j],k], 1.0);
  trisolv(mu, −1, gamma, j + 1, mu, j, k, h[k], v[j],
    P[ro[j],k], P[ro[j],j]);
  for i := 0 step 1 until gm1 do
  begin
    k := ro[i];
    h[i] := −ip2(j, gamma, mu, h[j], P[k,j], −b[k]);
    w[k] := −1.0
  end;
  comment Solve a linear system for the vector, w, of simplex
    multipliers;
  for i := gamma step 1 until mu do
  begin
    t1 := 0;
    for j := 0 step 1 until gm1 do t1 := t1 + P[ro[j],i];
    v[i] := t1
  end;
  trisolv(gamma, 1, mu, gamma, i − 1, i, j, v[j], v[i],
    P[ro[j], i] P[ro[i],i]);
  trisolv (mu, − 1, gamma, i + 1, mu, i, j, w[ro[j]], v[i], P[ro[j], i],
    1.0);
  go to new phase 1 cycle;
phase 2: ;
  comment Set up the appropriate phase 2 problem and make
    an initial LU decomposition if necessary;
  beta : = 0 ;
  if kappa < m then
  begin
    if gamma > 0 then
    begin
      kappa := m; nu := nu + 1; mu := m;
      ix[nu| := ix[mu]; ix[mu] := n + m
    end
  end;
  if kappa ≥ m then go to decomp
  else trisolv(0, 1, mu, 0, j − 1, j, k, q[k], if ro[j] = m then 0 else
    b[ro[j]], P[ro[j],k], 1.0);
new phase 2 cycle: ;
  comment Begin a new simplex step on the phase 1 problem.
    Solve a linear system for the vector, w, of simplex multipliers;
  trisolv(0, 1, mu, 0, i − 1, i, j, v[j], dvec(ix[i]), P[ro[j],i], P[ro[i],i]);
  trisolv(mu, −1, 0, i + 1, mu, i, j, w[ro[j]], v[i], P[ro[j],i], 1.0);
  comment Determine which nonbasic variable is to become
    basic;
  findalpha(Gmat(j,k), −dvec(k));
  comment Check whether the solution has been found;
  if t1 ≥ 0 then go to finished;
not done yet:
  i := ix[alpha];
  comment Solve a linear system for a vector y;
  trisolv(0, 1, mu, 0, j − 1, j, k, v[k], Gmat(ro[j],i), P[ro[j],k], 1.0);
  trisolv(mu, −1, 0, j + 1, mu, j, k, y[k], v[j], P[ro[j],k], P[ro[j],j]);
  comment Use y to determine which basic variable is to be-
    come nonbasic;
  findbeta;
  if t1 = infinity then go to unbounded;
  k := ix[beta]; ix[beta] := ix[alpha]; ix[alpha] := k;
decomp: ;
  comment Produce the LU decomposition of the new basis
    matrix;
  decompose(Gmat(ro[j],l), 0, beta);
  comment Compute the basic solution h;
  trisolv(beta, 1, mu, 0, j − 1, j, k, q[k], if ro[j] = m then 0 else
    b[ro[j]], P[ro[j],k], 1.0);
  trisolv(mu, −1, 0, j + 1, mu, j, k, h[k], q[j], P[ro[j],k], P[ro[j],j]);
  go to new phase 2 cycle;
finished: ;
  comment Refine w and the basic solution h. Compute the
    objective function. Check the refined results to determine
    whether the optimum has been reached. If the check indicates
    nonoptimality but the objective function is less than any
    value previously computed for it, return the best basic solu-
    tion obtained so far and print a warning that the solution
    has doubtful validity;
  refine(Gmat(ro[j],ix[i]), dvec(ix[i]), P[ro[j],i], P[ro[i],i], 1.0,
    w[ro[j]], i, j, mu, singular);
  z := ip2(i, 0, m − 1, w[i], b[i], 0);
  if z < prevz then
    begin comment Print out "doubtful solution"; end
  else
  begin
    prevz := z;
    refine(Gmat(ro[j], ix[k]), if ro[j] = m then 0 else b[ro[j]],
      P[ro[j],k], 1.0, P[ro[j],j], h[k], j, k, mu, singular);
    l := n − 1; kappa := nu + 1;
    for i := 0 step 1 until l do x[i] := 0;
    for i := 0 step 1 until nu do ind[i] := ix[i];
    for i := 0 step 1 until mu do
    begin
      j := ix[i];
      if j < n then x[j] := h[i]
    end;
    findalpha(Gmat(j,k), −dvec(k));
    if t1 < 0 then go to not done yet
  end
end linprog
```

ALGORITHM 351
MODIFIED ROMBERG QUADRATURE* [D1]

GRAEME FAIRWEATHER (Recd. 18 Sept. 1968 and 19 Feb. 1969)

Department of Applied Mathematics, University of St. Andrews, Fife, Scotland

KEY WORDS AND PHRASES: numerical integration, Romberg quadrature, trapezoid values, rectangle values, error bound

CR CATEGORIES: 5.16

Comments. *ROMINT* calculates the approximate value, *VAL*, of the definite integral

$$I = \int_A^B F(X)\, dX$$

and an error bound *ERR* for *VAL*, i.e. $| VAL - I | \leq ERR$. The integrand $F(X)$ must be given as a function subprogram with the heading *FUNCTION F(X)*. *VAL* is obtained from a modified form of Romberg quadrature which is less sensitive to the accumulation of rounding errors than the customary one. In this procedure, which was devised by Krasun and Prager [1], the following "skeleton" Romberg table is constructed:

$$\begin{array}{ccccccc}
T_0^0 & & & & & & \\
 & T_1^0 & & & & & \\
R_0^0 & & T_2^0 & & \cdot & & \\
 & R_1^0 & & \cdot & & \cdot & \\
R_0^1 & & R_2^0 & & & & T_m^0 \\
 & R_1^1 & & \cdot & & \cdot & \\
R_0^2 & \cdot & & \cdot & & \cdot & R_m^0 \\
\cdot & \cdot & R_2^{m-2} & & & & \\
\cdot & R_1^{m-1} & & & & & \\
R_0^m & & & & & &
\end{array}$$

where $m \leq MAXE$, *MAXE* being on entry the maximum number of extrapolations wanted. In this subroutine $MAXE \leq 15$. The quantities R_0^k ($k = 0, 1, \cdots, m$) are the rectangle values,

$$R_0^k = \frac{B - A}{2^k} \sum_{j=1}^{2^k} F\left(A + \left(j - \frac{1}{2}\right)\frac{B - A}{2^k}\right),$$

which are calculated using a procedure proposed by Rutishauser [2] to reduce the effect of rounding errors. The quantities R_j^k ($j > 0$) are computed using the usual extrapolation formula:

$$R_j^k = R_{j-1}^{k+1} + \frac{R_{j-1}^{k+1} - R_{j-1}^k}{4^j - 1}, \qquad k = 0, 1, \cdots, m - j.$$

The formula (see [1])

$$T_j^0 = R_{j-1}^0 + \frac{2 \cdot 4^{j-1} - 1}{4^j - 1}(T_{j-1}^0 - R_{j-1}^0), \qquad j = 1, \cdots, m,$$

enables one to determine the extrapolated trapezoid values $T_1^0, T_2^0, \cdots, T_m^0$ in the skeleton table from the trapezoid value

$$T_0^0 = \frac{B - A}{2}[F(A) + F(B)]$$

and the rectangle values $R_0^0, R_1^0, \cdots, R_{m-1}^0$. In this subroutine only one linear array for storing the quantities R_j^{m-j}, $j = 0, \cdots, m$ ($\leq MAXE$) is required.

The subroutine is left when (see [3])

$$ERR = \frac{| T_m^0 - R_m^0 |}{2} \leqq EPS,$$

where *EPS* specifies the desired accuracy, or when *MAXE* extrapolations have been performed. On exit, $VAL = (T_m^0 + R_m^0)/2$ ($m \leq MAXE$) and $N = 2^{(m+1)} + 1$ is the number of function evaluations. The exit value of *MAXE* is *m* unless the maximum number of extrapolations wanted has been performed without the desired accuracy being obtained, in which case the exit value of *MAXE* is zero.

This subroutine can be used to estimate the definite integral *I* provided $F(X)$ is at least three or four times differentiable and is not periodic with period $B - A$.

Test cases. Two test cases were carried out on the IBM 1620 of the Computing Laboratory, University of St. Andrews, to compare *ROMINT* with a FORTRAN II-D version of *havieintegrator* [4]. The calculations were carried through in single-precision, i.e. working to 8 significant decimal digits. The results are summarized in the following table.

Integrand	*A*	*B*	*EPS*	True value	*havieintegrator*	*ROMINT*	Number of extrapolations
$\cos x$	0.0	$\pi/2$	10^{-6}	1.0	0.99999985	0.99999995	3
e^{-x^2}	0.0	4.3	10^{-6}	0.88622692	0.88622665	0.88622675	5

REFERENCES:

1. KRASUN, A. M., AND PRAGER, W. Remark on Romberg quadrature. *Comm. ACM 8* (Apr. 1965), 236–237.
2. RUTISHAUSER, H. Description of Algol 60. In *Handbook for Automatic Computation, Vol. 1*, Springer-Verlag, Berlin, 1968, Part a.
3. HAVIE, T. On a modification of Romberg's algorithm. *BIT 6* (1966), 24–30.
4. KUBIK, R. N. Algorithm 257, Havie integrator. *Comm. ACM 8* (June 1965), 381.

```
      SUBROUTINE  ROMINT
C     ******************
     *  (VAL,ERR,EPS,A,B,N,MAXE)
        DIMENSION  RM(16)
C    INITIAL TRAPEZOID VALUE ..
      T = (B-A)*(F(A)+F(B))*0.5
C
C    INITIAL RECTANGLE VALUE ..
      RM(1) = (B-A)*F((A+B)*0.5)
```

* This work was based in part on work done at U.K.A.E.A., Culham Laboratory, Abingdon, England.

```
C
        N = 2
        R = 4
        DO 11  K = 1,MAXE
           BB = (R*0.5-1.)/(R-1.)
C     IMPROVED TRAPEZOID VALUE ..
           T = RM(1)+BB*(T-RM(1))
C
C     DOUBLE NUMBER OF SUBDIVISIONS
C     OF (A,B) ..
           N = 2*N
C
           S = 0
           H = (B-A)/FLOAT(N)
C
C     CALCULATE RECTANGLE VALUE ..
           IF(N-32) 1,1,2
    1         NO = N
              GO TO 3
    2         NO = 32
    3      IR(N-512) 4,4,5
    4         N1 = N
              GO TO 6
    5         N1 = 512
    6      DO 9  K2 = 1,N,512
              S1 = 0
              KK = K2+N1-1
              DO 8  K1 = K2,KK,32
                 SO = 0
                 KKK = K1+NO-1
                 DO 7  KO = K1,KKK,2
                    SO = SO+F(A+FLOAT(KO)*H)
    7            CONTINUE
                 S1 = SO+S1
    8         CONTINUE
              S = S+S1
    9      CONTINUE
           RM(K+1) = 2.*H*S
C     END CALCULATION OF RECTANGLE VALUE.
C
           R = 4
C     FORM ROMBERG TABLE FROM RECTANGLE
C     VALUES ..
           DO 10  J = 1,K
              L = K+1-J
              RM(L) = RM(L+1)+(RM(L+1)-RM(L))
     *                               /(R-1.)
              R = 4.*R
   10      CONTINUE
C
           ERR = ABS(T-RM(1))*0.5



C     CONVERGENCE TEST ..
           IF(ERR-EPS) 12,12,11
C
   11 CONTINUE
   12 VAL = (T+RM(1))*0.5
      N = N+1
      IF(K-MAXE) 14,13,13
   13    MAXE = 0
         GO TO 15
   14    MAXE = K
   15 RETURN
      END
```

REMARK ON ALGORITHM 351 [D1]
MODIFIED ROMBERG QUADRATURE [Graeme Fairweather, *Comm. ACM 12* (June 1969), 324]

N. D. Cook (Recd. 11 Sept. 1969)
Bettis Atomic Power Laboratory, P.O. Box 79, West Mifflin, PA 15122

KEY WORDS AND PHRASES: numerical integration, Romberg quadrature, trapezoid values, rectangle values, error bound
CR CATEGORIES: 5.16

There is an error in calculating the output value MAXE in the algorithm in the case where the desired accuracy is obtained by the last requested extrapolation. Statement 11 (the end of the DO loop on K) should be followed by:

```
   K = 0
12 VAL = (T+RM(1))*0.5
   N = N+1
   MAXE = K
   RETURN
   END
```

When the two test cases were repeated in single precision on the CDC-6600, the 14-digit arithmetic yielded results accurate to 10 digits with the same number of extrapolations as used to get 6-digit results on the 8-digit IBM-1620. The time spent in ROMINT was 0.7 and 2.0 msec for the cosine and e^{-x^2} integrals respectively, with a total time of 1.1 and 3.8 msec when the time spent evaluating the functions is included.

REMARK ON ALGORITHM 351 [D1]
MODIFIED ROMBERG QUADRATURE
[G. Fairweather, *Comm. ACM 12* (June 1969), 324]

George C. Wallick
Mobil Research and Development Corporation, Field Research Laboratory, P. O. Box 900, Dallas, TX 75221

KEY WORDS AND PHRASES: numerical integration, Romberg quadrature, modified Romberg quadrature, trapezoid values, rectangle values
CR CATEGORIES: 5.16

Algorithm 351 was compiled and run successfully in Fortran IV on a CDC 6400 computer. Computation times for equivalent orders were essentially the same as for a Fortran version of Algorithm 60 Romberg Integration [1]; storage requirements were approximately 20 percent greater.

Algorithm 351 incorporates two modifications to the standard Romberg algorithm, each designed to reduce roundoff: (1) the Krasun and Prager [3] replacement of the table of trapezoidal values T_j^k with a table of rectangular values R_j^k; (2) the method proposed by Rutishauser [6] for the evaluation of the rectangular sums R_0^k. Since neither of these modifications has been properly evaluated we have chosen to compare integral values returned by five variants of the Romberg algorithm:

1. Conventional Romberg integration as described by Algorithm 60
2. A Krasun and Prager modification of Algorithm 60 (T_j^k table replaced by R_j^k table)
3. A Rutishauser modification of Algorithm 60 (T_j^k table extrapolation with improved evaluation of the R_0^k)
4. Modified Romberg integration as described by Algorithm 351 (R_j^k table; improved R_0^k evaluation)
5. Algorithm 351 with the Rutishauser procedure replaced by the standard evaluation of the R_0^k (R_j^k table extrapolation)

The following test integrals were investigated.

A. $$\int_{.01}^{1.1} x^{-\alpha}\,dx, \quad \alpha = 3.0, 4.0, 5.0$$

B. $$\int_{0}^{1} (1 + x^{\alpha})^{-1}\,dx, \quad \alpha = 1.0, 4.0$$

C. $$\int_{1}^{10} \ln x\,dx$$

D. $$\int_{0}^{5} e^{-x^2}\,dx$$

Integral A was suggested by Thacher [7], Integral B by Rabinowitz [5], Integral C by Hillstrom [2], and Integral D by Hillstrom and by Kubik [4]. All computation was carried out in CDC 6400 single-precision floating-point arithmetic. Results were recorded to 14 decimal digits. (CDC 6400 word length corresponds to 14+ decimal digits.) The data obtained in this manner are summarized in Tables I-IV.

For a specified order of extrapolation m, Algorithm 60 variants require $2^m + 1$ function evaluations and return $T_m{}^0$. Algorithm 351 requires $2^{(m+1)} + 1$ function evaluations and returns $T_m{}^1$. Thus one cannot meaningfully compare integral values returned by the two algorithms for the same specified order. We have therefore chosen to compare integral values resulting from the same number of function evaluations and have tabulated these data in terms of the Algorithm 60 order m. The corresponding specified order for Algorithm 351 variants is $m - 1$.

In each example considered, Algorithm 351 returns integral values for the optimum extrapolation order that are more accurate than the Algorithm 60 solutions by from one to two significant figures. There is, of course, no increase in the rate of convergence and little difference in solution accuracy for approximation orders less than that corresponding to the maximum attainable accuracy. If one were interested in, e.g. six or eight significant figure accuracy, either algorithm would be satisfactory. If accuracy requirements are not severe and one is satisfied with integral values correct to a number of significant figures less than half the computer word length, either algorithm may be used. If one seeks the maximum achievable accuracy, Algorithm 351 is clearly the proper choice.

Tables I-IV include data recorded when the order was overspecified, i.e. when m was greater than that required for optimum accuracy. For both algorithms the accuracy at first increases with increasing order. This continues until an optimum accuracy obtains. With Algorithm 60 a further increase in m results in a decline, at times rather rapid, in evaluation accuracy. With Algorithm 351 there is little loss in accuracy with increasing order. The accuracy decline rate is strongly retarded and in many cases practically eliminated. This is a very significant result.

In routine use of the algorithms, the unwary may overestimate the order required for optimum convergence (Algorithm 60 terminates only when a specified order has been obtained) or may specify an accuracy criterion for termination that cannot be satisfied. With Algorithm 351 the only loss is that of computer time; with Algorithm 60 solution accuracy may be impaired.

From the data presented in Tables I-IV we may determine the extent to which each of the procedural modifications contributes to the overall superiority of Algorithm 351. It is immediately evident that the Krasun and Prager modification has little effect either on the accuracy of the algorithms or on the loss of accuracy as the optimum order is exceeded. Results obtained using this modification differ from those returned by Algorithm 60 by at most 2 in the 14th figure. When the Rutishauser procedure is subtracted from Algorithm 351, the algorithm becomes, for all practical purposes, equivalent in accuracy to Algorithm 60. This conclusion has been further supported by results obtained in the evaluation of eight additional test integrals selected from the literature.

If, on the other hand, the Rutishauser procedure is added to Algorithm 60, the results obtained are essentially the same as those recorded for Algorithm 351. Clearly the Rutishauser modification is the dominant factor determining the superiority of Algorithm 351.

The success of the Rutishauser modification tempts one to expand the procedure to include an additional summation level. Experiments with such expansions indicate that they may be of value where slow Romberg convergence requires the use of orders $m > 13$.

The following changes are suggested as possible improvements in the algorithm. The integration interval $(B-A)$ is now computed $K + 2$ times where K is the order of approximation on exit from the routine. We suggest an initial definition of a variable, e.g. $SH = (B-A)$ and the replacement of $(B-A)$ by SH in these statements where $(B-A)$ appears. Initialization should also include a test to insure that the maximum extrapolation order $MAXE$ permitted is less than or equal to 15 with a possible replacement $MAXE = 15$ if this condition is violated. Alternatively, one could replace the statement DO 11 $K = 1$, $MAXE$ with DO 11 $K = 1$, 15 and test for $K < MAXE$ prior to executing statement no. 11. The GO TO 3 statement following statement no. 1 should read GO TO 4. If $N \leq 32$, N is also ≤ 512.

Upon exit, the input parameter $MAXE$ is assigned either the value $MAXE = K$, where K is the approximation order, or $MAXE = 0$ if the accuracy criterion has not been satisfied. We believe that it is poor programming practice to have a subroutine alter the value of an input parameter. We suggest the addition of an output parameter, e.g. $MFIN = K$ which returns the order on exit. Where we now set $MAXE = 0$, we could set $MFIN = 16$. One can test as easily for $MFIN \leq 15$ as for $MAXE = 0$. This would eliminate the necessity for resetting $MAXE$ each time the subroutine is entered. It is also useful to return the final value of the accuracy ERR. In the event that $MAXE = 0$, one could test ERR to determine whether or not the returned integral value falls within acceptable limits.

In practical applications we prefer to express the procedure as a function subprogram and to add the name of the generating function F to the argument list. We also consider a test for relative error rather than absolute error to be more useful in routine use of the algorithm.

The author wishes to thank the Mobil Research and Development Corporation for permission to publish this information.

References:

1. Bauer, F. L. Algorithm 60, Romberg integration. *Comm. ACM 4* (June 1961), 255.
2. Hillstrom, K. Certification of Algorithm 257, Havie integrator. *Comm. ACM 9* (Nov. 1966), 795.
3. Krasun, A. M., and Prager, W. Remark on Romberg quadrature. *Comm. ACM 8* (Apr. 1965), 236-237.
4. Kubik, R. N. Algorithm 257, Havie integrator. *Comm. ACM 8* (June 1965), 381.
5. Rabinowitz, P. Automatic integration of a function with a parameter. *Comm. ACM 9* (Nov. 1966), 804-806.
6. Rutishauser, H. Description of Algol 60. In *Handbook for Automatic Computation, Vol. 1,* Springer-Verlag, New York, 1967, Part a, 105-106.
7. Thacher, H. C., Jr. Certification of Algorithm 60, Romberg integration. *Comm. ACM 5* (Mar. 1962), 168.

TABLES. Comparisons of Romberg Method Variations

(KP = Krasun-Prager Modification; RUT = Rutishauser Modification; NSF = Number of Significant Figures)

I. In the Evaluation of $I(\alpha) = \int_0^1 (1 + x^\alpha)^{-1}\, dx$

$I(1) = 0.69314\ 71805\ 59945$; $I(4) = 0.86697\ 29873\ 3991$

		Variations Returning T_m^0						*Variations Returning T_m^1*			
		Algorithm 60		*Algorithm 60 + KP*		*Algorithm 60 + RUT*		*Algorithm 351 (KP + RUT)*		*Algorithm 351 (KP only)*	
α	*Romberg Order m*	*Digits 1–14*	*NSF*	*Digits 11–14*	*NSF*	*Digits 11–14*	*NSF*	*Digits 6–14*	*NSF*	*Digits 11–14*	*NSF*
1.0	3	69314 74776 4482	6	4482	6	4482	6	79014 8123	5	8123	5
	4	69314 71819 1673	8	1673	8	1673	8	71830 7192	8	7192	8
	5	69314 71805 6227	11	6228	11	6227	11	71805 6360	11	6360	11
	6	69314 71805 5991	13	5992	13	5992	13	71805 5993	13	5992	13
	7	69314 71805 5987	12	5988	12	5991	13	71805 5992	13	5988	12
	8	69314 71805 5984	12	5984	12	5990	13	71805 5992	13	5984	12
	9	69314 71805 5971	12	5972	12	5989	12	71805 5990	13	5972	12
	10	69314 71805 5951	12	5951	12	5988	12	71805 5989	12	5951	12
	11	69314 71805 5906	11	5906	11	5991	13	71805 5990	13	5906	11
	12	69314 71805 5822	11	5822	11	5987	12	71805 5989	12	5822	11
4.0	4	86697 29736 8070	7	8070	7	8070	7	30046 3711	7	3711	7
	5	86697 29872 2539	9	2539	9	2539	9	29872 1216	9	1216	9
	6	86697 29873 4006	12	4006	12	4007	12	29873 4005	12	4003	12
	7	86697 29873 3983	12	3984	12	3987	13	29873 3988	13	3984	12
	8	86697 29873 3977	12	3978	12	3986	13	29873 3987	13	3979	12
	9	86697 29873 3963	12	3964	12	3985	12	29873 3986	13	3964	12
	10	86697 29873 3939	11	3940	11	3985	12	29873 3985	12	3940	11
	11	86697 29873 3890	11	3890	11	3984	12	29873 3986	13	3890	11
	12	86697 29873 3787	11	3788	11	3983	12	29873 3985	12	3788	11

II. In the Evaluation of $I(\alpha) = \int_{.01}^{1.1} x^{-\alpha}\, dx$

$I(3) = 0.49995\ 86776\ 85950 \times 10^4$; $I(4) = 0.33333\ 30828\ 95066 \times 10^6$;
$I(5) = 0.24999\ 99982\ 9247 \times 10^8$

		Variations Returning T_m^0						*Variations Returning T_m^1*			
		Algorithm 60		*Algorithm 60 + KP*		*Algorithm 60 + RUT*		*Algorithm 351 (KP + RUT)*		*Algorithm 351 (KP only)*	
α	*Romberg Order m*	*Digits 1–14*	*NSF*	*Digits 11–14*	*NSF*	*Digits 11–14*	*NSF*	*Digits 6–14*	*NSF*	*Digits 11–14*	*NSF*
3.0	8	50289 45604 1249	2	1249	2	1255	2	49952 9475	2	9469	2
	9	50007 88217 4010	3	4010	3	4037	3	88324 8156	3	8128	3
	10	49996 05996 3754	5	3755	5	3813	5	05997 5088	5	5029	5
	11	49995 86888 2917	7	2917	7	3041	7	86888 3087	7	2962	7
	12	49995 86777 0553	10	0553	10	0814	10	86777 0815	10	0553	10
	13	49995 86776 8069	10	8070	10	8588	12	86776 8590	12	8070	10
	14	49995 86776 7547	10	7549	10	8585	12	86776 8587	12	7549	10
	15	49995 86776 6495	10	6496	10	8581	12	86776 8583	12	6496	10
4.0	8	33918 76383 3713	1	3713	1	3717	1	83321 8573	1	8568	1
	9	33362 40891 0012	3	0011	3	0028	3	41103 2353	3	2337	3
	10	33333 86458 8643	4	8642	4	8682	4	86461 5904	4	5865	4
	11	33333 31207 4466	7	4466	7	4547	7	31207 4679	7	4598	7
	12	33333 30829 8056	9	8055	9	8220	9	30829 8220	9	8056	9
	13	33333 30828 9178	11	9178	11	9508	13	30828 9509	13	9178	11
	14	33333 30828 8842	10	8843	10	9500	12	30828 9501	12	8843	10
	15	33333 30828 8163	10	8163	10	9497	12	30828 9499	12	8163	10
5.0	8	25979 73076 7608	1	7608	1	7611	1	82577 2026	1	2023	1
	9	25058 17539 3846	2	3846	2	3857	2	17890 9312	2	9300	2
	10	25001 31264 6257	4	6257	4	6282	4	31270 0511	4	0486	4
	11	25000 01021 0524	6	0524	6	0576	6	01021 0887	6	0835	6
	12	24999 99985 6515	9	6515	9	6621	9	99985 6622	9	6516	9
	13	24999 99982 9053	11	9053	11	9267	12	99982 9268	12	9054	11
	14	24999 99982 8817	11	8818	11	9242	13	99982 9243	13	8818	11
	15	24999 99982 8379	10	8380	10	9241	12	99982 9242	13	8380	10

III. In the Evaluation of $I = \int_1^{10} \ln x\, dx =$ 14.025 85092 99404 6

		Variations Returning T_m^0						*Variations Returning T_m^1*			
		Algorithm 60		*Algorithm 60 + KP*		*Algorithm 60 + RUT*		*Algorithm 351 (KP + RUT)*		*Algorithm 351 (KP only)*	
α	*Romberg Order m*	*Digits 1–14*	*NSF*	*Digits 11–14*	*NSF*	*Digits 11–14*	*NSF*	*Digits 6–14*	*NSF*	*Digits 11–14*	*NSF*
	4	14025 60234 7275	5	7275	5	7275	5	60498 3885	5	3885	5
	5	14025 84455 4627	6	4627	6	4627	6	84433 5675	6	5675	6
	6	14025 85085 2042	8	2043	8	2043	8	85085 0505	8	0505	8
	7	14025 85092 9556	11	9556	11	9556	11	85092 9552	11	9551	11
	8	14025 85092 9938	13	9938	13	9939	13	85092 9939	13	9938	13
	9	14025 85092 9937	13	9937	13	9940	14	85092 9940	14	9937	13
	10	14025 85092 9934	12	9934	12	9939	13	85092 9940	14	9934	12
	11	14025 85092 9928	12	9929	12	9939	13	85092 9940	14	9929	12
	12	14025 85092 9916	12	9916	12	9940	14	85092 9939	13	9916	12

IV. In the Evaluation of $I = \int_0^5 e^{-x^2}\, dx =$ 0.88622 69254 51396

		Variations Returning T_m^0						*Variations Returning T_m^1*			
		Algorithm 60		*Algorithm 60 + KP*		*Algorithm 60 + RUT*		*Algorithm 351 (KP + RUT)*		*Algorithm 351 (KP only)*	
α	*Romberg Order m*	*Digits 1–14*	*NSF*	*Digits 11–14*	*NSF*	*Digits 11–14*	*NSF*	*Digits 6–14*	*NSF*	*Digits 11–14*	*NSF*
	5	88622 59970 9402	5	9043	5	9042	5	59296 9073	5	9073	5
	6	88622 69310 8538	7	8539	7	8541	7	69308 5739	7	5736	7
	7	88622 69254 4529	10	4529	10	4535	10	69254 4570	10	4564	10
	8	88622 69254 5117	12	5117	12	5134	12	69254 5135	13	5117	12
	9	88622 69254 5093	12	5094	12	5131	12	69254 5134	12	5095	12
	10	88622 69254 5053	11	5054	11	5135	13	69254 5134	12	5054	11
	11	88622 69254 4974	11	4975	11	5130	12	69254 5133	12	4976	11
	12	88622 69254 4801	11	4802	11	5129	12	69254 5131	12	4803	11
	13	88622 69254 4463	10	4463	10	5128	12	69254 5129	12	4464	10
	14	88622 69254 3801	10	3802	10	5125	12	69254 5127	12	3803	10

REMARKS ON:
ALGORITHM 332 [S22]
JACOBI POLYNOMIALS [Bruno F. W. Witte, *Comm. ACM 11* (June 1968), 436]
ALGORITHM 344 [S14]
STUDENT'S *t*-DISTRIBUTION [David A. Levine, *Comm. ACM 12* (Jan. 1969), 37]
ALGORITHM 351 [D1]
MODIFIED ROMBERG QUADRATURE [Graeme Fairweather, *Comm. 12* (June 1969), 324]
ALGORITHM 359 [G1]
FACTORIAL ANALYSIS OF VARIANCE [John R. Howell, *Comm. ACM 12* (Nov. 1969), 631]

Arthur H. J. Sale (Recd. 16 Feb. 1970)
Basser Computing Department, University of Sydney, Sydney, Australia

KEY WORDS AND PHRASES: Fortran standards
CR CATEGORIES: 4.0, 4.22

An unfortunate precedent has been set in several recent algorithms of using an illegal FORTRAN construction. This consists of separating an initial line from its continuation line by a comment line, and is forbidden by the standard (see sections 3.2.1, 3.2.3 and 3.2.4 of [1, 2]). The offending algorithms are to date: 332, 344, 351 and 359.

While this is perhaps a debatable decision by the compilers of the standard, and trivial to correct, it seems a pity to break the rules just for a pretty layout as has been done.

References:

1. ANSI Standard FORTRAN (ANSI X3.9-1966), American National Standards Institute, New York, 1966.
2. FORTRAN vs. Basic FORTRAN, *Comm. ACM 7* (Oct. 1964), 591–625.

ALGORITHM 352 CHARACTERISTIC VALUES AND ASSOCIATED SOLUTIONS OF MATHIEU'S DIFFERENTIAL EQUATION [S22]

DONALD S. CLEMM (Recd. 2 June 1967, 18 Apr. 1968, 6 Jan. 1969 and 10 Mar. 1969)

Aerospace Research Laboratories
Wright-Patterson Air Force Base
OH 45433

KEY WORDS AND PHRASES: Mathieu's differential equation, Mathieu function, characteristic value, periodic solution, radial solution
CR CATEGORIES: 5.12

Comments Algorithm 352 is a package of double-precision FORTRAN routines which consists of the following primary routines:

MFCVAL--referred to as Algorithm 352 (Part A)
MATH--referred to as Algorithm 352 (Part B)
BESSEL--referred to as Algorithm 352 (Part C)

MFCVAL computes characteristic values of Mathieu's differential equation. MATH computes the associated solutions of this equation, using BESSEL as an auxiliary routine to evaluate Bessel functions. This latter routine may be used independently.

There are other, secondary routines included in the package, and the numbering system (e.g. Algorithm 352 (Part A.1)) indicates somewhat the mutual relation between them, as well as their relation to the primary routines. The functioning of the routines and the linkages between them are explained in the comments prefacing each one. All literature citations refer to the following list.

REFERENCES:

1. ABRAMOWITZ, M., AND STEGUN, I. A. (Eds.). *Handbook of Mathematical Functions*. NBS Appl. Math. Ser. 55, US Govt. Print. Off., Washington, D.C., 1964.
2. BLANCH, G. Numerical evaluation of continued fractions. *SIAM Rev. 6*, 4 (1964), 383-421.
3. BLANCH, G. Numerical aspects of Mathieu eigenvalues. *Rend. Circ. Mat. Palermo* (2) *15* (1966), 51-97.
4. BLANCH, G., AND CLEMM, D. S. *Tables Relating to the Radial Mathieu Functions, Vol. 1, Functions of the First Kind.* US Govt. Print. Off., Washington, D.C., 1962.
5. BLANCH, G., AND CLEMM, D. S. *Tables Relating to the Radial Mathieu Functions, Vol. 2, Functions of the Second Kind.* US Govt. Print. Off., Washington, D.C., 1965.
6. INCE, E. L. Tables of the elliptic cylinder functions. *Proc. Roy. Soc. Edinburgh 52* (1932), 355-423; also Zeros and turning points. *Proc. Roy. Soc. Edinburgh 52* (1932), 424-433.
7. National Bureau of Standards. *Tables Relating to Mathieu Functions.* Appl. Math. Ser. 59, US Govt. Print. Off., Washington, D.C., 1967. (second ed.)
8. STRATTON, J. A., MORSE, P. M., CHU, L. J., AND HUTNER, R. A. *Elliptic Cylinder and Spheroidal Wave Functions.* Wiley, New York, 1941.

Algorithm 352 (Part A)
MFCVAL (Characteristic Values)

Comments The subrountine MFCVAL computes the first N characteristic values, a, together with upper and lower bounds, of Mathieu's differential equation for nonnegative values of the real parameter, q. The equation can be written in the form

$$y'' + (a - 2q \cos 2x)y = 0, \qquad (1)$$

where $a = a_r$ ($a = b_r$) indicates a characteristic value associated with the even (odd) periodic solutions.

The method consists of three steps: (1) calculate a rough approximation based on coefficients obtained from curve-fitting of available tabulations, (2) determine crude upper and lower bounds, and (3) iterate, using a variation of Newton's method. For a justification of this method, see [3].

Explanation of the arguments:

N the given number of characteristic values desired
R given as N−1 or N according as the characteristic values are to be associated with the even or odd solutions, respectively
QQ the given nonnegative parameter q
CV the computed 6 by N array of characteristic values and bounds
J the number of characteristic values successfully computed. J ≠ N indicates that J values were computed

with an error occurring on the J + 1 value. A printed message will accompany such an error condition.

The output array, CV, must be appropriately dimensioned in the calling program and upon return will contain the following data:

For the Kth characteristic value, K = 1, 2, $\cdots$, J,

CV (1, K)	the characteristic value a
CV (2, K)	the function $D(a) = -T_m(a)/T_m'(a)$
CV (3, K)	a_L , a lower bound of a
CV (4, K)	the function $D(a_L)$
CV (5, K)	a_U , an upper bound of a
CV (6, K)	the function $D(a_U)$.

Reference is again made to [3], where the function $T_m(a)$ is defined and it is proved that $T_m(a) = 0$ if and only if a is a characteristic value. From this, it can be said that the function D is an indication of the accuracy of its argument, since $a + D(a)$ would be the value of the next iteration.

The first executable statement in MFCVAL sets a tolerance of 10^{-13}. This may be changed by the user, but the following comments should be heeded if it is attempted.

If it is desired to reduce the tolerance in order to achieve the greatest possible accuracy, care should be taken that the tolerance is not less than $10^{-(n-2)}$ when executing the routines on a machine which uses n-digit arithmetic. In other words, if the user's computer employs 24-digit arithmetic, this tolerance should be no less than 10^{-22}. A too small tolerance will impose an unattainable accuracy requirement and overflow may occur.[1]

On the other hand, some time-saving may be achieved, at the expense of accuracy, by making the tolerance less stringent. A tolerance of 10^{-d} will produce results good to at least d digits. This is a conservative estimate, since one additional iteration is performed after the tolerance is met and, normally, the convergence of successive iterations is quadratic.

Perhaps it should be noted again that the accuracy of any characteristic value, a, can be determined from the size of it relative to the function $D(a)$. See the description of the contents of the output array CV. MFCVAL calls on the subroutines:

BOUNDS—referred to as Algorithm 352 (Part A.1)

MFITR8—referred to as Algorithm 352 (Part A.2)

TMOFA—referred to as Algorithm 352 (Part A.3)

[1] The constant in statement numbers 425 and 445 is introduced to avoid the possibility of a zero tolerance. This should not be altered unless the routines are being run on a machine which uses arithmetic of more than 16 digits, and then it must not be less than $10^{-(n-2)}$, with n defined as above.

```
      SUBROUTINE MFCVAL (N,R,QQ,CV,J)
C     *****************

      INTEGER
     *          J,K,KK,L,M,N,R,TYPE

      DOUBLE PRECISION
     *          A,CV,DL,DR,DTM,Q,QQ,
     *          T,TM,TOL,TOLA

      DIMENSION
     *          CV(6,N)

      EQUIVALENCE
     *          (DL,DR,T)

      COMMON /MF1/
     *          Q,TOL,TYPE,DUMMY(4)

      TOL = 1.D-13

      IF (N-R) 10,10,20

   10 L    = 1
                                      GO TO 30
   20 L    = 2
   30 Q    = QQ
      DO 500 K  = 1,N
         J      = K

         IF (Q) 960,490,40

   40    KK     = MINO(K,4)
         TYPE   = 2*MOD(L,2)+MOD(K-L+1,2)

C    FIRST APPROXIMATION
         GO TO (100,200,300,400), KK

  100    IF (Q-1.D0) 110,140,140

  110    GO TO (120,130), L
  120    A      = 1.D0-Q-.125D0*Q*Q
                                      GO TO 420
  130    A      = Q*Q
         A      = A*(-.5D0+.0546875D0*A)
                                      GO TO 420

  140    IF (Q-2.D0) 150,180,180

  150    GO TO (160,170), L
  160    A      = 1.033D0-1.0746D0*Q-
     *            .0688D0*Q*Q
                                      GO TO 420
  170    A      = .23D0-.495D0*Q-
     *            .191D0*Q*Q
                                      GO TO 420
  180    A      = -.25D0-2.D0*Q+
     *            2.D0*DSQRT(Q)
                                      GO TO 420

  200    DL     = L

         IF (Q*DL-6.D0) 210,350,350

  210    GO TO (220,230), L
  220    A      = 4.01521D0-Q*
     *            (.046D0+.0667857D0*Q)
                                      GO TO 420
  230    A      = 1.D0+1.05007D0*Q-
     *            .180143D0*Q*Q
                                      GO TO 420

  300    IF (Q-8.D0) 310,350,350

  310    GO TO (320,330), L
  320    A      = 8.93867D0+.178156D0*Q-
     *            .0252132D0*Q*Q
                                      GO TO 420
  330    A      = 3.70017D0+.953485D0*Q-
     *            .0475065D0*( *Q
                                      GO TO 420
  350    DR     = K-1
         A      = CV(1,K-1)-DR+

     *            4.D0*DSQRT(Q)
                                      GO TO 420

  400    A      = CV(1,K-1)-CV(1,K-2)
         A      = 3.D0*A+CV(1,K-3)

  420    IF (Q.GE.1.D0) GO TO 440

         IF (K.NE.1) GO TO 430
  425     TOLA   = DMAX1(DMIN1(TOL,DABS(A))
     *                  ,1.D-14)
                                GO TO 450
  430     TOLA   = TOL*DABS(A)
                                GO TO 450
  440     TOLA   = TOL*DMAX1(Q,DABS(A))
  445     TOLA   = DMAX1(DMIN1(TOLA,DABS(A)
     *                     ,.4D0*DSQRT(Q))
     *                  ,1.D-14)

C    CRUDE UPPER AND LOWER BOUNDS
  450     CALL BOUNDS (K,A,TOLA,CV,N,M)

          IF (M.NE.0)
     *       IF (M-1) 470,910,900

C    ITERATE
          CALL MFITR8 (TOLA,CV(1,K),
     *                    CV(2,K),M)

          IF (M.GT.0) GO TO 920

C    FINAL BOUNDS AND FUNCTIONS, D
  470     T         = CV(1,K)-TOLA
          CALL TMOFA (T,TM,DTM,M)

          IF (M.GT.0) GO TO 940

          CV(3,K) = T
          CV(4,K) = -TM/DTM
  480     T         = CV(1,K)+TOLA
          CALL TMOFA (T,TM,DTM,M)

          IF (M.GT.0) GO TO 950

          CV(5,K) = T
          CV(6,K) = -TM/DTM
                                GO TO 500

C    Q EQUALS ZERO
  490     CV(1,K) = (K-L+1)**2
          CV(2,K) = 0.D0
          CV(3,K) = CV(1,K)
          CV(4,K) = 0.D0
          CV(5,K) = CV(1,K)
          CV(6,K) = 0.D0
  500 CONTINUE
  550                           RETURN

C    PRINT ERROR MESSAGES
  900 WRITE (6,901) K
  901 FORMAT(20H0CRUDE BOUNDS CANNOT,
     *        22H BE LOCATED, NO OUTPUT,
     *         7H FOR K=I2)
                                GO TO 930
  910 WRITE (6,911) K
  911 FORMAT(20H0ERROR IN SUBPROGRAM,
     *        22H TMOFA, VIA SUBPROGRAM,
     *        18H BOUNDS, NO OUTPUT,
     *         7H FOR K=I2)
                                GO TO 930
  920 WRITE (6,921) K
  921 FORMAT(20H0ERROR IN SUBPROGRAM,
     *        22H TMOFA, VIA SUBPROGRAM,
     *        18H MFITR8, NO OUTPUT,
     *         7H FOR K=I2)
  930 J         = J-1
                                GO TO 550
  940 WRITE (6,941) K
  941 FORMAT(20H0ERROR IN SUBPROGRAM,
     *        22H TMOFA, NO LOWER BOUND,
     *         7H FOR K=I2)
      CV(3,K) = 0.D0
      CV(4,K) = 0.D0
                                GO TO 480
  950 WRITE (6,951) K
  951 FORMAT(20H0ERROR IN SUBPROGRAM,
     *        22H TMOFA, NO UPPER BOUND,
     *         7H FOR K=I2)
      CV(5,K) = 0.D0
      CV(6,K) = 0.D0
                                GO TO 500
  960 WRITE (6,961)
  961 FORMAT(20H0Q GIVEN NEGATIVELY,,
     *        20H USED ABSOLUTE VALUE)
      Q         = -Q
                                 GO TO 40
      END
```

Algorithm 352 (Part B)
MATH (Mathieu Functions)

Comments The subroutine MATH computes various solutions (and their derivatives), of either Mathieu's differential equation or Mathieu's modified equation, which are associated with the characteristic values.

The *even periodic* solution of equation (1) is

$$ce_r(x,q) = \sum_{k=0}^{\infty} A_{2k+p} \cos\,(2k+p)x, \quad (2)$$

associated with $a_r(q)$, and the *odd periodic* solution is

$$se_r(x,q) = \sum_{k=0}^{\infty} B_{2k-p} \sin\,(2k+p)x, \quad (3)$$

associated with $b_r(q)$. The order, r, is of the form $2n + p$. The n is a nonnegative integer while $p = 0$ or 1 indicates the solution is of period π or 2π. Calculation of the periodic solutions allows the following three options of normalization:

(a) Neutral. We define *neutral* coefficients such that $\bar{A}_{2k+p} = A_{2k+p}/A_{2s+p}$, where s is chosen so that A_{2s+p} is the numerically largest one of the set. The $\bar{B}_{2k+p}$ are similarly defined. This has the computationally convenient effect of making the largest coefficient equal to unity, hence all calculations are carried out with them. If a normalization other than *neutral* is selected, it is effected on the output array F only, the coefficients themselves remaining unchanged.

(b) Ince. The normalization adopted in [6] is defined so that if $y(x,q)$ represents either function (2) or (3) then

$$\int_0^{2\pi} y^2(x,\,q)dx = \pi.$$

(c) Stratton. As defined in [8], and in the notation of [7], this normalization is effected so that

$$Se_r(q,\,0) = \left[\frac{d}{dx}\,So_r(q,\,x)\right]_{x=0} = 1,$$

where Se is the even solution and So the odd.

If we replace x by ix in (1), we get

$$y'' - (a-2q\cosh 2x)\,y = 0, \quad (4)$$

known as Mathieu's modified equation. The solutions of (4) have been termed *radial* in [8] and, for characteristic values, can be put in the following form, using the notation of [4] and [5]:

$$Mc_r^{(j)}\ (x,q) = \sum_{k=0}^{\infty} (-1)^{n+k} A_{2k+p}\,[F_k + G_k]/A_{2s+p}\epsilon_{2s+p}\,, \quad (5)$$

associated with $a_r(q)$, and

$$Ms_r^{(j)}(x,q) = \sum_{k=0}^{\infty} (-1)^{n+k} B_{2k+p}\ [F_k - \ _k]/B_{2s+p}\,, \quad (6)$$

associated with $b_r(q)$. The order r equals $2n + p$, as in (2) and (3), and $\epsilon_m = 1$ if $m \neq 0$, but $\epsilon_0 = 2$. The choice of s is arbitrary here, but for numerical purposes we choose it in the manner described previously for *neutral* normalization. The coefficients are the same as defined in (2) and (3), while F_k and G_k involve the Bessel functions as follows:

$$F_k = J_{k-s}(u_1)\ Z^{(j)}_{k+p+s}(u_2), \qquad (7)$$

$$G_k = J_{k+p+s}(u_1)\ Z^{(j)}_{k-s}(u_2), \qquad (8)$$

$$u_1 = q^{\frac{1}{2}}e^{-x}, \qquad u_2 = q^{\frac{1}{2}}e^{x},$$

$$Z^{(1)}_m(u) = J_m(u),\ Z^{(2)}_m(u) = Y_m(u).$$

The solutions (5)-(6) are said to be of the first or second kind depending on whether $j = 1$ or 2 in (5)-(8).

Explanation of the arguments:

XX the given independent variable x

QQ the given positive parameter q

R the given order r

CV the given characteristic value, $a_r(q)$ or $b_r(q)$

SOL given as 1, 2, or 3 according as the desired solution is (1) radial of the first kind, (2) radial of the second kind, or (3) periodic

FNC given as 1, 2, 3, or 4 according as the desired solution is (1) associated with b_r, (2) associated with a_r, (3) the derivative of solution (1), or (4) the derivative of solution (2)

NORM given as 1, 2, or 3 according as the desired normalization is (1) defined as *neutral*, (2) defined by Ince, or (3) defined by Stratton. (This argument is decoded only if SOL = 3.)

F the computed three-element array, containing: (1) the solution value, (2) the series term of largest magnitude, and (3) the last term included in the summation

K the computed two-element array, containing: (1) the index, k, of the term in F(2), and (2) the index of the term in F(3)

M the error indicator cell: M = 0 indicates successful execution of subprogram, M = 1 signifies an error condition explained by an accompanying printed message.

The accuracy of results (within limits) and the speed of convergence may be altered by the user. See SUM (Algorithm 352 (Part B.2)) for details.

MATH calls on the subroutines:

COEF—referred to as Algorithm 352 (Part B.1)

SUM—referred to as Algorithm 352 (Part B.2)

BESSEL—referred to as Algorithm 352 (Part C)

```
      CUBROUTINE MATH (XX,QQ,R,CV,SOL,
     *                 FNC,NORM,F,K,M)
C     ***************

      INTEGER
     *        FNC,I,K(2),KLAST,KMAX,L,
     *        LL,M,MF,ML,MM,MO,M1,M2S,
     *        N,NORM,P,R,S,SOL,TYPE

      DOUBLE PRECISION
     *        A,AB,CV,DLAST,DMAX,F(3),G,
     *        J,Q,QQ,T,TOL,U1,U2,X,XX,Y

      EXTERNAL
     *        DC,DDC,DDS,DS,DPC,DPS,
     *        PC,PS

      COMMON
     *        J(250),Y(250),U1,U2,N,P,S,
     *        L,X,T,I,LL,G,DMAX,DLAST,
     *        KMAX,KLAST,DUM1(578),A,
     *        DUM2(6),MM,ML,AB(200)

      COMMON /MF1/
     *        Q,TOL,TYPE,M1,MO,M2S,MF

      M     = 0

      IF (SOL.LT.1 .OR.
     *    SOL.GT.3 .OR.
     *    FNC.LT.1 .OR.
     *    FNC.GT.4) GO TO 40C

      A     = CV
      Q     = QQ
      TOL   = 1.D-13
      TYPE  = 2*MOD(FNC,2)+MOD(R,2)
      CALL COEF (M)

      IF (M) 410,10,420

   10 N     = R/2
      P     = MOD(R,2)
      S     = MM/2
      L     = ML/2
      X     = XX
      T     = 1.D0

      IF (SOL.EQ.3)
     *   GO TO (150,160,170,180), FNC

      U1    = DSQRT(Q)*DEXP(-X)
      U2    = Q/U1
      LL    = L+S+P

C   COMPUTE BESSEL FUNCTIONS
      CALL BESSEL (1,U1,J,LL)
      CALL BESSEL (SOL,U2,Y,LL)

C   EVALUATE SELECTED FUNCTION
      GO TO (50,60,70,80), FNC
   50 CALL SUM (DS)
                                 GO TO 300
   60 CALL SUM (DC)
                                 GO TO 300
   70 CALL SUM (DDS)
                                 GO TO 300
   80 CALL SUM (DDC)
                                 GO TO 300
  150 CALL SUM (PS)
                                 GO TO 200
  160 CALL SUM (PC)
                                 GO TO 200
  170 CALL SUM (DPS)
                                 GO TO 200
  180 CALL SUM (DPC)

  200 IF (NORM-2) 300,210,250

C   INCE NORMALIZATION
  210 T     = AB(1)**2

      IF (TYPE.EQ.0) T = T+T

      DO 220 I = 1,L
         T     = T+AB(I+1)**2
  220 CONTINUE
      T     = DSQRT(T)
      I     = MO/2
      IF (AB(I).LT.0.D0) T = -T
                                    GO TO 300

C     STRATTON NORMALIZATION

  250 IF (TYPE.GT.1) GO TO 270

      T     = AB(1)
      DO 260 I = 1,L
         T      = T+AB(I+1)
  260 CONTINUE
                                    GO TO 300
  270 T     = DBLE(FLOAT(P))*AB(1)
      DO 280 I = 1,L
         T      = T+AB(I+1)*
     *             DBLE(FLOAT(2*I+P))
  280 CONTINUE

  300 F(1) = G/T
      F(2) = DMAX/T
      F(3) = DLAST/T
      K(1) = KMAX
      K(2) = KLAST
  350                                 RETURN

C     PRINT ERROR MESSAGES
  400 WRITE (6,401)
  401 FORMAT(18H0SOL OR FNC OUT OF,
     *         17H RANGE, NO OUTPUT)
                                    GO TO 450
  410 WRITE (6,411)
  411 FORMAT(15H0MORE THAN 200 ,
     *         22HCOEFFICIENTS REQUIRED,,
     *         20H QQ AND R TOO LARGE,,
     *         10H NO OUTPUT)
                                    GO TO 450
  420 WRITE (6,421)
  421 FORMAT(20H0ERROR IN SUBPROGRAM,
     *         22H TMOFA, VIA SUBPROGRAM,
     *         13H COEF, VERIFY,
     *         21H ARGUMENTS, NO OUTPUT)

  450 M     = 1
      F(1) = 0.D0
      F(2) = 0.D0
      F(3) = 0.D0
      K(1) = 0
      K(2) = 0
                                    GO TO 350
      END
```

Algorithm 352 (Part A.1)
BOUNDS (Crude Bounds)
(Called by MFCVAL)

Comments The subroutine BOUNDS determines crude upper and lower bounds for the Kth characteristic value, K ≦ N.

Explanation of the other arguments:

APPROX the first approximation
TOLA the tolerance determined by subroutine MFCVAL
CV the 6 by N array described in subroutine MFCVAL
N variable dimension of the CV array
MM an indicator cell used to communicate unusual and error conditions to subroutine MFCVAL

The output, $a_0 < a < a_1$, is put into the common block labeled MF2.

BOUNDS calls on the subroutine:
TMOFA—referred to as Algorithm 352 (Part A.3)

```
          SUBROUTINE BOUNDS (K,APPROX,
         *                   TOLA,CV,N,MM)
C         *****************

          INTEGER
         *        K,KA,M,MM,N

          DOUBLE PRECISION
         *        A,APPROX,A0,A1,CV,DTM,
         *        D0,D1,Q,TM,TOLA

          DIMENSION
         *        CV(6,N)

          COMMON /MF1/
         *        Q,DUMMY(7)

      COMMON /MF2/
     *        A0,A,A1

      KA = 0

      IF (K.EQ.1) GO TO 20

      IF (APPROX-CV(1,K-1)) 10,10,20

   10 A0 = CV(1,K-1)+1.D0
                                    GO TO 30
   20 A0 = APPROX
   30 CALL TMOFA (A0,TM,DTM,M)

      IF (M.GT.0) GO TO 250

      D0 = -TM/DTM

      IF (D0) 100,300,50

C     A0 IS LOWER BOUND,
C     SEARCH FOR UPPER BOUND
   50 A1 = A0+D0+.1D0
      CALL TMOFA (A1,TM,DTM,M)

      IF (M.GT.0) GO TO 250

      D1 = -TM/DTM

      IF (D1) 200,350,60

   60 A0 = A1
      D0 = D1
      KA = KA+1

      IF (KA-4) 50,400,400

C     A1 IS UPPER BOUND,
C     SEARCH FOR LOWER BOUND
  100 A1 = A0
      D1 = D0
      A0 = DMAX1(A1+D1-.1D0,-2.D0*Q)

      IF (K.EQ.1) GO TO 110

      IF (A0-CV(1,K-1)) 150,150,110

  110 CALL TMOFA (A0,TM,DTM,M)

      IF (M.GT.0) GO TO 250

      D0 = -TM/DTM

      IF (D0) 120,300,200

  120 KA = KA+1

      IF (KA-4) 100,400,400

  150 KA = KA+1

      IF (KA-4) 160,400,400

  160 A0 = A1+DMAX1(TOLA,DABS(D1))
                                    GO TO 30
  200 A  = .5D0*(A0+D0+A1+D1)

      IF (A.LE.A0 .OR.
     *     A.GE.A1) A = .5D0*(A0+A1)

  250 MM = M
                                      RETURN
  300 CV(1,K) = A0
  310 CV(2,K) = 0.D0
      M       = -1
                                    GO TO 250
  350 CV(1,K) = A1
                                    GO TO 310
  400 M       = 2
                                    GO TO 250
      END
```

Algorithm 352 (Part A.2)
MFITR8 (Improves Characteristic Value)
(Called by MFCVAL)

Comments Given $a_0 < a < a_1$, where a_0 is a lower and a_1 an upper bound, the subroutine MFITR8 iterates to the characteristic value, replacing one of the bounds with a better approximation at each step. The process terminates after 40 iterations unless one of the following conditions occurs first: (1) $a - a_0 \leqq$ TOLA, (2) $a_1 - a \leqq$ TOLA, or (3) $| D(a) | <$ TOLA. See Appendix 3, method 2, of [3] for a detailed description of this process.

Explanation of output:

CV the characteristic value, a
DCV the function $D(a)$
MM an indicator cell used to communicate an error condition to subroutine MFCVAL.

MFITR8 calls on the subroutine:
TMOFA—referred to as Algorithm 352 (Part A.3)

```
      SUBROUTINE MFITR8 (TOLA,CV,DCV,MM)
C     *****************

      INTEGER
     *          M,MM,N

      DOUBLE PRECISION
     *          A,A0,A1,A2,CV,D,DCV,DTM,
     *          TM,TOLA

      LOGICAL
     *          LAST

      COMMON /MF2/
     *          A0,A,A1

      N     = 0
      LAST  = .FALSE.
   50 N     = N+1
      CALL TMOFA (A,TM,DTM,M)

      IF (M.GT.0) GO TO 400

      D     = -TM/DTM

C   IS TOLERANCE MET
      IF (N       .EQ.  40 .OR.
     *    A-A0    .LE.TOLA .OR.
     *    A1-A    .LE.TOLA .OR.
     *    DABS(D).LT.TOLA) LAST = .TRUE.

      IF (D) 110,100,120

  100 CV    = A
      DCV   = 0.D0
                                      GO TO 320

C   REPLACE UPPER BOUND BY A
  110 A1    = A
                                      GO TO 200

C   REPLACE LOWER BOUND BY A
  120 A0    = A
  200 A2    = A+D

      IF (LAST) GO TO 300
      IF (A2.GT.A0.AND.A2.LT.A1)
     *                                GO TO 250

      A     = .5D0*(A0+A1)
                                      GO TO 50
  250 A     = A2
                                      GO TO 50

  300 IF (A2.LE.A0.OR.A2.GE.A1)
     *                         GO TO 350

      CALL TMOFA (A2,TM,DTM,M)

      IF (M.GT.0) GO TO 400

      D     = -TM/DTM
      CV    = A2
  310 DCV   = D
  320 MM    = M
                                      RETURN

  350 CV    = A
                                 GO TO 310
  400 CV    = 0.D0
      DCV   = 0.D0
                                 GO TO 320
      END
```

Algorithm 352 (Part A.3)
TMOFA (Accuracy Indicator)
(Called by MFCVAL, BOUNDS, MFITR8 and COEF)

Comments The subroutine TMOFA evaluates the function $T_m(a)$ and its derivative $dT_m(a)/da$. See [3] for the definitions, theorems, and numerical methods relating to the computation of these quantities.

Explanation of the arguments:

ALFA the given argument, a
TM $T_m(a)$
DTM $dT_m(a)/da$
ND internal error indicator cell

TMOFA calls no other subprograms.

```
      SUBROUTINE TMOFA (ALFA,TM,DTM,ND)
C     ****************

      INTEGER
     *          K,KK,KT,L,MF,M0,M1,M2S,
     *          ND,TYPE

      DOUBLE PRECISION
     *          A,AA,ALFA,B,DG,DTM,DTYPE,
     *          F,FL,G,H(200),HP,Q,QINV,
     *          Q1,Q2,T,TM,TOL,TT,V

      COMMON
     *          G(200,2),DG(200,2),AA,
     *          A(3),B(3),DTYPE,QINV,Q1,
     *          Q2,T,TT,K,L,KK,KT

      COMMON /MF1/
     *          Q,TOL,TYPE,M1,M0,M2S,MF

      EQUIVALENCE
     *          (H(1),G(1,1)),(Q1,HP),
     *          (Q2,F)

      DATA      FL /1.D+30/

C   STATEMENT FUNCTION
      V(K) = (AA-DBLE(FLOAT(K))**2)/Q

      ND        = 0
      KT        = 0
      AA        = ALFA
      DTYPE     = TYPE
      QINV      = 1.D0/Q
      DO 10          L = 1,2
         DO 5        K = 1,200
            G(K,L)   = 0.D0
            DG(K,L)  = 0.D0
    5    CONTINUE
   10 CONTINUE

      IF (MOD(TYPE,2)) 20,30,20

   20 M0        = 3
                                      GO TO 40
   30 M0        = TYPE+2
   40 K         = .5D0*DSQRT(DMAX1(
     *             3.D0*Q+AA,0.D0))
      M2S       = MIN0(2*K+M0+4,
     *                   398+MOD(M0,2))
```

```
C    EVALUATION OF THE TAIL OF A
C    CONTINUED FRACTION
        A(1)      = 1.D0
        A(2)      = V(M2S+2)
        B(1)      = V(M2S)
        B(2)      = A(2)*B(1)-1.D0
        Q1        = A(2)/B(2)
        DO 50 K = 1,200
            MF    = M2S+2+2*K
            T     = V(MF)
            A(3)  = T*A(2)-A(1)
            B(3)  = T*B(2)-B(1)
            Q2    = A(3)/B(3)

            IF (DABS(Q1-Q2).LT.TOL)
     *                        GO TO 70

            Q1    = Q2
            A(1)  = A(2)
            A(2)  = A(3)
            B(1)  = B(2)
            B(2)  = B(3)

   50 CONTINUE
        KT        = 1
   70 T           = 1.D0/T
        TT        = -T*T*QINV
        L         = MF-M2S
        DO 80 K = 2,L,2
            T     = 1.D0/(V(MF-K)-T)
            TT    = T*T*(TT-QINV)
   80 CONTINUE
        KK        = M2S/2+1

        IF (KT.EQ.1) Q2 = T

        G(KK,2) = .5D0*(Q2+T)
        DG(KK,2)= TT

C    STAGE 1
        G(2,1)    = 1.D0
        DO 140    K = MO,M2S,2
            KK        = K/2+1

            IF (K.LT.5)
     *          IF (K-3) 100,110,120

            G(KK,1) = V(K-2)-1.D0/G(KK-1,1)
            DG(KK,1)= QINV+DG(KK-1,1)/
     *                G(KK-1,1)**2
                                GO TO 130
  100       G(2,1)  = V(0)
            DG(2,1) = QINV
                                GO TO 130
  110       G(2,1)  = V(1)+DTYPE-2.D0
            DG(2,1) = QINV
                                GO TO 130
  120       G(3,1)  = V(2)+(DTYPE-2.D0)/
     *                G(2,1)
            DG(3,1) = QINV+(2.D0-DTYPE)*
     *                DG(2,1)/G(2,1)**2

            IF (TYPE.EQ.2) G(2,1) = 0.D0

  130       IF (DABS(G(KK,1)).LT.1.D0)
     *                        GO TO 200

  140 CONTINUE

C    BACKTRACK
        TM        = G(KK,2)-G(KK,1)
        DTM       = DG(KK,2)-DG(KK,1)
        M1        = M2S
        KT        = M2S-MO
        DO 180    L = 2,KT,2
            K         = M2S-L
            KK        = K/2+1
            G(KK,2)   = 1.D0/(V(K)-G(KK+1,2))
            DG(KK,2)= -G(KK,2)**2*
     *                (QINV-DG(KK+1,2))

            IF (K-2) 150,150,160

  150       G(2,2)    = 2.D0*G(2,2)
            DG(2,2)   = 2.D0*DG(2,2)
  160       T         = G(KK,2)-G(KK,1)

            IF (DABS(T)-DABS(TM))
     *                      170,180,180

  170       TM        = T
            DTM       = DG(KK,2)-DG(KK,1)
            M1        = K
  180 CONTINUE
                                GO TO 320
```

```
C    STAGE 2
  200 M1          = K
      K           = M2S
      KK          = K/2+1

  210 IF (K.EQ.M1)
     *      IF (K-2) 300,300,310

      K           = K-2
      KK          = KK-1
      T           = V(K)-G(KK+1,2)

      IF (DABS(T)-1.D0) 250,220,220

  220 G(KK,2) = 1.D0/T
      DG(KK,2)= (DG(KK+1,2)-QINV)/T**2
                                GO TO 210

C    STAGE 3
  250 IF (K.EQ.M1) IF (T) 220,290,220

      HP          = DG(KK+1,2)-QINV
  260 G(KK,2)     = FL
      H(KK)       = T
      K           = K-2
      KK          = KK-1
      F           = V(K)*T-1.D0

      IF (K.EQ.M1) IF (F) 280,290,280

      IF (DABS(F)-DABS(T)) 270,280,280

  270 HP          = HP/T**2-QINV
      T           = F/T
                                GO TO 260
  280 G(KK,2)     = T/F
      DG(KK,2)= (HP-QINV*T*T)/F**2
                                GO TO 210
  290 ND          = 1
                                GO TO 320

C    CHAINING M EQUALS 2
  300 G(2,2)      = 2.D0*G(2,2)
      DG(2,2)     = 2.D0*DG(2,2)
  310 TM          = G(KK,2)-G(KK,1)
      DTM         = DG(KK,2)-DG(KK,1)
  320                                RETURN
      END
```

Algorithm 352 (Part B.1)
COEF (Coefficients)
(Called by MATH)

Comments The subroutine COEF computes the *neutral* coefficients, as defined in the *Comments* of Algorithm 352 (Part B), and returns them via common array AB. Argument M is an internal error indicator cell. For details of the method used, see Appendix 6 of [3]. COEF calls on the subroutine: TMOFA—referred to as Algorithm 352 (Part A.3)

```
      SUBROUTINE COEF (M)
C     ***************

      INTEGER
     *        K,KA,KB,KK,M,MF,ML,MM,
     *        MO,M1,M2S,TYPE

      DOUBLE PRECISION
     *        A,AB,FL,G,H(200),Q,T,
     *        TOL,V,V2

      COMMON
     *        G(200,2),DUM1(800),A,T,K,
     *        KA,KB,KK,MM,ML,AB(200)

      COMMON /MF1/
     *        Q,TOL,TYPE,M1,MO,M2S,MF

      EQUIVALENCE
     *        (H(1),G(1,1))

      DATA    FL,V2/1.D+30,1.D-15/
```

```
C    STATEMENT FUNCTION
       V(K) = (A-DBLE(FLOAT(K))**2)/Q

       CALL TMOFA (A,T,T,M)

       IF (M.NE.0) GO TO 300

       DO 60    K = 1,200
           AB(K)    = 0.D0
   60 CONTINUE
       KA       = M1-M0+2
       DO 90    K = 2,KA,2
           KK       = (M1-K)/2+1

           IF (K-2) 70,70,80

   70      AB(KK)   = 1.DC
                                      GO TO 90
   80      AB(KK)   = AB(KK+1)/G(KK+1,1)
   90 CONTINUE
       KA       = 0
       DO 130   K = M1,M2S,2
           KK       = K/2+1
           ML       = K
           IF (G(KK,2).EQ.FL) GO TO 100

           AB(KK)   = AB(KK-1)*G(KK,2)
                                     GO TO 110
  100      T        = AB(KK-2)

           IF (K.EQ.4.AND.M1.EQ.2) T = T+T

           AB(KK)   = T/(V(K-2)*H(KK)-1.D0)

  110      IF (DABS(AB(KK)).GE.1.D-17)
     *                          KA = 0

           IF (KA.EQ.5) GO TO 260

           KA       = KA+1
  130 CONTINUE
       T        = DLOG(DABS(AB(KK))/V2)/
     *            DLOG(1.D0/DABS(G(KK,2)))
       KA       = 2*IDINT(T)
       ML       = KA+2+M2S

       IF (ML.GT.399) GO TO 400

       KB       = KA+2+MF
       T        = 1.D0/V(KB)
       KK       = MF-M2S
       DO 150   K = 2,KK,2
           T        = 1.D0/(V(KB-K)-T)
  150 CONTINUE
       KK       = ML/2+1
       G(KK,2) = T
       DO 200   K = 2,KA,2
           KK       = (ML-K)/2+1
           G(KK,2) = 1.D0/(V(ML-K)-
     *            G(KK+1,2))
  200 CONTINUE
       KA       = M2S+2
       DO 250   K = KA,ML,2
           KK       = K/2+1
           AB(KK)   = AB(KK-1)*G(KK,2)
  250 CONTINUE

C    NEUTRAL NORMALIZATION
  260 T        = AB(1)
       MM       = MOD(TYPE,2)
       KA       = MM+2
       DO 280   K = KA,ML,2
           KK       = K/2+1

           IF (DABS(T)-DABS(AB(KK)))
     *                          270,280,280

  270      T        = AB(KK)
           MM       = K
  280 CONTINUE
       DO 290   K = 1,KK
           AB(K)    = AB(K)/T
  290 CONTINUE
  300                                        RETURN
  400 M        = -1
                                        GO TO 300
       END
```

Algorithm 352 (Part B.2)
SUM (Series Evaluation)
(Called by MATH)

Comments The subroutine SUM performs the summation, truncating the series when the magnitude of two successive terms, relative to the magnitude of the largest term, is less than or equal to 10^{-13}.

If the user is willing to accept reduced accuracy, he may save some computing time by making this tolerance larger. On the other hand, however, a smaller tolerance will not necessarily increase the accuracy, since on a machine using 16-digit arithmetic the sum will be, at best, good to 16 digits.

The particular series being evaluated is determined by the arguments SOL and FNC within subroutine MATH and communicated to this subroutine via argument DUM.

Output is returned via common: variables F, DMAX, DLAST, KMAX, and KLAST.

SUM calls on one of the functions of Algorithm 352 (Part B.2.1).

```
      SUBROUTINE SUM (DUM)
C     **************

      INTEGER
     *           K,KLAST,KMAX,L,S

      DOUBLE PRECISION
     *           DLAST,DMAX,DUM,F,T

      COMMON
     *           DUM1(1006),S,L,DUM2(6),F,
     *           DMAX,DLAST,KMAX,KLAST,T

      K      = 0
      F      = DUM(0)
      DMAX   = F
      T      = DABS(F)
      KMAX   = 0
      DO 30 KLAST = 1,L
          DLAST = DUM(KLAST)
          F      = F+DLAST

          IF (T-DABS(DLAST)) 10,10,20

   10     DMAX   = DLAST
          T      = DABS(DMAX)
          KMAX   = KLAST

   20     IF (KLAST.LE.S) GO TO 30

          IF (DABS(DLAST)/T.GT.1.D-13)
     *                     K = 0

          K      = K+1

          IF (K.EQ.3) GO TO 40

   30 CONTINUE
      KLAST = L
   40                                  RETURN
      END
```

Algorithm 352 (Part C)
BESSEL (Bessel Functions)[2]
(Called by MATH)

Comments The subroutine BESSEL evaluates Bessel functions of the first or second kind, according as the argument SOL = 1 or 2, of orders 0, 1, $\cdots$, n and argument u, both of which must be nonnegative. Functions of order zero and one are always evaluated, regardless of the value of n. Results are returned via array JY, with element JY(K) containing the function of order K-1.

It should be noted that for SOL = 2 and $u = 0$, a large negative constant (-10^{37}) is returned as the function value for all orders and no warning is given.

Different methods of computation are used for $J_0(u)$, $J_1(u)$, $Y_0(u)$, and $Y_1(u)$, depending upon whether $u < 8$, or not. (See subroutines J0J1, Y0Y1, and LUKE for details.) The $J_n(u)$, $n = 2, 3, \cdots, m$, are computed by means of a continued fraction (see subroutine JNS), whereas the $Y_n(u)$ for corresponding orders are calculated directly from the recurrence relation:

$$Y_{n+1}(u) = \frac{2n}{u} Y_n(u) - Y_{n-1}(u)$$

BESSEL calls on the subroutines:

J0J1—referred to as Algorithm 352 (Part C.1)

Y0Y1—referred to as Algorithm 352 (Part C.2)

LUKE—referred to as Algorithm 352 (Part C.3)

JNS—referred to as Algorithm 352 (Part C.4)

```
      SUBROUTINE BESSEL (SOL,U,JY,N)
C     *****************

      INTEGER
     *           N,NN,SOL

      DOUBLE PRECISION
     *           JY(250),U

      NN = MIN0(N,249)

      IF (U.EQ.0.D0.AND.SOL.EQ.2)
     *                    GO TO 80

      IF (U.GE.8.D0) GO TO 30

      GO TO (10,20), SOL
   10 CALL J0J1 (U,JY)
                                      GO TO 40
   20 CALL Y0Y1 (U,JY)
                                      GO TO 40
   30 CALL LUKE (U,SOL,JY)

   40 IF (N.LT.2) GO TO 100

      GO TO (50,60), SOL
   50 CALL JNS (JY,U,NN)
                                     GO TO 100

C    RECURRENCE FORMULA
   60 DO 70     K = 2,NN
         JY(K+1) = 2.D0*
     *             DBLE(FLOAT(K-1))*
     *             JY(K)/U-JY(K-1)
   70 CONTINUE
                                     GO TO 100
   80 NN = NN+1
      DO 90   K = 1,NN
         JY(K) = -1.D+37
   90 CONTINUE
  100                                   RETURN
      END
```

* This subroutine (together with its subsidiary routines) may be removed in toto, with no changes, and used independently as a Bessel function algorithm. The results are good to 14 significant digits or decimal places, whichever is least accurate, with an error of no more than one unit in the last digit or place.

Algorithm 352 (Part C.1)
J0J1 (First Kind)
(Called by BESSEL)

Comments The subroutine J0J1 computes the Bessel functions of the first kind, $J_0(x)$ and $J_1(x)$, for $x < 8$. This is done by evaluating formula 9.1.10 of [1]. The results are returned via array J.

J0J1 calls no other subprograms.

```
      SUBROUTINE J0J1 (X,J)
C     ***************

      DOUBLE PRECISION
     *           J(2),T(5),X

      COMMON
     *           DUM(1014),T

     T(1) = X/2.D0
     J(1) = 1.D0
     J(2) = T(1)
     T(2) = -T(1)**2
      T(3) = 1.D0
      T(4) = 1.D0

   10 T(4) = T(4)*T(2)/T(3)**2
      J(1) = J(1)+T(4)
      T(5) = T(4)*T(1)/(T(3)+1.D0)
      J(2) = J(2)+T(5)

      IF (DMAX1(DABS(T(4)),DABS(T(5)))
     *    .LT.1.D-15) RETURN

      T(3) = T(3)+1.D0
                                  GO TO 10
      END
```

Algorithm 352 (Part C.2)
Y0Y1 (Second Kind)
(Called by BESSEL)

Comments The subroutine Y0Y1 computes the Bessel functions of the second kind, $Y_0(x)$ and $Y_1(x)$, for $x < 8$. This is done by evaluating formulas 9.1.13 and 9.1.11 of [1]. The results are returned via array Y.

Y0Y1 calls no other subprograms.

```
      SUBROUTINE Y0Y1 (X,Y)
C     ***************

      DOUBLE PRECISION
     *           T(10),X,Y(2)

      COMMON
     *           DUM(1014),T

      T(1) = X/2.D0
      T(2) = -T(1)**2
      Y(1) = 1.D0
      Y(2) = T(1)
      T(7) = 0.D0
      T(10)= -T(1)
      T(3) = 0.D0
      T(4) = 0.D0
      T(5) = 1.D0

   10 T(3) = T(3)+1.D0
      T(4) = T(4)+1.D0/T(3)
      T(5) = T(5)*T(2)/T(3)**2
      Y(1) = Y(1)+T(5)
      T(6) = -T(5)*T(4)
      T(7) = T(7)+T(6)
      T(8) = T(5)*T(1)/(T(3)+1.D0)
      Y(2) = Y(2)+T(8)
      T(9) = -T(8)*(2.D0*T(4)+
     *       1.D0/(T(3)+1.D0))
      T(10)= T(10)+T(9)
```

```
       IF (DMAX1(DABS(T(6)),DABS(T(9)))
     *     .GE.1.D-15) GO TO 10

       T(2) = .5772156649015328606D0+
     *        DLOG(T(1))
       Y(1) = .6366197723675813430D0*
     *        (Y(1)*T(2)+T(7))
       Y(2) = .6366197723675813430D0*
     *        (Y(2)*T(2)-1.D0/X)+T(10)/
     *        3.1415926535897932D0
                                RETURN
       END
```

Algorithm 352 (Part C.3)
LUKE
(Called by BESSEL)

Comments The subroutine LUKE evaluates Bessel functions of order zero and one, of the first or second kind, according as the argument KIND = 1 or 2, for $u \geqq 8$. The results are returned via the 2-element array JY.

The Bessel function of the third kind (Hankel function), $H_\nu^{(1)}(u) = J_\nu(u) + iY_\nu(u)$, can be expressed in terms of the Chebyshev polynomials, $T_n{}^*(x)$, as follows:

$$H_\nu^{(1)}(u) = \left(\frac{2}{\pi u}\right)^{\frac{1}{2}} e^{i\left(u-\frac{\nu\pi}{2}-\frac{\pi}{4}\right)} \cdot \sum_{k=0}^{\infty} (\alpha_k^{(\nu)} + i\beta_k^{(\nu)}) T_k{}^*(R/u). \tag{9}$$

We now define $\alpha_k^{(0)} = A_{k+1}$, $\beta_\nu^{(0)} = B_{k+1}$, $\alpha_k^{(1)} = C_{k+1}$, $\beta_k^{(1)} = D_{k+1}$, $x = R/u$, and $T_k{}^*(x) = G_{k+1}(x)$. The recurrence relations for the $G_k(x)$ are as follows:

$$G_1(x) = 1, \qquad G_2(x) = 2x - 1,$$

$$G_k(x) = (4x-2)\,G_{k-1}(x) - G_{k-2}(x), \qquad k \geqq 3.$$

If we let $\nu = 0$ and make other appropriate substitutions in (9), while remembering that $e^{i\theta} = \cos\theta + i\sin\theta$, we can separate the real and imaginary parts and get the following relations:

$$J_0(u) = \left(\frac{2}{\pi u}\right)^{\frac{1}{2}} \cdot \left[\cos\theta \sum_{k=1}^{\infty} A_k\,G_k(x) - \sin\theta \sum_{k=1}^{\infty} B_k\,G_k(x)\right],$$

$$Y_0(u) = \left(\frac{2}{\pi u}\right)^{\frac{1}{2}} \cdot \left[\cos\theta \sum_{k=1}^{\infty} B_k\,G_k(x) + \sin\theta \sum_{k=1}^{\infty} A_k\,G_k(x)\right],$$

where $\theta = u - \pi/4$.

Notice that if $\nu = 1$ in (9), then θ is replaced by $\theta - \pi/2$. Also, $\cos(\theta-\pi/2) = \sin\theta$ and $\sin(\theta-\pi/2) = -\cos\theta$. Therefore, proceeding as before, we get

$$J_1(u) = \left(\frac{2}{\pi u}\right)^{\frac{1}{2}} \cdot \left[\sin\theta \sum_{k=1}^{\infty} C_k\,G_k(x) + \cos\theta \sum_{k=1}^{\infty} D_k\,G_k(x)\right],$$

$$Y_1(u) = \left(\frac{2}{\pi u}\right)^{\frac{1}{2}} \cdot \left[\sin\theta \sum_{k=1}^{\infty} D_k\,G_k(x) - \cos\theta \sum_{k=1}^{\infty} C_k\,G_k(x)\right].$$

The coefficients A, B, C, and D have been computed for $R = 8$ in eq. (9) and are guaranteed to the number of digits given.

LUKE calls no other subprograms.

```
       SUBROUTINE LUKE (U,KIND,JY)
C      ***************

       INTEGER
     *         K,KIND

       DOUBLE PRECISION
     *         A(19),B(19),CS,C(19),
     *         D(19),G(3),JY(2),R(2),
     *         S(2),SN,T,U,X

       COMMON
     *         DUM(1014),R,S,G,X,T,SN,CS

C    WARNING - THE FOLLOWING DATA
C    STATEMENTS ARE NOT IN ASA
C    STANDARD FORTRAN


       DATA A /
     *          .99959506476867287416D0,
     *         -.5380795613960691 3D-3,
     *         -.1317967712336157 0D-3,
     *          .151422497048644D-5,
     *          .15846861792063D-6,
     *         -.856069553946D-8,
     *         -.29572343355D-9,
     *          .6573556254D-10,
     *         -.223749703D-11,
     *         -.44821140D-12,
     *          .6954827D-13,
     *         -.151340D-14,
     *         -.92422D-15,
     *          .15558D-15,
     *         -.476D-17,
     *         -.274D-17,
     *          .61D-18,
     *         -.4D-19,
     *         -.1D-19/

       DATA B /
     *         -.77693556942053213 6D-2,
     *         -.77480323096544767 0D-2,
     *          .253654116543079 6D-4,
     *          .394273598399711D-5,
     *         -.10723498299129D-6,
     *         -.721389799328D-8,
     *          .73764602893D-9,
     *          .150687811D-11,
     *         -.574589537D-11,
     *          .45996574D-12,
     *          .2270323D-13,
     *         -.887890D-14,
     *          .74497D-15,
     *          .5847D-16,
     *         -.2410D-16,
     *          .265D-17,
     *          .13D-18,
     *         -.10D-18,
     *          .2D-19/
```

```
 DATA C /
*            1.00067753586591346234D0,
*             .9010072519590818 3D-3,
*             .22172434918599454D-3,
*            -.196575946319104D-5,
*            -.208895311432 70D-6,
*             .1028144350894D-7,
*             .37597054789D-9,
*            -.7638891358D-10,
*             .238734670D-11,
*             .51825489D-12,
*            -.7693969D-13,
*             .144008D-14,
*             .103294D-14,
*            -.16821D-15,
*             .459D-17,
*             .302D-17,
*            -.65D-18,
*             .4D-19,
*             .1D-19/

 DATA D /
*             .23376829986285803 28D-1,
*             .23346801223545575 33D-1,
*            -.35760105909013 82D-4,
*            -.56086314949262 7D-5,
*             .1327389408434 0D-6,
*            ·.916975845066D-8,
*            -.86838880371D-9,
*            -.378073005D-11,
*             .663145586D-11,
*            -.50584390D-12,
*            -.2720782D-13,
*             .985381D-14,
*            -.79398D-15,
*            -.6757D-16,
*             .2625D-16,
*            -.280D-17,
*            -.15D-18,
*             .10D-18,
*            -.2D-19/

 X     = 8.D0/U
 G(1)  = 1.D0
 G(2)  = 2.D0*X-1.D0
 R(1)  = A(1)+A(2)*G(2)
 S(1)  = B(1)+B(2)*G(2)
 R(2)  = C(1)+C(2)*G(2)
 S(2)  = D(1)+D(2)*G(2)
 DO 10 K = 3,19
    G(3)  = (4.D0*X-2.D0)*G(2)-G(1)
    R(1)  = R(1)+A(K)*G(3)
    S(1)  = S(1)+B(K)*G(3)
    R(2)  = R(2)+C(K)*G(3)
    S(2)  = S(2)+D(K)*G(3)
    G(1)  = G(2)

       G(2) = G(3)
10 CONTINUE

   T  = .7978845608028654D0/DSQRT(U)
   SN = DSIN(U-.785398163397448 3D0)
   CS = DCOS(U-.785398163397448 3D0)

   GO TO (20,30), KIND
20 JY(1)  = T*(R(1)*CS-S(1)*SN)
   JY(2)  = T*(R(2)*SN+S(2)*CS)
                              GO TO 40
30 JY(1)  = T*(S(1)*CS+R(1)*SN)
   JY(2)  = T*(S(2)*SN-R(2)*CS)
40                               RETURN
   END
```

Algorithm 352 (Part C.4)
JNS
 (Called by BESSEL)

Comments The subroutine JNS evaluates Bessel functions of the first kind, of orders $n = 2, 3, \cdots, m$, for argument u, given $J_0(u)$ and $J_1(u)$. From the definition $G_n = J_n(u)/J_{n-1}(u)$ and the recurrence relation,

$$J_{n+1}(u) = (2n/u)\, J_n(u) - J_{n-1}(u),$$

we can derive the following equation:

$$G_n = \cfrac{1}{\cfrac{2n}{u} - G_{n+1}}. \tag{10}$$

Since G_{n+1} is of the same form as G_n, we can continue the process and obtain the continued fraction,

$$G_n = \frac{1}{\dfrac{2n}{u}} - \frac{1}{\dfrac{2(n+1)}{u}} - \cdots - \cfrac{1}{\cfrac{2(n+k)}{u} - G_{n+k+1}}. \tag{11}$$

G_m is evaluated using (11), then the other G_n are computed from (10) for $n = m - 1, m - 2, \cdots, 2$. Finally, the J_n are evaluated in a forward direction from $J_n = G_n J_{n-1}$ and returned via argument array JJ. See [2] for a more detailed treatment of this process.

JNS calls no other subprograms.

```
      SUBROUTINE JNS (JJ,U,M)
C     **************

      INTEGER
     *          K,KA,KK,M

      DOUBLE PRECISION
     *          A,B,D(2),DM,G(249),
     *          JJ(250),P(3),Q(3),U

      EQUIVALENCE
     *          (A,G),(D,G(2)),
     *          (P,G(4)),(Q,G(7)),
     *          (DM,G(10)),(B,G(11))

      COMMON
     *          DUM(1014),G,M,K,KK,KA

      DM    = 2*M
      P(1)  = 0.D0
      Q(1)  = 1.D0
      P(2)  = 1.D0
      Q(2)  = DM/U
      D(1)  = P(2)/Q(2)
      A     = 2.D0


   10 B     = (DM+A)/U
      P(3)  = B*P(2)-P(1)
      Q(3)  = B*Q(2)-Q(1)
      D(2)  = P(3)/Q(3)

      IF (DABS(D(1)-D(2))
     *     .LT.1.D-15) GO TO 20

      P(1)  = P(2)
      P(2)  = P(3)
      Q(1)  = Q(2)
      Q(2)  = Q(3)
      D(1)  = D(2)
      A     = A+2.D0
                                GO TO 10
   20 G(M)  = D(2)
      KA    = M-2
      DO 30  K = 1,KA
         KK     = M-K
         A      = 2*KK
         G(KK)  = U/(A-U*G(KK+1))

         IF (G(KK).EQ.0.D0)
     *                G(KK) = 1.D-35

   30 CONTINUE
      DO 40    K = 2,M
         JJ(K+1) = G(K)*JJ(K)
   40 CONTINUE
                                 RETURN
      END
```

Algorithm 352 (Part B.2.1)
DS, DC, DDS, DDC, PS, PC, DPS, DPC
(Called by MATH via SUM)

Comments The following collection of function subprograms is utilized by SUM to evaluate the *k*th term ($k = 0, 1, \cdots$) of one of the following: eq. (2), (3), (5), (6), or their derivatives.

DS and DC call on functions FJ and FY.

DDS and DDC call on functions FJ, FY, DJ and DY.

PS, PC, DPS, and DPC call no other subprograms.

```
      DOUBLE PRECISION FUNCTION DS(KK)
C     ************************
       INTEGER
     *          K,KK,N,N1,N2,P,S

      DOUBLE PRECISION
     *          AB,FJ,FY

      COMMON
     *          DUM1(1004),N,P,S,DUM2(17),
     *          K,N1,N2,DUM3(583),AB(200)

C   EVALUATES ONE TERM OF THE RADIAL
C   SOLUTION, ASSOCIATED WITH B(Q)
      K  = KK
      N1 = K-S
      N2 = K+S+P
      DS = AB(K+1)*(FJ(N1)*FY(N2)-
     *             FJ(N2)*FY(N1))

      IF (MOD(K+N,2).NE.0) DS = -DS

                                     RETURN
      END


      DOUBLE PRECISION FUNCTION DC(KK)
C     ************************

      INTEGER
     *          K,KK,N,N1,N2,P,S

      DOUBLE PRECISION
     *          AB,FJ,FY

      COMMON
     *          DUM1(1004),N,P,S,DUM2(17),
     *          K,N1,N2,DUM3(583),AB(200)

C   EVALUATES ONE TERM OF THE RADIAL
C   SOLUTION, ASSOCIATED WITH A(Q)
      K  = KK
      N1 = K-S
      N2 = K+S+P
      DC = AB(K+1)*(FJ(N1)*FY(N2)+
     *             FJ(N2)*FY(N1))

      IF (MOD(K+N,2).NE.0) DC = -DC

      IF (S+P.EQ.0) DC = .5D0*DC

                                     RETURN
      END


      DOUBLE PRECISION FUNCTION DDS(KK)
C     ************************

      INTEGER
     *          K,KK,N,N1,N2,P,S

      DOUBLE PRECISION
     *          AB,DJ,DY,FJ,FY,U1,U2

      COMMON
     *          DUM1(1000),U1,U2,N,P,S,
     *          DUM2(17),K,N1,N2,
     *          DUM3(583),AB(200)

C   EVALUATES ONE TERM OF THE DERIVATIVE
C   OF THE RADIAL SOLUTION,
C   ASSOCIATED WITH B(Q)
      K   = KK
      N1  = K-S
      N2  = K+S+P
      DDS = AB(K+1)*(U2*(FJ(N1)*DY(N2)-
     *        FJ(N2)*DY(N1))-U1*(FY(N2)*
     *        DJ(N1)-FY(N1)*DJ(N2)))

      IF (MOD(K+N,2).NE.0) DDS = -DDS

                                     RETURN
      END


      DOUBLE PRECISION FUNCTION DDC(KK)
C     ************************

      INTEGER
     *          K,KK,N,N1,N2,P,S

      DOUBLE PRECISION
     *          AB,DJ,DY,FJ,FY,U1,U2

      COMMON
     *          DUM1(1000),U1,U2,N,P,S,
     *          DUM2(17),K,N1,N2,
     *          DUM3(583),AB(200)

C   EVALUATES ONE TERM OF THE DERIVATIVE
C   OF THE RADIAL SOLUTION,
C   ASSOCIATED WITH A(Q)
      K   = KK
      N1  = K-S
      N2  = K+S+P
      DDC = AB(K+1)*(U2*(FJ(N1)*DY(N2)+
     *        FJ(N2)*DY(N1))-U1*(FY(N2)*
     *        DJ(N1)+FY(N1)*DJ(N2)))

      IF (MOD(K+N,2).NE.0) DDC = -DDC

      IF (S+P.EQ.0) DDC = .5D0*DDC

                                     RETURN
      END


      DOUBLE PRECISION FUNCTION PS(K)
C     ************************

      INTEGER
     *          K,P

      DOUBLE PRECISION
     *          AB,X

      COMMON
     *          DUM1(1005),P,DUM2(2),X,
     *          DUM3(600),AB(200)

C   EVALUATES ONE TERM OF THE ODD
C   PERIODIC SOLUTION
      PS = AB(K+1)*
     *     DSIN(DBLE(FLOAT(2*K+P))*X)
                                     RETURN
      END


      DOUBLE PRECISION FUNCTION PC(K)
C     ************************

      INTEGER
     *          K,P

      DOUBLE PRECISION
     *          AB,X

      COMMON
     *          DUM1(1005),P,DUM2(2),X,
     *          DUM3(600),AB(200)

C   EVALUATES ONE TERM OF THE EVEN
C   PERIODIC SOLUTION
      PC = AB(K+1)*
     *     DCOS(DBLE(FLOAT(2*K+P))*X)
                                     RETURN
      END


      DOUBLE PRECISION FUNCTION DPS(K)
C     ************************

      INTEGER
     *          K,P
```

```
      DOUBLE PRECISION
     *          AB,T,X

      COMMON
     *          DUM1(1005),P,DUM2(2),X,
     *          DUM3(14),T,DUM4(584),
     *          AB(200)

C   EVALUATES ONE TERM OF THE DERIVATIVE
C   OF THE ODD PERIODIC SOLUTION
      T   = 2*K+P
      DPS = AB(K+1)*T*DCOS(T*X)
                                        RETURN
      END


      DOUBLE PRECISION FUNCTION DPC(K)
C     *************************

      INTEGER
     *          K,P

      DOUBLE PRECISION
     *          AB,T,X

      COMMON
     *          DUM1(1005),P,DUM2(2),X,
     *          DUM3(14),T,DUM4(584),
     *          AB(200)

C   EVALUATES ONE TERM OF THE DERIVATIVE
C   OF THE EVEN PERIODIC SOLUTION
      T   = 2*K+P
      DPC = -AB(K+1)*T*DSIN(T*X)
                                        RETURN
      END
```

Algorithm 352 (Part B.2.2)
FJ, FY, DJ, DY (Bessel Functions and Derivatives)
(Called by DS, DC, DDS, DDC)

Comments The following collection of functionsubprograms produces Bessel functions or their derivatives for integer order n, n being positive or negative. This is accomplished by using the already computed functions of nonnegative order (Algorithm 352 (Part C)) and substituting them in one of the following formulas:

$$J_{-n}(u) = (-1)^n J_n(u),$$

$$Y_{-n}(u) = (-1)^n Y_n(u),$$

$$J_n'(u) = \frac{n}{u} J_n(u) - J_{n+1}(u),$$

$$Y_n'(u) = Y_{n-1}(u) - \frac{n}{u} Y_n(u),$$

whichever is appropriate.

DJ calls on function FJ.

DY calls on function FY.

FJ and FY call no other subprograms.

```
      DOUBLE PRECISION FUNCTION FJ(N)
C     *************************

      INTEGER
     *          K,N

      DOUBLE PRECISION
     *          J

      COMMON
     *          J(250),DUM(527),K

C   PRODUCES BESSEL FUNCTIONS
C   OF THE FIRST KIND
      K   = IABS(N)

      IF (K.GE.250) GO TO 20

      FJ = J(K+1)

      IF (MOD(N,2).LT.0) FJ = -FJ

   10                                   RETURN

   20 FJ = 0.D0
      WRITE (6,99) N
   99 FORMAT(2H0JI3,7H NEEDED)
                                      GO TO 10
      END


      DOUBLE PRECISION FUNCTION FY(N)
C     *************************

      INTEGER
     *          K,N

      DOUBLE PRECISION
     *          Y

      COMMON
     *          DUM1(500),Y(250),DUM2(27),K

C   PRODUCES BESSEL FUNCTIONS
C   OF THE SECOND KIND
      K   = IABS(N)

      IF (K.GE.250) GO TO 20

      FY = Y(K+1)

      IF (MOD(N,2).LT.0) FY = -FY

   10                                   RETURN

   20 FY = 0.D0
      WRITE (6,99) N
   99 FORMAT(2H0YI3,7H NEEDED)
                                      GO TO 10
      END


      DOUBLE PRECISION FUNCTION DJ(N)
C     *************************

      INTEGER
     *          N

      DOUBLE PRECISION
     *          FJ,FN,U1

      COMMON
     *          DUM1(1000),U1,DUM2(26),FN

C.  DERIVATIVES OF BESSEL FUNCTIONS
C   OF THE FIRST KIND

      FN = N

      IF (N-249) 10,20,40

   10 DJ = FN*FJ(N)/U1-FJ(N+1)
                                      GO TO 30
   20 DJ = FJ(N-1)-FN*FJ(N)/U1
   30                                   RETURN

   40 DJ = 0.D0
      WRITE (6,99) N
   99 FORMAT(3H0J@I3,7H NEEDED)
                                      GO TO 30
      END


      DOUBLE PRECISION FUNCTION DY(N)
C     *************************

      INTEGER
     *          N

      DOUBLE PRECISION
     *          FN,FY,U2

      COMMON
     *          DUM1(1002),U2,DUM2(24),FN

C   DERIVATIVES OF BESSEL FUNCTIONS
C   OF THE SECOND KIND

      IF (N.GE.250) GO TO 20
      FN = N
      DY = FY(N-1)-FN*FY(N)/U2
   10                                      RETURN

   20 DY = 0.D0
      WRITE (6,99) N
   99 FORMAT(3H0Y@I3,7H NEEDED)
                                         GO TO 10
      END
```

REMARK ON ALGORITHM 352 [S22] CHARACTERISTIC VALUES AND ASSOCIATED SOLUTIONS OF MATHIEU'S DIFFERENTIAL EQUATION [D. S. Clemm, *Comm. ACM 12* (July 1969), 399–407]

Arthur H. J. Sale (Recd. 4 May 1970 and 28 May 1970)
University of Sydney, Sydney, NSW, Australia

KEY WORDS AND PHRASES: Mathieu's differential equation, Mathieu function, characteristic value, periodic solution, radial solution
CR CATEGORIES: 5.12

This algorithm contains a number of syntactically incorrect *FORMAT* statements: labeled 901, 911, 921, 941, and 951 in subroutine *MFCVAL*, and 99 in the functions *FJ*, *FY*, *DJ*, and *DY*. The error consists of omitting a comma separating the Hollerith field descriptor and the integer field descriptor, as required by Sections 7.2.3 and 7.2.3.2 of the Fortran standard [1, 2]. In all cases this may be corrected by inserting a comma immediately preceding the field descriptor *I*3 in these statements.

It has also been pointed out by the referee and the Algorithms Editor that the two *FORMAT* statements in functions *DJ* and *DY* contain a character not in the standard Fortran character set. The standard is somewhat ambiguous on this point: any representable character is permitted in a Hollerith constant in a *CALL* or a *DATA* statement, and also in data to be read in with an *Aw* field descriptor (Sections 4.2.6, 5.1.1.6), but since Hollerith field descriptors are not Hollerith constants, it must be presumed that the prohibition of Section 3.1 applies. The "at" symbol (@) in these two statements should therefore be replaced by a blank or some other character in the standard set.

There is another, more serious, error: subroutines *BOUNDS* and *MFITR*8 both reference a named common block which is not referenced by the routine that calls them (*MFCVAL*). According to Section 10.2.5 of the standard, the contents of this block will therefore become undefined at the moment either of these two routines executes a *RETURN*, unless this common block is referenced by a routine which is directly or indirectly calling *MFCVAL*. This undefinition permits named common blocks to be overlaid, and since it is not the author's intention to allow this block to become undefined, the following two statements should be added to *MFCVAL* immediately following the existing *DOUBLE PRECISION* and *COMMON* statements respectively:

```
DOUBLE PRECISION FILL(3)
COMMON /MF2/ FILL
```

References:

1. ANSI Standard Fortran ANSI (USASI) X3.9–1966. American National Standards Institute, New York, 1966.
2. FORTRAN vs Basic FORTRAN. *Comm. ACM 7* (Oct. 1964), 591–625.

Remark on:
Algorithm 352 [S22]
Characteristic Values and Associated Solutions of Mathieu's Differential Equation
[Donald S. Clemm, *Comm. ACM 12* (July 1969), 399–407]

Michael J. Frisch [Recd. 27 Jan. 1971]
University Computer Center, University of Minnesota, Minneapolis, MN 55455

Key Words and Phrases: ANSI Fortran standard
CR Categories: 4.0, 4.22

The following items were found during compilation of the algorithms written in Fortran published to date in Communications. The MNF compiler written at the University of Minnesota for CDC 6000 Series machines by Lawrence A. Liddiard and E. James Mundstock was used to check the validity of the algorithms.

Algorithm 352 does not conform to the standard in subroutine *MATH* which calls subroutine *SUM* with arguments that were in an *EXTERNAL* statement but not in a type statement. The dummy argument in subroutine *SUM* has type *DOUBLE PRECISION* so a statement *DOUBLE PRECISION DS, DC, DDS, DDC, PS, PC, DPS, DPC* should be inserted before the *EXTERNAL* statement in subroutine *MATH* (Section 8.4.2).

In subroutine *JNS*, the dummy argument *M* is also in blank common, contrary to 7.2.1.3. In the same subroutine, arrays *D*, *G*, *P*, and *Q* are referenced by array name instead of array element name as required in Section 7.2.1.4. The statement should be:

EQUIVALENCE (*A*,*G*(1)), (*D*(1),*G*(2)), (*P*(1),*G*(4)), (*Q*(1), *G*(7)), (*DM*,*G*(10)), (*B*,*G*(11)).

ALGORITHM 353
FILON QUADRATURE [D1]

STEPHEN M. CHASE AND LLOYD D. FOSDICK (Recd. 7 July 1967 and 6 Jan. 1969)
Department of Computer Science, University of Illinois, Urbana, IL 61820

KEY WORDS AND PHRASES: quadrature, Filon quadrature, integration, Filon integration, Fourier coefficients, Fourier series
CR CATEGORIES: 5.16

comment FSER1 evaluates the integrals

$$C = \int_0^1 F(X) \cos (M\pi X)\, dX, \qquad S = \int_0^1 F(X) \sin (M\pi X)\, dX$$

using the Filon quadrature algorithm. The user may request an evaluation of C only, S only, or both C and S. FSER1 contains an automatic error-control feature which selects an integration step size on the basis of an error parameter supplied by the user. The Filon quadrature formulas, truncation error, rounding error, and automatic error control are described in a companion paper [1] by the authors.

The calling parameters for this subroutine are defined as follows. F is the name of a FUNCTION subprogram F(X), supplied by the user, which evaluates $F(X)$ appearing in the integrand. EPS is the name for ϵ appearing in inequalities (45) and (46) of [1]. It is used in the error control portion of the algorithm. The error in the computed values of C and S is related to ϵ by the inequality (76) given in [1]. The user must assign a value to EPS before calling FSER1. MAX specifies the maximum number of halvings of the step size that are allowed. The minimum step size, h in equation (16) of [1], is $2^{-\mathrm{MAX}}$. The user must assign a value to MAX before calling FSER1. M is the parameter appearing in the argument $M\pi X$ of the cosine and sine functions. The user must assign a value to M before calling FSER1. C is the value of the cosine integral determined by FSER1. S is the value of the sine integral determined by FSER1. LC is used on entry as a signal that the user does want C evaluated (LC = 1) or does not want C evaluated (LC = 0). It is used on exit to report the value of h used by the subroutine to evaluate C, this value being $2^{-\mathrm{LC}}$. The user must assign a value of 1 or 0 to LC before calling FSER1, and if LC = 1 on entry, then the subroutine will assign a new value to LC related to the step size by $2^{-\mathrm{LC}}$. LS is used on entry as a signal that the user does want S evaluated (LS = 1) or does not want S evaluated (LS = 0). It is used on exit to report the value of h used by the subroutine to evaluate S, this value being $2^{-\mathrm{LS}}$. The user must assign a value of 1 or 0 to LS before calling FSER1, and if LS = 1 on entry, then the subroutine will assign a new value to LS related to the step size by $2^{-\mathrm{LS}}$.

FSER1 calls a subroutine ENDT1 which is also listed below. The purpose of ENDT1 is to perform the end test described by inequalities (45) and (46) of [1].

REFERENCES:

1. FOSDICK, LLOYD D., AND CHASE, STEPHEN M. An algorithm for Filon quadrature. *Comm. ACM 12* (Aug. 1969), 453–457.

```
      SUBROUTINE FSER1(F,EPS,MAX,M, C, S, LC, LS)
      PI = 3.1415926535898
      XM = M
C F1 = COS(M*PI) TEMPORARY.
      F1 = 1 - 2 * ( M- (M/2 ) * 2 )
      F0 = F(0.0)
      F1 = F(1.0) * F1
C 'CIR' WILL BE USED THROUGHOUT THESE COMMENTS TO STAND FOR 'SIN' OR
C 'COS' WHEREVER THOSE TWO SYMBOLS MAY OCCUR.
C NOW DEFINE SUMCIR OF THE ENDPOINTS.
      SUMCOS = (F1 + F0 ) * .5
      SUMSIN = 0.0
      B1 = 2. / 3.
C TMAX IS THE SWITCH-OVER POINT IN THE ANGLE T.
C OUR ANALYSIS INDICATES THAT TMAX = 1/6 IS THE BEST FOR THE ILLIAC II
C WHICH HAS A 44 BIT FLOATING POINT MANTISSA.
      TMAX = 0.166
C N IS THE NUMBER OF THE ITERATION. NOTE THAT WE START AT THE
C FOURTH ITERATION STEP.
C ACTUALLY, THE FIRST EVALUATION OF AN INTEGRAL IS AT N = 5, AND
C THEREFORE, THE FIRST COMPARISON OF VALUES IS AT N= 6.
      N = 4
C BOTH TMAX AND N MAY BE CHANGED IF THE MACHINE FOR WHICH THIS
C ROUTINE IS INTENDED HAS GREATER OR LESS ACCURACY THAN ILLIAC II.
C IF N IS CHANGED , THEN THE CORRESPONDING CHANGES MUST BE MADE
C IN THE ASSIGNMENTS OF H AND NSTOP.
      H = 1. / 16.
C H = 2 ** -N.
      NSTOP = 15
C NSTOP = 2**N - 1
      T = H * XM
      TP = T * PI
      NST = 1
      ASSIGN 67 TO MSWTCH
C LLC AND LLS ARE USED BY THE ROUTINE IN COMPUTED-GO-TO STATEMENTS.
C AS SOON AS LLS AND LLC HAVE BEEN DEFINED, WE CAN USE LS AND LC
C AS RETURN PARAMETERS (SEE ABOVE).
      IF ( LS ) 1, 1, 2
 1    LLS = 2
      GO TO 3
 2    LLS = 1
      LS = MAX
 3    IF ( LC ) 4, 4, 5
 4    LLC = 2
      GO TO 7
 5    LLC = 1
      LC = MAX
 7    LN = 1
C ALL OF THE ABOVE IS EXECUTED ONLY ONCE PER CALL.
C NOW THE ITERATION BEGINS.
 10   ODCOS = 0.
      ODSIN = 0.
C BEGIN SUMMATION FOR ODCOS AND ODSIN.
      DO 65 I = 1, NSTOP, NST
        XI = I
        THA = XI * T
C THA*PI IS THE ANGLE USED IN THIS ITH TERM.

C CIR(I*T*PI) IS CALCULATED HERE USING THE IDENTITY
C CIR ( INTEGER MULTIPLE OF PI + FRACTIONAL MULT OF PI )
C = COS(INTEGER*PI) * CIR(FRAC*PI)
C = (+ OR -) * CIR(FRAC*PI).
        FRAC = THA
        IN = THA
        THA = IN
        FRAC = (FRAC - THA) * PI
C THA IS A FLOATING POINT INTEGER, FRAC IS THE FRACTIONAL PART *PI.
        COSIP = 1 - 2*(IN - 2*(IN/2))
        TEMP1 = COSIP * F(XI*H)
C TEMP1 = COS(INTEGER PART) * F(I*H).
        GO TO ( 50 , 55 ) , LLS
 50     ODSIN = TEMP1 * SIN(FRAC) + ODSIN
 55     GO TO ( 60 , 65 ) , LLC
 60     ODCOS = TEMP1 * COS(FRAC) + ODCOS
 65     CONTINUE
      GO TO MSWTCH,(67,70)
 67   NST = 2
C NOW HAVE MADE UP FOR THE FIRST 4 ITERATION STEPS, SO RESET THESE
C TWO NUMBERS TO LOOK LIKE THE GENERAL CASE.
      NSTOP = 16
C NSTOP = 2**N (IN CASE YOU CHANGE STARTING VALUE OF N).
      ASSIGN 70 TO MSWTCH
      GO TO 92
 70   TSQ = TP*TP
      IF (T -TMAX) 74, 74, 75
C 74 IS THE POWER SERIES FOR SMALL T, 75 IS THE CLOSED FORM USED WITH
C LARGER VALUES OF T.
C THE POWER SERIES ARE (WITH 'TN' = TP**N)
C A = (2./45.)*T3 - (2./315.)*T5 + (2./4725.)*T7
C B = (2./3.) + (2./15.)*T2 - (4./105.)*T4 + (2./567.)*T6
C      - (4./22275.)*T8
C G = (4./3.) - (2./15.)*T2 + (1./210.)*T4 - (1./11340.)*T6
C THE NEXT TERM IN G IS TOO SMALL. IT IS (1./997920.)*T8
 74   A = TP * TSQ * (1. - TSQ * (1. - TSQ / 15.) / 7.) / 22.5
      B2 = B1 * TSQ* .2
      B3 = B2 * TSQ * 2./7.
      B4 = B3 * TSQ / 10.8
      B5 = B4 * TSQ * 14./275.
      B = B1+ B2 - B3 + B4 - B5
      G = 2.*B1 - B2+ B3/ 8. - B4/40.
C G = 2.*B1 - B2 + B3/8. - B4/40. + 5.*B5/896.  IF YOU WANT THE T8
```

```
C TERM INCLUDED IN G.
      GO TO 80
C CLOSED FORM OF THE COEFFICIENTS, WHERE AGAIN 'TN' MEANS TP**N.
C A = 1./TP + COS(TP)*SIN(TP)/T2 - 2.*(SIN(TP))**2/T3
C B = 2.*((1 + (COS(TP))**2)/T2 - 2.*SIN(TP)*COS(TP)/T3)
C G = 4.*(SIN(TP)/T3 - COS(TP)/T2)
 75   IN = T
      TEMP1 = 1 - 2 * ( IN - 2 * ( IN / 2 ) )
      TEMP2 = IN
C TEMP1 IS COS ( INTEGER PART OF TP), TEMP2 IS FRACTIONAL PART OF TP.
      TEMP2 =(T - TEMP2 ) * PI
      S1 = TEMP1 * SIN (TEMP2)
C S1 = SIN(TP)
      C1 = TEMP1 * COS (TEMP2)
C C1 = COS(TP)
      P = S1 * C1
      S1SQ = S1 * S1
      A = (((-2.*S1SQ/TP) + P)/TP +1.)/ TP
      B = 2. * ((-2.* P/ TP)+ 2. -S1SQ) / TSQ
      G = 4. * (S1 / TP - C1)/ TSQ
 80   GO TO (81, 85), LLS
C HAVE CALCULATED THE COEFFICIENTS, NOW READY FOR THE INTEGRATION
C FORMULAS.
 81   T2 = H* (A * (F0 - F1) + B * SUMSIN + G * ODSIN)
C ENDT1 IS A SUBROUTINE WHICH CHECKS FOR THE CONVERGENCE OF THE
C ITERATIONS. ENDT1 REQUIRES THE PRESENT VALUE TO AGREE WITH THE
C PREVIOUS VALUE TO WITHIN EPS2, WHERE
C EPS2 = (1.0 + ABSF(PRESENT VALUE))*EPS
C EPS IS SUPPLIED BY THE USER.
      CALL ENDT1   (PVT2, T2, EPS, S, LLS, LN)
      GO TO ( 85, 84 ), LLS
 84   LS = N
 85   GO TO (86,90),LLC
C THIS IS THE COSINE INTEGRAL.
 86   T1 = H * ( B * SUMCOS + G * ODCOS)
      CALL ENDT1   (PVT1, T1, EPS, C, LLC, LN)
      GO TO ( 90, 89 ), LLC
 89   LC = N
 90   LN = 2
C NOW TEST TO SEE IF DONE.
      IF (LLC + LLS - 3) 92, 92, 100
 92   N = N + 1
C THIS IS THE BEGINING OF THE ITERATION.
      IF (N-MAX) 95, 95,100
 95   H = .5 * H
      T = .5 * T
      TP = .5 * TP
      NSTOP = 2 * NSTOP
      SUMSIN = SUMSIN + ODSIN
      SUMCOS = SUMCOS + ODCOS
      GO TO 10
 100  S = T2
      C = T1
      RETURN
      END
      SUBROUTINE ENDT1   (PREVQT, QUANT,EPS, VALUE, L1, L2)
      GO TO ( 29, 20), L2
 20   REPS = EPS * (1.0 + ABS(QUANT))
 23   IF (ABS(PREVQT - QUANT) - REPS) 25, 25, 29
 25   VALUE = QUANT
      L1 = 2
      GO TO 30
 29   PREVQT = QUANT
 30   RETURN
      END
```

REMARK ON ALGORITHM 353 [D1]
FILON QUADRATURE [Stephen M. Chase and Lloyd D. Fosdick, *Comm. ACM 12* (Aug. 1969), 457–458]

Bo Einarsson (Recd. 8 Dec. 1969)

Research Institute of National Defense, Box 98,
S-147 00 Tumba, Sweden

KEY WORDS AND PHRASES: quadrature, Filon quadrature, integration, Filon integration, Fourier coefficients, Fourier series
CR CATEGORIES: 5.16

The algorithm has been tested in double precision on an IBM 360/75 with great success. An improvement to the algorithm to take care of heavily oscillating functions can easily be made. The starting value of the number N of iterations is chosen to give at least four quadrature nodes for each full period of the trigonometric function. The following changes are therefore suggested:

line 22: N = ALOG(2.*XM)/0.693
line 27: H = 1.0/FLOAT (2**N)
line 29: NSTOP = 2**N − 1
line 79: NSTOP = 2**N

The following example shows the importance of this change at the computation of the sine integral for $m = 64$ with the function $f(x) = x^2(1-x)$, which is zero at both endpoints. Entry variables were in both cases MAX = 20, M = 64, LC = 1, LS = 1, and EPS = 1.0E-10. The computation in double precision gave the results:

original version LC = LS = 6   C = −0.2473661710D-04
                                S = 0.0
improved version LC = LS = 9   C = −0.2473661709D-04
                                S = −0.7381790409D-06

The exact values are

$$C = -1/(64\pi)^2 = -0.2473661710 \cdot 10^{-4}$$

and

$$S = -6/(64\pi)^3 = -0.7381790413 \cdot 10^{-6}.$$

The failure of the original computation is due to the fact that all the inner nodes of the sine integral are multiples of π, and the boundary contribution is zero. The connection with the sampling theorem is obvious.

The original version of the algorithm is not valid for negative values of M. The use of ALOG(2.*XM) is therefore no essential restriction, since the algorithm is rather slow for computing an ordinary quadrature (with M = 0).

It is important to observe that the present version will give correct values only if the maximum number MAX is larger than the computed value N.

ALGORITHM 354
GENERATOR OF SPANNING TREES [H]

M. Douglas McIlroy (Recd. 29 Apr. 1966, 9 Sept. 1968 and 6 Mar. 1969)

Bell Telephone Laboratories, Murray Hill, NJ 07971, and Oxford University Computing Laboratory, Oxford, England

KEY WORDS AND PHRASES: spanning trees, trees, graphs
CR CATEGORIES: 5.32

/* This procedure finds all trees that span a nondirected graph on n nodes. The essential step of this procedure partitions the set $T(G)$ of trees which span graph G into two classes. Trees of one class contain a branch connecting a selected pair of nodes, i and j; trees of the other class exclude such branches. To formalize the effect of partitioning with respect to nodes i and j, we let A_{ij} be the "attachment set" of branches between them, and G_{ij} be the graph derived from G by combining i and j into a single node. Then

$$T(G) = T(G_{ij}) \times A_{ij} \cup T(G-A_{ij}).$$

The algorithm generates $T(G)$ by a particular combination of recursive and iterative applications of this partition. A set S of combined nodes is "grown" by incorporating one node at each level of recursion. The attachment set for node i is the set of branches radiating from i to members of S. The recursion bottoms whenever S contains all nodes, and a "family" of trees is then produced, where a family is the Cartesian product of the attachment sets from each level.

The basic method would work for any graph, but to simplify data representation, this algorithm requires that G be free of paralleled branches and self-loops. All computations are done in terms of the original graph to save actually having to combine nodes and, incidentally, to avoid parallels arising from combination. A set of nodes is represented by a string of n bits, with 1's for nodes present and 0's for nodes absent. The original graph is represented by an array of n strings, where the ith string indicates the set of nodes neighboring node i. An attachment set for node i is a suitable subset of its neighborhood.

The algorithm maintains the graph G, the set of combined nodes S, and a boundary set B of nodes neighboring members of S. B is disjoint from S. Initially S contains only node 1; B is the neighborhood of node 1. The key recursive routine "grow" iterates over the nodes of B. For each node i in B it finds the attachment set (necessarily nonempty) connecting i to S. It then removes the attachment set from G and node i from B, and calls "grow" recursively with S augmented by node i and B augmented by neighbors of i (except those in S). The recursive call thus yields trees which include branches from node i to S, while the iteration over succeeding nodes in B yields trees which exclude such branches.

As an example, for the graph

```
1(2,3,4)  2(1,3)  3(1,2,4)  4(1,3)
```

the algorithm generates eight trees in four families:

```
1( )  2(1)  3(1,2)  4(1,3)
1( )  2(1)  3(4)    4(1)
1( )  2(3)  3(1)    4(1,3)
1( )  2(3)  3(4)    4(1)
```

In these lists a set is represented by its index together with a parenthesized list of contained nodes.

Unlike other algorithms in the literature [1, 2, 3], this produces unique trees and hence does not require storage for a checklist of trees already produced. An algorithm of Burstall [4] generalizes this strategy to a wide class of problems; however, a direct particularization of Burstall's algorithm for spanning trees would be less efficient.

Acknowledgment is due S. C. Johnson, A. J. Goldstein, J. B. Kruskal, and D. M. R. Park for discussion and help, also P. Seaman and IBM U. K. Laboratories for testing.

References

1. Hakimi, S. L., and Green, D. G. Generation and realization of trees and k-trees. *IEEE Trans. CT-11* (1964), 247–255.
2. Watanabe, H. A computational method for network topology. *IRE Trans. CT-7* (1960), 296–302.
3. MacWilliams, F. J. Topological network analysis as a computer program. *IRE Trans. CT-5* (1958), 228–229.
4. Burstall, R. M. Tree-searching methods with an application to a network design problem. In *Machine Intelligence 1*, N. L. Collins and D. Michie (Eds.), Oliver and Boyd, Edinburgh, 1967.*/

```
/* Nomenclature
     G = the graph, or modified graph,
     S = set of nodes covered by growing family,
     B = "boundary" set of uncovered nodes neighboring to
         members of S,
     A = array of attachment sets,
  f(A) = routine for processing each family once it has been
         generated.*/
trees:
  procedure(G, f);
    declare  G(*)  bit(*), f entry, n fixed binary;
    n = hbound(G, 1);
    begin;
      declare A(n) bit(n),
        unit entry(fixed binary) returns(bit(n));
        /*Start at node 1. Arguments by value.*/
      call grow((G), unit(1), (G(1)));
    grow:
      procedure(G, S, B) recursive;
        declare(G(n), S, B) bit(n), i fixed binary;
        if ¬S = '0' b then call f(A);
        else
        do i = 1 to n;
          if substr(B, i, 1) then
          do;
            substr(B, i, 1) = '0'b;
            A(i) = G(i) & S
            G(i) = G(i) & ¬S;
            call grow((G). S|unit(i), B|G(i));
            if G(i) = '0' b then return;
          end;
      end;
    end grow;
  unit:
    procedure(i) bit(n);
      declare i fixed binary, u bit(n) initial (' 'b);
      substr(u, i, 1) = '1'b;
      return(u);
    end  unit;
  end;
end trees;
```

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 $(S_1, \cdots, S_n)$ of zeros and ones where $x = \sum_{i=1}^{n} S_i$ and $t = \sum_{i=1}^{n-1} | S_{i+1} - S_i |$ are given. The main idea is to interleave compositions of x and $n - x$ objects and resort to a lexicographic generation of compositions. 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 distinct 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 ≥ 4 elements of the
  infinite sequence 2, 3, 5, 7, 11, ··· of prime numbers and stores
  them in IP[1], IP[2], ···, 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 √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] ≥ IQ[i ÷ 2]   for 2 ≤ i ≤ 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 func-
  tion of the two used here, and the information is totally ordered
  each time a prime is found. Between primes the unordered seg-
  ment of the information is searched to locate the smallest ele-
  ment. 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 pro-
  portional 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 propor-
  tional 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], ···, IP[m]. IP[m+1], IP[m+2], ···,
  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* = 10*m*, 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
```

Remark on Algorithm 357 [A1]

An Efficient Prime Number Generator [Richard C. Singleton, *Comm. ACM 10* (October, 1969), 563]

Richard M. De Morgan [Recd 8 August 1972], Digital Equipment Co. Ltd., Reading, England

On some Algol 60 implementations, the value of *ni* is destroyed between subsequent calls to the procedure. The second and third lines of the algorithm should be changed to make *ni* an **own integer**:

```
own integer ij, ik, inc, j, ni, nj;
integer i, jqi, k;
```

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 $\geq$ 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, $\cdots$, 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, $\cdots$, 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 $\leq$ 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 Q,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)=G*(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=I.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,C.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
```

ALGORITHM 359
FACTORIAL ANALYSIS OF VARIANCE* [G1]

John R. Howell (Recd. 2 Aug. 1968 and 12 May 1969)
Department of Biometry, Medical Center, Virginia Commonwealth University, Richmond, VA 23219

* This investigation was supported in part by Public Health Service Research Grant FR 00016-05, from the National Institutes of Health.

KEY WORDS AND PHRASES: factorial variance analysis, variance, statistical analysis
CR CATEGORIES: 5.5

Comments. This subroutine transforms a vectory $\mathbf{y}$, observed in a balanced complete $t_1 \times t_2 \times \cdots \times t_n$ factorial experiment, into an interaction vector $\mathbf{z}$, whose elements include mean and main effects.

The experimental observations y_s, $(\mathbf{s} = (s_1, s_2, \cdots, s_n)$; $s_i = 0, 1, \cdots, t_i - 1$; $i = 1, 2, \cdots, n)$ are assumed to be stored in the array Y in increasing order by the composite base integer $\mathbf{s}$. After the transformation, the array Z will contain the interactions in natural order.

The method used is Good's [1, 2] modification of Yates's [5] interaction algorithm. In [1, p. 367], the interactions are expressed in the form $\mathbf{z} = (M_1 \otimes M_2 \otimes \cdots \otimes M_n)\mathbf{y}$, where M_i is a $t_i \times t_i$ matrix of normalized orthogonal contrasts and where $\otimes$ denotes a direct (Kronecker, tensor) product. The interactions can also be written $\mathbf{z} = (C_1 C_2 \cdots C_n)\mathbf{y}$, where

$$C_1 = M_1 \otimes I_{t_2} \otimes \cdots \otimes I_{t_n}$$

$$C_2 = I_{t_1} \otimes M_2 \otimes \cdots \otimes I_{t_n}$$

$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$

$$C_n = I_{t_1} \otimes I_{t_2} \otimes \cdots \otimes M_n$$

and where I_{t_i} is the $t_i \times t_i$ identity matrix.

By performing elementary operations (row and column interchanges) on the C_i we get $\mathbf{z} = (D_1 D_2 \cdots D_n)\mathbf{y}$, where

$$D_i = \left(\begin{array}{c} M_{i1} \oplus \cdots \oplus M_{i1} \\ \hdashline M_{i2} \oplus \cdots \oplus M_{i2} \\ \hdashline \hdashline M_{it_i} \oplus \cdots \oplus M_{it_i} \end{array}\right)$$

and where M_{ij} is row j of M_i. The symbol $\oplus$ denotes a direct sum. For an example of this for an unnormalized matrix, see Good [1, p. 362].

Since each row of D_i consists of a row of M_i and zeros, we only need M_i for forming $\mathbf{z}$. The subroutine forms first $D_n\mathbf{y}$, then this result is premultiplied by D_{n-1}, and so on until we obtain $\mathbf{z}$. The elements of $\mathbf{z}$ are the required interactions.

This method can be mechanized for hand computation in the following way. (The subroutine was written from this point of view.) Write the observations in the order specified above. Write row one of M_n down the right edge of a strip of paper using the same spacing as for the observations. Now place this movable strip alongside the observation vector so that the top element on the paper strip is opposite the top element of the observation vector. Multiply adjacent elements and write the sum of these products at the top of a new column. Now slide the paper strip down t_n spaces. Form the indicated inner product as before and write the result in the new column below the previous entry. Continue in this manner until all the observations have been used. Now write row two of M_n on a strip of paper and proceed as before. If we continue this process with all the rows of M_n we will get a new vector $\mathbf{z}_n$ whose elements are linear transformations of the observation vector $\mathbf{y}$. The dimension of $\mathbf{z}_n$ is the same as that of $\mathbf{y}$. Similarly form $\mathbf{z}_{n-1}$ from $\mathbf{z}_n$ and M_{n-1}. Continuing this process we finally obtain $\mathbf{z}_1 = \mathbf{z}$ which is the desired interaction vector.

In all the foregoing we used the normalized contrast matrices; thus the sums of squares are the squares of the elements of $\mathbf{z}$. For hand computation, one might prefer using the unnormalized contrast matrices, since their elements are integers. But then we need a vector of divisors; it is obtained by performing the same operations on a column of ones as on $\mathbf{y}$, except that we use the squares of the elements of the contrast matrices. Then the ith sum of squares equals z_i^2 divided by the corresponding divisor.

This method might be called a "paper strip method" for analysis of variance and is similar to paper strip methods used for operations with polynomials. For examples of this, see Lanczos [3] and Prager [4].

We require $2t_1t_2 \cdots t_n$ locations for storing $\mathbf{y}$ and $\mathbf{z}$ plus $\sup(t_1, t_2, \cdots, t_n)$ locations for storing a row of M_i. The number of multiplications required is $(\prod t_i)(\sum t_i + 1)$.

Acknowledgments: The author wishes to thank Dr. A. E. Brandt for initiating his interest in programming analysis of variance. He wishes to thank Dr. W. H. Carter, Jr., and the referee, for helpful comments.

References:

1. Good, I. J. The interaction algorithm and practical Fourier analysis. *J. Roy. Statist. Soc.* {B} *20*, 2 (1958), 361–372.
2. Good, I. J. The interaction algorithm and practical Fourier analysis: An addendum. *J. Roy. Statist. Soc.* {B} *22*, 2 (1960). 372–375.
3. Lanczos, C. *Applied Analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1956.
4. Prager, W. *Introduction to Basic Fortran Programming and Numerical Methods.* Blaisdell, Waltham, Mass., 1965.
5. Yates, F. The design and analysis of factorial experiments. Imperial Bureau of Soil Science, Harpenden, England, 1937.

```
      SUBROUTINE FNCVA
C     ****************
     *  (Y,Z,ROW,MSIZE,NCLS,NFCTR)
      DIMENSION  Y(1),Z(1),
     *              ROW(1),MSIZE(1)
C     LOOP FOR NFCTR CONTRAST MATRICES
      DO 5  NF = 1,NFCTR
         I     = 1
C     GET SIZE OF THE MATRIX
         K     = NFCTR-NF+1
         NRNC  = MSIZE(K)
      DO 3   J = 1,NRNC
C     ROW OF A CONTRAST MATRIX
      CALL AROW (ROW,NRNC,J)
C     PERFORM THE 'PAPER STRIP'
C     OPERATION FOR A MATRIX ROW
      DO 2   K = 1,NCLS,NRNC
         Z(I)  = 0.
      DO 1   L = 1,NRNC
         KL1   = K+L-1
    1    Z(I)  = Z(I)+ROW(L)*Y(KL1)
    2    I     = I+1
    3    CONTINUE
C     MOVE Z INTO Y
      DO 4   J = 1,NCLS
    4    Y(J)  = Z(J)
    5    CONTINUE
      DO 6   J = 1,NCLS
    6    Y(J)  = Y(J)*Y(J)
      RETURN
      END
```

```
      SUBROUTINE AROW
C     ***************
     *  (ROW,NRNC,J)
      DIMENSION ROW(1)
C     IF ROW ONE
      IF(J-1)3,1,3
    1    A       = NRNC
         EL      = 1./SQRT(A)
      DO 2    I  = 1,NRNC
    2    ROW(I)= EL
C     AND
      RETURN
C     ELSE
    3    JM1     = J-1
         RJ      = J
         A       = SQRT(RJ*RJ-RJ)
         EL      = 1./A
      DO 4    I  = 1,JM1
    4    ROW(I)= EL
      DO 5    I  = J,NRNC
    5    ROW(I)= 0.
         ROW(J)= (1.-RJ)/A
      RETURN
      END
```

REMARKS ON:
ALGORITHM 332 [S22]
JACOBI POLYNOMIALS [Bruno F. W. Witte, *Comm. ACM 11* (June 1968), 436]
ALGORITHM 344 [S14]
STUDENT'S *t*-DISTRIBUTION [David A. Levine, *Comm. ACM 12* (Jan. 1969), 37]
ALGORITHM 351 [D1]
MODIFIED ROMBERG QUADRATURE [Graeme Fairweather, *Comm. 12* (June 1969), 324]
ALGORITHM 359 [G1]
FACTORIAL ANALYSIS OF VARIANCE [John R. Howell, *Comm. ACM 12* (Nov. 1969), 631]

Arthur H. J. Sale (Recd. 16 Feb. 1970)
Basser Computing Department, University of Sydney,
Sydney, Australia

KEY WORDS AND PHRASES: Fortran standards
CR CATEGORIES: 4.0, 4.22

An unfortunate precedent has been set in several recent algorithms of using an illegal FORTRAN construction. This consists of separating an initial line from its continuation line by a comment line, and is forbidden by the standard (see sections 3.2.1, 3.2.3 and 3.2.4 of [1, 2]). The offending algorithms are to date: 332, 344, 351 and 359.

While this is perhaps a debatable decision by the compilers of the standard, and trivial to correct, it seems a pity to break the rules just for a pretty layout as has been done.

References:
1. ANSI Standard FORTRAN (ANSI X3.9-1966), American National Standards Institute, New York, 1966.
2. FORTRAN vs. Basic FORTRAN, *Comm. ACM 7* (Oct. 1964), 591–625.

ALGORITHM 360
SHORTEST-PATH FOREST WITH TOPOLOGICAL ORDERING [H]

ROBERT B. DIAL (Recd. 21 Nov. 1968, 27 Nov. 1968 and 30 Apr. 1969)
Alan M. Voorhees and Associates, Inc., McLean, VA 22101, and Department of Civil Engineering, University of Washington, Seattle, WA 98105

KEY WORDS AND PHRASES: shortest path, tree, network, directed graph
CR CATEGORIES: 5.32, 5.42

```
procedure MOORE (INDEX, J, D, maxd, n, DIST, I, NEXT,
    LAST, maxdist, ROOT, m);
  value maxd, n, maxdist, m;
  integer array INDEX, J, D, DIST, I, NEXT, LAST, ROOT;
  integer maxd, n, maxdist, m;
```

comment Given a subset (called "roots") of the nodes (numbered from 1 to *n*) spanned by a directed graph composed of arcs of known length, *MOORE* finds for each node in the network the shortest path connecting it to its closest root node. The result is a disjoint set of shortest-path trees, referred to here as a "shortest-path forest." *MOORE*'s output describes all the paths in the forest and gives their lengths. It also provides two lists which sequence the nodes spanned by the forest in forward and backward topological order. In the algorithm's terminology, "forward topological order" is a sequence in which any given node is listed after any other node which lies on the path between it and its root node. Conversely, the "backward topological order" has the nodes arranged in decreasing distance from their nearest root node.

The procedure below implements a well-known, widely-used algorithm by E. F. Moore [1] and is particularly suited for a large, sparse network whose arc lengths are short and which have a small variance, e.g. an urban highway system. As an indication of its efficiency, an Assembly Language routine patterned after *MOORE* for the IBM 360 model 65 found all shortest paths from a single root node to the remaining 12,000 nodes of a 36,000-arc network (i.e. built a minimum-path tree) in one (1) second. In general, for a connected graph, *MOORE*'s "running time" is directly proportional to the number of arcs in the network and is independent of the number of roots. The mechanics of the algorithm are summarized in the following three steps:

0. Mark each root node *r* "reached but not scanned" and associate with it a distance of zero (*DIST*[*r*]=0). Mark each nonroot node *i* "not reached" and associate with it a distance of infinity (i.e. *DIST*[*i*]=*maxdist*). Go to Step 1.
1. From among the nodes marked "reached but not scanned," select the node *i* whose distance is smallest. If there is no node so marked, the forest is complete. Otherwise go to Step 2.
2. For each arc (*i*, *j*) in the network (i.e. all arcs exiting the selected node *i*), compare *DIST*[*j*] with the sum of *DIST*[*i*] and the arc length of (*i*, *j*). Whenever this latter sum is less than the former quantity, set *DIST*[*j*] equal to it, mark node *j* "reached but not scanned," and put the arc (*i*, *j*) in the forest, removing any other arc whose final node is *j*. When all arcs exiting node *i* have been so examined mark node *i* "reached and scanned" and go to Step 1.

While Moore's algorithm possesses the important attribute of examining each arc in the network only once, the speed achieved in its implementation depends primarily on its efficiency in Step 1. To facilitate this node selection, the procedure below uses a topological ordering of the final nodes of the arcs in the partial forest. It effects Step 1 by referring to a forward-ordering list, *NEXT*, to determine which node should be selected next from the "reached but not scanned" category. A backward-ordering list, *LAST*, aids updating the ordering when a previously found path to a node is superseded by a newly found, shorter one. Also used in this updating process are two short local vectors, *HEAD* and *TAIL*. *HEAD*[*d*] and *TAIL*[*d*] contain the first and last node of a sublist of nodes, whose associated distance is not less than the distance of the node selected in Step 1 and is congruent to *d* modulo the net's maximum arc length. The use of these latter two arrays becomes clear while studying the ALGOL below.

Besides the *m* root nodes stored in *ROOT*[1], ··· , *ROOT*[*m*], input to *MOORE* consists of a network description in three vectors, *J*, *D*, and *INDEX*, together with the scalar parameters *n*, *maxd*, and *maxdist*. The array *J* contains the final node numbers of all arcs in the network stored in ascending sequence with respect to their initial node number. The second vector, *D*, is parallel to the array *J* and holds the corresponding arc lengths—against which paths are to be minimized. *INDEX*[*i*] points to the first element of *J* representing an arc exiting node *i*. *INDEX* is dimensioned from 1 to $n + 1$, where the parameter *n* is the highest node number in the network, and *INDEX*[*n*+1] contains one plus the total number of arcs in the network. The arc lengths stored in the array *D* must be positive integers strictly less than the parameter *maxd*. Similarly, as *maxd* exclusively limits the length of an arc, so does the other input scalar parameter *maxdist* limit the length of a path. *MOORE* only considers paths which are shorter than *maxdist*.

The algorithm's output describes the minimum-path forest in two vectors, *I* and *DIST*. *I*[*j*] contains the initial node of the forest's unique are whose final node is *j*. Thus the sequence of nodes representing the shortest path from the nearest root to *j* is found in reverse order by looking at *I*[*j*], *I*[*I*[*j*]], etc., until a root node is encountered. *DIST*[*j*] returns the minimized distance from the closest root node to *j*. If *j* is not reachable from any root node via a path shorter than *maxdist*, *MOORE* returns with *DIST*[*j*] = *maxdist* and *I*[*j*] = 0. The forest's topological orderings are returned in list form in the pointer vectors *NEXT* and *LAST*. *NEXT* is a circular successor list. The number of the node closest to its root node is stored in *NEXT*[*ROOT*[1]]. The next closest node is contained in *NEXT*[*NEXT*[*ROOT*[1]]], etc., until *ROOT*[1] is encountered in some *NEXT*[*j*], where *j* is the number of the node farthest from its root node. Similarly, *LAST* is a circular predecessor list. The backward topological order is obtained by starting at *LAST*[*ROOT*[1]], which contains the number of the most distant node. *LAST*[*LAST*[*ROOT*[1]]] has the next most distant, etc., until *LAST*[*j*] = *ROOT*[1], *j* being the closest node to its root. When no path shorter than *maxdist* exists between a root node and *j*, then *j* appears in neither the *NEXT* nor the *LAST* list.

REFERENCE:

1. MOORE, E. F. The shortest path through a maze. In *International Symposium on the Theory of Switching Proceedings.* Harvard U. Press, Cambridge, Mass., Apr. 1957, pp. 285–292;

```
begin
  integer procedure mod(d, maxd); value d, maxd; integer
    d, maxd; mod := d − maxd × entier(d÷maxd);
  integer array HEAD[0:maxd−1], TAIL[0:maxd−1]; integer
    i, pt, k, v, j, q, ct;
  for i := 1 step 1 until maxd−1 do HEAD[i] := TAIL[i] := 0;
  for i := 1 step 1 until n do
  begin DIST[i] := maxdist; I[i] := 0 end;
  for i := 2 step 1 until m do
  begin
    NEXT[ROOT[i−1]] := ROOT[i]; LAST[ROOT[i]] := ROOT
    [i−1];
    DIST[ROOT[i]] := 0
  end;
  LAST[ROOT[1]] := NEXT[ROOT[m]] := DIST[ROOT[1]] :=
    pt := 0;
  i := HEAD[0] := ROOT[1]; TAIL[0] := ROOT[m];
comment Examine all exits from selected node (Step 2 above);
r: for k := INDEX[i] step 1 until INDEX[i+1] − 1 do
  begin
    v := DIST[i] + D[k]; j := J[k];
    if v < DIST[j] then
    begin
      comment Path to j via i is shortest so far—put arc (i, j)
      in forest;
      if DIST[j] ≠ maxdist then
      begin
        comment Delete node j from its prior sublist;
        q := mod(DIST[j], maxd);
        if HEAD[q] = j then HEAD[q] := NEXT[j]
        else
        begin
          if TAIL[q] = j then
          begin TAIL[q] := LAST[j]; NEXT[LAST[j]] := 0
            end
          else
          begin LAST[NEXT[j]] := LAST[j]; NEXT[LAST
            [j]] := NEXT[j] end
        end
      end;
      comment Hook j to its new sublist, and put arc (i, j) in
        forest;
      q := mod(v, maxd);
      if HEAD[q] = 0 then
      begin HEAD[q] := j; LAST[j] := 0 end
      else
      begin LAST[j] := TAIL[q]; NEXT[TAIL[q]] := j end;
      comment Update forest and forward ordering;
      I[j] := i; DIST[j] := v; TAIL[q] := j; NEXT[j] := 0
    end
  end;
  comment Select next node i whose exit arcs are to be examined
    (Step 1 above);
  if NEXT[i] ≠ 0 then
  begin
    comment Sublist containing i not empty—use successor of
    i; i := NEXT[i]; go to r
  end;
  comment Sublist containing i empty—use first node in next
    nonempty sublist;
  HEAD[pt] := 0;
  for ct := 1 step 1 until maxd − 1 do
  begin
    pt := mod(pt+1, maxd);
    if HEAD[pt] ≠ 0 then
    begin
      comment Found a nonempty sublist—hook it to lists;
      LAST[HEAD[pt]] := i; i := NEXT[i] := HEAD[pt];
        go to r
    end;
  end;
  comment All sublists empty, forest built—circularize lists
    and quit;
  LAST[ROOT[1]] := i; NEXT[i] := ROOT[1]
end MOORE
```

ALGORITHM 361
PERMANENT FUNCTION OF A SQUARE MATRIX I AND II [G6]

Bruce Shriver, P. J. Eberlein, and R. D. Dixon (Recd. 19 Feb. 1969, 7 Mar. 1969 and 9 July 1969)
State University of New York at Buffalo, Amherst, NY 14226

KEY WORDS AND PHRASES: matrix, permanent, determinant
CR CATEGORIES: 5.30

```
real procedure per1(A, n);
  integer n;  array A;
```

comment Let A be an $n \times n$ real matrix, $n > 1$. The permanent function of A, denoted per(A), is computed by H. J. Ryser's [1] expansion formula:

$$\text{per}(A) = \sum_{r=0}^{n-1} (-1)^r \sum_{\mathbf{x} \in T_{n-r}} \prod_{i=1}^{r} x_i$$

where Tj, $j = n, n-1, \cdots, 2, 1$, is the set of vectors $\mathbf{x} = (x_i)$, $i = 1, 2, \cdots, n$ which are obtained by adding j columns of A together in all $\binom{n}{j}$ possible ways. To effect the sum over vectors in T_j, $n - 1$ sums are computed. The natural 1-1 map from the binary integers to all r-combinations, $r = 1, 2, \cdots, n-1$, is used to increment the sums over the sets T_j.

Reference:

1. Ryser, H. J. *Combinatorial Mathematics, Carus Monograph* #14. Wiley, New York, 1963, p. 27;

```
begin
  real sig, pera, prod, rowsum;
  integer number, limit, mod, gen, g, i, j, r;
  array sum[0:n−1];
  integer array d[1:n];
  sig := −1;  pera := 0;  limit := (2 ↑ n) − 1;
  for r := 0 step 1 until n − 1 do sum[r] := 0;
  for number := 1 step 1 until limit do
  begin
    r := 0;  gen := number;
    for mod := 1 step 1 until n do
    begin
      g := gen ÷ 2;  if (gen−g×2) = 1 then
      begin r := r + 1;  d[r] := mod end;
      gen := g
    end;
    prod := 1;
    for i := 1 step 1 until n do
    begin
      rowsum := 0;
      for j := 1 step 1 until r do
      rowsum := rowsum + A[i, d[j]];
      prod := prod × rowsum
    end;
    sum[n−r] := sum[n−r] + prod
  end;
  for r := 0 step 1 until n − 1 do
  begin sig := −sig;  pira := pera + sig × sum[r] end;
  per := pera
end of real procedure per1;
```

```
real procedure per2(A, n);
  integer n;  array A;
```

comment Let A be an $n \times n$ real matrix, $n > 1$. The permanent function of A, denoted by per(A) is computed by Jurkat and Ryser's [1] method of inductively generating the vectors $p_1, \cdots, p_n$ where p_r is the vector of permanents of r by r submatrices of the first r rows of A. This vector has $\binom{n}{r}$ components indexed by the r-combinations of $\{1, \cdots, n\}$. The natural 1-1 map from the binary integers $\{1, \cdots, 2 \uparrow n-1\}$ to the r-combinations of $\{1, \cdots, n\}$ for $r = 1, \cdots, n$ is used to index the p's and thus they are generated in an order somewhat different from that of Jurkat and Ryser.

Reference:

1. Jurkat, W. B. and Ryser, H. J. Matrix factorizations of determinants and permanents. *J. Algebra 3* (1966), 1–27;

```
begin
  integer number, limit, mod, gen, g, r, dig, sub, j;
  array list [1:2 ↑ n−1];
  limit := 2 ↑ n − 1;
  comment Initialize list as accumulators;
  for j := 1 step 1 until limit do list [j] := 0;
  for j := 1 step 1 until n do list [2 ↑ (j−1)] := A[1, j];
  for number := 1 step 1 until limit do
  begin
    if list [number] ≠ 0 then
    begin
      r := 1; gen := number;
      for mod := 1 step 1 until n do
      begin
        g := gen ÷ 2;
        if gen − 2 × g = 1 then r := r + 1;
        gen := g
      end count of 1's in number;
      dig := 1;  gen := number;
      for mod := 1 step 1 until n do
      begin
        g := gen ÷ 2;
        if gen − 2 × g = 0 then
        begin
          sub := number + dig;
          list [sub] := list [sub] + list [number] × A [r, mod]
        end;
        gen := g;  dig := 2 × dig
      end computations with list [number];
    end
  end;
  per := list [limit]
end of real procedure per2;
```

Note. On the Permanent Function of a Square Matrix I and II: Program I is slower than Program II. However Program II uses approximately 2^n more locations of store. The running times for both programs double when n is incremented by 1.

REMARK ON ALGORITHM 361 [G6] PERMANENT FUNCTION OF A SQUARE MATRIX I AND II [Bruce Shriver, P. J. Eberlein, and R. D. Dixon, *Comm. ACM 12* (Nov. 1969), 634]

Bruce Shriver, P. J. Eberlein, and R. D. Dixon

(Recd. 22 Jan. 1970)

State University of New York at Buffalo, Amherst, NY 14226

KEY WORDS AND PHRASES: matrix, permanent, determinant
CR CATEGORIES: 5.30

The authors would like to cite the following misprints in the above two algorithms:

(A) In procedure *per*1(*A*, *n*)
- (1) in line 43, the variable name *pira* should be *pera*
- (2) in line 44, the variable name *per* should be *per*1.

(B) In procedure *per*2(*A*, *n*)
- (1) in line 47, the variable name *per* should be *per*2.

ALGORITHM 362
GENERATION OF RANDOM PERMUTATIONS [G6]

J. M. Robson (Recd. 1 Apr. 1969)
Programming Research Group, 45 Banbury Road, Oxford,
England

KEY WORDS AND PHRASES: permutation, random permutation, transposition
CR CATEGORIES: 5.5

```
procedure perm(n, r, A); value n, r; integer n, r; integer
  array A;
comment This procedure produces in the vector A a permuta-
  tion on the integers 1, 2, ··· , n, each of the n! permutations
  being given by one value of r between 1 and n! inclusive. It is
  thus similar in effect to the procedure given in [1] but it is con-
  siderably faster, especially for large values of n, since it uses a
  single loop rather than a double one.
    A permutation is generated as the product of n − 1 transpo-
  sitions of which the jth transposes A[n+1−j] and A[x] for
  some x ≤ n + 1 − j.
    If the line
  for i := 1 step 1 until n do A[i] := i
  is omitted the procedure will permute the original values
  A[1], ··· , A[n] in the same manner.
  REFERENCE:
1. ROBINSON, C. L. Algorithm 317, Permutation. Comm. ACM 10
     (Nov. 1967), 729;
begin
  integer i, x, y;
  for i := 1 step 1 until n do A[i] := i;
  for i := n step −1 until 2 do
  begin
    x := r − (r÷i) × i + 1;  r := r ÷ i;
    y := A[x];  A[x] := A[i];  A[i] := y
  end
end
```

ALGORITHM 363
COMPLEX ERROR FUNCTION* [S15]

WALTER GAUTSCHI (Recd. 11 June 1969)
Computer Sciences Department, Purdue University, Lafayette, IN 47907

* Work supported, in part, by the National Aeronautics and Space Administration (NASA) under grant NGR 15-005-039 and, in part, by Argonne National Laboratory.

KEY WORDS AND PHRASES: error function for complex argument, Voigt function, Laplace continued fraction, Gauss-Hermite quadrature, recursive computation
CR CATEGORIES: 5.12

```
procedure wofz(x, y, re, im);  value x, y;  real x, y, re, im;
comment This procedure evaluates the real and imaginary
  part of the function w(z) = exp(−z²)erfc(−iz) for argument
  z = x + iy in the first quadrant of the complex plane. The accu
  racy is 10 decimal places after the decimal point, or better.
  For the underlying analysis, see W. Gautschi, "Efficient com-
  putation of the complex error function," to appear in SIAM
  J. Math. Anal.;
begin
  integer capn, nu, n, np1;
  real h, h2, lambda, r1, r2, s, s1, s2, t1, t2, c;
  Boolean b;
  if y < 4.29 ∧ x < 5.33 then
  begin
    s := (1−y/4.29) × sqrt(1−x × x/28.41);
    h := 1.6 × s;  h2 := 2 × h;
    capn := 6 + 23 × s;  nu := 9 + 21 × s
  end
  else
  begin h := 0;  capn := 0;  nu := 8 end;
  if h > 0 then lambda := h2 ↑ capn;
  b := h = 0 ∨ lambda = 0;
  r1 := r2 := s1 := s2 := 0;
  for n := nu step −1 until 0 do
  begin
    np1 := n + 1;
    t1 := y + h + np1 × r1;  t2 := x − np1 × r2;
    c := .5/(t1 × t1 + t2 × t2);
    r1 := c × t1;  r2 := c × t2;
    if h > 0 ∧ n ≦ capn then
    begin
      t1 := lambda + s1;  s1 := r1 × t1 − r2 × s2;
      s2 := r2 × t1 + r1 × s2;
      lambda := lambda/h2
    end
  end;
  re := if y = 0 then exp(−x×x) else
      1.12837916709551 × (if b then r1 else s1);
  im := 1.12837916709551 × (if b then r2 else s2)
end wofz
```

Certification of Algorithm 363 [S15]

Complex Error Function [Walter Gautschi, *Comm. ACM 12* (Nov. 1969), 635]

K.S. Kölbig* (Recd. 8 Oct. 1970)
Data Handling Division, European Organization for Nuclear Research (CERN), 1211 Geneva 23, Switzerland.

Key Words and Phrases: error function for complex argument, Voigt function, special functions, function evaluation
CR Categories: 5.12

As a result of an exchange of letters with W. Gautschi it became apparent that the following alterations simplify somewhat the procedure *wofz*:

(i) insert the statement
lambda := *h*2 ↑ *capn*;

between the statements

capn := 6 + 23 × *s*; *nu* := 9 + 21 × *s*

(ii) delete the statement

if *h* > 0 **then** *lambda* := *h*2 ↑ *capn*;

Furthermore, for clarification, a comment could be inserted before the statement *b* := *h* = 0 ∨ *lambda* = 0; namely

comment In the following statement, *lambda* = 0 covers the underflow case when *h* > 0 is very small;

After these slight modifications, the procedure *wofz* was translated into Fortran and extended to the whole z-plane ($z = x+iy$) by means of [1, No. 7.1.11, 12]

$$w(-z) = 2e^{-z^2} - w(z), \quad w(\bar{z}) = \overline{w(-z)}.$$

It was then tested on a CDC 6500 computer at CERN. The tests included the following:

(i) Calculation of the seven examples No. 12–18 for $w(z)$, $erf(z)$, and the Fresnel integral $S_1(z)$ given in [1, No. 7.5]. At least 11 significant digits agreed with the values obtained by

$$w(z) = e^{-z^2}[1-erf(-iz)] = e^{-z^2}erfc(-iz). \qquad (1)$$

The error function $erf(z)$ for complex z in (1) was calculated using Salzer's formula, which is reproduced in the NBS Handbook [1, No. 7.1.29]. This formula requires the computation of $erf(x)$ for real x, which was done with the help of a library program based on the approximation given by Cody [2]. (Note that the correct value of $ImS_1[(\frac{1}{2}+i)\sqrt{2}]$ in example 18 is −0.681620 instead of −0.681619.)

(ii) Calculation of $w(z)$ for

$$z = 4.29 + 10^{-10}p + i(5.33+10^{-10}q)$$

with $p, q = -1, 0, 1$. These values of z lie near the line which separates two branches in the procedure *wofz*. Eight to nine significant digits, corresponding to nine to ten figures after the decimal point, agreed with the values obtained from (1).

(iii) Calculation of $w(z)$ along the diagonal $z = (1+i)u$ for $u = -27(1)100$, 1000, 10000. For $u < 10$, the formula [1, No. 7.9]

$$w[(1 + i)u] = e^{-2iu^2}\left\{1 + (i - 1)\left[C\left(\frac{2u}{\sqrt{\pi}}\right) + iS\left(\frac{2u}{\sqrt{\pi}}\right)\right]\right\} \tag{2}$$

was used for comparison. The Fresnel integrals $C(x)$ and $S(x)$ were computed with a library program based on the Algol procedure *Fresnel* written by Bulirsch [3]. Twelve to fourteen significant digits agreed. For $u > 10$, the results of *wofz* were checked against the asymptotic expansion [1, No. 7.1.23]

$$w(z) \sim \frac{i}{\sqrt{\pi}z}\left(1 + \sum_{n=1}^{\infty} \frac{1.3 \cdots (2n - 1)}{(2z^2)^n}\right) \qquad \left(z \to \infty, -\frac{\pi}{4} < arg(z) < \frac{5\pi}{4}\right). \tag{3}$$

Thirteen to fourteen significant digits agreed.

(iv) Calculation of $w(z)$ along the imaginary axis $x = 0$ for $y = -27(1)100$, 1000, 10000. For $y < 25$, the formula

$$w(iy) = e^{y^2}erfc(y) \tag{4}$$

was used for comparison. The complementary error function $erfc(x)$ was computed by means of a library program based on [2]. For $-27 \leq y \leq -2$ and for $10 \leq y < 25$, 12 to 14 significant digits agreed, whereas for $-2 < y < 10$, ten to thirteen significant digits were found to be in agreement. For $y \geq 25$, the results were checked against the asymptotic expansion (3). Thirteen to fourteen significant digits agreed.

(v) Calculation of $w(x) - e^{-x^2}$ along the real axis $y = 0$ for $x = 0(1)100$, 1000, 10000 using the formula [1, No. 7.9]

$$w(x) - e^{-x^2} = \frac{2i}{\sqrt{\pi}} e^{-x^2} \int_0^x e^{t^2}\, dt = \frac{2i}{\sqrt{\pi}} F(x) \tag{5}$$

for comparison. The Dawson integral $F(x)$ was computed with the help of the rational approximations given by Cody et al. [4]. For $x \leq 7$, 10 to 12 significant digits agreed, whereas for $x > 7$, 13 to 14 significant digits were found to be in agreement.

(vi) Calculation of $w(z)$ for $z = (1+i/\sqrt{3})u$ and $z = (1+i\sqrt{3})u$ for $u = 10^k$, $k = -10(1)4$. For $k \leq 0$, the results were compared with the values obtained from the power series

$$w(z) = \sum_{n=0}^{\infty} \frac{(iz)^n}{\Gamma\left(\frac{n}{2} + 1\right)} \cdot \tag{6}$$

Ten significant digits agreed. For $k > 0$, 13 to 14 significant digits agreed with the values obtained from the asymptotic expansion (3).

(vii) Calculation of $w(z)$ for $z = x + 10^{-8}i$ for $x = 1(1)100$, 1000, 10000. For $x < 5$, the results were compared with the values obtained from formula (1). Six to eight significant digits, corresponding to at least nine to ten decimals, agreed for the real part. However, the accuracy of *wofz* may be higher, since the values from formula (1) are possibly inaccurate. The imaginary part agreed to ten to twelve significant digits. For $x > 5$, the asymptotic expansion (3) was used for comparison. For $6 \leq x \leq 8$, ten to twelve significant digits, and for $x > 8$, thirteen to fourteen significant digits agreed in both the real and imaginary part. For $x = 5$, it was not possible to calculate accurate values for the real part of $w(z)$ either by means of formula (1) or from the asymptotic expansion (3).

References

1. Gautschi, W. Error function and Fresnel integrals. Chap. 7 in Handbook of Mathematical Functions, M. Abramowitz and J.A. Stegun, Eds. NBS Appl. Math. Ser. 55, U.S. Govt. Printing Office, Washington, D.C., 1965.

2. Cody, W.J. Rational Chebyshev approximations for the error function. *Math. Comp. 22* (1968), 631-637.

3. Bulirsch, R. Numerical calculation of the sine, cosine and Fresnel integrals, Handbook Series Special Functions. *Numer. Math. 9* (1967), 380-385.

4. Cody, W.J., Paciorek, K.A., and Thacher, H.C.Jr. Chebyshev approximations for Dawson's integral. *Math. Comp. 24* (1970), 171-178.

ALGORITHM 364
COLORING POLYGONAL REGIONS [Z]

Robert G. Herriot (Recd. 30 Jan. 1967, 31 Oct. 1968 and 2 July 1969)
University of Wisconsin, Computer Science Department, Madison, WI 53706

KEY WORDS AND PHRASES: coloring polygonal regions, coloring planar surfaces, drawing pictures, shading enclosed regions
CR CATEGORIES: 4.9

```
procedure drawarea (x, y, firstpoint, lastpoint, section, numrows,
    numseats, regcolor, paintflag, paintcolor, sgn, dir, edge);
  value firstpoint, lastpoint, numrows, numseats, regcolor,
    paintflag, paintcolor, sgn, dir, edge;
  integer firstpoint, lastpoint, numrows, numseats, regcolor,
    paintcolor, sgn;
  real edge;
  Boolean paintflag, dir;
  real array x, y;
  integer array section;
```

comment This procedure is a part of a large program which produces the card stunts for the Stanford University football game half-times. The initial development was done by L. Breed, L. Tesler, and J. Sauter. The author (a Stanford student at the time) made many further developments on this program which included producing an algorithm for coloring in polygonal regions. Prior to the development of this algorithm, there were many cases which did not work. The larger program takes as input an English description of the stunts and produces as output an image of each flip (similar to a frame in a movie film), as a rectangle that has 45 rows with 77 seats in each row. The main program, which will be considered the driver program for the purpose of the procedure *drawarea*, does all of the handling of the definition of regions and also the printing of the images. It should be mentioned that the procedure *drawarea* in the actual program is just part of a larger procedure and that all of the parameters are global in order to increase efficiency. The purpose of *drawarea* is to take the current regions and draw them in the two-dimensional array *section*, which is to be declared as *section* [1: *numrows*, 1: *numseats*] (the array is 45 by 77 for Stanford). Each completed picture in *section* is then printed and also written out on tape. Another program later takes this tape and processes it to produce an instruction card for each student holding a set of colored cards in the rooters section.

The larger program allows objects of any shape to be defined by a series of x, y-coordinates. It will accept a series of points which are given an identifying name by the user and which can then be used as (1) a group of points, (2) a series of connected line segments, (3) a polygonal region enclosed by the points (with the first and last point connected by a straight line). It also allows ellipses to be defined. Once an object is defined, it can be expanded and contracted in size, rotated about any fixed point, or moved anywhere, including all or partially out of sight. As soon as all objects are in place, the user can ask that an image of the picture be made. Except for polygonal regions, producing the image of these objects is trivial. The procedure *drawarea* is the routine which places the polygonal regions in the array section.

The array *section* is presumed to have a background color associated with it. All objects, which also have an associated color, are then drawn into the array in a specified order so that the objects which are to be superimposed over other objects are drawn last. The procedure *drawarea* takes the coordinates of the point (which may not be integral) from arrays x and y with subscript values ranging from *firstpoint* to *lastpoint* and decides which seats in array *section* will form the left and right boundaries of this new region. After the boundary is determined, the interior must be colored in. The algorithm colors the region by taking each row and then examining each seat from left to right. For optimization, only the area of a minimal circumscribing rectangle is examined. At the beginning of each row the variable *count* is set to *leftcount* [*row*, 0]$-$*rightcount* [*row*, 0], which will be zero unless the object is partially out of sight on the left. Then as long as *count* remains zero, the seat is on the exterior and is not colored. As each seat is encountered, *leftcount* [*row*, *seat*] is added to *count*. When *count* is positive, the seat is in the interior or on a boundary and is colored. After each seat is processed, *rightcount* [*row*, *seat*] is subtracted from *count*. When *count* returns to zero, the seat is an exterior seat and is not colored. In any row it is possible to have the color turned on and off several times. Arrays *leftcount* and *rightcount* contain twice the number of left and right boundaries which pass through each individual seat. These two arrays solve the problem created by having several boundaries passing through one seat.

A further complication to the routine is added by allowing a region to be gradually changing color. Thus each region always has a color (*regcolor*) associated with it, and if the region is being swept with a new color, then *paintflag* is **true** and *paintcolor*, *sgn*, *dir*, and *edge* are used to determine the section of the region which is to be of the new color (*paintcolor*). The roles of the parameters for painting are: *sgn* and *dir* indicate the direction in which the imaginary paintbrush is moving. *dir* = **true** means the direction is horizontal and *dir* = **false** means vertical. *sgn* = -1 means the direction is left or down and *sgn* = 1 means the direction is right or up. *edge* is the row or seat (column) where the new color (*paintcolor*) ends and the old color (*regcolor*) begins. The driver program is expected to change *edge* with each new image so that the region looks as if it is being swept by a new color.

A related algorithm which determines whether a point is inside a polygon is presented in Algorithm 112 [1, 2].

References:

1. Hacker, Richard. Certification of Algorithm 112, Position of point relative to polygon. *Comm. ACM 5* (Dec. 1962), 606.
2. Shimrat, M. Algorithm 112, Position of point relative to polygon. *Comm. ACM 5* (Aug. 1962), 434;

```
begin
  integer row, seat, toprow, rightseat, rit, lef, top, bot, iox, ioy,
    inx, iny, sdx, sdy, j, ix, iy, count;
  real ox, oy, nx, ny, dx, dy, dxdy, const;
  integer array leftcount, rightcount [0: numrows+1,
    0: numseats+1];
  integer procedure max(x, y); value x, y; integer x, y;
    max := if x ≥ y then x else y;
  integer procedure min(x, y); value x, y; integer x, y;
    min := if x ≤ y then x else y;
  toprow := numrows + 1;
  rightseat := numseats + 1;
  for row := 0 step 1 until toprow do
    for seat := 0 step 1 until rightseat do
      leftcount [row, seat] := rightcount [row, seat] := 0;
  ox := x[lastpoint];  rit := left := iox := ox;
  oy := y[lastpoint];  top := bot := ioy := oy;
  comment  Draw the boundary by iterating through the points;
  for j := firstpoint step 1 until lastpoint do
begin
  nx := x[j];  inx := nx;
  ny := y[j];  iny := ny;
  dx := nx − ox;
  dy := ny − oy;
  sdx := if dx < 0 then −1 else 1;
  sdy := if dy < 0 then −1 else 1;
  if ioy = iny then
  begin
    comment  The line is horizontal, or almost so;
    comment  min and max keep the point in the section;
    row := max(min(ioy, toprow), 0);
    seat := max(min(max(iox, inx), rightseat), 0);
    rightcount [row, seat] := rightcount [row, seat] + 1;
    seat := max(min(min(iox, inx), rightseat), 0);
    leftcount [row, seat] := leftcount [row, seat] + 1;
  end horizontal line
  else
  begin
    comment  The line is not horizontal;
    dxdy := dx/dy;
    const := if abs(dx) ≤ abs(dy)
             then ox − dxdy × oy
             else ox − dxdy × (oy − sdx/2) − sdy/2;
    comment  Draw line between two points by stepping
      through each row and determining which seat should be
      marked as the boundary;
    for iy := ioy step sdy until iny do
    begin
      ix := dxdy × iy + const;
      row := max(min(iy, toprow), 0);
      seat := max(min(ix, rightseat), 0);
      comment  Because end points are each processed twice,
        we add only 1 to them instead of the usual 2;
      if dy > 0 then
      begin
        comment  Boundary on right side of area;
        rightcount[row,seat] := rightcount[row,seat]
          + (if iy=ioy∨iy=iny then 1 else 2)
      end
      else
      begin
        comment  Boundary on left side of area;
        leftcount[row,seat] := leftcount[row,seat]
          + (if iy=ioy∨iy=iny then 1 else 2)
      end
    end drawing of line;
  end sloping line;
  comment  Move on to next line segment;
  ox := nx;    iox := ox;
  oy := ny;    ioy := oy;
  comment  Find rectangle which circumscribes the area;
  if rit < iox then rit := iox
  else if lef > iox then lef := iox;
  if top < ioy then top := ioy
  else if bot > ioy then bot := ioy;
end bordering area;
```

```
lef := max(1, lef);  rit := min(rit, numseats);
bot := max(1, bot);  top := min(top, numrows);
comment  Color the area. It is only necessary to look within
  the circumscribing rectangle;
for row := bot step 1 until top do
begin
  count := leftcount [row, 0] − rightcount [row, 0];
  for seat := lef step 1 until rit do
  begin
    count := count + leftcount [row, seat];
      if count > 0 then
        section [row, seat] := if paintflag then
              (if sgn×((if dir then seat else row)−edge) >0
                then
              regcolor
              else paintcolor)
            else regcolor;
      count := count − rightcount [row, seat];
    end coloring of one seat;
  end coloring of one row;
end drawarea;
```

ALGORITHM 365
COMPLEX ROOT FINDING [C5]

H. Bach (Recd. 18 Apr. 1968 and 15 July 1969)
Laboratory of Electromagnetic Theory, Technical University of Denmark, Lyngby, Denmark

KEY WORDS AND PHRASES: downhill method, complex relaxation method, complex iteration, complex equation, transcendental complex equation, algebraic complex equation
CR CATEGORIES: 5.15

Comment. The present subroutine determines, within a certain region, a root of a complex transcendental equation $f(z) = 0$, on which the only restriction is that the function $w = f(z)$ must be analytic in the region considered. The iterative method used, the downhill method, was originally described in [2] and is discussed and modified in [1].

The program uses a complex function subprogram FUNC(Z) for the computation of $f(z)$. From a given complex starting point ZS, the iteration is performed in steps of initial length HS. The iterations stop at the root approximation ZE when either the function value DE at the end point is less than the prescribed minimum deviation DM or when the step length HE has become less than the prescribed minimum step length HM. For reference, the subroutine also returns DS, the function value at the starting point ZS, and N, the number of iterations used. There are thus four input parameters, namely the starting point ZS, the initial step length HS, the minimum step length HM, and the minimum deviation DM.

Acknowledgment. Thanks are due to Mr. Frank Jensen, M.Sc., who helped in the testing of this algorithm.

References:
1. Bach, H. On the downhill method. *Comm. ACM 12* (Dec. 1969) 675–677.
2. Ward, J. A. The downhill method of solving $f(z) = 0$. *J. ACM 4* (Mar. 1957), 148–150.

```
      SUBROUTINE CRF(ZS,HS,HM,DM,FUNC,DS,ZE,HE,DE,N)
C
C  THE SUBROUTINE DETERMINES A ROOT OF A TRANSCEN-
C  DENTAL COMPLEX EQUATION F(Z)=0 BY STEP-WISE ITE-
C  RATION.(THE DOWN HILL METHOD)
C
C  INPUT-PARAMETERS.
C
C  ZS = START VALUE OF Z.(COMPLEX)
C  HS = LENGTH OF STEP AT START.
C  HM = MINIMUM LENGTH OF STEP.
C  DM = MINIMUM DEVIATION.
C
C  SUBPROGRAM.
C
C  FUNC(Z), A COMPLEX FUNCTION SUBPROGRAM FOR THE
C  CALCULATION OF THE VALUE OF F(Z) FOR A COMPLEX
C  ARGUMENT Z.
C
C  OUTPUT-PARAMETERS.
C
C  DS = CABS(FUNC(ZS))=DEVIATION AT START.
C  ZE = END VALUE OF Z.(COMPLEX)
C  HE = LENGTH OF STEP AT END.
C  DE = CABS(FUNC(ZE))=DEVIATION AT END.
C  N  = NUMBER OF ITERATIONS.
C
C  RESTRICTIONS.
C
C  THE FUNCTION W=F(Z) MUST BE ANALYTICAL IN THE
C  REGION WHERE ROOTS ARE SOUGHT.
C
      REAL W(3)
      COMPLEX ZO,ZS,ZE,ZD,ZZ,Z(3),CW,A,V,U(7),FUNC
      U(1)=(1.,0.)
      U(2)=(0.8660254,0.5000000)
      U(3)=(0.0000000,1.0000000)
      U(4)=(0.9659258,0.2588190)
      U(5)=(0.7071068,0.7071068)
      U(6)=(0.2588190,0.9659258)
      U(7)=(-0.2588190,0.9659258)
      H=HS
      ZO=ZS
      N=0
C
C  CALCULATION OF DS.
C
      CW=FUNC(ZO)
      WO=ABS(REAL(CW))+ABS(AIMAG(CW))
      DS=WO
      IF(WO-DM) 18,18,1
    1 K=1
      I=0
    2 V=(-1.,0.)
C
C  EQUILATERAL TRIANGULAR WALK PATTERN.
C
    3 A=(-0.5,0.866)
C
C  CALCULATION OF DEVIATIONS W IN THE NEW TEST POINTS.
C
    4 Z(1)=ZO+H*V*A
      CW=FUNC(Z(1))
      W(1)=ABS(REAL(CW))+ABS(AIMAG(CW))
      Z(2)=ZO+H*V
      CW=FUNC(Z(2))
      W(2)=ABS(REAL(CW))+ABS(AIMAG(CW))
      Z(3)=ZO+H*CONJG(A)*V
      CW=FUNC(Z(3))
      W(3)=ABS(REAL(CW))+ABS(AIMAG(CW))
      N=N+1
C
C  DETERMINATION OF W(NR), THE SMALLEST OF W(I).
C
      IF(W(1)-W(3)) 5,5,6
    5 IF(W(1)-W(2)) 7,8,8
    6 IF(W(2)-W(3)) 8,8,9
    7 NR=1
      GOTO 10
    8 NR=2
      GOTO 10
    9 NR=3
   10 IF(WO-W(NR)) 11,12,12
   11 GOTO (13,14,15),K
   12 K=1
      I=0
C
C  FORWARD DIRECTED WALK PATTERN.
C
      A=(0.707,0.707)
      V=(Z(NR)-ZO)/H
      WO=W(NR)
      ZO=Z(NR)
      IF(WO-DM) 18,18,4
   13 K=2
C  REDUCTION OF STEP LENGTH.
C
      IF(H.LT.HM) GOTO 18
      H=H*0.25
      GOTO 3
   14 K=3
C
C  RESTORATION OF STEP LENGTH.
C
      H=H*4.
      GOTO 2
   15 I=I+1
C
C  ROTATION OF WALK PATTERN.
C
      IF(I-7) 16,16,17
   16 V=U(I)
      GOTO 3
C
C  REDUCTION OF STEP LENGTH.
C
   17 IF(H.LT.HM) GOTO 18
      H=H*0.25
      I=0
      GOTO 2
   18 ZE=ZO
      HE=H
      DE=WO
      RETURN
      END
```

ALGORITHM 366
REGRESSION USING CERTAIN DIRECT PRODUCT MATRICES [G2]

P. J. Claringbold (Recd. 10 May 1968 and 8 July 1969)
Division of Animal Genetics, C.S.I.R.O., P.O. Box 90, Epping, N.S.W., Australia, 2121

KEY WORDS AND PHRASES: analysis of variance, analysis of covariance, regression analysis, experimental design, matrix direct product, projection operator, orthogonal matrix
CR CATEGORIES: 5.14, 5.5

```
procedure regressor (vec, kobs, levs, code, kfac, nfac, ndf);
  value nfac;
  integer kobs, levs, code, kfac, nfac, ndf;
  real vec;
```

comment The mathematical basis of the algorithm which forms the kernel of a very general analysis of variance and covariance procedure (Algorithm 367) is set out in [5, 6]. An overwhelming majority of the experimental designs in [2] may be analyzed in this way. Statistical nomenclature is given in parentheses.

A vector *vec*, of *nobs* elements (*observations*) traced by *kobs*, is replaced by $ndf \leqq nobs$ elements (*regression coefficients*) obtained by the matrix product $C^T \cdot vec$, since the matrix is semiorthogonal. The number of initial elements is implied as the product of the *nfac* values of the variable *levs* which are traced by *kfac*. Values of *code*, similarly traced, specify matrices which enter a direct product [4] to form the transforming matrix C^T (*independent variates transposed*). As *code* takes the values 0, 1, or 2, the matrices selected are I, j, or V, i.e. the unit matrix of order *levs*, the unit vector of *levs* equal elements, or a matrix made up of $levs - 1$ mutually orthogonal unit vectors which are also orthogonal to the previous vector ($V^T \cdot j = 0$ and $V^T \cdot V = I$). A direct product of the transposes of the selected matrices forms the transforming matrix. An example of an actual call is shown to illustrate tracing: *example*: *regressor* (*vec*[*kobs*], *kobs*, *levs*[*kfac*], *code*[*kfac*], *kfac*, *nfac*, *ndf*).

The squared length of the resultant vector (*sum of squares on ndf degrees of freedom*) is equal to the squared length of the projection of the original vector in the subspace spanned by an idempotent symmetric matrix (*idix*) P. Eigenvectors associated with unit eigenvalues of this projection operator [1] comprise the rows of the transforming matrix.

$$l^2 = vec^T \cdot P \cdot vec = vec^T \cdot C \cdot C^T \cdot vec. \qquad (1)$$

The cosine of the angle between two similarly transformed vectors (*correlation coefficient*) is obtained in an analogous manner from a scalar product (*sum of cross products*).

$$l_{vec} l_{wec} \cos(\theta) = vec^T \cdot P \cdot wec. \qquad (2)$$

Prior evaluation of direct products is very wasteful of operations [3], and use is made of an identity which involves ordinary $(\cdot)$ and direct $(\times)$ products:

$$(A \times B \times C) \cdot y = (A \times I \times I) \cdot (I \times B \times I) \cdot (I \times I \times C) \cdot y. \qquad (3)$$

Although shown for a triple product the identity obviously holds for any number of factors. The identity, however, is only valid for square matrices and the rectangular j or V factors must therefore be bordered by zeros to satisfy. In the algorithm multiplication by these zeros is bypassed, and after each transformation the vector is packed ready for the next.

Another identity:

$$(A \times B) \cdot (C \times D) = (A \cdot C) \times (B \cdot D), \qquad (4)$$

implies that the ordinary products in (3) may be taken in any order, since the direct product factors commute. The transformations should therefore be taken in the order which achieves the largest reduction in the number of elements. Since *j-factors* achieve a reduction in the ratio *levs*:1, while *V-factors* merely achieve *levs*:*levs* − 1, the transformations are arranged in descending order of levels for *j-factors* followed by an ascending order of levels for *V-factors*. Transformations requiring the unit matrix are, of course, skipped.

References:

1. Banerjee, K. S. A note on idempotent matrices. *Ann. Math. Statist. 35* (1964), 880–882.
2. Cochran, W. G. and Cox, Gertrude M. *Experimental Designs* (2 Ed.) Wiley, New York, 1957.
3. Good, I. J. The interaction algorithm and practical Fourier analysis. *J. Roy. Statist. Soc.* {B} *20* (1958), 361–373.
4. Marcus, M. Basic theorems in matrix theory. *Nat. Bur. Standards Appl. Mathl Ser. 57* (1960), Washington, D.C.
5. Nelder, J. A. The analysis of randomised experiments with orthogonal block structure. I. Block structure and the null analysis of variance. *Proc. Roy. Soc.* {A} *283* (1965), 147–162.
6. Nelder, J. A. The analysis of randomised experiments with orthogonal block structure. II. Treatment structure and the general analysis of variance. *Proc. Roy. Soc.* {A} *283* (1965), 163–178;

```
begin
  integer ifac, jgo, nlft, nrgt, jfac, jump, ilft, irgt, jumphold, ilev,
    jumpo, jumper, iup, idown, nlev, maxp;
  real x, v;
  integer array ranks[1:nfac];
  maxp := ndf := 1;
  for kfac := 1 step 1 until nfac do
  begin
    comment Transmit levels and determine largest factor;
    ranks[kfac] := nlev := levs;  ndf := ndf×nlev;
    if nlev > maxp then maxp := nlev
  end with degrees of freedom set in null case;
  maxp := −(maxp+1);
  for jgo := 1, 2 do
  begin
    comment Averaging before differencing transformations;
mfac:
    begin
      comment Search for best remaining factor;
      nlev := maxp;  ifac := 0;
      for kfac := 1 step 1 until nfac do
      begin
        ilev := (3−2×jgo) × ranks[kfac];
        if code = jgo ∧ ranks[kfac] = levs ∧ ilev > nlev then
        begin
          nlev := ilev;  ifac := kfac
        end if a better factor
      end search;
      if ifac > 0 then
```

```
begin
  comment Process a factor;
  kfac := ifac; nlev := levs; nlft := nrgt := 1;
  for jfac := 1 step 1 until nfac do
    if ifac ≠ jfac then
    begin
      comment Determine orders of unit matrices to left
        and right;
      if jfac < ifac then nlft := nlft × ranks[jfac]
      else nrgt := nrgt × ranks[jfac]
    end products;
  begin
    comment Evaluate normalization constants;
    array root[jgo : if jgo=1 then 1 else nlev];
    if jgo = 1 then root[1] := sqrt(1/nlev)
    else
    for ilev := 2 step 1 until nlev do
      root[ilev] := sqrt(1/(ilev×(ilev−1)));
    comment Begin transformation of vector;
    jump := 0;
    comment Loop over all combinations to the left;
    for ilft := 1 step 1 until nlft do
    begin
      jump := jump + 1;
      comment Loop over all combinations to the right;
      for irgt := 1 step 1 until nrgt do
      begin
        jumphold := jump; jump := jump − nrgt; x := 0;
        comment Loop over active factor;
        for ilev := 1 step 1 until nlev do
        begin
          comment Form sum;
          jumpo := jump; kobs := jump := jump + nrgt;
          if jgo = 2 ∧ ilev > 1 then
          begin
            comment Form difference when appropriate;
            v := vec; kobs := jumpo;
            vec := (x−(ilev−1)×v)×root[ilev]
          end now do sum;
          kobs := jump; x := x + vec
        end sum and difference loop;
        if jgo = 1 then
        begin
          comment Insert normalized average;
          kobs := jumphold; vec := x × root[1]
        end insertion;
        jumper := jump; jump := jumphold + 1
      end loop over all combinations to the right;
      jump := jumper;
    end loop over all combinations to the left
  end block;
  iup := nrgt × nlev; idown := if jgo = 1 then nrgt else
    iup − nrgt;
  for ilft := 2 step 1 until nlft do
  begin
    comment Compact vector;
    for irgt := 1 step 1 until nrgt do
    for ilev := 2 step 1 until nlev do
      if ilev < 3 ∨ jgo = 2 then
      begin
        kobs := iup := iup + 1; v := vec;
        kobs := idown := idown + 1; vec := v
      end within block moves;
    iup := if jgo = 1 then iup + (nlev−1) × nrgt else
      iup + nrgt
  end block moves;
            comment Adjust dimensions of pseudoarray;
            ranks[ifac] := if jgo = 1 then 1 else nlev − 1;
            ndf := idown;
            go to mfac
          end
          else go to end jgo
        end labeled compound statement;
end jgo:
  end loop over factor types
end regressor
```

ALGORITHM 367
ANALYSIS OF VARIANCE FOR BALANCED EXPERIMENTS [G2]

P. J. Claringbold (Recd. 27 May 1968 and 8 July 1969)
Division of Animal Genetics, C.S.I.R.O., P.O. Box 90, Epping, N.S.W., Australia, 2121

KEY WORDS AND PHRASES: analysis of variance, analysis of covariance, regression analysis, experimental design, balanced experiment, missing data, interblock estimate, intrablock estimate

CR CATEGORIES: 5.14, 5.5

integer procedure *balanced anova* (*y, missing y, x, fixed effect, estimate, error level, error code, all y, all x, length y, length x, pooled beta, se beta, normalized beta, error, df total, df error, tolcor, tolength, tolmpss, ispace, nspace, ires, jres, nres, itrt, ntrt, iobs, nobs, ifac, nfac, max cycle, check diagonality, projector, putpy, getpy, putpx, getpx*);

value *tolcor, tolength, tolmpss, nspace, nres, ntrt, nobs, nfac, max cycle, check diagonality*;

real *y, x, all y, all x, length y, length x, pooled beta, se beta, normalized beta, error, tolcor, tolength, tolmpss*;

integer *error level, error code, df total, df error, ispace, nspace, ires, jres, nres, itrt, ntrt, iobs, nobs, ifac, nfac, max cycle*;

Boolean *missing y, fixed effect, estimate, check diagonality*;

procedure *projector, putpy, getpy, putpx, getpx*;

comment The algorithm provides analyses of variance, covariance, and regression for data collected according to a wide variety of experimental designs. The vector of elements comprising either a response (*y or dependent*) or a treatment (*x or independent*) variate forms a conceptual complete array of *nfac* dimensions. The implied subscripts are a set of discrete variables which define an error classification. Designs of this type include the *fully randomized, randomized block, incomplete block, split* (*to any order*) *plot, Latin* (*and higher*) *squares, lattices*, et cetera, and make up the overwhelming majority in use [3]. By means of an appropriate transformation the frequency data of contingency tables may be processed to provide partitions of chi-square [1]. A comprehensive account of the mathematical basis is given in [4, 5].

In this implementation extensive use is made of the *call-by-name* facility so that generators and routines involving auxiliary store may freely be used for all input variables. Usually data sets are quite small and storage of intermediate quantities within the immediate access store is possible. In the following notes on the formal parameters relevant tracer variables are shown in brackets. An arrow (→) indicates that the variable is used only as a source of information.

balanced anova: If the projection of *x-variate* numbered *jtrt* has a correlation coefficient exceeding *tolcor* with the projection of *x-variate* numbered *ktrt* in subspace *ispace* of the design, then abnormial termination is forced with *balanced anova* $= 10^6 \times$ *ispace* $+ 10^3 \times$ *jtrt* $+$ *ktrt*. Zero is returned as the value of the procedure in the case of normal termination. Note that this time-consuming check of the *balance* of the treatment model with respect to the error model is only performed if *check diagonality* is set **true**.

y, missing y (*ires, iobs*) → : The *y-variate* generator or array must provide trial values, e.g. the average of present elements for the variate, for any missing data. These elements are flagged by **true** in the **Boolean** *missing y* which may take the form of an expression in terms of *ires, iobs*, and **integer** constants.

x (*itrt, iobs*) → : A complete specification of the orthogonal decomposition of the total sum of squares (and products) using polynomials or some other form of contrast representation is required. In the case of treatment classifications (for example *factorial experiment*) the *x-variate* values may be generated as a direct product (or as a selection of elements from such a matrix) of a number of small contrast matrices, i.e. orthogonal matrices with first column having elements greater than zero (usually constant).

fixed effect (*itrt, ispace*) → : By setting this variable **true** the flagged regression coefficients, i.e. *beta* number *itrt* in estimation subspace number *ispace*, are declared to be error free or invariants. In most practical cases this facility is only relevant to the constant term of the regression model.

estimate (*itrt, ispace*) → : By setting this variable **false** the flagged regression coefficients are declared to be zero and are not estimated in the indicated subspaces. Usually this facility is not required, and the constant **true** is used as actual parameter.

error level (*ifac*) → : The variable sets the number of levels of the error classifications. If it is assumed that the conceptual subscripts have unit lower bounds, then the upper bounds are set. Variates (traced by *iobs*) must be in lexical order by the implied subscripts, and use of a permutation array or function may be required to achieve this end.

error code (*ifac, ispace*) → : Error sources of variation (estimation or error subspaces) are specified by integer codes 0, 1, or 2. The codes could be generated by means of a procedure which interpreted a string of input characters denoting the *error structure* of the experimental design, see [4, 5]. A set of *nfac* integers specifies a projection operator which spans a subspace. The operator is formed as the direct product of (0) *identity matrix* I, (1) *averaging matrix* J, or (2) *differencing matrix* $K = I - J$. Every element of the averaging matrix is equal to the reciprocal of the order,

e.g.: $2, 0, 1, 2, 1 \leftrightarrow K_1 \times I_2 \times J_3 \times K_4 \times J_5 = P_i$, *say.*

It is required that the error subspaces be mutually orthogonal, $P_iP_j = \delta_{ij}P_i$.

Code Sets for Some Common Designs

Design	Codes							P_1+P^2
Fully randomized	1	2						0
Randomized or incomplete block	11	21	02					01
Split plot	111	211	021	002				011
Split split plot	1111	2111	0211	0021	0002			0111
Square or rectangle	11	21	12	22				01
Revlicated square or rectangle	111	211	021	012	022			011
Three-way crossed error	111	211	121	112	221	212	122 222	011

In certain circumstances it may be desired to work *mod*$(J{\times}J{\times}\cdots{\times}J)$, that is the *y-variates* are adjusted to have zero mean. In this case the first code is omitted from the analysis. Usually it is convenient to pool the subspaces defined by $J \times J \times \cdots \times J$ and $K \times J \times \cdots \times J$ yielding (by addition) $I \times J \times \cdots \times J$, and if this is required the first two columns of the table are replaced by the rightmost auxiliary column.

all y [*ires*], *all x* [*itrt*], *length y* [*ires, ispace*], *length x* [*itrt, ispace*]: The lengths of the *y*, *x*, projected *y*, and projected *x* vectors are returned. Null variates (which have zero length) should be indicated in, or excluded from, analysis of variance tables (et cetera) derived from an activation of the procedure.

pooled beta, se beta [*ires, itrt*]: The weighted mean regression coefficient relating *y-variate* number *ires* to *x-variate* number *itrt* is returned in *pooled beta*, and the standard error of the estimate in *se beta*.

normalized beta [*ires, itrt, ispace*]: Within each subspace the regression coefficients are scaled so that it may be assumed that the sum of squares of each (nonnull) projected *x-variate* is unity. The *dyad* obtained by forming all pairwise products over the tracer *ires* (fixing the other tracers) is a single degree of freedom contribution due to treatment (*x-variate*) number *itrt* to subspace number *ispace* of the analysis of variance (and covariance if $nres > 1$).

error [*ires, jres, ispace*]: For each subspace an error covariance matrix is computed. This is the only variable bearing the tracer *jres* which is constrained so that $jres \leqq ires$. The calling program may make provision to pack the matrices in triangular form using a subscript function: $pack[ires] + jres$, where $pack[ires] = (ires \times (ires-1)) \div 2$.

df total, df error [*ispace*]: The variables return the total and error degrees of freedom for each subspace.

tolcor: If the activation calls for a check of the orthogonality of projected *x-variates*, then this constant sets the value of the correlation coefficient, which should not be exceeded in the test.

tolength: A projected vector is assigned zero length if the ratio of the computed length to that of the unprojected vector, multiplied by the square root of the ratio of the number of observations to degrees of freedom of the subspace, fails to exceed this criterion.

tolmpss: As a single measure of all missing data a sum of squares is computed. If the ratio of the absolute value of the difference between this sum and that of the previous iteration (or 0), to the current sum, fails to exceed this constant, no further iterations are made.

ispace, nspace, ires, jres, nres, itrt, ntrt, iobs, nobs, ifac, nfac: The identifiers with initial letter *i* or *j* are tracers mnemonically related to the remaining identifiers which define the number of subspaces, *y-variates*, *x-variates*, observations and error factors, respectively.

max cycle: An upper limit to the number of iterations required for the convergence of estimates of missing data is provided by this parameter.

check diagonality: If this parameter is **true** then the projected *x-variates* are checked for orthogonality. While computing time is saved by the opposite setting, incorrect results are computed if an invalid assumption of orthogonality is made.

projector: In order to compute the consequences of projection of variates, a choice between at least two procedures is made: $P \cdot x = C \cdot C^T \cdot x$ or $C^T \cdot x$. The idempotent symmetric projection operator P (see [4, 5]), or the rectangular matrix made up of the eigenvectors corresponding with unit eigenvalues (see [2]) is used. The second alternative is preferred since the transforming matrix is then thin, and Algorithm 366 is an implementation of this approach.

putpy, getpy, putpx, getpx: These procedures are concerned with the transmission of transformed variates between arrays internal to the algorithm and auxiliary store. While immediate access store may be used as auxiliary store with small problems, backing media such as magnetic drum, disk, or tape are required for large problems. The procedure *putpy* transmits all *nelm* elements of a transformed *y-variate* to auxiliary store, while *getpy* performs the reverse transmission. Similar actions on the *x-variates* are carried out by the other two procedures. All four routines have similar calling sequences: (*vec*[*ielm*], *ielm, nelm, ivar, ispace*), where *vec* identifies the vector to be moved, *ielm* traces the elements of the vector, *nelm* (returned by *projector*) specifies the number of elements to be moved, *ivar* gives the variate number, and *ispace* gives the subspace number. The elements to be moved are in the leading position in *vec*, and an appropriate instruction begins **for** *ielm* := 1 **step** 1 **until** *nelm* **do**. The last two formal parameters may be used to index an array listing the starting positions of the vectors in auxiliary storage.

References:

1. Claringbold, P. J. The use of orthogonal polynomials in the partition of chi-square. *Austral. J. Statist. 3* (1961), 48–63.
2. Claringbold, P. J. Algorithm 366. Regression using certain direct product matrices. *Comm. ACM 12* (Dec. 1969), 687–688.
3. Cochran, W. G., and Cox, Gertrude M. Experimental Designs (Ed. 2). Wiley, New York, 1957.
4. Nelder, J. A. The analysis of randomised experiments with orthogonal block structure. I. Block structure and the null analysis of variance. *Proc. Roy. Soc.* {*A*} *283* (1965), 147–162.
5. Nelder, J. A. The analysis of randomised experiments with orthogonal block structure. II. Treatment structure and the general analysis of variance. *Proc. Roy. Soc.* {*A*} *283* (1965), 163–178;

```
begin
  array yy, xx[1:nobs];  real s, t, v, ssmp;
  integer i cycle, ndf, jtrt, ktrt, kres, nelm, nmis;
  real procedure sigma (x, i, n);
    value n;
    real x;  integer i, n;
begin
  real xx;  xx := 0;
  for i := 1 step 1 until n do xx := xx + x;
  sigma := xx
end sigma;
comment  Count missing data items;
nmis := 0;  ssmp := 0;
for ires := 1 step 1 until nres do
for iobs := 1 step 1 until nobs do
  if missing y then nmis := nmis + 1;
begin
  comment  Get space for estimates of missing data;
  array y missing[1 : if nmis=0 then 1 else nmis];
  comment  Set up loop for missing data iteration;
  for i cycle := 1 step 1 until max cycle do
  begin
    comment  Analyze data in various error subspaces;
    for ispace := 1 step 1 until nspace do
    begin
      comment  Determine subspace degrees of freedom;
      if i cycle = 1 then
      begin
        comment  Only compute degrees of freedom once;
        ndf := 1;
        for ifac := 1 step 1 until nfac do
          ndf := ndf × (if error code=0 then error level
              else if error code=1 then 1 else error level−1);
        df total := ndf
      end
      else ndf := df total;
      comment  Project response vectors;
      nmis := 0;
```

```
for ires := 1 step 1 until nres do
begin
  comment Fetch a vector, and possibly fit missing
    data;
  for iobs := 1 step 1 until nobs do
    if missing y then
    begin
      nmis := nmis + 1;
      if ispace = 1 then y missing[nmis] := if i cycle = 1
        then y
          else sigma (pooled beta×x, itrt, ntrt);
      yy[iobs] := y missing[nmis]
    end
    else yy[iobs] := y;
  if ispace = 1 then all y := sqrt(sigma(yy[iobs]↑2, iobs,
      nobs));
  projector(yy[iobs], iobs, error level, error code, ifac, nfac,
    nelm);
  jres := ires;
  error := sigma(yy[iobs]↑2, iobs, nelm);
  length y := if sqrt((error×nobs)/ndf)/all y > tolength
      then sqrt(error) else 0;
  putpy(yy[iobs], iobs, nelm, jres, ispace);
  for jres := 1 step 1 until ires − 1 do
  begin
    comment Determine sums of cross products;
    getpy(xx[iobs], iobs, nelm, jres, ispace);
    error := sigma(yy[iobs]×xx[iobs], iobs, nelm)
  end cross products
end dependent variates;
comment In the first cycle project treatment vectors;
if i cycle = 1 then
for jtrt := 1 step 1 until ntrt do
  if estimate then
  begin
    comment Only work on variates included in regression;
    itrt := jtrt;
    for iobs := 1 step 1 until nobs do xx[iobs] := x;
    if ispace = 1 then all x := sqrt(sigma(xx[iobs] ↑ 2,
      iobs, nobs));
    projector(xx[iobs], iobs, error level, error code, ifac, nfac,
      nelm);
    t := sigma(xx[iobs]↑2, iobs, nelm);
    s := length x := if sqrt((t×nobs)/ndf)/all x > tolength
        then sqrt(t) else 0;
    if s > 0 then
    begin
      comment Null variates are skipped;
      putpx(xx[iobs], iobs, nelm, itrt, ispace);
      if check diagonality then
      for ktrt := 1 step 1 until jtrt − 1 do
        if estimate then
        begin
          comment Orthogonality checked for variates
            in regression;
          itrt := ktrt; v := length x;
          if v > 0 then
          begin
            comment Null variates are skipped;
            getpx(yy[iobs], iobs, nelm, itrt, ispace);
            if abs(sigma(xx[iobs]×yy[iobs], iobs, nelm))/
              (s×v) > tolcor then
            begin
              comment Force termination since excessive correlation;
              balanced anova := 1000 × (1000×ispace+
                jtrt) + ktrt;
              go to exit
            end large correlation
          end if secondary variate has projection
        end secondary variate loop
    end if primary variate has projection
  end primary variate loop;
comment Compute normalized regression coefficients;
for itrt := 1 step 1 until ntrt do
  if length x > 0 ∧ estimate then
  begin
    comment Skip null or not in regression independent
      variates;
    ndf := ndf − 1;
    getpx(xx[iobs], iobs, nelm, itrt, ispace);
    for ires := 1 step 1 until nres do
      if length y > 0 then
      begin
        comment Skip null dependent variates;
        getpy(yy[iobs], iobs, nelm, ires, ispace);
        normalized beta := sigma(xx[iobs]×yy[iobs], iobs,
          nelm)/length x
      end
      else normalized beta := 0
  end
  else for ires := 1 step 1 until nres do normalized beta
    := 0;
df error := ndf;
comment Reduce sums of squares and products for
  regression;
for itrt := 1 step 1 until ntrt do
  if length x > 0 ∧ estimate then
  begin
    for kres := 1 step 1 until nres do
    for jres := 1 step 1 until kres do
    begin
      ires := jres; s := normalized beta;
      ires := kres; error := error − s × normalized beta
    end dyad reduction loops
  end normalized regression coefficient computation;
comment Determine true regressions and information;
for ires := 1 step 1 until nres do
begin
  for jres := 1 step 1 until ires do
  error := if length y = 0 ∨ ndf = 0 then 0 else error/ndf;
  jres := ires;
  for itrt := 1 step 1 until ntrt do
  begin
    comment Clear areas at start;
    if ispace = 1 then pooled beta := se beta := 0;
    if estimate then
    begin
      comment Set information as unity for fixed
        effects;
      t := if fixed effect ∧ length x > 0 then 1 else
      if ndf = 0 then 0 else length x↑ 2/ (if error=0
        then 1 else error);
      se beta := se beta + t;
      pooled beta := pooled beta + t × (if length x=0
        then 0 else normalized beta/length x)
    end of addition to pools
  end independent variate loop
end dependent variate loop
end error subspace loop;
for ires := 1 step 1 until nres do
for itrt := 1 step 1 until ntrt do
```

```
        if se beta > 0 then
        begin
          comment  Compute weighted means and standard
            errors;
          pooled beta := pooled beta/se beta;
          se beta := sqrt(1/se beta)
        end average;
      if nmis > 0 then
      begin
        comment  Check convergence of missing items;
        s := sigma(y missing[iobs]↑2, iobs, nmis);
        if abs(s−ssmp)/s > tolmpss then ssmp := s
        else go to finish
      end missing data convergence test
    end cycle;
finish:  balanced anova := 0;
exit:
  end block
end balanced anova
```

ALGORITHM 368
NUMERICAL INVERSION OF LAPLACE TRANSFORMS [D5]

HARALD STEHFEST* (Recd. 29 July 1968, 14 Jan. 1969 and 24 July 1969)
Institut f. angew. Physik, J. W. Goethe Universität, 6000 Frankfurt am Main, W. Germany

* The work forms part of a research program supported by the Bundesministerium für wissenschaftliche Forschung and the Fritz ter Meer-Stiftung.

KEY WORDS AND PHRASES: Laplace transform inversion, integral transformations, integral equations
CR CATEGORIES: 5.15, 5.18

procedure *Linv*(P, N, T, Fa, V, M);
  value N, T;
  integer M, N; **real** T, Fa; **array** V; **real procedure** P;
comment If a Laplace transform $P(s)$ is given in the form of a real procedure, *Linv* produces an approximate value Fa of the inverse $F(t)$ at T. Fa is evaluated according to

$$Fa = \frac{\ln 2}{T} \sum_{i=1}^{N} V_i \, P\left(\frac{\ln 2}{T} i\right).$$

N must be even. Since the V_i depend on N only, in case of repeated procedure calls with the same N the array V is to be evaluated only once. That is why the formal parameter M has been introduced: that part of the algorithm which computes the V_i is run through only if $M \neq N$, and after every call of *Linv* M equals N. At the first call M may be any integer different from N.

The calculation method originates from Gaver [2], who considered the expectation of $F(t)$ with respect to the probability density

$$f_n(a, t) = a \frac{(2n)!}{n!(n-1)!} (1 - e^{-at})^n e^{-nat}, \qquad a > 0:$$

$$\bar{F}_n = \int_0^\infty F(t) f_n(a, t) \, dt \tag{1}$$
$$= a \frac{(2n)!}{n!(n-1)!} \sum_{i=0}^{n} \binom{n}{i} (-1)^i P((n+i)a).$$

$f_n(a, t)$ has the following properties:

1. $\int_0^\infty f_n(a, t) \, dt = 1,$
2. modal value of $f_n(a, t) = \ln 2/a$
3. $\mathrm{var}(t) = 1/a^2 \sum_{i=0}^{n} 1/(n+i)^2.$

They imply that $\bar{F}_n$ converges to $F(\ln 2/a)$ for $n \to \infty$. $\bar{F}_n$ has the asymptotic expansion [2]

$$\bar{F}_n \sim F\left(\frac{\ln 2}{a}\right) + \frac{\alpha_1}{n} + \frac{\alpha_2}{n^2} + \frac{\alpha_3}{n^3} + \cdots.$$

For a given number N of P-values a much better approximation to $F(\ln 2/a)$ than $\bar{F}_{N-1}$ is attainable, and that by linear combination of $\bar{F}_1, \bar{F}_2, \cdots, \bar{F}_{N/2}$: requiring

$$\sum_{i=1}^{K} x_i(K) \frac{1}{(N/2 + 1 - i)^k} = \delta_{k0},$$
$$k = 0, 1, \cdots, K-1, \quad K \leq N/2,$$

we find

$$x_i(K) = \frac{(-1)^{i-1}}{K!} \binom{K}{i} i \, (N/2 + 1 - i)^{K-1}$$

and thus

$$\sum_{i=1}^{K} x_i(K) \bar{F}_{N/2+1-i} = F\left(\frac{\ln 2}{a}\right) + (-1)^{k+1} \alpha \frac{(N/2 - K)!}{(N/2)!} + o\left(\frac{(N/2 - K)!}{(N/2)!}\right).$$

Setting $K = N/2$, $a = \ln(2)/T$, and using (1) we get the expression the procedure evaluates:

$$Fa = \sum_{i=1}^{N/2} x_i(N/2) \bar{F}_{N/2+1-i} = \frac{\ln 2}{T} \sum_{i=1}^{N} V_i \, P\left(\frac{\ln 2}{T} i\right)$$

with

$$V_i = (-1)^{N/2+i} \sum_{k=\left[\frac{i+1}{2}\right]}^{Min(i, N/2)} \frac{k^{N/2+1} (2k)!}{(N/2-k)! k! (k-1)! (i-k)! (2k-i)!}.$$

(The method of "extrapolation to the limit," which Gaver [2] used, leads to less accurate results for the same N, because not so many powers of n cancel out. Moreover, with this method N must be a power of 2, so that in general one cannot make the best use of the available computer precision.)

Theoretically Fa becomes the more accurate the greater N. Practically, however, rounding errors worsen the results if N becomes too large, because V_i with greater and greater absolute values occurs. (This reflects the unboundedness of the inverse Laplace operator.) For given $P(s)$ and T the N at which the accuracy is maximal increases with the number of significant figures used. For fixed computer precision the optimum value of N is the smaller, i.e. the maximum accuracy is the greater, the faster $\bar{F}_n$ (see eq. (1)) converges to $F(T)$. In the following the term "smooth" is used to express that the rate of convergence is sufficiently great. An oscillating $F(t)$ certainly is not smooth enough unless the wavelength of the oscillations is large compared with the half-width of the peak which $f_{N/2}(\ln 2/T, t)$ has at T. No accurate results are to be expected, too, if $F(t)$ has discontinuities near T. If $F(t)$ behaves equally in the neighborhood of two different T-values the result at the smaller T-value will be the better one, because the peak of $f_n(\ln 2/T, t)$ broadens as T increases.

The only way to sharpen these qualitative statements is to apply *Linv* to many Laplace transforms the inverses of which are known. This was done with 50 transforms. The numbers of significant figures used ranged from 8 to 17 (IBM 7094, single and double precision, CDC 3300). The T-values lay between 0 and 50. It was found that with increasing N the number of correct figures first increases nearly linearly and then, owing to the rounding errors, decreases linearly. The optimum N is approximately proportional to the number of digits the machine is working with. Table I was calculated using 8-digit arithmetic and $N = 10$.

TABLE I

T	$F(T)$	Fa	$F(T)$	Fa
	$F(t) = \frac{1}{\sqrt{\pi t}}$, $P(s) = \frac{1}{\sqrt{s}}$		$F(t) = -C - \ln(t)$, $P(s) = \ln(s)/s$	
1.0	0.56419	0.56555	−0.57722	−0.57782
2.0	0.39894	0.39912	−1.27036	−1.27084
3.0	0.32574	0.32655	−1.67583	−1.67544
4.0	0.28209	0.28278	−1.96351	−1.96392
5.0	0.25231	0.25174	−2.18665	−2.18727
6.0	0.23333	0.22989	−2.36898	−2.36870
7.0	0.21324	0.21322	−2.52313	−2.52270
8.0	0.19947	0.19956	−2.65666	−2.65740
9.0	0.18806	0.18814	−2.77444	−2.77390
10.0	0.17841	0.17796	−2.87980	−2.88091
	$F(t) = t^3/6$, $P(s) = 1/s^4$		$F(t) = e^{-t}$, $P(s) = 1/(s+1)$	
1.0	0.16667	0.16568	0.36788	0.36798
2.0	1.33333	1.32543	0.13534	0.13557
3.0	4.50000	4.47354	0.04979	0.05043
4.0	10.66667	10.60342	0.01832	0.01849
5.0	20.83333	20.70845	0.00674	0.00640
6.0	36.00000	35.78832	0.00248	0.00195
7.0	57.16667	56.82535	0,00091	0.00036
8.0	85.33333	84.82735	0.00034	−0.00006
9.0	121.50000	120.78473	0.00012	−0.00047
10.0	166.66667	165.66759	0.00005	−0.00020
	$F(t) = \sin(\sqrt{2t})$, $P(s) = \sqrt{\frac{\pi}{2s^3}}\, e^{-1/(2s)}$		$F(t) = L_3(t)$, $P(s) = \frac{(s-1)^3}{s^4}$	
1.0	0.98777	0.98775	−0.66667	−0.66533
2.0	0.90930	0.91001	−0.33333	−0.32531
3.0	0.63816	0.63826	1.00000	1.02575
4.0	0.30807	0.30968	2.33333	2.39533
5.0	−0.02068	−0.02119	2.66667	2.78844
6.0	−0.31695	−0.31927	1.00000	1.21092
7.0	−0.56470	−0.57254	−3.66667	−3.32956
8.0	−0.75680	−0.76869	−12.33333	−11.82953
9.0	−0.89168	−0.91049	−26.00000	−25.28393
10.0	−0.97128	−0.98949	−45.66667	−44.88511

With double precision arithmetic and $N = 18$ the number of correct figures doubles. The chosen N-values are about the optimum N for all functions of the table. Evaluating an unknown function from its Laplace transform, one should, nevertheless, compare the results for different N, to see whether the function is smooth enough, what accuracy can be reached, and what the optimum N is. Even then it is risky to rely solely on the results of *Linv*. One ought to be sure a priori that the unknown function $F(t)$ has not any discontinuities, salient points, sharp peaks, or rapid oscillations. Moreover, the accuracy should be checked by employing other inversion techniques.

The inverses of the 50 test functions were also evaluated according to the inversion technique of Bellman et al. [1], which is based on the approximation of $F(t)$ by a polynomial in e^{-t}. It appeared that the algorithm *Linv* generally produces better results, i.e. the condition "$F(t)$ is everywhere smooth (in the sense described above)" is less restrictive than the condition "$F(-\ln(r))$ can be well approximated by a polynomial in $r = e^{-t}$ for $0 \leq r \leq 1$". The evaluation of the function $F(t) = t^2/2$ from its Laplace transform $P(s) = 1/s^3$ illustrates the difference between the two conditions: using *Linv* the inverse is correct within 0.1 percent, using the inversion technique described in [1] errors of hundreds of percents occur ($N = 10$, $0.1 < T < 10$).

The algorithm was successfully applied to renewal equations, differential-difference equations, and systems of partial differential equations. Reference [1] includes many other problems to which the algorithm can be applied.

REFERENCES:

1. BELLMAN, R. E., KALABA, R. E., AND LOCKETT, J. *Numerical Inversion of the Laplace Transform*. American Elsevier, New York, 1966.
2. GAVER, D. P. Observing stochastic processes, and approximate transform inversion. *Oper. Res. 14*, 3 (1966), 444-459;

```
begin
  integer i, ih, k, Nh, sn;  real a;  array G[0:N], H[1:N/2];
  if M = N then go to C;
  G[0] := 1;  Nh := N/2;
  for i := 1 step 1 until N do G[i] := G[i−1] × i;
  H[1] := 2/G[Nh−1];
  for i := 2 step 1 until Nh do
    H[i] := i ↑ Nh × G[2×i]/(G[Nh−i]×G[i]×G[i−1]);
  sn := 2 × sign (Nh−Nh÷2×2) − 1;
  for i := 1 step 1 until N do
  begin
    V[i] := 0;
    for k := (i+1) ÷ 2 step 1 until if i < Nh then i else Nh do
      V[i] := V[i] + H[k]/(G[i−k]×G[2×k−i]);
    V[i] := sn × V[i];
    sn := −sn
  end;
  M := N;
C: Fa := 0;  a := ln(2)/T;
  comment ln(2) should be replaced by its actual value
    0.69314... ;
  for i := 1 step 1 until N do
    Fa := Fa + V[i] × P(i×a);
  Fa := a × Fa
end
```

REMARK ON ALGORITHM 368 [D5] NUMERICAL INVERSION OF LAPLACE TRANSFORMS [Harald Stehfest, *Comm. ACM 13* (Jan. 1970),47]

HARALD STEHFEST (Recd. 6 May 1970)
Institut f. angew. Physik, J. W. Goethe-Universität
6000 Frankfurt a.M., W. Germany

KEY WORDS AND PHRASES: Laplace transform inversion, integral transformations, integral equations
CR CATEGORIES: 5.15, 5.18

Some errors have crept into the comment of the procedure after proof-reading:
The formula following "and thus" should read

$$\sum_{i=1}^{K} x_i(K) F_{N/2+1-i} = F\left(\frac{\ln 2}{\alpha}\right) + (-1)^{K+1} \alpha_K \frac{(N/2-K)!}{(N/2)!} + o\left(\frac{(N/2-K)!}{(N/2)!}\right).$$

The formula following "with" should read

$$V_i = (-1)^{N/2+i} \sum_{k=\left[\frac{i+1}{2}\right]}^{Min(i,N/2)} \frac{k^{N/2}(2k)!}{(N/2-k)!k!(k-1)!(i-k)!(2k-i)!}.$$

ALGORITHM 369
GENERATOR OF RANDOM NUMBERS SATISFYING THE POISSON DISTRIBUTION [G5]

HENRY E. SCHAFFER* (Recd. 27 Jan. 1969 and 16 July 1969)
North Carolina State University, Genetics Department,
Raleigh, NC 27607

* This work was supported by Grants PR-00011 and GM-11546 of the National Institutes of Health.

KEY WORDS AND PHRASES: Poisson distribution, random number generator
CR CATEGORIES: 5.5

```
integer procedure poissrn (lambda);
  value lambda; real lambda;
comment At each call this procedure returns an observation
  from a Poisson distribution with parameter lambda. The rejection
  method discussed by Kahn [1] is used. It requires an average
  of lambda + 1 (pseudo) random numbers (uniformly distributed
  on the 0, 1 interval) per call. For efficiency the random
  number generator should be coded in-line.
    This procedure is especially suitable when a small number of
  random numbers are needed from each of a large number of
  different Poisson distributions. This can occur when the Poisson
  parameter used in each call is itself chosen according to some
  probability distribution. Algorithm 342 [2] is more efficient for
  repeated use of the same value of the Poisson parameter.
    A value of −1 is returned to signal a value of lambda which
  is not positive. A value of −2 is returned to signal a value of
  lambda which is too large for the significance of the computer.
    I thank the referee for his suggestions and comments.
  REFERENCES:
1. KAHN, H. Applications of Monte Carlo. RM-1237-AEC, Rand
     Corp. 1956 (revised version).
2. SNOW, R. H. Algorithm 342, Generator of random numbers
     satisfying the Poisson distribution. Comm. ACM 11 (Dec.
     1968), 819;
if lambda ≤ 0.0 then poissrn := −1
else
begin
real z;
z := exp (−laamdb);
  if z = 0.0 then poissrn := −2
  else
  begin
    real t; integer k;
    real procedure random;
    begin
      comment The body of this procedure must be provided
        by the user to generate the uniformly distributed random
        numbers required by poissrn. The random number generator
        is placed here rather than called as a global procedure
        to decrease the time taken to obtain each random
        number. For the same reason a fast generator should be
        chosen. It is also important that this generator should
        have negligible serial correlation;
      ⟨procedure body⟩;
    end random;
    k := 0;  t := 1.0;
    for t := t × random while t > z do k := k + 1;
    poissrn := k
  end
end poissrn
```

ALGORITHM 370
GENERAL RANDOM NUMBER GENERATOR [G5]

Edgar L Butler (Recd. 20 June 1969 and 11 Aug. 1969)
Texas A & M University, College Station, TX 77840

KEY WORDS AND PHRASES: random number generator, probability density function, transformation, cumulative density function
CR CATEGORIES: 5.13, 5.5

Introduction. The algorithm below will generate random numbers from any probability density function, whether it be analytical, hypothetical, or experimentally acquired. Although there are in existence some fast and some general routines, the fast ones are for specific densities whereas the general algorithms are slow. As an example of a general algorithm, IBM's GPSS [7] uses the transformation theory of random deviates [4] to generate random numbers from any density function which can be described by data points. The GPSS algorithm is simple, and its precision is dependent upon the degree of interpolation and the number of points used for estimating the transformation function.

The program below has made the transformation method more accurate than the GPSS routine by using 257 points and linear approximation to the probability density function. Speed was acquired by appropriate organization of necessary tables. A time estimate for the performance of an assembly language program of the algorithm RANDG on an IBM 360/65 is about 33μsec for each generation.

Initialization. The operation of RANDG is based on vectors Q and R which can be derived by RANDGI as indicated below. An explanation of the routine RANDGI will give the reader some insight into the theory of RANDG.

1. Let (x_i, y_i), $i = 1, 2, \cdots, n$ be coordinates describing the probability density function, $y = f(x)$.

2. Using the trapezoidal rule, find $p_i = \int_{x_1}^{x_i} f(x)\, dx$ so that $p(x)$ approximates the cumulative density function of $f(x)$.

3. Let $x = p^{-1}(v)$, the inverse cumulative density function.

4. Find $q_j = p^{-1}(v_j)$ by using Lagrange's quadratic interpolation formula on $p^{-1}(v)$ for values of $v_j = j/256$ and $j = 0, 1, 2, \cdots 256$ [5].

5. Compute $f(q_j)$ and let $r_j = (f(q_{j+1}) - f(q_j))/(f(q_{j+1}) + f(q_j))$ for $j = 0, 1, 2, \cdots, 255$. The $|\, r_j \,|$ is the ratio of the triangular area to the total area of a trapezoid approximating the probability density function between $x = q_j$ and $x = q_{j+1}$ (Figure 1) and the sign of r_j is the sign of the derivative. If the vectors Q and R are available to the experimenter, it is not necessary to use RANDGI. It should also be noted that RANDGI need be used only once for a given density function and, therefore, does not usually affect the speed of generation.

Program. The routine RANDG then uses Q and R to generate the random ordinates in the following manner:

1. Select the jth interval with probability 1/256.

2. Let L_1 and L_2 be uniform random numbers on the interval (0, 1). It follows that $Y_1 = Q_j + (Q_{j+1} - Q_j)*L_1$ is uniformly random over the interval (Q_j, Q_{j+1}) and $Y_2 = Q_j + (Q_{j+1} - Q_j)*\max(L_1, L_2)$ is triangularly distributed on the same interval and is skewed left.

3. Let $P[Y = Y_1] = |\, R_j$ and $P[Y = Y_2] = 1 - |\, R_j$. Then Y is trapezoidally distributed with

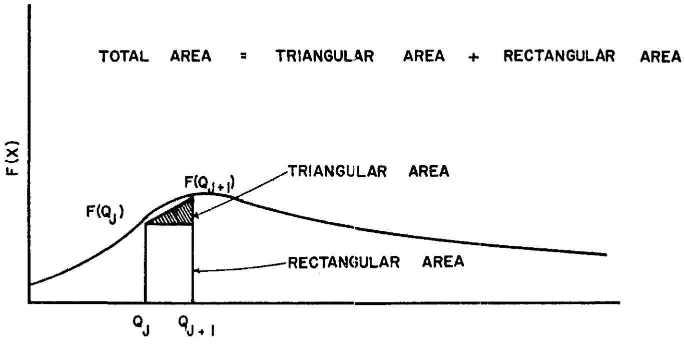


Fig. 1. Trapezoid approximating area under the probability density function from Q_J to Q_{J+1}

$$f(Y) = \begin{cases} 2R_j(Y - Q_j)/(Q_{j+1} - Q_j)^2 & \\ \quad + (1 - R_j)/(Q_{j+1} - Q_j), & Q_j < Y < Q_{j+1}, \\ 0, & \text{otherwise.} \end{cases}$$

4. If $R_j < 0$ then use $Y_2 = Q_j + (Q_{j+1} - Q_j)*\min(L_1, L_2)$. The use of 256 intervals was arbitrary. For speed in assembly language on a binary machine a power of 2 should be used. It is possible that 128 or 64 values are adequate and the use of fewer than 256 would certainly save storage. (Note: Any good uniform random number generator may be used for selecting the interval and finding L_1 and L_2 [1, 2, 3, 6].)

References

1. Hull, T. E., and Dobell, A. R. Random number generators. *SIAM Rev. 4* (July 1962), 230–254.
2. Lewis, P. A. W., Goodman, A. S., and Miller, J. M. A pseudo-random number generator for the System/360. *IBM Syst. J. 8*, 2 (1969), 136.
3. Marsaglia, G., and Bray, T. A. One-line random number generators and their use in combinations. *Comm. ACM 11* (Nov. 1968), 757–759.
4. Mood, A. M. *Introduction to the Theory of Statistics.* McGraw-Hill, New York, 1950, pp. 107–108.
5. Salvadori, M. G., and Baron, M. L. *Numerical Methods in Engineering* (2nd ed.). Prentice-Hall, Englewood Cliffs, N. J., 1964, pp. 88–91.
6. Whittlesey, J. RB. A comparison of the correlational behavior of random number generators for the IBM 360. *Comm. ACM 11* (Sept. 1968), 641–644.
7. General purpose simulation System/360 user's manual. No. H20-0326-3 (1968), IBM, White Plains, N. Y., pp. 26–35.

```
C *************************************
C
C   SUBROUTINE RANDG
C
C   PURPOSE
C      COMPUTE RANDOM NUMBERS FROM ANY GENERAL DISTRIBUTION.
C
C   USAGE
C      CALL RANDG (L,X,R,Y)
```

```
C
C DESCRIPTION OF PARAMETERS
C   INPUT
C     L  -A NON ZERO ODD RANDOM INTEGER
C     X  -VECTOR OF LENGTH 257 CONTAINING ORDINATE POINTS
C         SEPERATED BY EQUAL PROBABILITY ON DESIRED DISTRIBUTION.
C         (CAN BE CALCULATED IN RANDGI).
C     R  -VECTOR OF LENGTH 256 CONTAINING RATIOS OF DERIVATIVE*DX
C         TO AREA/DX FOR EACH ORDINATE POINT IN X.
C         (CAN BE CALCULATED IN RANDGI).
C   OUTPUT
C     Y  -RANDOM NUMBER
C
C REMARKS
C   QUADRATIC APPROXIMATION OF CDF (CUMULATIVE DENSITY FUNCTION)
C   WHICH IMPLIES LINEAR APPROXIMATION OF PDF (PROBABILITY DENSITY
C   FUNCTION).
C
C SUBROUTINES AND FUNCTION SUBPROGRAMS REQUIRED
C   NONE DIRECTLY.  RANDGI MAY BE USED FOR INITIALIZATION.
C
C METHOD
C   TABLE LOOK UP PLUS UNIFORM AND TRIANGULAR DISTRIBUTION
C   VARIABLES ARE USED.
C
C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      SUBROUTINE RANDG (L,X,R,Y)
      DIMENSION X(257),R(256)
C
C GENERATE TWO UNIFORM RANDOM NUMBERS ON INTERVAL (1 - 2**31)
C ANY GOOD GENERATOR MAY BE SUBSTITUTED.
C
      L1=IABS(65539*L)
      L=IABS(65539*L1)
      L2=L
C
C CALCULATE TWO UNIFORM RANDOM NUMBERS
C K1 INTEGER ON INTERVAL (1 - 256)
C AK2 REAL ON INTERVAL (0 - 1.0)
C
      K1=L1/8388608+1
      AK2=FLOAT(MOD(L1,8388608))*1.192093E-7
      IF(AK2-ABS(R(K1))) 8,8,30
    8 IF(R(K1)) 20,10,10
C
C CALCULATE TRIANGULAR RANDOM SKEWED LEFT
C
   10 Y=X(K1)+(X(K1+1)-X(K1))*AMAX0(L1,L2)*4.656613E-10
      RETURN
C
C CALCULATE TRIANGULAR RANDOM SKEWED RIGHT
C
   20 Y=X(K1)+(X(K1+1)-X(K1))*AMIN0(L1,L2)*4.656613E-10
      RETURN
C
C CALCULATE UNIFORM RANDOM
C
   30 Y=X(K1)+(X(K1+1)-X(K1))*FLOAT(L2)*4.656613E-10
      RETURN
      END
C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
C
C SUBROUTINE RANDGI
C
C PURPOSE
C   COMPUTE INITIALIZING VECTORS FOR RANDG
C
C USAGE
C   CALL RANDGI (N,X,Y,P,Q,R,IER)
C
C DESCRIPTION OF PARAMETERS
C   INPUT
C     N  -NUMBER OF (X,Y) POINTS OF APPROXIMATION TO PDF
C         (PROBABILITY DENSITY FUNCTION)
C     X  -VECTOR OF LENGTH N CONTAINING ORDINATE OF PDF
C     Y  -VECTOR OF LENGTH N CONTAINING ABSCISSA OF PDF
C   OUTPUT
C     P  -WORK VECTOR OF LENGTH N
C     Q  -VECTOR OF LENGTH 257 CONTAINING ORDINATE POINTS
C         SEPERATED BY EQUAL PROBABILITY ON DESIRED DISTRIBUTION.
C     R  -VECTOR OF LENGTH 256 CONTAINING RATIOS OF DERIVATIVE*DX
C         TO AREA/DX FOR EACH ORDINATE POINT IN Q.
C     IER-ERROR INDICATOR
C           1 - ERROR IN SCALING. I.E. TOTAL AREA OF PDF NOT EQUAL TO
C               1.  ASSUMING ESTIMATION ERRORS A FUDGE FACTOR IS USED
C               TO SCALE A RESULT.
C           2 - DENSITY NOT POSITIVE. I.E. SOME Y(I) LT 0.  ABORT
C           3 - NOT IN SORT. I.E. SOME X(I) LT X(I-1).  ABORT
C           4 - SEARCH ERROR.  SHOULD NEVER OCCUR BECAUSE OF FUDGE
C               FACTOR USED.  THIS MEANS SOME P(I) NOT LARGE ENOUGH
C               FOR SEARCH OF PROPER Q.  INVESTIGATION IS NEEDED.
C
C REMARKS
C   NONE
C
C SUBROUTINES AND FUNCTION SUBPROGRAMS REQUIRED
C   NONE
C
C METHOD
C   LINEAR APPROXIMATION OF PDF TO FIND CDF (CUMULATIVE DENSITY
C   FUNCTION) AND ICDF (INVERSE CDF).  QUADRATIC INTERPOLATION ON
C   ICDF TO FIND Q AND R.
C * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      SUBROUTINE RANDGI (N,X,Y,P,Q,R,IER)
      DIMENSION X(300),Y(300),P(300),Q(257),R(256)
C
C CALCULATE CUMULATIVE PROBABILITIES
C
      IER=0
      IF(Y(1)) 5,10,10
C
C ERROR 2
```

```
C
    5 IER=2
      RETURN
   10 P(1)=0.0
      DO 15 I=2,N
         IF(Y(I)) 5,11,11
   11    IF(X(I)-X(I-1)) 6,12,12
C
C  ERROR 3
C
    6    IER=3
         RETURN
   12    P(I)=(Y(I)+Y(I-1))*(X(I)-X(I-1))*0.5+P(I-1)
   15    CONTINUE
      IF(P(N)-0.996094) 7,7,16
   16 IF(P(N)-1.003906) 3,7,7
C
C  ERROR 1
C
    7 IER=1
    3 F=1.0/P(N)
      DO 4 I=2,N
    4    P(I)=P(I)*F
C
C  CALCULATE X POINTS FOR EQUAL-DISTANT CUMULATIVE PROBABILITIES
C
      V=0.0
      Q(1)=X(1)
      T1=Y(1)
      J1=2
  100 DO 150 I=2,257
         IF(I-257) 102,103,103
  102    V=V+3.90625E-3
C
C  LOCATE BEST POINT FOR INTERPOLATION
C
         DO 101 J=J1,N
            IF(P(J)-V) 101,104,105
  101       CONTINUE
C
C  ERROR 4
C
         IER=4
  103    J=N
  104    Q(I)=X(J)
         T2=Y(J)
         GO TO 125
  105    IF(J-3) 113,108,107
  107    IF(J-N) 108,111,111
  108    IF((P(J)-V)-(V-P(J-1))) 110,110,111
  110    J1=J-1
         GO TO 120
  111    J1=J-2
         GO TO 120
  113    J1=1
C
C  QUADRATIC INTERPOLATION OF P INVERSE FOR Q
C
  120    XT2=P(J1+2)-P(J1)
         XT3=P(J1+2)-P(J1+1)
         XT1=P(J1+1)-P(J1)
         XV1=V-P(J1)
         XV2=V-P(J1+1)
         XV3=V-P(J1+2)
         Q(I)=(XV3*XV2*X(J1))/(XT1*XT2)-(XV3*XV1*X(J1+1))/(XT1*XT3)+
     1      (XV2*XV1*X(J1+2))/(XT2*XT3)
C
C  LINEAR INTERPOLATION OF Y FOR T2 AND R
C
         T2=(Y(J)-Y(J-1))*(Q(I)-X(J-1))/(X(J)-X(J-1))+Y(J-1)
  125    R(I-1)=(T2-T1)/(T2+T1)
         T1=T2
         J1=J
  150    CONTINUE
      RETURN
      END
```

Remark on Algorithm 370 [G5]
General Random Number Generator [Edgar L. Butler, *Comm. ACM 13* (Jan. 1970), 49–52]

L.G. Proll* (Recd. Nov. 1970)
Department of Mathematics, University of Southampton, U.K.

Key Words and Phrases: random number generator, probability density function, transformation, cumulative distribution function
CR Categories: 5.13, 5.5

Algorithm 370 was translated into Algol and run on an ICL 1907 computer. Tests revealed that, in several instances, the subroutine *RANDGI* generated incorrect values for the vector *Q* and, consequently, for *R*. In particular, *RANDGI* does not guarantee that *Q(I)* increases with *I* as clearly should be the case. For example, a selection of the results for *Q* and *R*, rounded to four decimal places, obtained by *RANDGI* with

$N = 4$
$X = (0.0, 0.5, 1.0, 2.0)$
$Y = (0.0, 0.5, 1.0, 0.0)$

corresponding to a symmetric triangular distribution [1] on [0, 2], is as follows:

I	*Q(I)*	*R(I)*
78	0.9211	0.0031
79	0.9268	0.0030
80	0.9322	0.0029
81	0.7232	−0.1262
82	0.7284	0.0036

Similar results were obtained for several other distributions.

The error lies in changing the interpolating quadratic between two interpolation points and will always arise when, for some J,

(i) interpolation takes place at points between $P(J)$ and $P(J+1)$,

(ii) the interpolating quadratic on the points $P(J-1)$, $P(J)$ and $P(J+1)$ is convex,

(iii) the interpolating quadratic on the points $P(J)$, $P(J+1)$ and $P(J+2)$ is concave.

Alteration of the interpolating quadratic only at an interpolation point will avoid this error; an appropriate alteration to the algorithm is given later.

In addition, the following remarks can be made about the algorithm:

(i) The statements labeled 105, 110, and 120 in the subroutine *RANDGI* imply that $N \geq 4$. However only three points are needed for quadratic interpolation, and moreover, it is meaningful to specify a probability distribution by only three points, e.g. any triangular distribution.

(ii) A trivial alteration would allow the subroutine *RANDGI* to trap the condition $X(I) = X(I-1)$ which would otherwise cause an overflow in calculating an element of Q.

(iii) The usefulness of Algorithm 370 can be enhanced by allowing the vector Y to represent either a probability density function or a cumulative distribution function as required. The experimenter may, for instance, have directly available the cumulative polygon [2] of an empirical distribution.

The following alterations to the subroutine *RANDGI* incorporate the above correction and remarks:

(i) In the opening comment,

(a) replace line 9 by

```
C CALL RANDGI (N,X,Y,P,Q,R,K,IER)
```

(b) replace line 14 by

```
C (PROBABILITY DENSITY FUNCTION) OR CDF
C (CUMULATIVE DISTRIBUTION FUNCTION), N.GE.3.
```

(c) add the words OR CDF to lines 15 and 16

(d) insert after line 16,

```
C      K - K SHOULD BE SET TO 1 IF Y REPRESENTS
C        A CDF, OTHERWISE Y WILL BE INTER-
C          PRETED AS A PDF
```

(e) replace line 28 by

```
C  3 - NOT IN SORT, I.E. SOME X(I) LE X(I-1).ABORT
```

(ii) Change the subroutine statement to

```
      SUBROUTINE RANDGI (N,X,Y,P,Q,R,K,IER)
```

(iii) Change the statement labeled 11 to

```
   11    IF (X(I)-X(I-1)) 6,6,12
```

(iv) Delete the statement labeled 12 and insert

```
   12    IF (K.EQ.1) GO TO 13
         P(I) = (Y(I)+Y(I-1))*(X(I)-X(I-1))*0.5 +
       1  P(I-1)
C    Y IS A PDF
         GO TO 15
   13    P(I) = Y(I)
C    Y IS A CDF
```

(v) Delete the five statements commencing at label 105 and insert

```
  105    IF (J.LE.3) GO TO 113
```

With these alterations to the subroutine *RANDGI* and with the incorporation of locally available routines for generating uniform and triangular deviates [3] into *RANDG*, satisfactory results were obtained for the first two moments of several distributions including the beta, symmetric triangular, nonsymmetric triangular, and various empirical distributions. Table I contains a selection of the results obtained for various values of N for samples of size 1000 from a *beta*(4, 3) distribution. In each case, the distribution was specified at the points

$X(I) = (I-1)/(N-1), \quad I = 1, 2 -- N.$

With the exception of the case when $N = 5$, the true mean and variance lie within the appropriate 95 percent confidence intervals obtained from the samples.

In addition to tests on the first two moments, the samples were also subjected to Q - Q plots [4]; i.e t.he ordered observations were plotted against the quantiles of the parent distribution. The procedure indicates a perfect match by a straight line of slope 1 passing through the origin and is especially sensitive to differences in the tails of the distributions. The quantiles of the beta distribution were calculated by interpolation in values of the *beta c.d.f.* obtained by the method of Hill and Pike [5]. Serious departures from the desired shape were observed for $N = 5$, 10 in both the cases $K = 1$ and $K \neq 1$. The results obtained for $N \geq 20$ were satisfactory for both cases.

Table I.

	$K \neq 1$		$K = 1$	
N	*Mean*	*Variance*	*Mean*	*Variance*
5	0.605	0.037	0.663	0.026
10	0.576	0.031	0.580	0.028
20	0.578	0.029	0.580	0.029
50	0.571	0.030	0.574	0.030
100	0.565	0.029	0.573	0.029
True value	0.571	0.030	0.571	0.030

References

1. Feller, W. *An Introduction to Probability Theory and Its Applications, Vol. II.* Wiley, New York, 1968.

2. Guttman, I., and Wilks, S.S. *Introducing Engineering Statistics.* Wiley, New York, 1965.

3. Proll, L.G. A subroutine package for the generation of random deviates on an ICL 1900 computer. Mathematics Depart. Tech. Rep. 70/1, U. of Southampton, U.K. July 1970.

4. Wilk, M.B., and Gnanadesikan, R. Probability plotting methods for the analysis of data. *Biometrika 55* (Mar. 1968), 1–17.

5. Pike, M.C., and Hill, I.D. Remark on Algorithm 179: Incomplete beta ratio. *Comm. ACM 10* (June 1967), 375.

ALGORITHM 371
PARTITIONS IN NATURAL ORDER [A1]

J. K. S. McKay (Recd. 28 Apr. 1967)
California Institute of Technology, Mathematics Division,
Pasadena, CA 91109.

KEY WORDS AND PHRASES: partitions, number theory
CR CATEGORIES: 5.39

```
procedure partition (p, k, last); integer n, k;
  integer array p; Boolean last;
comment Partition may be used to generate partitions in their
  natural (reverse lexicographical) order. On entry the first k
  elements of the global integer array p[1:n] should contain a parti-
  tion, p[1] ≥ p[2] ≥ ··· ≥ p[k], of n into k parts. In order to ini-
  tialize m, the first entry must be made with last set true: this will
  result in p[1], p[2], ··· , p[k] and k remaining unaltered and last
  set false on exit. On all subsequent entries with last false, k is
  updated and p[1], p[2], ··· , p[k] will be found to contain the
  next partition of n with parts in descending order. On returning
  with the last partition, p[1] = p[2] = ··· = p[n], last is set
  true. To generate all partitions of n, p[1], k, last should be set
  to n, 1, true, respectively for the initial call: these variables
  must not be altered between successive calls for partition;
begin
  own integer m; integer t;
  if last then
  begin
    last := false;
    for m := 1 step 1 until k do
      if p[m] = 1 then go to c;
    m := k; go to c
  end;
  t := k − m;
  k := m;
  p[m] := p[m] − 1;
a: if p[k] > t then go to b;
  t := t − p[k];
  k := k + 1;
  p[k] := p[k−1];
  go to a;
b: k := k + 1;
  p[k] := t + 1;
  if p[m] ≠ 1 then m := k;
c: if p[m] = 1 then m := m − 1;
  if m = 0 then last := true;
end partition
```

ALGORITHM 372
AN ALGORITHM TO PRODUCE COMPLEX PRIMES, CSIEVE [A1]

K. B. Dunham (Recd. 29 July 1968 and 7 Oct. 1968)
Georgia Institute of Technology, School of Information Service, Atlanta, GA 30332

KEY WORDS AND PHRASES: primes, complex numbers
CR CATEGORIES: 5.39

```
procedure CSIEVE (m, PR, PI);
  value m;  integer m;  integer array PR, PI;
```

comment Primes can be defined in the complex domain, $a + bi$, where a and b are integers. A unity is ± 1 or $\pm i$. A unity times a prime is its associate. Primes are not unique among associates; but except for that ambiguity, all the ordinary rules of real primes, such as the unique factorization law, apply to complex primes.

It can be shown that a complex integer is prime if and only if its conjugate is prime. Therefore it is sufficient to search for primes in the one-eighth plane area with a closed bound along $y = 0$ and an open bound along $x = y$, where x is positive and y is less than x but nonnegative. Any prime found in that area has seven more associated primes: $-x + yi$, $\pm x - yi$, $\pm y + xi$, $\pm y - xi$. A discussion of complex primes can be found in [1]. It should be pointed out that numbers prime in the real domain are not necessarily prime in the complex domain, e.g. $2 = 2 + 0i = (1+i)(1-i)$.

Algorithms 35 [2], 310 [3], and 311 [4] generate real primes. The simplistic technique used by Algorithm 35 applies equally well to generating complex primes. Unfortunately the more efficient techniques of Algorithms 310 and 311 cannot easily be translated into complex prime sifters. This algorithm, *CSIEVE*, uses the result that a complex integer is prime if the square of its modulus is relatively prime to the square of the moduli of all previous primes. The procedure is called with a value m, the number of complex primes to generate, *PR* and *PI*, the real and imaginary parts of the prime list generated where $PR > PI \geq 0$ for each prime. The seven other associated primes must be generated externally to *CSIEVE*.

References:

1. Hardy, G. H., and Wright, E. M. *An Introduction to the Theory of Numbers*. Clarendon Press, Oxford, 1954, Ch. 12.
2. Wood, T. C. Algorithm 35, Sieve. *Comm. ACM 4* (Mar. 1961), 151.
3. Chartres, B. A. Algorithm 310, Prime number generator 1. *Comm. ACM 10* (Sept. 1967), 569.
4. Chartres, B. A. Algorithm 311, Prime number generator 2. *Comm. ACM 10* (Sept. 1967), 570;

```
begin
  integer dn, nr, ni, sq, root, i, j, k;
  integer array PM[2:m];
  dn := PR[1] := PI[1] := PI[2] := 1;  PM[2] := 5;
    j := PR[2] := 2;
  for nr := 3 step 1 until m do
  begin
    dn := 1 - dn;
    for ni := dn step 2 until nr - 1 do
    begin
      sq := nr × nr + ni × ni;
      root := entier (1.5×nr);
      for i := 2 step 1 until j do
      begin
        if ((sq÷PM[i]) × PM[i]) = sq then go to C;
        if root < PM [i] then go to A;
      end;
A:    for i := 2 step 1 until j do
      begin
        if PM[i] > sq then
        begin
          for k := j step -1 until i do
            PM[k+1] := PM[k];
          go to B;
        end
      end;
B:    PM[i] := sq;  j := j + 1;  PR[j] := nr;  PI[j] := ni;
      if j = m then go to D;
C:    end
  end;
D:
end CSIEVE
```

REMARKS ON ALGORITHM 372 [A1]
AN ALGORITHM TO PRODUCE COMPLEX PRIMES, CSIEVE [K. B. Dunham. *Comm. ACM 13* (Jan. 1970), 52–53]
ALGORITHM 401 [A1]
AN IMPROVED ALGORITHM TO PRODUCE COMPLEX PRIMES [P. Bratley. *Comm. ACM 13* (Nov. 1970), 693]

Paul Bratley (Recd. 25 Feb. 1970)
Département d'informatiqué, Universite de Montréal, C.P. 6128, Montréal 101, Quebec, Canada

KEY WORDS AND PHRASES: number theory, prime numbers, complex numbers
CR CATEGORIES: 5.39

Algorithm 372 was run on the CDC 6400 at the University of Montreal. The variable i is undefined if the **for**-loop at label A is completed. The statement

$$i := j + 1;$$

should be added immediately before label B. Algol purists may also care to remove redundant semicolons after **go to** A and **go to** B, and the redundant parentheses in one **if**-statement. With these changes the algorithm produced correct results for several values of m.

The comment in Algorithm 372 is slightly inaccurate. The first prime generated by the algorithm is $1 + i$, which does not have $PR > PI$, and which has not seven but three associated primes.

It is not possible to compare the speeds of Algorithm 372 and Algorithm 401 directly since they generate primes in a different order. However, the following test was run. A value of m was chosen, and Algorithm 401 was used to list all the complex primes with modulus less than m. The time taken and the number of primes produced were noted. Then Algorithm 372 was used to produce an equal number of primes, the time taken again being noted. Times observed are shown in Table I.

TABLE I

Limit on modulus	*Algorithm 401 produced this number of primes*	*Time taken (secs)*	*Time taken by Algorithm 372 to produce the same number of primes (secs)*	*Ratio of times taken*
25	60	0.278	0.331	1.2
50	189	1.577	2.140	1.4
75	373	4.217	7.602	1.8
100	623	8.618	20.214	2.4
150	1266	23.732	79.481	3.4

The conclusion from the figures in Table I is that if the speed with which the complex primes are generated is of paramount importance then Algorithm 401 should be preferred to Algorithm 372.

As written Algorithm 401 will use more memory than Algorithm 372 since it is convenient and perspicuous to use *sieve*2 in an unmodified form, which makes it necessary to store temporarily all the rational primes less than m^2. However, if space is tight then *sieve*2 can easily be modified so as to generate rational primes one at a time on successive calls, and in this way the use of the long array $P2$ can be avoided. If this modification is made Algorithm 401 will in fact use less store than Algorithm 372, which wastefully stores many useless values in PM. It is also to be noticed that the factors 0.7 and 1.4 occurring in the declarations of $P2$ and $P3$ may be diminished for large m: all that is necessary is that $P2$ should be long enough to hold the rational primes less than m^2, and that $P3$ should be long enough to hold the rational primes which are not greater than m and which are of the form $4n + 3$. Some space may be saved similarly in *sieve*2, which is called from Algorithm 401.

ALGORITHM 373
NUMBER OF DOUBLY RESTRICTED
PARTITIONS [A1]

JOHN S. WHITE (Recd. 4 Mar. 1969)
University of Minnesota, Department of Mechanical Engineering, Minneapolis, MN 55455

KEY WORDS AND PHRASES: partitions, restricted partitions, sums of integers, restricted sums
CR CATEGORIES: 5.39

```
procedure setk (P, N, K);  value N, K;
  integer N, K;  integer array P;
comment The number of partitions of L with parts greater than
  or equal to K and less than or equal to M is set in P[L, M] for
  all L, M such that N ≥ L ≥ M ≥ 0. This algorithm is a general-
  ization of [1] which treats the case K = 1.
    REFERENCE:
  1. McKAY, J. K. S.  Algorithm 262, Number of restricted par-
      titions on N. Comm. ACM 8 (Aug. 1965), 493;
begin integer L, M;
  for L := 0 step 1 until N do
    for M := 0 step 1 until L do P[L, M] := 0;
  P[0, 0] := 1;
  for L := K step 1 until N do
    for M := K step 1 until L do
      P[L, M] := P[L, M−1] + P[L−M, if L−M<M then L−M
        else M]
end
```

ALGORITHM 374
RESTRICTED PARTITION GENERATOR [A1]

JOHN S. WHITE (Recd. 4 Mar. 1969)
University of Minnesota, Department of Mechanical Engineering, Minneapolis, MN 55455

KEY WORDS AND PHRASES: partitions, restricted partitions, sums of integers, restricted sums
CR CATEGORIES: 5.39

```
procedure gen (P, N, K, position, ptn, len);
  value N, K, position;
  integer N, K, position, len;  integer array P, ptn;
comment  The partitions of N with smallest part greater than
  or equal to K are mapped in their natural order, one-one, onto
  the consecutive integers from 0 to P[N, N] − 1, where P[N, N]
  is the number of partitions of N with smallest part greater than
  or equal to K. The array P is set by the procedure setk. On entry,
  position contains the integer onto which the partition is mapped.
  On exit, len contains the number of parts of the partition and
  ptn[1:len] contains the parts of the partition in descending order.
  This algorithm is a generalization of [1] which considers the
  case K = 1.
    REFERENCE:
  1. McKAY, J. K. S. Algorithm 263, Partition generator. Comm.
      ACM 8, (Aug. 1965), 493;
begin integer L, M, psn;
  L := N;  psn := position;  len := 0;
A:
  len := len + 1;  M := K;
B:
  if P[L, M] < psn then
  begin
    M := M + 1;  go to B
  end
  else if P[L, M] > psn then
C:
  begin
    ptn[len] := M;  psn := psn − P[L, M−1];
    L := L − M;  if L < K then go to D;  go to A
  end
  else M := M + 1;
  if M = L then go to C else go to B;
D:
end;
begin integer N, I, J, K, len, position;
  integer array P[0:20, 0:20], ptn[0:20];
comment driver for setk and gen;
Next:
  outstring (1, " ");  outstring (1, " ");
  outstring (1, "partitions of N, N=");  ininteger (2, N);
  outstring (1, "with parts ≥ K, K=");  ininteger (2, K);
  for I := 0 step 1 until N do
    for J := 0 step 1 until N do P[I, J] := 0;
  setk (P, N, K);
  outstring (1, "P array");
  for I := 0 step 1 until N do
  begin
    for J := 0 step 1 until N do outinteger (1, P[I, J]);
    outstring (1, " ")
  end;
  outstring (1, " ");
  outstring (1, "pos. partition");
  for position := 0 step 1 until P[N, N] − 1 do
  begin
    gen (P, N, K, position, ptn, len);
    outinteger (1, position);
    for I := 1 step 1 until len do outinteger (1, ptn[I]);
    outstring (1, " ");
  end;
  go to Next
end
```

ALGORITHM 375
FITTING DATA TO ONE EXPONENTIAL [E2]

H. Späth (Recd. 23 Oct. 1967)
Institut für Neutronenphysik und Reaktortechnik, Kernforschungszentrum Karlsruhe, Germany

KEY WORDS AND PHRASES: nonlinear least squares fit
CR CATEGORIES: 5.15

```
procedure abfit (x, y, p, n, eps, a, b, ab, eb, bool, exit);
  value n, eps;  integer n;  real eps, a, b, ab, eb;
  label exit;  array x, y, p;  Boolean bool;
```

comment If you want to fit data points (x_i, y_i) $(i=1, \cdots, n)$ with associated weights p_i to $f(x) = ae^{-bx}$ the usual approach is to do a linear fit in the sense of least squares with $\ln(f(x)) = \ln(a) - bx$ to the data $(x_i, \ln(y_i))$ that is to minimize

$$S^* = \sum_{i=1}^{n} g_i(\ln(y_i) - \ln(a) + bx_i)^2. \qquad (1)$$

In [1] it is shown that this approach for finding a and b that are minimizing

$$S(a, b) = \sum_{i=1}^{n} p_i(y_i - ae^{-bx_i})^2 \qquad (2)$$

is in general bad if you do not choose

$$g_i = p_i y_i^2 \quad (i=1, \cdots, n). \qquad (3)$$

Proceeding similarly as in [2] from the necessary conditions for S having a minimum

$$\frac{\partial S}{\partial a} = \frac{\partial S}{\partial b} = 0, \qquad (4)$$

we eliminate $a = a(b)$ from the first equation of (4) and put this into the second one. We result in an equation

$$F(b) = 0. \qquad (5)$$

If we have found a zero b of (5) then $(a(b), b)$ is a solution of (4).

The procedure *abfit* has two possibilities to do this. For *bool* = **false** we use the result b^* from minimizing (1) with weights (3) to set up the intervals

$$\left[b^*\left(1 - \frac{j}{20}\right), b^*\left(1 - \frac{j+1}{20}\right)\right] (j = 0, \pm1, \pm2, \cdots, \pm19) \qquad (6)$$

and to look if F has opposite signs at the endpoints of one of these intervals [*ab*, *eb*]. Experience has shown that for realistic data this method is a good one. If we do not find such an interval, *abfit* is left through *exit* and we can deliver *ab* and *eb* as input parameters to *abfit* with *bool* = **true**.

In both cases a global procedure *Rootfinder* must be made available to find an existing zero b with relative accuracy *eps* in the calculated or given interval otherwise leaving to the label *exit*.

The label *exit* would further be used if for *bool* = **false** the condition $y_i > 0$ for $i = 1, \cdots, n$ is not fulfilled.

References:
1. Böttger, H. Über Gewichtsverteilung beim Fit mit Exponentialfunktionen. *ZfK-TPh 22* (1966).
2. Späth, H. Algorithm 295, Exponential curve fit. *Comm. ACM 10* (Feb. 1967), 87;

```
begin integer k;
  real h1, h2, h3, h4, h5, h6, h7, h8, b1, b2, F1, F2, F3, F4, h;
  procedure Fb(b, F);  value b;  real b, F;
  comment  For given b this procedure calculates F = F(b);
  begin
    h1 := h2 := h3 := h4 := 0;
    for k := 1 step 1 until n do
    begin
      h5 := exp(-b×x[k]);  h6 := p[k]×y[k];
      h8 := h5 × h6;  h7 := p[k] × h5 × h5;
      h1 := h1 + h8;  h2 := h2 + h7;
      h3 := h3 + x[k] × h8;  h4 := h4 + x[k] × h7
    end;
    a := h1/h2;  F := h3 × h2 - h1 × h4
  end Fb;
  if bool then go to ROOT;
  h1 := h2 := h3 := h4 := h5 := 0;
  comment  The linear fit is done to get the estimate b*;
  for k := 1 step 1 until n do
  begin
    if y[k] ≤ 0 then go to exit;
    h8 := ln(y[k]);  h6 := p[k] × y[k] × y[k];  h7 := h6 × x[k];
    h1 := h1 + h6;  h2 := h2 + h7 × x[k];  h3 := h3 + h7;
    h4 := h4 + h7 × h8;  h5 := h5 + h6 × h8
  end;
  h8 := 1.0/(h1×h2-h3×h3);  b := -h8 × (h1×h4-h3×h5);
  b1 := b2 := b;  k := 0;  h := 0;  Fb(b, F1);  F2 := F1;
SEARCH:  k := k + 1;  if k > 20 then go to exit;
  h := h + .05;  ab := b1 × (1.0-h);  Fb(ab, F3);
  if F1 × F3 < 0 then begin eb := b1;  go to ROOT end;
  eb := b2 × (1.0+h);  Fb(eb, F4);
  if F2 × F4 < 0 then begin ab := b2;  go to ROOT end;
  b1 := ab;  b2 := eb;  F1 := F3;  F2 := F4;  go to SEARCH;
ROOT: Rootfinder (Fb, ab, eb, eps, b, exit)
end abfit
```

ALGORITHM 376
LEAST SQUARES FIT BY $f(x) = A\ cos\ (Bx+C)$ [E2]

H. Späth (Recd. 26 June 1967 and 28 Oct. 1968)
Institut für Neutronenphysik und Reaktortechnik, Kernforschungszentrum Karlsruhe, Germany

KEY WORDS AND PHRASES: nonlinear least squares fit
CR CATEGORIES: 5.15

```
procedure cosfit(x, y, p, n, beginB, endB, eps, A, B, C, fB, s, fx,
  exit);
  value n, beginB, endB, eps;  integer n;
  real beginB, endB, eps, A, B, C, fB, s;
  array x, y, p, fx;  label exit;
```

comment Let (x_k, y_k) be n given data points with associated weights p_k. We want to find the three parameters A, B, and C of a curve $f(x) = Acos(Bx+C)$ such that f fits the data in the least squares sense. Introducing the parameters $\alpha = -Asin(C)$, $\beta = Acos(C)$, $\gamma = B$, we have $f(x) = \alpha sin(\gamma x) + \beta cos(\gamma x)$ and thus only one nonlinear parameter γ. Now we can use the same method as in [1]. From the necessary conditions for

$$s(\alpha, \beta, \gamma) = \sum_{k=1}^{n} p_k(x_k - f(x_k))^2$$

having a minimum we eliminate α and β getting one equation in one nonlinear parameter γ, $F(\gamma) = 0$. If we obtain a root γ^* of F then the triple $(\alpha(\gamma^*), \beta(\gamma^*), \gamma^*)$ is a stationary point of s and we finally get the desired parameters by

$$B = \gamma,$$
$$C = arctan(\alpha/\beta),$$
$$A = -sign(a) \times sign(sin(C)) \times (\alpha^2 + \beta^2)^{\frac{1}{2}}.$$

A global procedure named *Rootfinder* must be made available to *cosfit* which is able to get a zero $\gamma = B$ of a function $F(\gamma)$ in a given interval [*beginB*, *endB*] with relative accuracy *eps*, if *sign* $(F(beginB)) \neq sign(F(endB))$ otherwise leaving to the global label *exit*. A bisection routine is possible, but an interpolation method like that in [2] is to be preferred.

By setting *beginB* equal to *endB*, the procedure *cosfit* can be used to tabulate the functions $fB = F(B) = F(beginB)$, $s = s(beginB)$, $A = A(beginB)$, and $C = C(beginB)$ and thus allows to get all minima in a given range. Often, the tabulation is made superfluous by proceeding as follows. In a rough graph we gather two intervals (x_1^*, x_1^{**}) and (x_2^*, x_2^{**}) including two successive zeros x_1 and x_2 of the desired function f. Then the two values $beginB = 2\pi/(x_2^* - x_1^{**})$, $endB = 2\pi/(x_2^{**} - x_1^*)$ in general form an interval that contains the value B for which s has the absolute minimum. As s has in general infinitely many minima, our method is superior to general purpose minimizing methods. If the found zero of F is not a minimum of s in the sense that the Jacobian s'' is numerically not positive definite, the program puts s equal to $-s$. As rounding errors may cause here a wrong decision it is recommended to look also at the magnitude of s.

The label *exit* is further used if, during the zero locating process, it would happen that the elimination of α and β were not possible. Variables fB and s finally have the values $F(B)$ and $s(B)$ at the found zero. The array fx will contain the fitted values $fx[k] = A \times cos(B \times x[k]+C)$.

References:

1. Späth, H. Algorithm 295, Exponential curve fit. *Comm. ACM 10* (Feb. 1967), 87.
2. Kristiansen, G. K. Contribution No. 6, Zero of arbitrary function. *BIT 3* (1963), 205-207;

```
begin
  integer k;  real h1, h2, h3, h4, h5, h6, h7, h8, h9, h11, h12, h13,
  h14, hh, alpha, beta, gamm, t, u, v, w, z, q, r, h, d, e, f;
  procedure Fgamma(gamm, Fgamm);
  value gamm;  real·gamm, Fgamm;
  begin
    if gamm = 0 then go to exit;
    h1 := h2 := h3 := h4 := h5 := h6 := h7 := h8 := h9 := 0;
    for k := 1 step 1 until n do
    begin
      t := x[k];  u := gamm × t;  v := sin(u);  u := cos(u);
      w := v × v;  z := u × u;  q := p[k];  r := v × u;
      h := y[k];  d := q × h;  e := q × t;  f := e × h;
      h1 := h1 + q × w;  h2 := h2 + q × z;  h3 := h3 + q × r;
      h4 := h4 + d × v;  h5 := h5 + d × u;  h6 := h6 + e × r;
      h7 := h7 + e × (z−w);  h8 := h8 + f × u;  h9 := h9 + f × v
    end;
    hh := h1 × h2 − h3 × h3;
    if hh = 0 then go to exit;  h = 1/hh;
    alpha := h × (h4×h2−h3×h5);  beta := h × (h1×h5−h3×h4);
    Fgamm := fB := h6 × (alpha+beta) × (alpha−beta)
      + alpha × beta × h7 − alpha × h8 + beta × h9
  end Fgamma;
  if beginB = endB then begin Fgamma(B, fB);  go to CC end
  Rootfinder (Fgamma, beginB, endB, eps, gamm, exit);
    B := gamm;
CC : if beta =0 then C := − 1.5707963 else C := −arctan
    (alpha/beta);
  A := −sign(alpha) × sign(sin(C)) × sqrt(alpha×alpha+beta×
    beta);  h := 0;
  for k := 1 step 1 until n do
  begin
    v := fx[k] := A × cos(B×x[k]+C);
    v := v − y[k];  h := h + p[k] × v × v
  end;
  s := h;  if beginB = endB then go to END;
  if h1 ≤ 0 ∨ hh < 0 then begin s := −s;  go to END end;
  h11 := h12 := h13 := h14 := 0;
  for k := 1 step 1 until n do
  begin
    u := B × x[k];  v := sin(u);  u := cos(u);
    e := x[k] × x[k];  r := p[k] × e;  f := r × y[k];
    h11 := h11 + r × (u×u−v×v);  h12 := h12 + r × u × v;
    h13 := h13 + f × v;  h14 := h14 + f × u
  end;
  h11 := h11 × (alpha+beta) × (alpha−beta)
    − 4 × alpha × beta × h12 + alpha × h13 + beta × h14;
  h12 := 2 × alpha × h6 + beta × h7 − h8;  h13 := alpha × h7
    − 2 × beta × h6 + h9;
  if h11 × hh − h13 × (h1 × h13 − h3 × h12)
    + h12 × (h3 × h13 − h2 × h12) ≤ 0 then s := −s;
END: end
```

ALGORITHM 377
SYMBOLIC EXPANSION OF ALGEBRAIC EXPRESSIONS [R2]

Michael J. Levine*
Department of Physics, Carnegie-Mellon University,
Pittsburgh, PA 15213
and Stanley M. Swanson† (Recd. 27 Jan. 1969)
89 Mid Oaks Lane, St. Paul, MN 55113

* This work was done in part at the Division of Theory, CERN, Geneva, Switzerland.

† This work was donein part at the Institute of Theoretical Physics, Stanford University, Stanford, California.

KEY WORDS AND PHRASES: algebra, symbolic algebra, symbolic multiplication, algebraic distribution, algebraic multiplication, distribution algorithm, multiplication algorithm, product algorithm, polynomial distribution, polynomial expansion
CR CATEGORIES: 3.10, 3.17, 3.20, 4.13, 4.90

```
procedure EXPAND(M); integer M;
comment This algorithm algebraically expands arbitrarily
  parenthesized expressions into monomials. Distribution is direct,
  without intermediate expansion of lower level expressions. The
  algorithm has been used as a part of algebra programs in the-
  oretical physics [2, 3]. It was devised by H. J. Kaiser [1] and re-
  constructed by M. J. Levine. Expansion proceeds in two steps:
  First, parsing an input expression into a sequence of variable-
  operator pairs with associated parenthesis-level information,
  and then picking out the variables which belong together as
  factors of monomial terms. EXPAND accepts an abbreviated
  ALGOL-like syntax:
          ⟨variable⟩ ::= A | B | C | D | E | F | G
           ⟨primary⟩ ::= ⟨variable⟩ | (⟨expression⟩)
              ⟨term⟩ ::= ⟨primary⟩ | ⟨term⟩ × ⟨primary⟩
        ⟨expression⟩ ::= ⟨term⟩ | ⟨expression⟩ + ⟨term⟩
  REFERENCES:
  1. KAISER, H. J. Trace calculation on electronic computer. Nu-
       clear Physics 43 (1963), 620.
  2. LEVINE, M. J. Dirac matrix and tensor algebras on a compu-
       ter. J. Computat. Phys. 1 (1967), 454.
  3. SWANSON, S. M. Computer algorithms for Dirac algebra.
       J. Computat. Phys. 4, 1 (1969), 171;
begin
  integer LVL, N, T, U;  Boolean array MULT[0:M];
  integer array V, VL, OPL, INDEX[0:M];
  integer procedure CHAR;
  begin
    integer C;
A: insymbol (2, '×) + (ABCDEFG␣;', C);  if C = 12 then go
  to A;
    CHAR := C
  end CHAR;
  procedure DISTRIBUTE(N);  integer N;
  comment  There are two problems in distribution: first, to se-
    lect the variables in an expression which belong together as
    factors of the current monomial, and then to alter the reference
    marks in USED to indicate the next monomial. A Boolean
    value in USED is associated with each variable-operator pair.
    The expression is scanned from the left to select the first un-
    used variable, and then any variables in an additive relation
    to the selected variable are skipped before continuing the
    scanning for other factors. For the next monomial, the first
    selected variable followed by a "+" is marked used, and the
    marks on all the variables to the left are altered, depending on
    their operator type and level relation to the "+". Distribu-
    tion is from left to right (initial factors change most often);
  begin
    integer I, J, K, L, LEVEL;
    Boolean ALTER, PRODUCT, TERM;
      Boolean array USED[0:N];
    for K := 0 step 1 until N do USED[K] := false;
NEXT:  ALTER := true;  J := I := -1;
FACTOR: I := I + 1;  if USED[I] then go to FACTOR;
    J := J + 1;  INDEX[J] := I;
SKIP:  if MULT[I] then go to FACTOR;  LEVEL := OPL[I];
    if LEVEL > 0 then
    begin
      if ALTER then
      begin
        L := LEVEL;  LEVEL := VL[I] + 1;
        USED[I] := PRODUCT := TERM := true;
        ALTER := false;
        for K := I - 1 step -1 until 0 do
        begin
          if OPL[K] < LEVEL then
          begin
            LEVEL := OPL[K];  PRODUCT := MULT[K];
            if PRODUCT then LEVEL := LEVEL + 1;
            if LEVEL ≦ L then TERM := false
          end;
          if PRODUCT then USED[K] := TERM
        end
      end
      else
      begin
R:  I := I + 1;  if LEVEL ≦ OPL[I] then go to R
      end;
      go to SKIP
    end;
    PROCESS(J);  if ¬ ALTER then go to NEXT;
  end DISTRIBUTE;
  procedure PROCESS(J);  integer J;
  comment  A skeletal output routine (normally, monomials are
    further manipulated, sorted, and accumulated);

  begin
    integer I;  outstring (1, '+');
    for I := 0 step 1 until J do
    begin
      outsymbol (1, "×) + (ABCDEFG", V[INDEX[I]]);
      if I ≠ J then outstring (1, "×")
    end
  end PROCESS;
  comment  The following statements parse the input. A full-
    fledged input routine would extend ⟨primary⟩ to include num-
    bers and would class both "−" and "+" together as ⟨adding
    operators⟩. DISTRIBUTE still works with only "+" and "×"
    since a "−" is either absorbed into a following unsigned num-
    ber or replaced by the string "−1×". Only a single subexpres-
    sion, followed by an unparenthesized "+", is expanded at a
    time. M limits the size of this subexpression. A syntax error or
    a semicolon terminates the processing of input;
```

```
   LVL := N := 0;  U := CHAR;  if U < 4 then go to ERR;
 A:  T := U;  if U = 13 then T := 3 else U := CHAR;
   if U ≧ 4 then
   begin
     if T = 1 then
     begin
       MULT[N] := true;  OPL[N] := LVL;  N := N + 1
     end
     else if T = 3 then
     begin
       MULT[N] := false;  OPL[N] := LVL;
       if LVL = 0 then begin DISTRIBUTE(N);  N := 0 end
       else N := N + 1
     end
     else if T = 4 then LVL := LVL + 1
     else go to ERR
   end
   else
   begin
     if T = 2 ∧ LVL > 0 then LVL := LVL − 1 else
     if T ≧ 5 then begin V[N] := T;  VL[N] := LVL end
     else go to ERR;
   end;
   if U ≠ 13 then go to A else if LVL = 0 then go to B;
 ERR: outstring (1, 'syntax error');
 B:  end EXPAND
```

ALGORITHM 378
DISCRETIZED NEWTON-LIKE METHOD FOR SOLVING A SYSTEM OF SIMULTANEOUS NONLINEAR EQUATIONS [C5]

W. Pankiewicz (Recd. 24 May 1967, 13 July 1967 and 2 Oct. 1968)
Warszawa-90, Wyszatycka 28, Poland

KEY WORDS AND PHRASES: Newton's method, nonlinear equations, interpolating polynomials
CR CATEGORIES: 5.15, 5.13

```
integer procedure nielin (n, h, w, eps, psi, y, z);
  value n, h, w, eps, psi;
  integer n; real h, w, eps, psi; array y, z;
comment Functional procedure nielin, of the integer type,
  solves a system of simultaneous nonlinear algebraic or trans-
  cendental equations.
```

Let us consider a given system of n equations with n variables:

$$f_i(y_1 \ , y_2 \ , \cdots, y_n) = 0, \quad i = 1, 2, \cdots, n. \tag{1}$$

A kth approximation of the solution of the system (1) is supposed to be given:

$$Y_0^{(k)} = (y_1^{(k)}, y_2^{(k)}, \cdots, y_n^{(k)}). \tag{2}$$

If for every i,

$$|f_i(Y_0^{(k)})| < \epsilon, \tag{3}$$

where $\epsilon > 0$ is a given number, then the approximation (2) is considered as a solution of the system (1), otherwise a further approximation is calculated.

Let $h^{(k)} > 0$ be given and construct the n new points:

$$Y_i^{(k)} = (y_1^{(k)}, \cdots, y_{i-1}^{(k)} \ , y_i^{(k)} + h^{(k)}, y_{i+1}^{(k)} \ , \cdots, y_n^{(k)}), i = 1, 2, \cdots, n. \tag{4}$$

For every function of the system (1) a new interpolating polynomial of the first order is constructed on the points (2) and (4) such that:

$$w_i(Y_j^{(k)}) = f_i(Y_j^{(k)}), \ j = 0, 1, \cdots, n, \ i = 1, 2, \cdots, n. \tag{5}$$

A solution of the linear system:

$$w_i(y_1 \ , y_2 \ , \cdots, y_n) = 0, \quad i = 1, 2, \cdots, n, \tag{6}$$

is used as the $(k+1)$-th successive approximation.

The special choice of the interpolation points (2) and (4) assures existence and uniqueness of the interpolating polynomials w_i (5). Namely, the kth approximation has for the ith function the form:

$$w_i^{(k)}(Y) = f_i(Y_0^{(k)}) + \sum_{j=1}^{n} g_{ij}^{(k)}(y_j - y_j^{(k)}), \tag{7}$$

where

$$g_{ij}^{(k)} = (f_i(Y_j^{(k)}) - f_i(Y_0^{(k)})/h^{(k)}. \tag{8}$$

The solution of the system (6) where w_i is given by (7) can be written in the form (see [2]):

$$y^{(k+1)} = y_i^{(k)} - (1/\alpha^{(k)})z_i^{(k)} \times h^{(k)}, \quad i = 1, 2, \cdots, n, \tag{9}$$

where $z^{(k)} = (z_1^{(k)}, z_2^{(k)}, \cdots, z_n^{(k)})$ is a solution of the following linear system:

$$\sum_{j=1}^{n} f_i(Y^{(k)}) \times z_j = f_i(Y_0^{(k)}), \quad i = 1, 2, \cdots, n, \tag{10}$$

and

$$\alpha^{(k)} = 1 - \sum_{m=1}^{n} z_m^{(k)}. \tag{11}$$

If the sequence $\{Y^{(k)}\}$ is convergent when $k \rightarrow \infty$ and, $\{h^{(k)}\} \rightarrow 0$ then the solution of the system (1) is the limit of the sequence.

The algorithm described above is realized by means of the procedure *nielin*, which in turn uses the following two additional procedures:

(1) nonlocal procedure $f(y, z)$, which calculates for a given vector y values of the left-hand sides of the system (1), and
(2) local procedure *gauss* (u, a, y), see [1].

Input parameters:

- n number of equations in the system (1),
- h number which is used for the construction of auxiliary points (4),
- w factor multiplying the number h in every iteration,
- *eps* number used in the checking of condition (2),
- *psi* maximal admissible absolute value of the left-hand sides of the system (1).

Input/output parameters:

- y vector of dimension [1:n]. Initially this vector must contain the starting approximation; subsequently y will contain the successive approximations to the solution.

Output parameters:

- z vector of dimension [1:n] which contains the values of the equations in (1) evaluated at y,
- *nielin* assigned one of the following values:
  - −1 if any left-hand side exceeds the given value *psi*,
  - −2 if the linear system (10) is singular,
  - −3 if the sum of the roots of the system equals 1, i.e. if *alpha* = 0 (11),
  - m number of iterations, if the required accuracy *eps* is attained.

Example. To solve the system

$$y_1^2 + y_2^2 - 1 = 0,$$

$$0.75y_1^3 - y_2 + 0.9 = 0,$$

the procedure (see footnote*) was applied.

For $eps = {}_{10}{-7}$, $psi = {}_{10}3$ and $w = .1$ the following results were obtained:

*y*0	*h*	*k*	*y*	*z*	*k*	*y*	*z*
−.4	.1	7	−.9817026	$-4_{10}{-9}$			
−.1			.1904203	$9_{10}{-9}$			
−.7	.1	6	−.9817026	0			
−.2			.1904203	0			
−.7	1	−2	.3581622	$-7_{10}{-1}$	4	.3569699	$-4_{10}{-8}$
−.2			.9366243	$5_{10}{-1}$		.9341159	$-4_{10}{-8}$

* The procedure applied was:

```
procedure f(y, z);
  array y, z;
begin
  z[1] := y[1] ↑ 2 + y[2] ↑ 2 − 1;
  z[2] := .75 × y[1] ↑ 3 − y[2] + .9
end
```

The second result of the third set was obtained from a repeated call as indicated below.

Procedure *nielin* was tested on many 2 × 2 and 3 × 3 systems. If, from a given starting guess the process was divergent, the divergence was apparent after two or three iterations.

In the case when the auxiliary linear system (10) was singular or $\alpha = 0$ (11), the obtained approximation was close to the required approximation. Then the repeated call of the procedure with the obtained approximation and the starting value *h* gave the desired result after 3–4 iterations. The last remark suggests the following construction of the call of procedure *nielin*:

```
REPEAT:  k := nielin (n, h, w, eps, psi, y, z)
          ..........
         if k = −2 ∨ k = −3 then go to REPEAT
```

REFERENCES:

1. COUNTS, J. W. Algorithm 126, Gauss' method. *Comm. ACM 5* (Oct. 1962), 511.
2. PANKIEWICZ, W. About some method for solving a system of simultaneous nonlinear equations. Proc. of the Symposium: Systems of the Computers, Novosibirsk, USSR, 1967, pp. 102–105 (in Russian);

```
begin
  integer m, i, k; real alpha, r;
  Boolean b1, b2; array A[1 :n, 1 :n+1], v[1 :n];
  procedure gauss (u, a, y);
    integer u; array a, y;
  begin
    comment At this point the body of a procedure named
      Gauss (see [1]) must be supplied by the user to solve a
      u × u linear system whose coefficient matrix is stored in
      the first u rows and u columns of a, whose vector of con-
      stants (right-hand side) is stored in the (u + 1)-th column
      of a, and whose solution is given as y. If the system is singu-
      lar it should execute go to error;
  end gauss;
  m := 0;
POCZATEK:
  b1 := true;  b2 := false;  f(y, z);
  for i := 1 step 1 until n do
  begin
    A[i, n+1] := r := z[i];
    r := abs(r);
    b1 := b1 ∧ r < eps;
    b2 := b2 ∨ r > psi
  end:
  if b1 then go to KONIEC;
  if b2 then go to ALARM;
  for i := 1 step 1 until n do
  begin
    r := y[i];  y[i] := r + h;  f(y, z);
    for k := 1 step 1 until n do
      A[k, i] := z[k];
    y[i] := r
  end;
  gauss (n, A, v);
  alpha := 1;
  for i := 1 step 1 until n do
    alpha := alpha − v[i];
  if alpha = 0 then go to ALPHA;
  alpha := h/alpha;
  for i := 1 step 1 until n do
    y[i] := y[i] − v[i] × alpha;
  h := h × w; m := m + 1;
  go to POCZATEK;
KONIEC:
  nielin := m;  go to END;
ALARM:
  nielin := −1;  go to END;
error:
  nielin := −2;  go to END;
ALPHA:
  nielin := −3;
END:  end nielin
```

ALGORITHM 379
SQUANK (SIMPSON QUADRATURE USED ADAPTIVELY—NOISE KILLED)* [D1]

J. N. LYNESS (Recd. 21 Apr. 1969 and 25 Nov. 1969)
Applied Mathematics Division, Argonne National Laboratory, Argonne, IL 60439

* Work performed under the auspices of the US Atomic Energy Commission.

KEY WORDS AND PHRASES: numerical integration, integration rule, adaptive integration, automatic integration, Simpson's rule, numerical quadrature, quadrature rule, adaptive quadrature, automatic quadrature, round-off error control
CR CATEGORIES: 5.16

DESCRIPTION:

Purpose. SQUANK is an automatic numerical quadrature routine. The user provides a = A, and b = BIG, the lower and upper limits of integration, the tolerance ϵ_{tol} = ERROR he requires, and a function subprogram FUN(X) for the integrand $f(x)$. The routine returns Rf = SQUANK, where Rf is an expression of the form $Rf = \sum_{i=1}^{N} w_i f(x_i)$ which is an approximation to the integral $If = \int_a^b f(x)\,dx$.

Hopefully, this approximation is within the claimed accuracy ϵ'_{tol}, i.e. $|Rf - If| = |\epsilon_{act}| \leq \epsilon'_{tol}$.

The routine returns three other quantities, as arguments. These are

FIFTH—the fifth-order adjustment term. This may be used as an error estimate in cases in which round-off error is not significant.

NO = N—the number of calls to the function subprogram.

RUM = ϵ'_{tol}—the claimed accuracy. This is normally the same as ϵ_{tol}, the required tolerance, except in cases in which round-off error is significant, when it is higher than ϵ_{tol}.

Like many other routines, SQUANK is a special purpose routine. It is designed to treat efficiently integrands $f(x)$ having *both* the following properties:

(a) $f(x)$ and its first four derivatives are continuous in the open interval (a, b).

(b) $f(x)$ does not have high frequency oscillations.

By experiment the routine has been found efficient for the wider class of functions

(c) $g(x) = f(x)|x - x_0|^\alpha$, $\alpha \geq 0$, where $x_0 = a$ or $x_0 = b$ or $x_0 = (a+b)/2$ and $f(x)$ satisfies both (a) and (b) above.

Construction. The construction of this routine is described in detail in [3]. Briefly, it is based on the ideas of the Adaptive Simpson Quadrature routine [5-8], referred to below as ASQ, but embodies four major modifications:

(1) a different assignment of allowed error to interval and a different interval convergence criterion;

(2) interval bisection in place of trisection;

(3) inclusion of an adjustment term to give a result of polynomial degree 5 in place of degree 3;

(4) a round-off error guard (which guards against the effects of excessive round-off error in function values).

The first three modifications are of a standard nature. Their effect is described below under *Comparisons.* The fourth modification is somewhat unusual and is described by means of an example below.

Round-off Error Guard. The accuracy attainable by any quadrature routine is clearly limited by the accuracy to which the function is evaluated. The effect in an automatic routine of requesting an accuracy in excess of the accuracy of the function evaluation is described elsewhere [4] and can be catastrophic. SQUANK contains a "round-off error guard" which is Modification 4 of [3]. Thus the user may request any tolerance ϵ_{tol}, even $\epsilon_{tol} = 0$. The routine provides a result which may reflect different accuracies over different ranges of x, the local tolerance level being constrained to remain above the level of the apparent local round-off error. The overall estimated accuracy ϵ'_{tol} is returned as argument RUM.

As an example, the same problem was treated using SQUANK on two different computers. These have machine accuracy parameters $\epsilon_M = 10^{-11}$ and $\epsilon_M = 10^{-7}$, respectively. The problem was to evaluate

$$If = \int_{-1}^{1} (x^2 + 10^{-6})^{-1}\,dx \simeq 3 \times 10^3$$

with various tolerances ϵ_{tol}. A selection of the results is tabulated below.

	$\epsilon_M = 10^{-11}$			$\epsilon_M = 10^{-7}$		
ϵ_{tol}	ϵ'_{tol}	ϵ_{act}	N	ϵ'_{tol}	ϵ_{act}	N
10^{-3}	10^{-3}	-2.4×10^{-5}	1081	2.4×10^{-3}	9.1×10^{-4}	889
10^{-9}	9.9×10^{-8}	-3.2×10^{-10}	12057	1.5×10^{-3}	9.1×10^{-4}	2513
0	9.9×10^{-8}	-3.2×10^{-10}	14809	1.5×10^{-3}	9.1×10^{-4}	2513

Here ϵ_{act} is the difference between Rf and If.

It should be borne in mind that the peak of the integrand is of magnitude 10^6. Thus the accuracy in function evaluation near the peak is about 10^{-5} or 10^{-1}, respectively. Naturally, the machine with smaller word length produced a less accurate result, but at a lower cost in function evaluation. No intervention by the user was necessary. For a further comparison, the round-off error guard was disabled. For $\epsilon_{tol} \leq 10^{-7}$, the routine then required 577,197 function values, but the resulting value Rf was about the same. Thus in this example, the round-off error guard cut the computation time by a factor of 40.

The inclusion of this round-off error guard has one serious drawback. If the routine is used with an integrand which is discontinuous, or has a low order discontinuous derivative, SQUANK may take this to be evidence of round-off error and may adjust the tolerance. In these cases, the result may have a much lower accuracy than requested. However, this value of the accuracy is estimated and returned in argument RUM. The number of function values required for such a less accurate result is correspondingly lower.

Comparisons. Besides the testing carried out by the author, SQUANK has been subjected to two independent sets of extensive tests in comparison with other quadrature routines [1, 2]. The respective authors have kindly made some of their results available to me. These tests involve a set of routines, a set of functions, and eight different tolerances, all large enough so that round-off error is not significant.

Restricting attention only to functions of type (c) and to the two routines SQUANK and ASQ, the following information is reported. Of a set of 47 functions, both routines are equally reliable; ASQ is more economic than SQUANK for only one of these. For the other 46, SQUANK is more economic, generally by factors of about two [1]. Of a set of 14 functions, in all cases SQUANK is more economic, by factors ranging from 1.4 (at high accuracies) to 3 or 4 (at low accuracies) [2].

Turning to a general comparison with other routines, certain trends are apparent, although there are no clear simple conclusions. In some cases SQUANK is more economic than other routines; in other cases it is obviously much worse.

References:

1. Casaletto, J., Picket, M., and Rice, J. A comparison of some numerical integration programs. CSD TR 37, Purdue U., Lafayette, Ind., June 1969, and "SIGNUM Newsletter" *4*, 3 (Oct. 1969), 30–40.
2. Kahaner, D. K. Private communication. See also Comparison of numerical quadrature formulas, LA-4137, Los Alamos Sci. Lab., Los Alamos, N.M., June 1969.
3. Lyness, J. N. Notes on the adaptive Simpson quadrature routine. *J. ACM 16* (July 1969), 483–495.
4. Lyness, J. N. The effect of inadequate convergence criteria in automatic routines. *Comput. J. 12* (1969), 279–281.
5. McKeeman, W. M. Algorithm 145, Adaptive numerical integration by Simpson's rule. *Comm. ACM 5* (Dec. 1962), 604.
6. ——. Certification of algorithm 145, Adaptive numerical integration by Simpson's rule. *Comm. ACM 6* (Apr. 1963), 167–168.
7. ——, and Tesler, L. Algorithm 182, Nonrecursive adaptive integration. *Comm. ACM 6* (June 1963), 315.
8. ——. Algorithm 198, Adaptive integration and multiple integration. *Comm. ACM 6* (Aug. 1963), 443.

Algorithm:

```
      FUNCTION SQUANK (A,BIG,ERROR,FIFTH,RUM,NO,FUN)
C
C   S*Q*U*A*N*K  STANDS FOR * SIMPSON QUADRATURE USED ADAPTIVELY. NOISE KILLED.*
C
C
C        CALLING PROGRAM REQUIRES
C      EXTERNAL FUN
C      THIS IS FUNCTION TO BE INTEGRATED
C      A         THE LOWER LIMIT OF INTEGRATION
C      BIG       THE UPPER LIMIT
C      ERROR     THE REQUIRED TOLERANCE (ABSOLUTE ERROR)
C
C   OUTPUT
C
C      SQUANK    THE FIFTH ORDER RESULT  = THIRD + FIFTH
C      FIFTH     THE FIFTH ORDER ADJUSTMENT TERM
C      RUM       THE CLAIMED  TOLERANCE ( ADJUSTED FOR ROUNDOFF ERROR)
C      NO        THE NUMBER OF FUNCTION EVALUATIONS REQUIRED
C
C
C   NCTES ON USE.  (1)  DISCONTINUOUS FUNCTIONS
C
C      THIS ROUTINE IS BASED ON DEGREE 3 AND DEGREE 5 LOCAL POLYNOMIAL APPROX-
C      IMATION.  CONSEQUENTLY  IT SHOULD NOT BE USED WITH FUNCTIONS WHICH HAVE
C      DISCONTINUITIES IN THE FOURTH OR LOWER DERIVATIVES WITHIN THE INTERVAL OF
C      INTEGRATION.  IF THERE ARE SUCH DISCONTINUITIES,  THIS ROUTINE WILL TAKE
C      THIS TO BE EVIDENCE OF ROUND OFF ERROR IN FUNCTION VALUES AND WILL ADJUST
C      THE TOLERANCE.
C         IF THE LOCATIONS  OF SUCH DISCONTINUITIES ARE KNOWN, THE ROUTINE MAY
C      BE USEC SEPARATELY FOR EACH INTERVAL BETWEEN CONSECUTIVE DISCONTINUITIES.
C      WHILE, LIKE ALL SUCH ROUTINES, IT DISLIKES DISCONTINUITIES, IT CAN HANDLE
C      THEM IF THEY ARE LOCATED AT THE END POINTS OF THE INTEGRATION INTERVAL.
C
C
C   NCTES ON USE.  (2)  FUNCTIONS WITH HIGH-FREQUENCY OSCILLATIONS.
C
C      THE ROUTINE WILL RETURN UNRELIABLE RESULTS FOR FUNCTIONS LIKE  G(X)
C      TIMES  COS(100*X).   IF THE HIGHEST PERIOD LIKELY TO BE ENCOUNTERED IS
C      KNOWN,  THE INTERVAL SHOULD BE SUB-DIVIDED IN SUCH A WAY THAT,
C        (ONE) THERE ARE NOT MORE THAN THREE PERIODS PER INTERVAL,  AND
C        (TWO)  THE PERIOD *P* DIVIDED BY THE SUB-INTERVAL, *(B - A)*  IS NOT A
C        A SIMPLE FRACTION *N/M* WITH N OR M LESS THAN 9.
C
C
C   NCTES ON USE.  (3)  INTERVAL SUB-DIVISION
C
C      THE FAILURES DESCRIBED ABOVE ARE GENERALLY WORSE FOR *SQUANK*  THAN FOR
C      OTHER RCUTINES BECAUSE *SQUANK*  TAKES THE INCONVENIENT BEHAVIOR AS AN
C      INDICATION OF ROUND OFF ERROR. IN GENERAL SUB-DIVISION OF THE INTERVAL IS
C      ADVOCATED.  ESSENTIALLY THE USER CARRIES OUT, UNDER DRIVING PROGRAM
C      CONTROL,  A SEQUENCE OF CALCULATIONS WHICH SHOULD HAVE BEEN CARRIED OUT
C      IN THE SUBROUTINE IN ANY CASE.  IN THIS WAY HE PREVENTS CHANCE LOW ORDER
C      FALSE CCNVERGENCE AT VIRTUALLY NO ADDITIONAL COST.  NOTE THAT THE SUM OF
C      THE PARAMETERS *ERROR* FOR THE SUB-INTERVALS SHOULD CORRESPOND TO THE
C      VALUE REQUIRED FOR THE WHOLE INTERVAL.
C
C
C   NIM NUMBERING SYSTEM AND LOGIC
C
C      THE INTERVAL (A,B) IS DEFINED  NIM = 1 ,  LEVEL = 0.
C      THE INTERVAL  NIM = N, LEVEL = L  IS BISECTED, IF NECESSARY, INTO
C        TWO INTERVALS,  NIM = 2*N  AND  NIM = 2*N + 1,  BOTH AT LEVEL = L+1.
C      IF INTERVAL  NIM = N, LEVEL = L  DOES NOT CONVERGE,  THE NEXT INTERVAL
C        CONSICERED  IS  NIM = 2*N, LEVEL = L+1.
C      IF INTERVAL NIM=N, LEVEL = L  DOES CONVERGE, THE NEXT INTERVAL CONSIDERED
C        IS  NIM = M(R) + 1 , LEVEL = L-R , WHERE  M(R) IS THE FIRST
C        EVEN MEMBER OF THE SEQUENCE M(O) = N, M(S+1) = (M(9)-1)/2.  IF THIS
C      GIVES LEVEL = 0, THE CALCULATION IS COMPLETE.
C
C
C   SCALING TC AVOID EXCESSIVE DIVISION BY TWO.
C
C      THE INTERVAL(X1,X5) IS OF LENGTH  H = X5-X1.   THE POINTS X1,X2,X3,X4,X5
C        ARE THE POINTS OF QUARTERSECTION OF THIS INTERVAL AND FX1,FX2,FX3,FX4,
C        FX5 ARE THE CORRESPONDING FUNCTION VALUES.
C      EST  IS APPROXIMATION TO (6.0/H)* INTEGRAL(X1,X5).
C      (EST1+EST2) IS APPROXIMATION TO (12.0/H) * INTEGRAL(X1,X5).
C      SUM  IS APPROXIMATION TO (12.0) * INTEGRAL(A,X1).
C
C
C   STORAGE
C
C      X3ST(L) = 0.5*(X5ST(L) + X1).  THUS  X3ST(L)  COULD BE RECALCULATED
C        AT EACH STAGE TO AVOID STORAGE. ESTST(L) IS SAME IN THIS RESPECT.
C        THE RESULTS OF ABOVE RECALCULATION ARE IDENTICAL MACHINE NUMBERS.
C      X5ST(L) = X1 + (B-A)*(2**(-L)) .  THIS COULD ALSO BE RECALCULATED.  BUT
C        IN THIS CASE CALCULATION IS EXCESSIVE AND THERE IS A POSSIBILITY OF
C        ROUNC OFF ERROR ARISING BECAUSE THE SAME POINT IS BEING CALCULATED
C        IN TWO OR MORE DIFFERENT WAYS.
C
C
C   AVOIDANCE OF ROUND OFF ERROR TROUBLE
C
C      IF INTERVAL DOES NOT CONVERGE, FOLLOWING INTERVAL SHOULD HAVE  ADIFF
C      VALUE  APPROXIMATELY EQUAL TO (1/16) TIMES PREVIOUS ADIFF VALUE, CALLED
C      ADIFF1 IN THE CODE.  THERE IS A THEOREM WHICH STATES THAT, UNLESS THE
C      FOURTH DERIVATIVE OF FUN(X) VANISHES IN THE PREVIOUS INTERVAL,  ADIFF
C      IS LESS THAN OR EQUAL TO ADIFF1.  IF THIS DOES NOT HAPPEN, IT IS TAKEN
C      TO BE AN INDICATION OF POSSIBLE ROUND OFF LEVEL.   IN THIS CASE,
C      UNLESS LEV IS LESS THAN FIVE,  THE CURRENT TOLERANCE LEVEL,  CEPS,
C      IS APPROPRIATELY ADJUSTED.  HOWEVER CEPS IS RESET AS AND WHEN
C      APPEARS THAT IT SHOULD BE ADJUSTED EITHER UP OR DOWN.  IT IS REDUCED
C      IF CONVERGENCE OCCURS WITH A NON-ZERO ADIFF STRICTLY LESS THAN
C      0.25*CEPS.  AN INVOLVED SECTION OF CODING GUARDS TO SOME EXTENT AGAINST
C      AN UNREALISTIC VALUE ARISING AS A RESULT OF A ZERO IN THE FOURTH
C      DERIVATIVE.   A FACTOR  EFACT  IS CALCULATED WHICH ADJUSTS THE CLAIMED
C      TOLERANCE TO TAKE INTO ACCOUNT THESE ALTERATIONS IN THE TOLERANCE LEVEL.
C          THE ROUTINE ENTERS THESE INVOLVED SECTIONS OF CODING ONLY IF ROUND
C      OFF ERRCR APPEARS TO BE PRESENT.  IN A NORMAL (ROUND OFF ERROR FREE)
C      RUN,  THESE SECTIONS ARE SKIPPED AT A COST OF A SINGLE COMPARISON PER
C      ITERATICN (TWO FUNCTION EVALUATIONS).
C
C
C   ARBITRARY CONSTANTS
C
C      THE FOLLOWING CONSTANTS HAVE BEEN ASSIGNED IN THE LIGHT OF EXPERIENCE
C      WITH NC THEORETICAL JUSTIFICATION.
C    (1)  NO CONVERGENCE IS ALLOWED AT LEVEL = **ZERO**.  THIS MEANS THAT THE
C      ROUTINE  IS CONSTRAINED TO BASE THE RESULT ON AT LEAST 9 FUNCTION VALUES.
C    (2)  NO UPWARD ADJUSTMENT OF THE TOLERANCE LEVEL IS CONSIDERED AT LEVELS
C      LOWER THAN LEVEL = **FIVE**.  THE POINT SPACING IS THEN (BIG-A)/128.0.
C    (3)  PHYSICAL LIMIT.  HIGHEST LEVEL ALLOWED IS LEVEL = **THIRTY**.  HERE
C      CONVERGENCE IS ASSIGNED WHETHER OR NOT THE INTERVAL HAS CONVERGED.  THE
C      POINT SPACING IS THEN ABOUT (BIG-A)*2.0* 10**-10.
C    (4)  UPWARD ADJUSTMENT OF TOLERANCE LEVEL IS LIMITED IN GENERAL TO
C      A FACTOR **2.0** OR LESS.
C    (5)  DOWNWARD ADJUSTMENT OF TOLERANCE LEVEL IS INHIBITED IN GENERAL UNLESS
C      BY A FACTOR GREATER THAN **4.0**.
C
C
C   SOME NOTATION
C
C      SUM ANC SIM ARE RUNNING SUMS,  INCREASED AT STAGE EIGHT. THEY ARE
C        RESPECTIVELY 12.0 * ( THIRD ORDER APPROXIMATION TO THE INTEGRAL )
C        AND -180.0 * ( FIFTH ORDER ADJUSTMENT TO THE INTEGRAL).
C      CEPSF IS THE REQUIRED (SCALED) TOLERANCE.
C      CEPS  IS THE RUNNING VALUE OF THE ADJUSTED TOLERANCE.
C      QCEPS   =  0.25 * CEPS
C      LEVTAG = -1  OR  0,2,3  INDICATES WHETHER TOLERANCE IS NOT OR IS
C      CURRENTLY ADJUSTED.  ( SEE COMMENT IN STAGE SEVEN.)
C      EFACT  IS RUNNING SUM CORRESPONDING TO 180.0 * RUM
C      FACERR  = 15.0  OR  1.0  DEPENDING ON WHETHER TOLERANCE IS OR IS NOT
C        CURRENTLY ADJUSTED.  IF IT IS,  THERE IS NO JUSTIFICATION FOR THE
C        FIFTH ORCER ADJUSTMENT AND ACCURACY  IS NOT EXPECTED TO BE (1/15) TIMES
C        DIFFERENCE OF APPROXIMATICNS.  FACERR = 15.0  REMOVES THE BUILT IN 15.0
C        FACTCR FOR CALCULATION OF EFACT.
C      EPMACH  THE MACHINE ACCURACY PARAMETER. THE ROUND OFF ERROR GUARD DOES
C        NOT REQUIRE THIS NUMBER.  IT IS MACHINE INDEPENDENT.  THIS IS ONLY
C        USED TO HELP IN AN INITIAL GUESS IN STAGE TWO IF THE VALUE OF ERROR
C        HAPPENS TO BE ZERO.  ANY NON-ZERO NUMBER MAY BE USED INSTEAD, WITH A
C        VERY SMALL PENALTY IN NUMBER OF FUNCTION EVALUATIONS IF A COMPLETELY
C        UNREASONABLE NUMBER IS USED.
C
C
      DIMENSICN  FX3ST(30),X3ST(30),ESTST(30),FX5ST(30),X5ST(30)
      DIMENSICN PREDIF(3C)
      DOUBLE PRECISION SUM,SIM
      EPMACH = 0.0000000000075
C
C          ****    STAGE ONE    ****
C       INITIALISE ALL QUANTITIES  REQUIRED FOR CENTRAL CALCULATION (STAGE 3).
C
      SUM = 0.0
      SIM = 0.C
      CEPSF = 180.0*ERROR/(BIG - A)
      CEPS = CEPSF
      ADIFF  = 0.C
      LEVTAG = -1
      FACERR = 1.C
      XZERO = A
      EFACT = C.0
      NIM = 1
      LEV = 0
C     FIRST INTERVAL
      X1 = A
      X5 = BIG
      X3 = 0.5*(A+BIG)
      FX1= FUN(X1)
      FX3= FUN(X3)
      FX5= FUN(X5)
      NO = 3
      EST = FX1 + FX5 + 4.0*FX3
C
C          ****    STAGE TWO    ****
C       SET A STARTING VALUE FOR TOLERANCE IN CASE THAT CEPSF = 0.0
C
      IF(CEPSF) 295,205,295
  205 LEVTAG = 0
      FACERR = 15.0
      CEPS = EPMACH*ABS (FX1)
      IF(FX1) 295,210,295
  210 CEPS = EPMACH*ABS (FX3)
      LEVTAG = 3
      IF(FX3) 295,215,295
  215 CEPS = EPMACH*ABS (FX5)
      IF(FX5) 295,220,295
  220 CEPS = EPMACH
  295 QCEPS = C.25*CEPS
C     INITIALISING COMPLETE
C
C          ****    STAGE THREE    ****
```

```
C        CENTRAL CALCULATION.
C                         REQUIRES X1,X3,X5,FX1,FX3,FX5,EST,ADIFF.
C
  300 CONTINUE
      X2 = 0.5*(X1 + X3)
      X4 = 0.5*(X3 + X5)
      FX2= FUN(X2)
      FX4= FUN(X4)
      NO = NO + 2
      EST1 = FX1 + 4.0*FX2 + FX3
      EST2 = FX3 + 4.0*FX4 + FX5
      ADIFF1 = ADIFF
      DIFF = EST + EST - EST1 - EST2
      IF(LEV - 30) 305,800,800
  305 ADIFF= ABS (DIFF)
      CRIT = ADIFF - CEPS
      IF(CRIT)  700,700,400
C     END OF CENTRAL LOOP
C      NEXT STAGE IS STAGE FOUR  IN CASE OF NO NATURAL CONVERGENCE
C      NEXT STAGE IS STAGE SEVEN IN CASE OF    NATURAL CONVERGENCE
C
C           ****    STAGE FOUR     ****
C      NO NATURAL CONVERGENCE.  A COMPLEX SEQUENCE OF INSTRUCTIONS
C      FOLLOWS WHICH ASSIGNS CONVERGENCE AND / OR ALTERS TOLERANCE
C      LEVEL IN UPWARD DIRECTION IF THERE ARE INDICATIONS OF ROUND OFF
C      ERROR.
C
  400 CONTINUE
      IF(ADIFF1 - ADIFF)  410, 410, 500
C      IN A NORMAL RUN WITH NO ROUND OFF ERROR PROBLEM,  ADIFF1  IS GREATER THAN
C      ADIFF  AND THE REST OF STAGE FOUR IS OMITTED.
  410 IF(LEV - 5) 500,415,415
  415 EFACT = EFACT + CEPS *(X1 - XZERO)*FACERR
      XZERO = X1
      FACERR = 15.0
C      THE REST OF STAGE FOUR DEALS WITH UPWARD ADJUSTMENT OF TOLERANCE (CEPS)
C       BECAUSE OF SUSPECTED ROUND OFF ERROR TROUBLE.
      IF(ADIFF-2.0*CEPS)  420,420,425
C      SMALL JUMP IN CEPS. ASSIGN CONVERGENCE
  420 CEPS =      ADIFF
      LEVTAG = 0
      GO TO 78C
  425 IF( ADIFF1 - ADIFF)  435,430,435
C      LARGE JUMP IN CEPS
  430 CEPS = ADIFF
      GO TO 445
C      FACTOR TWO  JUMP IN CEPS
  435 CEPS = 2.0*CEPS
      IF(LEVTAG - 3) 440,445,445
  440 LEVTAG = 2
  445 QCEPS = G.25*CEPS
C
C           ****    STAGE FIVE     ****
C       NO ACTUAL CONVERGENCE.
C     STORE RIGHT HAND ELEMENTS
C
  500 CONTINUE
      NIM = 2*NIM
      LEV = LEV + 1
      ESTST(LEV) = EST2
      X3ST(LEV)= X4
      X5ST(LEV)= X5
      FX3ST(LEV)=FX4
      FX5ST(LEV)=FX5
      PREDIF(LEV) = ADIFF
C
C           ****    STAGE SIX     ****
C      SET UP CUANTITIES FOR CENTRAL CALCULATION.
C
C     READY TC GO AHEAD AT LEVEL LCWER WITH LEFT HAND ELEMENTS
C     X1 AND FX1 ARE THE SAME AS BEFORE
      X5 = X3
      X3 = X2
      FX5 = FX3
      FX3 = FX2
      EST = EST1
      GO TO 30C
C
C           ****    STAGE SEVEN     ****
C      NATURAL CONVERGENCE IN PREVIOUS INTERVAL. THE FOLLOWING COMPLEX SEQUENCE
C      CHECKS PRIMARILY THAT TOLERANCE LEVEL IS NOT TOO HIGH.  UNDER CERTAIN
C      CIRCUMSTANCES  NON CONVERGENCE IS ASSIGNED AND / OR TOLERANCE LEVEL
C      IS RE-SET.
C
  700 CONTINUE
C     CHECK THAT IT WAS NOT LEVEL ZERO INTERVAL.IF SO ASSIGN NON CONVERGENCE
      IF( LEV ) 400,400, 705
C
C      LEVTAG =-1   CEPS = CEPSF, ITS ORIGINAL VALUE.
C      LEVTAG = 0   CEPS IS GREATER THANCEPSF. REGULAR SITUATION.
C      LEVTAG = 2   CEPS IS GREATER THANCEPSF.CEPS PREVIOUSLY ASKED FOR A BIG
C                              JUMP, BUT DID NOT GET ONE.
C      LEVTAG = 3   CEPS IS GREATER THANCEPSF.CEPS PREVIOUSLY HAD A BIG JUMP.
C
  705 IF(LEVTAG) 800,710,710
C      IN A NCRMAL RUN WITH NO ROUND OFF ERROR PROBLEM,  LEVTAG = -1  AND THE
C      REST OF STAGE SEVEN IS OMITTED.
  710 CEPST = 15.0*CEPS
C      CEPST HERE IS FACERR*CURRENT VALUE OF CEPS
      IF(CRIT) 715,800,800
  715 IF(LEVTAG - 2)  720,740,750
C      LEVTAG = 0
  720 IF(ADIFF) 8C0,800,725
  725 IF(ADIFF - QCEPS) 730,800,80C
  730 IF(ADIFF - CEPSF) 770,77C,735
  735 LEVTAG = 0
      CEPS = ADIFF
      ÉFACT = EFACT + CEPST*(X1 - XZERO)
      XZERO = X1
      GO TO 445
C      LEVTAG = 2
  740 LEVTAG = 0
      IF(ADIFF) 765,765,725
C      LEVTAG = 3
  750 LEVTAG = 0
      IF(ADIFF) 775,775,730
  765 CEPS = ACIFF1
      GO TO 775
  770 LEVTAG = -1
      FACERR = 1.0
      CEPS = CEPSF
  775 EFACT = EFACT + CEPST*(X1 - XZERO)
      XZERO = X1
  780 CONTINUE
      QCEPS = C.25*CEPS
C           ****    STAGE EIGHT     ****
C      ACTUAL CONVERGENCE IN PREVIOUS INTERVAL.  INCREMENTS ADDED INTO
C      RUNNING SUMS
C
C     ADC INTO SUM AND SIM
  800 CONTINUE
      SUM = SUM +(EST1+EST2)*(X5-X1)
      IF(LEVTAG) 8C5,810,810
C      WE ADD INTO SIM ONLY IF WE ARE CLEAR OF ROUND OFF LEVEL.
  805 SIM = SIM + DIFF*(X5-X1)
  810 CONTINUE
C
C           ****    STAGE NINE     ****
C      SORT OUT WHICH LEVEL TO GO TO.  THIS INVOLVES NIM NUMBERING SYSTEM
C      DESCRIBED BEFORE STAGE ONE.
C
  905 NUM = NIM/2
      NOM = NIM - 2*NUM
      IF(NOM) 910,915,910
  910 NIM = NUM
      LEV = LEV - 1
      GO TO 9C5
  915 NIM = NIM + 1
C     NEW LEVEL IS SET. IF LEV=0 WE HAVE FINISHED
      IF( LEV ) 1100,11C0,1000
C
C           ****    STAGE TEN     ****
C      SET UP CUANTITIES FOR CENTRAL CALCULATION.
C
 1000 CONTINUE
      X1 = X5
      FX1= FX5
      X3 = X3ST(LEV)
      X5 = X5ST(LEV)
      FX3= FX3ST(LEV)
      FX5= FX5ST(LEV)
      EST= ESTST(LEV)
      ADIFF = PRECIF(LEV)
      GO TO 300
C
C           ****    STAGE ELEVEN     ****
C      CALCULATION NOW COMPLETE. FINALISE.
C
 1100 CONTINUE
      EFACT = EFACT + CEPS *(BIG- XZERO)*FACERR
      RUM      = EFACT/180.0
      THIRD= SUM/12.0
      FIFTH =-SIM/180.0
      SQUANK  = THIRD + FIFTH
      RETURN
C      END OF SQUANK
      END
```

Certification of Algorithm 379 [D1]

Squank (Simpson Quadrature Used Adaptively—Noise Killed) [J.N. Lyness, *Comm. ACM 13* (Apr. 1970), 260-263]

P. Hallet and E. Mund [Recd. 18 Jan. 1971 and 27 Apr. 1971]
Service de Métrologie Nucléaire, Université Libre de Bruxelles, Brussels, Belgium

Work supported by the Belgian Fonds National de la Recherche Scientifique.

Key Words and Phrases: numerical integration, integration rule, adaptive integration, automatic integration, Simpson's rule, numerical quadrature, quadrature rule, adaptive quadrature, automatic quadrature, round-off error control
CR Categories: 5.16

The algorithm was compiled and run without corrections on a CDC-6400 with a machine accuracy parameter of 0.7 × 10^{-14}. Our purpose was to test *SQUANK*'s ability to integrate a function blurred by random noise, and so the function *FUN*(*X*) is the result of applying a random perturbation *R* to some regular function $f(x)$, either by adding *R* to *x* before computing *f*, hereafter referred to as "*x*-noise", or by adding *R* to *f* after having computed it, "*y*-noise". *R* is taken as

$$R = C * (2. * RANF(X) - 1.)$$

where *C* is the noise amplitude and *RANF* is a system function generating pseudorandom numbers ($0. \leq RANF(X) \leq 1.$)

Our test program called *SQUANK* 100 times in situations involving all combinations of noise amplitude $C = 10^{-2}, 10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}$, required tolerance $\epsilon_1 = 10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}$, both noise types and the two functions $f_1(x) = k_1\, exp(x)$ and $f_2(x) = k_2(1 + 10^4 x^2)^{-1}$ integrated on [0, 1]. The constants k_1 and k_2 were chosen to normalize unblurred integrals to unity so that errors and tolerances may be seen as absolute or relative.

A rough calculation shows that y-noise causes in both integrals a deviation D that shouldn't exceed C. For x-noise, with $f(x + R) \simeq f(x) + Rf'(x)$, D shouldn't exceed respectively C and k_2C (meaning that the second function is oversensitive to x-noise by a factor $k_2 \simeq 200/\pi$).

The test program was run five times, yielding different results because the random perturbations were irreproducible. The following quantities were kept and averaged over the five runs.

$|\epsilon_2|$ actual error (specifically, ϵ_2 is the difference $SQUANK(\cdots) - 1.0$).

ϵ_3 error estimate (specifically, ϵ_3 is the value of parameter RUM as returned by $SQUANK$).

N number of function evaluations.

A sample of these averaged results is given in Table I.

Table I

C	ϵ_1	$f_1(x)$, x-noise			$f_1(x)$, y-noise		
		$\|\epsilon_2\|$	ϵ_3	N	$\|\epsilon_2\|$	ϵ_3	N
10^{-8}	10^{-12}	6.3 10^{-10}	5.5 10^{-9}	11278	8.4 10^{-10}	5.9 10^{-9}	8416
10^{-8}	10^{-4}	7.9 10^{-9}	1.0 10^{-4}	9	9.2 10^{-9}	1.0 10^{-4}	9
10^{-2}	10^{-12}	7.6 10^{-4}	4.6 10^{-3}	6986	8.9 10^{-4}	5.3 10^{-3}	8747
10^{-2}	10^{-4}	1.6 10^{-2}	1.5 10^{-3}	106	1.3 10^{-3}	1.6 10^{-3}	59
C	ϵ_1	$f_2(x)$, x-noise			$f_2(x)$, y-noise		
		$\|\epsilon_2\|$	ϵ_3	N	$\|\epsilon_2\|$	ϵ_3	N
10^{-8}	10^{-12}	2.9 10^{-9}	4.4 10^{-7}	35956	1.8 10^{-10}	5.5 10^{-9}	63254
10^{-8}	10^{-4}	5.8 10^{-6}	1.0 10^{-4}	77	5.8 10^{-6}	1.0 10^{-4}	77
10^{-2}	10^{-12}	6.3 10^{-2}	3.5 10^{-1}	9127	1.3 10^{-3}	5.2 10^{-3}	8496
10^{-2}	10^{-4}	6.9 10^{-2}	4.2 10^{-1}	3896	7.2 10^{-4}	2.5 10^{-3}	205

In 487 of the 500 calls it was found that *SQUANK*'s accuracy estimate of its own result was reliable, i.e. that $\epsilon_3 > |\epsilon_2|$. For the remaining 13 calls, the ratio $|\epsilon_2|/\epsilon_3$ ranged from 1 to 20 (in the worst cases, $\epsilon_3 \simeq \epsilon_1 < C$, just as if *SQUANK* had failed to notice the presence of the noise).

In 473 of the 500 calls it was found that *SQUANK*'s estimation was as good as could be reasonably expected, i.e., that $\epsilon_3 < \max(D, \epsilon_1) = \epsilon_4$. For the remaining 27 calls (all of them for f_2) the ratio ϵ_3/ϵ_4 never exceeded 1.15 (note that the test was made on ϵ_3, not on the actual error $|\epsilon_2|$).

ALGORITHM 380
IN-SITU TRANSPOSITION OF A RECTANGULAR MATRIX [F1]

Susan Laflin and M. A. Brebner* (Recd. 21 July 1969 and 31 Oct. 1969)
Computer Centre, University of Birmingham,
Birmingham 15, England

* Present address: Department of Mathematics, Statistics, and Computing Science, The University of Calgary, Calgary 44, Alberta, Canada.

KEY WORDS AND PHRASES: rectangular matrix, transpose
CR CATEGORIES: 5.14

Description:

The matrix $(n \times m)$ is assumed to be stored, column by column, in the one-dimensional array A, of length $m \times n$. Then the position J1 of the element a_{ij} is A(J1) where $\mathrm{J1} = n \times (j-1) + i$. This element must be moved to the position J2 of the element a_{ji} which is given by $\mathrm{J2} = m \times (i-1) + j$ and these two locations are related by the expression

$$(\mathrm{J2}-1) = m \times (\mathrm{J1}-1) - (m \times n-1) \times [(\mathrm{J1}-1)/n]$$

where $[e]$ indicates integer part of e.

It is more convenient to work in terms of index I1, taking values from 0 to $\mathrm{K} = m \times n - 1$, which gives the expression $\mathrm{I2} = m \times \mathrm{I1} - \mathrm{K} \times [\mathrm{I1}/n]$ and the value in A(I1+1) must be moved to A(I2+1). By repeating this formula we find that the transposition consists of a series of "loops", $\mathrm{I1} \to \mathrm{I2} \to \mathrm{I3} \to \cdots \to \mathrm{I1}$.

We also note that this process is symmetric. For example, if I is the smallest value in a loop then K − I is the largest value of a loop, although both these values may in fact belong to the same loop.

This is a special case of a more general result, which may be stated as follows:

Theorem. If $I1 \to I2 \to I3 \cdots \to I1$ *is a loop, then* $(K-I1) \to (K-I2) \to (K-I3) \cdots \to (K-I1)$ *is also a loop.*

Comments on Theorem. This may be two representations of the same loop or it may describe a "symmetric" pair of loops. For a 2×8 matrix the process generates the following loops, where $\mathrm{K} = 2 \times 8 - 1 = 15$.

(a)	I	$1 \to 2 \to 4 \to 8 \to 1$
	K − I	$14 \to 7 \to 11 \to 13 \to 14$
(b)	I	$5 \to 10 \to 5$
	K − I	$10 \to 5 \to 10$
(c)	I	$3 \to 6 \to 12 \to 9 \to 3$
	K − I	$12 \to 9 \to 3 \to 6 \to 12$

Case (a) is an example of a "symmetric" pair of loops. Cases (b) and (c) are both examples of a duplicated loop. An idealized picture of the circuits for this example is given in Figure 1, where the closed curves only indicate the range of the circuits, since the actual directed paths will be intertwined in a more complex manner.

Proof of Theorem. It is sufficient to show that

$$(\mathrm{K}-\mathrm{I2}) = m \times (\mathrm{K}-\mathrm{I1}) - \mathrm{K} \times [(\mathrm{K}-\mathrm{I1})/n] \cdots \qquad (1)$$

where I2 is generated from I1 by the expression given in the first paragraph.

Now

$$(\mathrm{K}-\mathrm{I2}) = \mathrm{K} - m \times \mathrm{I1} + \mathrm{K} \times [\mathrm{I1}/n].$$

Let

$$[\mathrm{I1}/n] = L_1\,, \quad [(\mathrm{K}-\mathrm{I1})/n] = L_2\,.$$

Hence it is required to prove that

$$\mathrm{K} - m \times \mathrm{I1} + \mathrm{K} \times L_1 = m \times (\mathrm{K}-\mathrm{I1}) - \mathrm{K} \times L_2\,,$$

or

$$\mathrm{K} \times L_2 = \mathrm{K} \times (m-1-L_1),$$

or

$$\mathrm{K} \times L_2 = \mathrm{K} \times L_3\,,$$

where

$$L_3 = m - 1 - L_1\,.$$

Note L_1, L_2 and L_3 are integer.

Let

$$(\mathrm{I1}/n) - L_1 = \epsilon_1\,, \quad ((\mathrm{K}-\mathrm{I1})/n) - L_2 = \epsilon_2\,.$$

From the definition of [ ] in paragraph one, it is obvious that ϵ_1 and ϵ_2 satisfy the inequalities

$$0 \le \epsilon_1 \le 1 - 1/n, \quad 0 \le \epsilon_2 \le 1 - 1/n.$$

Now

$$\begin{aligned} \mathrm{K} \times ((\mathrm{K}-\mathrm{I1})/n) &= \mathrm{K} \times ((m\times n-1-\mathrm{I1})/n) \\ &= \mathrm{K} \times (m-1/n-\mathrm{I1}/n) \\ &= \mathrm{K} \times (m-1-\mathrm{I1}/n+(1-1/n)) \\ &= \mathrm{K} \times (L_3-\epsilon_1+(1-1/n)). \end{aligned}$$

Also

$$\begin{aligned} &\mathrm{K} \times ((\mathrm{K}-\mathrm{I1})/n) = \mathrm{K} \times (L_2+\epsilon_2), \\ &\mathrm{K} \times (L_3-\epsilon_1+(1-1/n)) = \mathrm{K} \times (L_2+\epsilon_2), \end{aligned}$$

or

$$L_3 - \epsilon_1 + (1-1/n) = L_2 + \epsilon_2\,,$$

or

$$L_3 - L_2 = \epsilon_1 + \epsilon_2 - (1-1/n).$$

Therefore

$$0 + 0 - (1-1/n) \le L_3 - L_2 \le 2 \times (1-1/n) - (1-1/n),$$

or

$$| L_3 - L_2 | \le 1 - \frac{1}{n}.$$

Since L_2 and L_3 are integer and differ by less than unity, L_2 must equal L_3, hence $\mathrm{K} \times L_2 = \mathrm{K} \times L_3$, which implies that (1) is true.

Method. Each matrix will contain two or more "single elements," that is, loops consisting of only one point. The condition

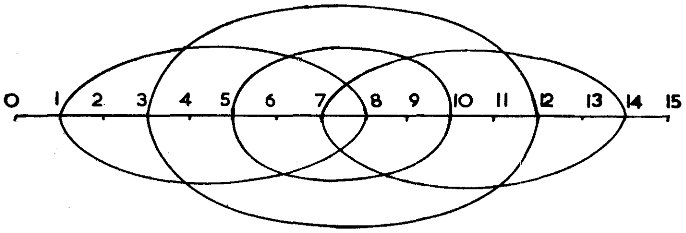


Fig. 1

for this is

$$I = m \times I - K \times [I/n].$$

Writing $I = a \times n + b$ and inserting $K = m \times n - 1$, this condition becomes

$$a(n-1) = b(m-1).$$

We shall always have the two pairs of integers (0, 0) and $(m-1, n-1)$ giving I = 0 and I = K as single elements.

The method used in the subroutine is as follows. First the number of "single elements" in the array is calculated and NCOUNT is set equal to this value; then, starting with variables I = 1 and MAX = K + 1, we search through the array, moving the elements in each loop once until all the elements have been moved. The variable NCOUNT is used to record how many elements have been moved and the process is terminated when NCOUNT $\geq mn$. For each value of I, the loop generated by I is examined, and if it contains any values less than 1 or greater than MAX, then we know that this loop has already been moved and so go on to examine the next value of I. If, however, I is the smallest value, the elements in this loop are moved round and at the same time a test is made to see if K − I also belongs to this loop. Each time a loop is completed NCOUNT is tested against mn. If K − I has been included in the loop for which I was smallest value, MAX is set equal to K − I and we return to statement number 20 to examine the next value of I. If K − I does not belong to the same loop as I, then the elements in the loop generated by K − I are also moved before returning to label 20.

The process is further speeded up by use of an array MOVE, dimension IWRK. Initially all elements of MOVE are set equal to 0 and whenever element I is moved, MOVE(I) is set equal to 2. Hence, so long as I ≤ IWRK, it is possible to detect whether or not the loop generated by I has been moved without calculating values around the loop. The value of IWRK to give the shortest possible time depends only on m and n, but so far it has not been possible to give a theoretical expression for this value. In 93 percent of the cases examined, the value of IWRK = $|\frac{1}{2}(m+n)|$ was large enough for the transposition to be completed before I exceeded IWRK. Since this condition gives the minimum execution time, we suggest the value $|\frac{1}{2}(m+n)|$ for the length of the array MOVE.

TABLE I

Size M×N	Alg. 302	Trans IWRK= (M+N)/2	2×(T1−T2) / (T1+T2)	Trans No Move	2×(T1−T3) / (T1+T3)
M×N	T1 (sec.)	T2 (sec.)		T3(sec.)	
7 × 60	0.56	0.29	0.640	0.37	0.426
7 × 70	0.96	0.32	1.012	0.37	0.897
7 × 80	0.68**	0.57	0.180	0.63	0.078
7 × 90	1.24	0.43	0.978	0.71	0.547
7 × 100	1.49	0.46	1.057	0.56	0.911
8 × 60	0.97	0.31	1.045	0.25	1.192
8 × 70	0.83	0.61	0.300	0.75	0.105
8 × 80	1.08**	0.78	0.319	1.02	0.061
8 × 90	1.50	0.46	1.065	0.37	1.216
8 × 100	1.60	0.71	0.766	1.05	0.419
9 × 60	0.91	0.47	0.649	0.63	0.365
9 × 70	1.11	0.57	0.641	0.82	0.298
9 × 80	1.53	0.46	1.082	0.37	1.223
9 × 90	1.92	0.51	1.156	0.62	1.028
9 × 100	1.88	0.63	0.997	1.10	0.528
45 × 50	4.89	2.09	0.804	2.89	0.515
45 × 60	6.10	1.59	1.173	1.38	1.261
46 × 50	4.57	2.69	0.519	3.76	0.195
46 × 60	5.99	2.88	0.701	4.16	0.361
47 × 50	4.92	3.22	0.416	3.96	0.216
47 × 60	6.59	1.66	1.195	1.45	1.279

TABLE II

Range of values of (T1 − T2)/½(T1 + T2)	*Percentage of results lying within this range*
−0.5 to 0.0	6.4%
0.0 to 0.5	29.5%
0.5 to 1.2	64.1%

This routine has been compared with a FORTRAN version of Algorithm 302 [2]. In the cases where the transpose is effected in a few loops each containing a large number of elements, our routine is very efficient, in many cases halving the time needed by Algorithm 302. It is less efficient for cases with a large number of loops, but in the cases where Algorithm 302 is faster, the difference in time is small.

Our method for the rectangular arrays is similar to that attributed to J. G. Gower in the paper by P. F. Windley [1] but using our concept of "symmetry" greatly improves the efficiency of the process. The case of a square array is detected and treated separately, exchanging pairs a_{ij} and a_{ji} instead of testing for loops.

Results. The execution times T1, for the FORTRAN version of Algorithm 302, and T2, for our routine TRANS with IWRK = $|\frac{1}{2}(m+n)|$, are given in Table I for a selection of matrices. The column T3 gives execution times for a version of TRANS from which all references to the array MOVE have been deleted. On the basis of more than 150 tests of this type, in which the relative difference between T1 and T2 was determined, only 6.4 percent gave a result favorable to Algorithm 302. A summary of the results of these tests is given in Table II.

The stars by the values of T1 indicate the condition T4 ≤ T1 ≤ T5 where T4 and T5 are execution times for TRANS with IWRK = $|\frac{1}{2}(m \times n)|$ and IWRK = 1 respectively. In these cases, the length of IWRK determines whether Algorithm 302 or TRANS is the quicker.

All the execution times refer to the ICL KDF9 computer.

Acknowledgments. The authors wish to thank Dr. S. H. Hollingdale, director of the Computer Centre, for his support and encouragement.

REFERENCES:

1. WINDLEY, P. F. Transposing matrices in a digital computer. *Comput. J. 2* (Apr. 1959), 47–48.
2. BOOTHROYD, J. Algorithm 302, Transpose vector stored array. *Comm. ACM 10* (May 1967), 292–293.

ALGORITHM:

```
      SUBROUTINE TRANS (A,M,N,MN,MOVE,IWRK,IOK)
C A IS A ONE-DIMENSIONAL ARRAY OF LENGTH MN=M*N, WHICH
C CONTAINS THE MXN MATRIX TO BE TRANSPOSED (STORED
C COLUMWISE).MOVE IS A ONE-DIMENSIONAL ARRAY OF LENGTH IWRK
C USED TO STORE IMFORMATION TO SPEED UP THE PROCESS. THE
C VALUE IWRK=(M+N)/2 IS RECOMMENDED. IOK INDICATES THE
C SUCCESS OR FAILURE OF THE ROUTINE.
C NORMAL RETURN  IOK =0
C ERRORS              IOK= -1 ,MN NOT EQUAL TO M*N.
C                     IOK= -2 ,IWRK NEGATIVE OR ZERO.
C                     IOK.GT.0, (SHOULD NEVER OCCUR).IN THIS CASE
C WE SET IOK EQUAL TO THE FINAL VALUE OF I WHEN THE SEARCH
C IS COMPLETED BUT SOME LOOPS HAVE NOT BEEN MOVED.
      DIMENSION A(MN),MOVE(IWRK)
C
C CHECK ARGUMENTS AND INITIALISE
C
      IF(M.LT.2.OR.N.LT.2)GO TO 60
      IF(MN.NE.M*N) GO TO 92
      IF(IWRK.LT.1)GO TO 93
      IF(M.EQ.N) GO TO 70
      NCOUNT=2
      M2=M-2
      DO 10 I=1,IWRK
   10   MOVE(I)=0
      IF(M2.LT.1)GO TO 12
C
C COUNT NUMBER,NCOUNT,OF SINGLE POINTS.
```

```
C
      DO 11 IA=1,M2
        IB=IA*(N-1)/(M-1)
        IF(IA*(N-1).NE.IB*(M-1))GO TO 11
        NCOUNT=NCOUNT+1
        I=IA*N+IB
        IF(I.GT.IWRK)GO TO 11
        MOVE(I)=1
   11   CONTINUE
C
C SET INTITAL VALUES FOR SEARCH.
C
   12 K=MN-1
      KMI=K-1
      MAX=MN
      I=1
C
C AT LEAST ONE LOOP MUST BE RE-ARRANGED.
C
      GO TO 30
C
C SEARCH FOR LOOPS TO REARRANGE.
C
   20          MAX=K-I
               I=I+1
               KMI=K-I
               IF(I.GT.MAX) GO TO 90
               IF(I.GT.IWRK)GO TO 21
               IF(MOVE(I).LT.1)GO TO 30
               GO TO 20
   21          IF(I.EQ.M*I-K*(I/N)) GO TO 20
               I1=I
   22          I2=M*I1-K*(I1/N)
               IF(I2.LE.I .OR. I2.GE.MAX) GO TO 23
               I1=I2
               GO TO 22
   23          IF(I2.NE.I)GO TO 20
C
C REARRANGE ELEMENTS OF A LOOP.
C
   30                I1=I
   31                B=A(I1+1)
   32                I2=M*I1-K*(I1/N)
                     IF(I1.LE.IWRK)MOVE(I1)=2
   33                NCOUNT=NCOUNT+1
                     IF(I2.EQ.I.OR.I2.GE.KMI) GO TC 35
   34                A(I1+1)=A(I2+1)
                     I1=I2
                     GO TO 32
   35                IF(MAX.EQ.KMI.OR.I2.EQ.I) GO TO 41
                     MAX=KMI
                     GO TO 34
C
C TEST FOR SYMMETRIC PAIR OF LOOPS.
C
   41          A(I1+1)=B
               IF(NCOUNT.GE.MN) GO TO 60
               IF(I2.EQ.MAX.OR.MAX.EQ.KMI) GO TO 20
               MAX=KMI
               I1=MAX
               GO TO 31
C
C NORMAL RETURN.
C
   60 IOK=0
      RETURN
C
C IF MATRIX IS SQUARE,EXCHANGE ELEMENTS A(I,J) AND  A(J,I).
C
   70 N1=N-1
      DO 71 I=1,N1
        J1=I+1
        DO 71 J=J1,N
          I1=I+(J-1)*N
          I2=J+(I-1)*M
          B=A(I1)
          A(I1)=A(I2)
          A(I2)=B
   71     CONTINUE
      GO TO 60
C
C ERROR RETURNS.
C
   90 IOK=I
   91 RETURN
   92 IOK=-1
      GO TO 91
   93 IOK=-2
      GO TO 91
C
      END
```

REMARK ON ALGORITHM 380 SUBROUTINE TO PERFORM IN-SITU TRANSPOSITION OF A RECTANGULAR MATRIX

[Susan Laflin and M. A. Brebner, *Comm. ACM* 13
(May 1970), 324–326]

RALPH LACHENMAIER
University of Colorado Graduate School Computing Center, Boulder, CO 80302

KEY WORDS AND PHRASES: rectangular matrix, transpose
CR CATEGORIES: 5.14

Laflin and Brebner compared the execution times of their transposition algorithm (Algorithm 380) and Algorithm 302 [1] when run on an ICL KDF 9 computer. This comparison showed Algorithm 380 to be faster than Algorithm 302, in most cases. In order to generalize this comparison, the same matrix transpositions were run on the CU CDC 6400 computer. Table I shows the results from the KDF 9 computer and Table II, the results from the 6400. It should be noted that Algorithm 380 did not enjoy as great an advantage on the 6400 as on the KDF 9.

TABLE I. ON THE ICL KDF—9

Range of values of (T1 − T2)/½(T1 + T2)*	*Percentage of results lying within this range*
−0.5 to 0.0	6.4%
0.0 to 0.5	29.5%
0.5 to 1.2	64.1%

TABLE II. ON THE CDC 6400

Range of values of (T1 − T2)/½(T1 + T2)*	*Percentage of results lying within this range*
−0.7 to 0.0	15.4%
0.0 to 0.5	28.9%
0.5 to 1.3	55.7%

*T1 refers to execution time for Algorithm 380. T2 refers to execution for Algorithm 302.

REFERENCES:

1. BOOTHROYD, J. Algorithm 302, Transpose vector stored array. *Comm. ACM 10* (May 1967), 292–293.

Certification of Algorithm 380 [F1]

In-Situ Transposition of a Rectangular Matrix [Susan Laflin and M.A. Brebner, *Comm. ACM 13* (May 1970), 324–326]

I.D.G. Macleod [Recd. 25 Aug. 1970]
Department of Engineering Physics, Research School of Physical Sciences, The Australian National University, Canberra, Australia, 2600

Key Words and Phrases: rectangular matrix, transpose
CR Categories: 5.14

Algorithm 380 (i.e. subroutine *TRANS*) has been extensively tested using FORTRAN IV (level G) on the A.N.U's IBM System 360 model 50; the test matrices were correctly transposed in every case. It should be pointed out that the FORTRAN convention of column-major storage of the input matrix is assumed in *TRANS*. Implementations which assume row-major matrix storage will have to be appropriately modified.

Some unnecessary computation can be avoided by changing:

```
21      IF(I.EQ.M*I-K*(I/N)) GO TO 20
        I1 =I
22      I2 =M*I1 -K*(I1/N)
        IF(I2.LE.I.OR.I2.GE.MAX) GO TO 23
        I1 = I2
        GO TO 22
23      IF(I2.NE.I) GO TO 20
```

to:

```
21      I2 =M*I -K*(I/N)
        IF(I2.LE.I.OR.I2.GE.MAX) GO TO 20
22      I2 =M*I2 -K*(I2/N)
        IF(I2.GT.I.AND.I2.LT.MAX) GO TO 22
        IF(I2.NE.I) GO TO 20
        ...
```

As an extension of the timing tests reported by Laflin and Brebner, and Lachenmaier [1], four versions of *TRANS* were timed against *TRANSPOSE* [2] and *PERMUTE* [3], using FORTRAN IV G for all routines. As in the case of *TRANS*, the method employed in *PERMUTE* is similar to that attributed by Windley [4] to J.G. Gower, but *PERMUTE* is intended for general permutations and hence does not take advantage of the symmetry present in in-situ transpositions. Execution times on the A.N.U's IBM System 360 model 50 for the test set of 21 matrices given by Laflin and Brebner are summarized in Table I; further tests on this machine have confirmed the relative efficiencies indicated.

Table I

Routine	*Execution time (sec)*
Original version of *TRANS*	
(i) *IWRK* = $(M + N)/2$	9.0
(ii) No *MOVE*	12.6
TRANS modified as recommended above	
(i) *IWRK* = $(M + N)/2$	8.2
(ii) No *MOVE*	10.8
TRANSPOSE	18.9
PERMUTE	13.7

References

1. Lachenmaier, R. Remark on Algorithm 380. *Comm. ACM 13* (May 1970), 327.
2. Boothroyd, J. Algorithm 302, Transpose vector stored array. *Comm. ACM 10* (May 1967), 292–293.
3. Macleod, I.D.G. An algorithm for in-situ permutation. *Austral. Comput. J. 2* (Feb. 1970), 16–19: (May 1970), 92 (Errata).
4. Windley, P. F. Transposing matrices in a digital computer. *Comput. J. 2* (Apr. 1959), 47–48.

ALGORITHM 381
RANDOM VECTORS UNIFORM IN SOLID ANGLE [G5]

ROBERT E. KNOP* (Recd 20 Nov. 1969 and 20 Jan. 1970)
Department of Physics, Rutgers University, New Brunswick, NJ 08903

* This work was supported in part by the National Science Foundation.

KEY WORDS AND PHRASES: random number, random vector, random number generator, probability distribution, frequency distribution, simulation, Monte Carlo
CR CATEGORIES: 5.5

```
procedure unisph (X, Y, Z);
  real X, Y, Z;
comment This procedure generates the components of random
  unit vectors distributed uniformly in solid angle. Let Z be the
  polar axis, θ the polar angle, and φ the azimuthal angle. The
  arguments returned may then be written as:
  X = sin(θ) × cos(φ)  Y, = sin(θ) × sin(φ),  Z = cos(θ)
   In this algorithm, R11 represents a procedure which returns
  random numbers which are distributed uniformly over the in-
  terval (−1, 1) [1]. The algorithm operates by the method of re-
  jection [2]. The variables X and Y are first sampled from the
  uniform distribution over the interval (−1, 1). After rejecting
  points outside of the unit disk, we may transform variables
  from X, Y to φ, S by use of the formulas X = sqrt(S) × cos(φ)
  and Y = sqrt(S) × sin(φ). It can be demonstrated that S is a
  random variable uniformly distributed over the interval (0, 1).
  The distribution of the cosine of the polar angle must be uniform
  over the interval (−1, 1). Thus Z is determined from S by the
  formula Z = 2 × S − 1 [3]. Finally, the X and Y components of
  the vector are normalized using the constraint that the vector
  be of unit length [3, 4].
   A modification of this algorithm could be used to generate
  vectors which were azimuthally uniform but have a specified
  nonuniform distribution in the cosine of the polar angle. This
  would be achieved by replacing the statement Z := 2 × S − 1
  with Z := F(S), where F is a procedure to calculate the inverse
  distribution function of Z.
   The author wishes to express his gratitude to B. Kehoe for
  comments concerning this algorithm, and to R. Nelson for doing
  much of the programming involved in testing it.
   REFERENCES:
  1. VAN GELDER, A. Some new results in pseudo-random num-
      ber generation. J. ACM 14 (Oct. 1967), 785–792.
  2. VON NEUMANN, J. Various techniques used in connection
      with random digits. Nat. Bur. of Standards Appl. Math.
      Ser. 12, 1959, p. 36.
  3. KNUTH, DONALD E. The Art of Computer Programming,
      Volume 2, Seminumerical Algorithms. Addison-Wesley,
      Reading, Mass., 1968, p. 34.
  4. KNOP, R. Remark on algorithm 334. Comm. ACM 12 (May
      1969), 281.;
begin
  real X, Y, Z, S;
  comment Rejection method yields two independent random
    variables, the azimuthal angle φ, and the square of the radius
    S.;
A:
  X := R11;  Y := R11;
  S := X ↑ 2 + Y ↑ 2;
  if S > 1 then go to A;
  comment Z must be uniform over the interval (−1, 1). It can
    be demonstrated that S is uniform over the interval (0, 1).;
  Z := 2 × S − 1;
  comment Given Z, X and Y are normalized by the constraint
    that the vector be of unit length.;
  S := sqrt((1−Z ↑ 2)/S);
  X := X × S;  Y := Y × S;
end unisph
```

Remark on Algorithm 381 [G5]

Random Vectors Uniform in Solid Angle [Robert E. Knop, *Comm. ACM 13* (May 1970), 326]

Günther F. Schrack [Recd. 1 Aug. 1970, 7 June 1971, and 4 Oct. 1971]
The University of British Columbia, Departments of Electrical Engineering and Computer Science, Vancouver 8, B.C., Canada

Key Words and Phrases: random vector generator, points uniform on sphere, spherically symmetric probability distribution
CR Categories: 5.5

Syntax corrections: The type declaration of the procedure body should be

real *S*;

and not

real *X*, *Y*, *Z*, *S*;

The sequential operator **if** in the conditional statement should be boldface and not in italic. The semicolon following the last assignment statement should be deleted. Also, in reference [3], p. 34 should be replaced by paragraph 3.4.

The following three cases are considered in this remark.

Case 1: the original algorithm, Algorithm 381.

Case 2: the modification of case 1 obtained by replacing the third last arithmetic assignment statement by

$S := 2 \times sqrt(1-S)$;

Case 3: an alternative modification of case 1 obtained by replacing the assignment statement for Z by

$Z := R11$;

possible because Z is uniformly distributed in $[-1,1]$.

The three cases were translated into Fortran IV and tested on a /360-67 running under the Michigan Terminal System. The generated vectors were all normalized. Two statistical tests were conducted in order to investigate some characteristics of these versions.

For these tests, $R11$ was replaced by $2*FRAND - 1$, where $FRAND$ is the fast random number generator in [4] with the multiplier replaced with 78125005. Each of the following two tests were repeated six times, initializing the random number generator once only with 0.461000. The sample size used for all tests was 1000.

(i) *Chi-square test for goodness of fit for each variable.* The number of categories used was 20. For case 1 the null hypothesis H_0 that each variable X, Y, and Z is uniformly distributed was rejected at the 1 percent significance level for variable X once out of the six tests; for variable Y, H_0 was rejected once at the 5 percent significance level for too good a fit; and was not rejected for variable Z. For case 3, no rejection of H_0 occurred.

(ii) *Linear correlation coefficient between pairs of the variables.* As the correlation coefficient ρ of the population has the theoretical value zero, two-tailed tests of the null hyposesis $H_0:\rho = 0$ were conducted. For case 1, all sample correlation coefficients were sufficiently small as not to reject H_0 at the 5 percent level of significance. For case 3, H_0 was rejected at the 1 percent significance level but not rejected at the 5 percent level for one out of the 18 sample correlation coefficients.

Case 2 saves one division compared to case 1 but otherwise does not change the behavior of the algorithm as tested above. Case 3 was slightly slower (less than 7 percent) than case 1 in execution time.

Finally, a comparison in execution time of case 1 with three other methods published previously [1, 2, 3] was carried out. Algorithm 381 showed a considerable advantage in speed, the three algorithms in [1, 2, and 3] were between 30 and 100 percent slower.

References

1. Cook, J.M. Rational formulae for the production of a spherically symmetric probability distribution. *Math. Tables Other Aids Comp. 11* (1957), 81–82.

2. Hicks, J.S., and Wheeling, R.F. An efficient method for generating uniformly distributed points on the surface of an n-dimensional sphere. *Comm. ACM 2* (Apr. 1959), 17–19.

3. Muller, M.E. A note on a method for generating points uniformly on n-dimensional spheres. *Comm. ACM 2* (Apr. 1959), 19–20.

4. Seraphin, D.S. A fast random number generator for IBM 360. *Comm. ACM 12* (Dec. 1969), 695.

ALGORITHM 382
COMBINATIONS OF *M* OUT OF *N* OBJECTS [G6]

Phillip J. Chase (Recd. 18 Mar. 1969 and 31 Oct. 1969)
Department of Defense, Fort Meade, MD 20755

KEY WORDS AND PHRASES: permutations and combinations, permutations
CR CATEGORIES: 5.39

```
procedure TWIDDLE (x, y, z, done, p); integer x, y, z;
  Boolean done; integer array p;
comment TWIDDLE can be used (1) in generating all combi-
  nations of m out of n objects, or (2) in generating all n-length
  sequences containing m 1's and (n-m) 0's.
    In the case (1), suppose the n objects are given by an array
  a[1:n], and let us successively store combinations in another
  array, say, c[1:m]. For the first combination, c[1] through c[m]
  are equated, respectively, to a[n-m+1] through a[n].
  TWIDDLE (x, y, z, done, p) is called. If done = true, then all
  combinations have been processed and we therefore stop. If
  not, a new combination is made available by setting c[z] equal
  to a[x]. TWIDDLE is called, and we continue on this loop until
  done = true.
    In the case (2), let the sequences of m 1's and (n - m) 0's be
  stored successively in an integer array, say, b[1:n]. The first
  sequence is obtained by setting b[1] through b[n-m] equal to 0,
  and b[n-m+1] through b[n] equal to 1. TWIDDLE (x, y, z,
  done, p) is called. If done = true, then all required sequences
  have been processed, and we therefore stop. If not, a new se-
  quence is made available by setting b[x] equal to 1, and b[y]
  equal to 0. TWIDDLE is again called, and we continue on this
  loop until done = true.
  m and n are used only in the initialization of the auxiliary
  integer array p[0:n+1], which is done in the main program as
  follows. (It is assumed that 0 ≤ m ≤ n and 1 ≤ n.) p[0] is set
  equal to n + 1, and p[n+1] is set equal to -2. p[1] through
  p[n-m] are set equal to 0. p[n-m+1] through p[n] are set
  equal, respectively, to 1 through m. If m = 0, then set p[1]
  equal to 1. done is set equal to false.
  The algorithm has several features which deserve mention.
  When used in generating combinations: (a) at each stage, only
  one combination number, namely c[z], is changed, (b)
  TWIDDLE is order preserving in the sense that at each stage
  c[1] through c[m] will equal, respectively, some a[i1] through
  a[im] where i1 through im are strictly increasing. When used in
  generating fixed-density 0-1 sequences: (c) at each stage, it is
  only necessary to change two numbers of the sequence, b[x]
  and b[y], and these are changed in a specific manner.
  The algorithm underlying this procedure was discovered by
  Leo W. Lathroum in 1965. Another algorithm which accom-
  plishes combinations by transpositions was discovered by
  Donald E. Knuth in 1964. The author has knowledge of the work
  of Lathroum and Knuth from private communications. He will
  include further detail in a mathematical paper, which will in-
  clude justification of this procedure, to be published elsewhere;
begin integer i, j, k; j := 0;
L1:
  j := j + 1; if p[j] ≤ 0 then go to L1;
  if p[j-1] = 0 then
  begin
    for i := j - 1 step -1 until 2 do p[i] := -1; p[j] = 0;
    p[1] := x := z := 1; y := j; go to L4
  end;
  if j > 1 then p[j-1] := 0;
L2:
  j := j + 1; if p[j] > 0 then go to L2;
  i := k := j - 1;
L3:
  i := i + 1; if p[i] = 0 then
  begin p[i] := -1; go to L3 end;
  if p[i] = -1 then
  begin
    p[i] := z := p[k]; x := i; y := k;
    p[k] := -1; go to L4
  end;
  if i = p[0] then begin done := true; go to L4 end;
  z := p[j] := p[i]; p[i] := 0; x := j; y := i;
L4:
end of TWIDDLE
```

REMARK ON ALGORITHM 382 [G6]
COMBINATIONS OF *M* OUT OF *N* OBJECTS

[Phillip J. Chase, *Comm. ACM 13* (June 1970), 368]

Phillip J. Chase (Recd. 18 Mar. 1969 and 31 Oct. 1969)
Department of Defense, Fort Meade, MD 20755

KEY WORDS AND PHRASES: permutations and combinations, permutations
CR CATEGORIES: 5.39

The following driver program illustrates the use of Algorithm 382.

```
begin integer m, n, i, x, y, z, q, r; Boolean done;
  integer array a, b, c[1:30], p[0:31];
  procedure TWIDDLE (x, y, z, done, p);
  comment Body of TWIDDLE is to be inserted here;
  comment TWIDDLE is here used to generate: (1) all combi-
    nations c[1:m] of a[1:n]. Here we take a[i] equal to i, each i.
    (2) all sequences b[1:n] consisting of m 1's and (n-m) 0's.
    The user must supply m and n such that 0 ≤ m ≤ n and 1 ≤ n.
    (Our declarations here require n ≤ 30.);
  ininteger (2, m); ininteger (2, n);
  for i := n step -1 until 1 do a[i] := i;
  comment We initialize the parameters p and done of
    TWIDDLE as follows;
  r := n - m;
  for i := r step -1 until 1 do p[i] := 0;
  for i := m step -1 until 1 do p[r+i] := i;
  p[0] := n + 1; p[n+1] := -2; done := false;
  if m = 0 then p[1] := 1;
  comment We initialize c[1:m];
  for i := m step -1 until 1 do c[i] := a[r+i];
  comment Next we initialize b[1:n];
  for i := m step -1 until 1 do b[r+i] := 1;
  for i := r step -1 until 1 do b[i] := 0;
  comment Now we generate and output our successive com-
    binations and sequences;
  q := 0;
```

```
L:
  q := q + 1;
  outinteger (1, q);
  for i := m − 1 step −1 until 0 do outinteger (1, c[m−i]);
  for i := n − 1 step −1 until 0 do outinteger (1, b[n−i]);
  TWIDDLE (x, y, z, done, p);
  if ¬ done then
  begin
    c[z] := a[x];  b[x] := 1;  b[y] := 0;  go to L
  end
end of driver program
```

ALGORITHM 383 PERMUTATIONS OF A SET WITH REPETITIONS [G6]

PHILLIP J. CHASE (Recd. 4 Aug. 1969 and 13 Feb. 1970)
Department of Defense, Fort Meade, MD 20755

KEY WORDS AND PHRASES: permutations and combinations, permutations
CR CATEGORIES: 5.39

```
procedure EXTENDED TWIDDLE (x, y, k, u, done, p);
  value k, u;  integer x, y, k, u;  Boolean done;  integer array
    p;
```

comment *EXTENDED TWIDDLE* is a generalization both of *TWIDDLE* [2], which is used in generating combinations by transpositions, and of the Trotter-Johnson adjacent-transposition permutation algorithms [5, 3].

In the main program, to successively store all distinct permutations of $C[I]$ numbers equal to $N[I]$ ($I=1$ to J) in an array A, take, as the first permutation, that obtained by dividing $A[1:C[1]+\cdots+C[J]]$ into J intervals and setting the $C[I]$ numbers of interval I equal to $N[I]$ ($I=1$ to J). (We assume that $J \geq 2$ and that each $C[I] \geq 1$. For *distinct* permutations, we need $N[I'] \neq N[I'']$ whenever $I' \neq I''$. For somewhat better efficiency, it is desirable, but not necessary, that the sequence $C[I]$ be non-increasing.)

EXTENDED TWIDDLE (*x*, *y*, *k*, *u*, *done*, *p*) is called. If *done* = **true**, then all permutations have been processed and we therefore stop. If not, a new permutation is made available by transposing $A[x]$ and $A[y]$, *EXTENDED TWIDDLE* is called, and we continue on this loop until *done* = **true**.

EXTENDED TWIDDLE is initialized in the main program. k is equated to J, u is equated to $C[1] + \cdots + C[J] + 1$, *done* is equated to **false**, and $p[0]$ and $p[u]$ are equated to $J + 1$. $p[1:u-1]$ is initialized by setting the members of the Ith interval, of length $C[I]$, equal to $J - I + 1 (I=1$ to $J)$;

That the procedure proceeds by transpositions (not necessarily *adjacent*, this being impossible in general) will introduce a special economy in some cases. If this feature is of no value in a particular application, then the algorithm of Bratley [1] or of Sagg [4] might be appropriate. For $J = 2$, *TWIDDLE* [2], which also has the transposition feature, will be more efficient than *EXTENDED TWIDDLE*. If each $C[I] = 1$, then Trotter's algorithm [5] for generating permutations by transpositions, is appropriate.

REFERENCES:

1. BRATLEY, P. Algorithm 306, Permutations with repetitions. *Comm. ACM 10* (July 1967), 450–451.
2. CHASE, P. J. Algorithm 382, Combinations of *M* out of *N* objects. *Comm. ACM 13* (June 1970), 368.
3. JOHNSON, S. M. Generation of permutations by adjacent transpositions. *Math. Comp. 17* (1963), 282–285.
4. SAGG, T. W. Algorithm 242, Permutations of a set with repetitions. *Comm. ACM 7* (Oct. 1964), 585.
5. TROTTER, H. F. Algorithm 115, PERM. *Comm. ACM 5* (Aug. 1962), 434–435.;

```
begin integer s, i, j, b;
  j := b := s := 0;
L1:
  j := j + 1;  if abs (p[j]) = k then
  begin if p[j] < 0 then s := j;  go to L1 end;
  if p[j−1] = k then
  begin
    for i := j − s − 1 step −1 until 2 do p[s + i] := −k;
    if s > b then p[s] := k;
    p[s+1] := p[j];  p[j] := k;  x := s + 1;  y := j;  go to L4
  end;
  if s > b then p[s] := k;
L2:
  j := j + 1;  if abs (p[j]) < k then go to L2;
  if j = u then
  begin
    if k = 2 then begin done := true;  go to L4 end;
    j := b := s;  k := k − 1;  go to L1
  end;
  i := b := j − 1;
L3:
  i := i + 1;  if p[i] = k then
  begin p[i] := −k;  go to L3 end;
  if p[i] = −k then
  begin
    p[i] := p[b];  p[b] := −k;  x := b;  y := i;  go to L4
  end;
  if i = u then
  begin
    if k = 2 then begin done := true;  go to L4 end;
    u := j;  j := b := s;  k := k − 1;  go to L1
  end;
  x := j;  y := i;  p[j] := p[i];  p[i] := k;
L4:
end EXTENDED TWIDDLE
```

REMARK ON ALGORITHM 383 [G6] PERMUTATIONS OF A SET WITH REPETITIONS [Phillip J. Chase, *Comm. ACM 13* (June 1970), 368]

PHILLIP J. CHASE (Recd. 4 Aug. 1969 and 13 Feb. 1970)
Department of Defense, Fort Meade, MD 20755

KEY WORDS AND PHRASES: permutations and combinations, permutations
CR CATEGORIES: 5.39

The following driver program illustrates the use of Algorithm 383.

```
begin integer x, y, k, u, J, Q, I, L;  Boolean done;
  integer array p[0:31], A, C, N[1:30];
  procedure EXTENDED TWIDDLE (x, y, k, u, done, p);
  comment Body of EXTENDED TWIDDLE is to be inserted
    here;
  comment Program uses EXTENDED TWIDDLE in generat-
    ing all permutations of C[I] numbers equal to N[I] (I=1 to J).
    They are successively stored in A and output. The user must
    supply: 1. J (indexing above requires J≤30); 2. C[I] (I=1 to
    J), each ≥ 1 (indexing above requires C[1]+···+C[J]≤30);
    3. N[I] (I=1 to J), distinct numbers (declarations above
    requires integer type);
  ininteger (2, J);
  for I := 1 step 1 until J do
  begin ininteger (2, C[I]);  ininteger (2, N[I]) end;
  comment The array A is initialized;
  L := 1;
  for I := 1 step 1 until J do
  for Q := C[I] step −1 until 1 do
  begin A[L] := N[I]; L := L + 1 end;
  comment EXTENDED TWIDDLE is initialized;
  L := 1;
  for I := 1 step 1 until J do
  for Q := C[I] step −1 until 1 do
  begin p[L] := J − I + 1; L := L + 1 end;
  p[0] := p[L] := J + 1;
  done := false;
  k := J; u := L;
  comment Permutations are successively generated and
    output;
  Q := 0; L := u − 1;
L1:
  Q := Q + 1;
  outinteger (1, Q);
  for I := u − 2 step −1 intil 0 do outinteger (1, A[L−I]);
  EXTENDED TWIDDLE (x, y, k, u, done, p);
  I := A[x]; A[x] := A[y]; A[y] := I;
  if ¬ done then go to L1
end of driver program
```

The following algorithm by G. W. Stewart relates to the paper by the same author in the Numerical Mathematics department of this issue on pages 365–367. This concurrent publication in Communications follows a policy announced by the Editors of the two departments in the March 1967 issue.

ALGORITHM 384
EIGENVALUES AND EIGENVECTORS OF A REAL SYMMETRIC MATRIX [F2]

G. W. Stewart (Recd. 7 Nov. 1969)

*Department of Computer Sciences, The University of Texas at Austin, *Austin, TX 78712*

* Work on this algorithm was supported in part by the National Science Foundation under grant GP-8442 and by the US Army Research Office (Durham) under grant DA-ARO(D)-31-124-G1050 at the University of Texas at Austin.

KEY WORDS AND PHRASES: real symmetric matrix, eigenvalues, eigenvectors, *QR* algorithm

CR CATEGORIES: 5.14

Description:

SYMQR finds the eigenvalues and, at the users option, the eigenvectors of a real symetric matrix. If the matrix is not initially tridiagonal, it is reduced to tridiagonal form by Householder's method [2, p. 290]. The eigenvalues of the tridiagonal matrix are calculated by a variant of the QR algorithm with origin shifts [1]. Eigenvectors are calculated by accumulating the products of the transformations used in the Householder transformations and the QR steps, a procedure which guarantees a nearly orthonormal set of approximate eigenvectors.

At each QR step the eigenvalues of the 2 × 2 submatrix in the lower right-hand corner are computed, and the one nearest the last diagonal element is distinguished. When these numbers settle down they are used as origin shifts.

The user may choose between absolute and relative convergence criteria. The former accepts the last diagonal element as an approximate eigenvalue when the last off-diagonal element is a small multiple (EPS) of the infinity norm of the matrix. The latter requires that the last off-diagonal be small compared to the last two diagonal elements. To avoid an excessive number of QR steps, an important consideration when eigenvectors are computed, the following guidelines should be followed. The convergence tolerance should not be smaller than the data warrants [2, p. 102]. The relative convergence criterion should be used only when there are eigenvalues, small compared to the elements of the matrix, that are nonetheless determined to high relative accuracy. Finally, when there is a wide disparity in the sizes of the elements of the matrix, the matrix should be arranged so that the smaller elements appear in the lower right hand corner.

The program will work with matrices whose elements very nearly underflow or overflow the range of a floating-point word. Some accuracy may be gained by accumulating inner products. The places where this should be done are signaled by the appearance of the variables SUM and SUM1.

References:

1. Stewart, G. W. Incorporating origin shifts into the symmetric QR algorithm for symmetric tridiagonal matrices. *Comm. ACM 13* (June 1970), 365–367.
2. Wilkinson, J. H. *The Algebraic Eigenvalue Problem.* Clarendon Press, Oxford, 1965.

Algorithm:

```
      SUBROUTINE SYMQR(A,D,E,KO,N,NA,EPS,ABSCNV,VEC,TRD,FAIL)
C
C
C     EXPLANATION OF THE PARAMETERS IN THE CALLING SEQUENCE.
C
C        A      A DOUBLE DIMENSIONED ARRAY.  IF THE MATRIX IS NOT
C               INITIALLY TRIDIAGONAL, IT IS CONTAINED IN THE LOWER
C               TRIANGLE OF A.  IF EIGENVECTORS ARE NOT REQUESTED
C               THE LOWER TRIANGLF OF A IS DESTROYED WHILE THE
C               ELEMENTS ABOVE THF DIAGONAL ARE LEFT UNDISTURBED.
C               IF EIGENVECTORS ARE REQUESTED, THEY ARE RETURNED IN THE
C               COLUMNS OF A.
C
C        D      A SINGLY SUBSCRIPTED ARRAY.  IF THE MATRIX IS
C               INITIALLY TRIDIAGONAL, D CONTAINS ITS DIAGONAL
C               ELEMENTS.  ON RETURN D CONTAINS THE EIGENVALUES OF
C               THE MATRIX.
C
C        E      A SINGLY SUBSCRIPTED ARRAY.  IF THE MATRIX IS
C               INITIALLY TRIDIAGONAL, E CONTAINS ITS OFF-DIAGONAL
C               ELEMENTS.  UPON RETURN E(I) CONTAINS THE NUMBER OF
C               ITERATIONS REQUIRED TO COMPUTE THE APPROXIMATE
C               EIGENVALUE D(I).
C
C        KO     A REAL VARIABLE CONTAINING AN INITIAL ORIGIN SHIFT TO
C               BE USED UNTIL THE COMPUTED SHIFTS SETTLE DOWN.
C
C        N      AN INTEGER VARIABLE CONTAINING THE ORDER OF THE
C               MATRIX.
C
C        NA     AN INTEGER VARIABLE CONTAINING THE FIRST DIMENSION
C               OF THE ARRAY A.
C
C        EPS    A REAL VARIABLE CONTAINING A CONVERGENCE TOLERANCE.
C
C        ABSCNV A LOGICAL VARIABLF CONTAINING THE VALUE .TRUE. IF
C               THE ARSOLUTE CONVERGENCE CRITERION IS TO BE USED
C               OR THE VALUE .FALSE. IF THE RELATIVE CRITERION
C               IS TO BE USED.
C
C        VEC    A LOGICAL VARIABLE CONTAINING THE VALUE .TRUE. IF
C               EIGENVECTORS ARE TO BE COMPUTED AND RETURNED IN
C               THE ARRAY A AND OTHERWISE CONTAINING THE VALUE
C               .FALSE..
C
C        TRD    A LOGICAL VARIABLF CONTAINING THF VALUE .TRUF.
C               IF THF MATRIX IS TRIDIAGONAL AND LOCATED IN THE ARRAYS
C               D AND E AND OTHERWISE CONTAINING THE VALUE .FALSE..
C
C        FAIL   AN INTEGER VARIABLE CONTAINING AN ERROR SIGNAL.
C               ON RETURN THE EIGENVALUES IN D(FAIL+1),...,D(N)
C               AND THEIR CORRESPONDING EIGENVECTORS MAY BE PRESUMED
-               ACCURATE.
C
      REAL
     1A(NA,1),D(1),E(1),KO,D1,D2,K,EPS,S2,CON,NINF,TEST,CB,CC,CD,
     2C,S,TEMP,P,PP,Q,QQ,NORM,R,TITTER,SUM,SUM1,MAX
      INTEGER
     1N,NM1,NM2,NA,FAIL,I,I1,J,L,L1,LL,LL1,NL,NU,NUM1,SINCOS,RETURN
      LOGICAL
     1ABSCNV,VEC,TRD,SHFT
      TITTER = 50.
      NM1 = N-1
      NM2 = N-2
      NINF = 0.
      ASSIGN 500 TO SINCOS
C
C     SIGNAL ERROR IF N IS NOT POSITIVE.
C
      IF(N.GT.0) GO TO 1
      FAIL = -1
      RETURN
C
C     SPECIAL TREATMENT FOR A MATRIX OF ORDER ONE.
C
    1 IF(N.GT.1) GO TO 5
      IF(.NOT.TRD) D(1) = A(1,1)
      IF(VEC) A(1,1) = 1.
      FAIL = 0
      RETURN
C
C     IF THE MATRIX IS TRIDIAGONAL, SKIP THE REDUCTION.
C
    5 IF(TRD) GO TO 100
      IF(N.EQ.2) GO TO 80
C
C     REDUCE THE MATRIX TO TRIDIAGONAL FORM BY HOUSEHOLDERS METHOD.
C
      DO 70 L=1,NM2
      L1 = L+1
      D(L) = A(L,L)
      MAX = 0.
      DO 10 I=L1,N
   10 MAX = AMAX1(MAX,ABS(A(I,L)))
      IF(MAX.NE.0.) GO TO 13
      E(L) = 0.
      A(L,L) = 1.
      GO TO 70
   13 SUM = 0.
      DO 17 I=L1,N
      A(I,L) = A(I,L)/MAX
   17 SUM = SUM + A(I,L)**2
      S2 = SUM
      S2 = SQRT(S2)
      IF(A(L1,L) .LT. 0.) S2 = -S2
      E(L) = -S2*MAX
      A(L1,L) = A(L1,L) + S2
      A(L,L) = S2*A(L1,L)
      SUM1 = 0.
      DO 50 I=L1,N
      SUM = 0.
      DO 20 J=L1,I
   20 SUM = SUM + A(I,J)*A(J,L)
      IF(I.EQ.N) GO TO 40
      I1 = I+1
      DO 30 J=I1,N
   30 SUM = SUM + A(J,L)*A(J,I)
   40 E(I) = SUM/A(L,L)
   50 SUM1 = SUM1 + A(I,L)*E(I)
      CON = .5*SUM1/A(L,L)
      DO 60 I=L1,N
      E(I) = E(I) - CON*A(I,L)
      DO 60 J=L1,I
   60 A(I,J) = A(I,J) - A(I,L)*E(J) - A(J,L)*E(I)
   70 CONTINUE
   80 D(NM1) = A(NM1,NM1)
      D(N) = A(N,N)
      F(NM1) = A(N,NM1)
C
C     IF EIGENVECTORS ARE REQUIRED, INITIALIZE A.
C
  100 IF(.NOT.VEC) GO TO 180
C
C     IF THE MATRIX WAS TRIDIAGONAL, SET A EQUAL TO THE IDENTITY MATRIX.
C
      IF(.NOT.TRD .AND. N.NE.2) GO TO 130
      DO 120 I=1,N
      DO 110 J=1,N
  110 A(I,J) = 0.
  120 A(I,I) = 1.
      GO TO 180
C
C     IF THE MATRIX WAS NOT TRIDIAGONAL, MULTIPLY OUT THE
C     TRANSFORMATIONS OBTAINED IN THE HOUSEHOLDER REDUCTION.
C
  130 A(N,N) = 1.
      A(NM1,NM1) = 1.
      A(NM1,N) = 0.
      A(N,NM1) = 0.
      DO 170 L=1,NM2
      LL = NM2-L+1
      LL1 = LL+1
      DO 140 I=LL1,N
      SUM = 0.
      DO 135 J=LL1,N
  135 SUM = SUM + A(J,LL)*A(J,I)
  140 A(LL,I) = SUM/A(LL,LL)
      DO 150 I=LL1,N
      DO 150 J=LL1,N
  150 A(I,J) = A(I,J) - A(I,LL)*A(LL,J)
      DO 160 I=LL1,N
      A(I,LL) = 0.
  160 A(LL,I) = 0.
  170 A(LL,LL) = 1.
C
C     IF AN ABSOLUTE CONVERGENCE CRITERION IS REQUESTED
C     (ABSCNV=.TRUE.), COMPUTE THE INFINITY NORM OF THE MATRIX.
C
  180 IF(.NOT.ABSCNV) GO TO 200
      NINF = AMAX1(ABS(D(1))+ABS(E(1)),ABS(D(N))+ABS(E(NM1)))
      IF(N.EQ.2) GO TO 200
      DO 190 I=2,NM1
  190 NINF = AMAX1(NINF,ABS(D(I))+ABS(F(I))+ABS(E(I-1)))
C
C     START THE QR ITERATION.
C
  200 NU = N
      NUM1 = N-1
      SHFT = .FALSE.
      K1 = K0
      TEST = NINF*EPS
      E(N) = 0.
C
C     CHECK FOR CONVERGENCE AND LOCATE THE SUBMATRIX IN WHICH THE
C     QR STEP IS TO BE PERFORMED.
C
  210 DO 220 NNL=1,NUM1
      NL = NUM1-NNL+1
      IF(.NOT.ABSCNV) TEST = EPS*AMIN1(ABS(D(NL)),ABS(D(NL+1)))
      IF(ABS(E(NL)) .LE. TEST) GO TO 230
  220 CONTINUE
      GO TO 240
  230 E(NL) = 0.
      NL = NL+1
      IF(NL .NE. NU) GO TO 240
      IF(NUM1 .EQ. 1) RETURN
      IF(E(200).NE.0.) PRINT 2000,(D(I),E(I),I=1,NU)
 2000 FORMAT(1H010E12.4/(1H 10E12.4))
      NU = NUM1
      NUM1 = NU-1
      GO TO 210
  240 E(NU) = E(NU)+FLOAT(NUM1-NL)
      IF(1. .EQ. 1.) GO TO 250
      IF(0. .EQ. 1.) GO TO 250
      FAIL = NU
      RETURN
C
C     CALCULATE THE SHIFT.
C
  250 CB = (D(NUM1)-D(NU))/2.
      MAX = AMAX1(ABS(CB),ABS(E(NUM1)))
      CB = CB/MAX
      CC = (E(NUM1)/MAX)**2
      CD = SQRT(CB**2 + CC)
      IF(CB .NE. 0.) CD = SIGN(CD,CB)
      K2 = D(NU) - MAX*CC/(CB+CD)
      IF(SHFT) GO TO 270
      IF(ABS(K2-K1) .LT. .5*ABS(K2)) GO TO 260
      K1 = K2
      K = K0
      GO TO 300
  260 SHFT = .TRUE.
  270 K = K2
C
C     PERFORM ONE QR STEP WITH SHIFT K ON ROWS AND COLUMNS
C     NL THROUGH NU
C
  300 IF(E(200).NE.0. .AND. K.LE.1.E-14*ABS(D(NL))) K=0.
      P = D(NL) - K
      Q = E(NL)
      ASSIGN 310 TO RETURN
      GO TO SINCOS,(500)
  310 DO 380 I=NL,NUM1
C
C     IF REQUIRED, ROTATE THE EIGENVECTORS.
C
      IF(.NOT.VEC) GO TO 330
      DO 320 J=1,N
      TEMP = C*A(J,I) + S*A(J,I+1)
      A(J,I+1) = -S*A(J,I) + C*A(J,I+1)
  320 A(J,I) = TEMP
C
C     PERFORM THE SIMILARITY TRANSFORMATION AND CALCULATE THE NEXT
C     ROTATION.
C
  330 D(I) = C*D(I) + S*E(I)
      TEMP = C*E(I) + S*D(I+1)
      D(I+1) = -S*E(I) + C*D(I+1)
      F(I) = -S*K
      D(I) = C*D(I) + S*TEMP
      IF(I .EQ. NUM1) GO TO 380
      IF(ABS(S) .GT. ABS(C)) GO TO 350
      R = S/C
      D(I+1) = -S*E(I) + C*D(I+1)
      P = D(I+1) - K
      Q = C*F(I+1)
      ASSIGN 340 TO RETURN
      GO TO SINCOS,(500)
  340 E(I) = R*NORM
      F(I+1) = Q
      GO TO 380
  350 P = C*E(I) + S*D(I+1)
      Q = S*E(I+1)
      D(I+1) = C*P/S + K
      E(I+1) = C*E(I+1)
      ASSIGN 360 TO RETURN
      GO TO SINCOS,(500)
  360 E(I) = NORM
  380 CONTINUE
      TEMP = C*E(NUM1) + S*D(NU)
      D(NU) = -S*E(NUM1) + C*D(NU)
      E(NUM1) = TEMP
      GO TO 210
C
C     INTERNAL PROCEDURE TO CALCULATE THE ROTATION CORRESPONDING TC
C     THE VECTOR(P,Q).
C
  500 PP = ABS(P)
      QQ = ABS(Q)
      IF(QQ .GT. PP) GO TO 510
      NORM = PP*SQRT(1. + (QQ/PP)**2)
      GO TO 520
  510 IF(QQ .EQ. 0.) GO TO 530
      NORM = QQ*SQRT(1. + (PP/QQ)**2)
  520 C = P/NORM
      S = Q/NORM
      GO TO RETURN,(310,340,360)
  530 C = 1.
      S = 0.
      NORM = 0.
      GO TO RETURN,(310,340,360)
      END
```

REMARK ON ALGORITHM 384 [F2] EIGENVALUES AND EIGENVECTORS OF A REAL SYMMETRIC MATRIX [G. W. Stewart, *Comm. ACM* 6 (June 1970), 369–371]

G. W. STEWART
Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712

KEY WORDS AND PHRASES: real symmetric matrix, eigenvalues, eigenvectors, *QR* algorithm
CR CATEGORIES: 5.14

The following changes should be made in the subroutine *SYMQR*. Change the statement:

```
  REAL
1A(NA,1),D(1),E(1),K0,D1,D2,...
```

to:

```
  REAL
1A(NA,1),D(1),E(1),K0,K1,K2,...
```

After statement number 230 delete the statements:

```
      IF(E(200).NE.0.) PRINT 2000,(D(I),E(I),I=1,NU)
2000 FORMAT(1H010E12.4/(1H 10E12.4))
```

Replace the statements:

```
240 E(NU) = E(NU)+FLOAT(NUM1-NL)
    IF(1. .EQ. 1.) GO TO 250
    IF(0. .EQ. 1.) GO TO 250
```

by:

```
240 E(NU) = E(NU)+1.
    IF(E(NU) .LE. TITTER) GO TO 250
```

Replace the statements:

```
300 IF(E(200).NE.0. .AND. K.LE.1.E-14*ABS(D(NL)))K=0.
    P = D(NL) - K
```

by:

```
300 P = D(NL) - K
```

ALGORITHM 385
EXPONENTIAL INTEGRAL $E_i(x)$ [S13]

KATHLEEN A. PACIOREK* [Recd. 16 May 1969 and 11 March 1970]

Argonne National Laboratory, Argonne, IL 60439

* Work performed under the auspices of the US Atomic Energy Commission.

KEY WORDS AND PHRASES: exponential integral, special functions, rational Chebyshev approximation

CR CATEGORIES: 5.12

DESCRIPTION:

The classical exponential integral is defined by

$$E_i(x) \equiv \int_{-\infty}^{x} \frac{e^t}{t}\,dt = -\int_{-x}^{\infty} \frac{e^{-t}}{t}\,dt, \quad x > 0$$

where the integral is to be interpreted as the Cauchy principal value. Except for the sign, it represents the natural extension of the function

$$E_1(z) \equiv \int_{z}^{\infty} \frac{e^{-t}}{t}\,dt = -E_i(-z), \qquad |\arg z| < \pi$$

to the negative real axis.

The rational approximations and corresponding intervals used in this routine are:

$$\begin{aligned}
E_{lm}(x) &= \frac{e^x}{x}\left[1 - \frac{1}{x} R_{lm}(-1/x)\right], & x \le -4 \\
&= -e^x R_{lm}(-1/x), & -4 \le x \le -1 \\
&= ln(-x) - R_{lm}(-x), & -1 \le x < 0 \\
&= ln(x/x_0) + (x - x_0) R_{lm}(x), & 0 < x \le 6 \\
&= \frac{e^x}{x} R_{lm}(1/x), & 6 \le x \le 12,\ 12 \le x \le 24 \\
&= \frac{e^x}{x}\left[1 + \frac{1}{x} R_{lm}(1/x)\right], & 24 \le x
\end{aligned}$$

where the $R_{lm}(t)$ are rational functions of degree l in the numerator and m in the denominator, and

$$x_0 = .372507410781366634461991866580$$

is the zero of $E_i(x)$. See [2, 3] for the derivation of these approximations.

In several of the ranges, it was necessary to express the rational functions either as J-fractions or as ratios of finite sums of Chebyshev polynomials, since the original forms were found to be poorly conditioned, i.e. subject to cancellation errors (subtraction of nearly equal quantities), large roundoff errors, etc. The approximations chosen for this routine have the following maximum relative errors.

Range	*Maximum Relative Error*
x less than -4	1.32D-19
$(-4, -1)$	6.33D-20
$(-1, 0)$	1.12D-21
(0, 6)	1.24D-18
(6, 12)	2.35D-18
(12, 24)	6.0D-20
x greater than 24	7.85D-19

Different approximations would naturally be required for use on computers with different word lengths. See [2, 3].

Test results. This routine was tested on an IBM System 360 model 75, where truncation is approximately 7.0D-18, usual for long normalization form. However, since this is a base 16 machine, truncation may be 1.1D-16, maximum for short normalization. The testing procedure is described in [1]. The maximum relative errors (MRE) and root mean square relative errors (RMS) follow. (Note—Argonne National Laboratory versions of DEXP and DLOG, rather than the IBM subroutine library routines, were used in these tests.)

Range	*MRE*	*RMS*
$(-150, -4)$	4.44D-16	1.52D-16
$(-4, -1)$	6.02D-16	2.50D-16
$(-1, 0)$	4.41D-16	1.08D-16
(0, 6)	6.68D-16	2.71D-16
(6, 12)	7.45D-16	3.58D-16
(12, 24)	8.18D-16	2.56D-16
(24, 100)	3.88D-16	1.36D-16

On the IBM System 360 model 75 the average time per call, excluding the jump from the calling program, was 245 microseconds. Using small perturbations of the constant coefficients in the numerators of the rational functions, in order to compensate for the biased arithmetic on the IBM System 360, it is possible to reduce the MRE by an average of 25 percent and the RMS by an average of 45 percent.

Machine dependent features. Since $E_i(0) = -\infty$, an argument of zero results in a function value which is the smallest negative floating point number on the IBM System 360. Both the Argonne version of DEXP and that of the IBM System 360 Subroutine Library treat an argument greater than 174.673 as an error and return the largest possible floating point number. Since DEXP(X) is used for X greater than 24, this exponential integral routine returns the largest possible floating point number on the IBM System 360 whenever the argument is greater than 174.673, eliminating the call to the DEXP routine. In order to maintain good relative accuracy in the vicinity of x_0, the quantity $(x - x_0)$ should be computed to higher than machine precision to preserve the low order bits of x_0. This can be readily accomplished by breaking x_0 into two parts, x_1 and x_2, such that, to the precision desired, $x_0 = x_1 + x_2$ and the floating point exponent on x_2 is much less than that of x_1. See [2]. Examining the hexadecimal representation x_0 = .5F5CA54AD2D7F0F264C3 (base 16), we see that for the IBM System 360 we might, and in fact this routine does, use x_1 = .5F5CA54AD2 (base 16) and x_2 = .0000000000D7F0F264C3 (base 16) or, x_1 = 409576229586./2**40 (base 10), in a form which will avoid decimal to hexadecimal conversion errors and x_2 = .7671772501993940D-12 (base 10). Then, $(x - x_0)$ is computed as $(x - x_0) = (x - x_1) - x_2$. Additional precautions will have to be taken to compute $ln(x/x_0)$ for x near x_0. We use a low order rational approximation to $ln(x/x_0) = log\ (1 + y)$, for $|y| < .1$, where $y = (x - x_0)/x_0$. However, a few terms in the Taylor series for $ln\ (1 + y)$ will usually suffice.

REFERENCES:

1. CLARK, N. A., CODY, W. J., HILLSTROM, K. E., AND THIELEKER, E. A. Performance statistics of the FORTRAN IV(H) library for the IBM System/360. Argonne National Laboratory Rep. ANL-7321, May 1967.
2. CODY, W. J., AND THACHER, HENRY C., JR. Chebyshev approximations for the exponential integral $E_i(x)$. *Math. Comp. 23* (Apr. 1969), 289–303.
3. CODY, W. J., AND THACHER, HENRY C., JR. Rational Cheby-

shev approximations for the exponential integral $E_1(x)$. *Math. Comp. 22* (July 1968), 641–649.

4. Rice, J. R. On the conditioning of polynomial and rational forms. *Numer. Math. 7* (1965), 426–435.

Algorithm:

```
      FUNCTION DEI(X1)
C AN EXPONENTIAL INTEGRAL ROUTINE
C FOR X GREATER THAN 0, THE EXPONENTIAL INTEGRAL, EI, IS DEFINED BY
C EI(X)=INTEGRAL(EXP(T)/T DT) , FROM T=-INFINITY TO T=X
C WHERE THE INTEGRAL IS TO BE INTERPRETED AS THE CAUCHY PRINCIPAL
C VALUE.  FOR X LESS THAN 0, EI(X)=-E1(-X), WHERE
C E1(Z)=INTEGRAL(EXP(-T)/T DT) FROM T=Z TO T=INFINITY.
      DOUBLE PRECISION DEI,X1,X,X0,XMX0,Y,R,DENM,FRAC,W,A,B,C,D,E,
     XF,P0,P1,P2,P3,P4,Q0,Q1,Q2,Q3,Q4,PX,QX,T,SUMP,SUMQ
      DIMENSION P1(9),Q1(9),P2(9),Q2(8),P3(10),Q3(9),PX(10),QX(10),
     XP4(10),Q4(9),P0(6),Q0(6)
      DIMENSION A(6),B(6),C(8),D(8),E(8),F(8)
      DATA P0/1.D0,2.23069937666899751D0,  1.70277059606809295D0,
     X5.10499279623219400D-1,  4.89089253789279154D-2,
     X3.65462224132368429D-4/
      DATA Q0/1.D0,2.73069937666899751D0,  2.73478695106925836D0,
     X1.21765962960151532D0,  2.28817933990526412D-1,
     X1.31114151194977706D-2/
      DATA P1/5.99569946892370010D9, -2.50389994886351362D8,
     X7.05921609590056747D8,  -3.36899564201591901D6,
     X8.98683291643758313D6,  7.37147790184657443D4,  2.85446881813647O1
     X5D4,  4.12626667248911939D2,  1.06395472416395801D1/
      DATA Q1/2.55926497607616350D9,  -2.79673351122984591D9,
     X8.02827782946956507D8,  -1.44980714393023883D8,  1.771583080107998
     X84D7,  -1.49575457202559218D6,  8.53771000180749097D4,  -3.0252368
     X22382274100D3, 5.12578125D1/
      DATA P2/9.98957666516551704D-1,  5.73116705744508018D0,
     X4.18102422562856622D0,  5.88658240753281111D0,  -1.941329675144307
     X02D1,  7.89472209294457221D0,  2.32730233839039141D1,  -3.67783113
     X478311458D1,  -2.46940983448361265D0/
      DATA Q2/1.14625253249016191D0,  -1.99149600231235164D2,
     X3.41365212524375539D2,  5.23165568734558614D1,  3.172794892543693 2
     X8D2,  -8.38767084189640707D0,  9.65405217429280303D2,  2.639830073
     X18024593D0/
      DATA P3/9.99933106160568740D-1,.   -1.84508623239127867D0,
     X2.65257581845279982D1,    2.49548773040205944D1,
     X-3.32361257934396228D1,    -9.13483569999874255D-1,
     X-2.10574079954804045D1,    -1.00064191398928483D1,
     X-1.86009212172643758D1,    -1.64772117246346314D0/
      DATA Q3/1.00153385204534270D0,    -1.09355619539109124D1,
     X1.99100447081774247D2,    1.19283242396860101D3,
     X4.42941317833792840D1,    2.53881931563070803D2,
     X5.99493232566740736D1,    6.40380040535241555D1,
     X9.79240359921729030D1/
      DATA P4/1.00000000000000486D0,    -3.00000000320981266D0,
     X-5.00006640413131002D0,    -7.06810977895029359D0,
     X-1.52856623636929637D1,    -7.63147701620253631D0,
     X-2.79798528624305389D1,    -1.81949664929868906D1,
     X-2.23127670777632410D2,    1.75338801265465972D2/
      DATA Q4/1.99999999999048104D0,    -2.99999894040324960D0,
     X-7.99243595776339741D0,    -1.20187763547154743D1,
     X7.04831847180424676D1,    1.17179220502086455D2,
     X1.37790390235747999D2,    3.97277109100414518D0,
     X3.97845977167414721D4/
      DATA A/-5.77215664901532863D-1,    7.54164313663016620D-1,
     X1.29849232927373234D-1,    2.40681355683977413D-2,
     X1.32084309209609371D-3,    6.57739399753264501D-5/
      DATA B/1.0D0,4.25899193811589822D-1,    7.97794718410228220-2,
     X8.30208476098771677D-3,    4.86427138393016416D-4,
     X1.30655195822848878D-5/
      DATA C/8.67745954838443744D-8, 9.99995519301390302D-1,
     X1.18483105554945844D1, 4.55930644253389823D1,
     X6.99279451291003023D1, 4.25202034768840779D1, 8.83671808803843939D
     X0, 4.01377664940664720D-1/
      DATA D/1.D0,1.28481935379156650D1, 5.64433569561803199D1,
     X1.06645183769913883D2, 8.97311097125289802D1, 3.14971849170440750D
     X1, 3.79559003762122243D0, 9.08804569188869219D-2/
      DATA E/-9.99999999999973414D-1,  -3.44061995006684895D1,
     X-4.27532671201988539D2,  -2.39601943247490540D3,
     X-6.16885210055476351D3,  -6.57609698748021179D3,
     X-2.10607737142633289D3,  -1.48990849972948169D1/
      DATA F/1.D0,3.64061995006459804D1,4.94345070209903645D2,
     X3.19027237489543304D3,  1.03370753085840977D4,
     X1.63241453557783503D4,  1.11497752871096620D4,
     X2.37813899102160221D3/
      DATA X0/.372507410781366634D0/
      X=X1
    1 IF(X.LE.0.0D0) GO TO 100
      IF(X.GE.12.D0) GO TO 60
      IF(X.GE.6.D0) GO TO 40
C X IN (0,6)
      T=X+X
      T=T/3.0D0-2.0D0
      PX(10)=0.0D0
      QX(10)=0.0D0
      PX(9)=P1(9)
      QX(9)=Q1(9)
C THE RATIONAL FUNCTION IS EXPRESSED AS A RATIO OF FINITE SUMS OF
C SHIFTED CHEBYSHEV POLYNOMIALS AND IS EVALUATED BY NOTING THAT
C T*(X)=T(2X-1) AND USING THE CLENSHAW-RICE ALGORITHM FOUND IN
C REFERENCE(4).
      DO 10 L=2,8
      I=10-L
      PX(I)=T*PX(I+1)-PX(I+2)+P1(I)
   10 QX(I)=T*QX(I+1)-QX(I+2)+Q1(I)
      R=(.5D0*T*PX(2)-PX(3)+P1(1))/(.5D0*T*QX(2)-QX(3)+Q1(1))
C (X-X0)=(X-X1)-X2,         WHERE X1=409576229586./2**40 AND
C X2=-.7671772501993940D-12.
      XMX0=(X-409576229586.D0/1099511627776.D0)-.7671772501993940D-12
      IF(DABS(XMX0) .LT.  .037D0) GO TO 15
      DEI=DLOG(X/X0)+XMX0*R
      RETURN
   15 Y=XMX0/X0
C A RATIONAL APPROXIMATION TO LOG(X/X0)=LOG(1+Y), WHERE Y=(X-X0)/X0
C AND DABS(Y) IS LESS THAN .1, THAT IS FOR DABS(X-X0) LESS THAN .037
      SUMP=((((P0(6)*Y+P0(5))*Y+P0(4))*Y+P0(3))*Y+P0(2))*Y+P0(1)
      SUMQ=((((Q0(6)*Y+Q0(5))*Y+Q0(4))*Y+Q0(3))*Y+Q0(2))*Y+Q0(1)
      DEI=(SUMP/(SUMQ*X0)+R)*XMX0
      RETURN
C X IN (6,12)
   40 DENM=P2(9)+X
      FRAC=Q2(8)/DENM
C THE RATIONAL FUNCTION IS EXPRESSED AS A J-FRACTION.
      DO 25 J=2,8
      I=9-J
      DENM=P2(I+1)+X+FRAC
   25 FRAC=Q2(I)/DENM
      DEI=DEXP(X)*((P2(1)+FRAC)/X)
      RETURN
   60 IF(X.GE.24.D0) GO TO 80
C X IN (12,24)
      DENM=P3(10)+X
      FRAC=Q3(9)/DENM
C THE RATIONAL FUNCTION IS EXPRESSED AS A J-FRACTION.
      DO 26 J=2,9
      I=10-J
      DENM=P3(I+1)+X+FRAC
   26 FRAC=Q3(I)/DENM
      DEI=DEXP(X)*((P3(1)+FRAC)/X)
      RETURN
C X GREATER THAN 24
   80 IF(X.LE.174.673D0) GO TO 90
C X IS GREATER THAN 174.673 AND DEI IS SET TO INFINITY ON IBM S/360
      DEI=7.2D75
      RETURN
   90 Y=1.0D0/X
      DENM=P4(10)+X
      FRAC=Q4(9)/DENM
C THE RATIONAL FUNCTION IS EXPRESSED AS A J-FRACTION.
      DO 28 J=2,9
      I=10-J
      DENM=P4(I+1)+X+FRAC
   28 FRAC=Q4(I)/DENM
      DEI=DEXP(X)*(Y+Y*Y*(P4(1)+FRAC))
      RETURN
  100 IF(X.NE.0.D0) GO TO 101
C X =0 AND DEI IS SET TO -INFINITY ON IBM S/360
      DEI=-7.2D75
      PRINT 500
  500 FORMAT(57H0DEI CALLED WITH A ZERO ARGUMENT, RESULT SET TO -INFINIT
     XY)
      RETURN
  101 Y=-X
  110 W=1.0D0/Y
      IF(Y.GT.4.0D0) GO TO 300
      IF(Y.GT.1.0D0) GO TO 200
C X IN (-1,0)
      DEI=DLOG(Y)-(((((A(6)*Y+A(5))*Y+A(4))*Y+A(3))*Y+A(2))*Y+A(1))/
     X (((((B(6)*Y+B(5))*Y+B(4))*Y+B(3))*Y+B(2))*Y+B(1))
      RETURN
  200 DEI=-DEXP(-Y)*((((((((C(8)*W+C(7))*W+C(6))*W+C(5))*W+C(4))*W+C(3))
C X IN (-4,-1)
     X*W+C(2))*W+C(1))/((((((((D(8)*W+D(7))*W+D(6))*W+D(5))*W+D(4))*W+
     XD(3))*W+D(2))*W+D(1)))
      RETURN
C X LESS THAN -4
  300 DEI=-DEXP(-Y)*(W*(1.0D0+W*(((((((E(8)*W+E(7))*W+E(6))*W+E(5))*W+
     XE(4))*W+E(3))*W+E(2))*W+E(1))/((((((((F(8)*W+F(7))*W+F(6))*W+F(5))
     X*W+F(4))*W+F(3))*W+F(2))*W+F(1))))
      RETURN
      END
```

CERTIFICATION OF ALGORITHM 385 [S13] EXPONENTIAL INTEGRAL *Ei* (*x*) (Kathleen A. Paciorek, *Comm. ACM 13* (July 1970) 444–445)

Edward W. Ng* (Recd. 2 Jan. 1970)
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91103

* This paper presents the results of one phase of research carried out at the Jet Propulsion Laboratory, California Institute of Technology, under Contract NAS7-100, sponsored by the National Aeronautics and Space Administration.

KEY WORDS AND PHRASES: exponential integral, special functions, rational Chebyshev approximation
CR CATEGORIES: 5.12

General discussion. This algorithm computes, for $x \geq 0$, $Ei(x) = \int_{-\infty}^{x} \frac{e^t}{t}\, dt$ and $-E_1(x) = Ei(-x)$. It is a straightforward implementation of approximations produced by Cody and Thacher, the references as given in the algorithm. It fills a gap left by previously published algorithms, e.g. Clenshaw et al. [1] and IBMSSP [2], in that it computes $Ei(x)$ for all values of real x within computer restrictions and that it is done with comparably high precision. Moreover, it is based on more efficient approximations than those used in the algorithms mentioned above. However, it is inferior in one aspect to Clenshaw et al. in that the type of approximations used makes it difficult for implementing an algorithm of variable precision, a feature included in Clenshaw et al.

The documentation and design of this algorithm are very good with clear reference to the method used, the amount and result of testing, the machine dependent features, etc. A minor defect is that the data are not identified by comments, probably because they can be recognized readily in the main body of the code.

Testing. This algorithm was compiled and executed without any modification on a UNIVAC 1108 computer. It was tested against a reference subprogram QE1EI which computes $Ei(x)$ in extended precision using a package of subroutines in 70-bit (about 21 decimal) arithmetic, written by Dr. C. L. Lawson and associates at the Jet Propulsion Laboratory. The subprogram QE1EI, written by the present author, computes $Ei(x)$ from truncated Chebyshev series for negative x [3], and from Taylor and asymptotic expansions for positive x [4, eqs. (2.1), (2.2), (2.3)]. QE1EI itself has been tested for some overlap ranges of values of x where more than one computational method is applicable and is believed to be correct to at least 19 significant decimal digits (except when x is very close to the zero of Ei, $x \approx 0.3725$ where relative accuracy is poor).

For the seven intervals of x as indicated in this algorithm, tests were made of the algorithm against QE1EI which was considered as producing the "correct" function values. Each interval was partitioned into 1000 subintervals of equal length and in each subinterval one test value of x was selected using a uniform pseudorandom number generator. The results of the tests are as follows:

Interval of x	*Maximum Relative Error*	*RMS Relative Error*
[−150, −4]	2.2D-16	5.1D-17
[−4, −1]	7.5D-17	1.2D-17
[−1, 0]	8.7D-18	1.4D-18
[0, 0.5]	1.8D-16*	3.2D-17*
[0.5, 6]	5.5D-17	1.0D-17
[6, 12]	1.6D-17	3.0D-18
[12, 24]	2.9D-17	7.1D-18
[24, 100]	8.9D-17	1.9D-17

The errors marked by * are adjusted to exclude the subinterval [0.37245, 0.37255] in which QE1EI does not have sufficient relative accuracy to give meaningful comparison with this algorithm, which is coded in such a way as to retain good relative accuracy near $x_0 \approx 0.3725 \cdots$. In fact, x_0 is given in the data as two constants with a total accuracy of 79 bits, so that on the computer with an N-bit mantissa, this algorithm produces good relative accuracy for $|x - x_0| > 2^{N-79}$. However, the additional relative accuracy thus obtained is based on the assumption that x is exactly zero in the $(N + 1)$th through 79th bits—an assumption not too realistic in most applications.

We observe that the errors found are smaller than those obtained by the author of the algorithm. This is due to smaller truncation error for long precision on the UNIVAC computer (~1.D-18).

In the range $| x - x_0 | < 0.37$, the author supplied an approximation for $\log(x/x_0)$. Such approximation is in the form of a 5-5 rational function (i.e. a fifth degree polynomial divided by another fifth degree polynomial). It should be noted that there exists in the literature a 2-2 rational approximation suitable for the same purpose. (See [5, p. 111, Index 2720].)

The two error exits occur for $Ei(0) = -\infty$, and $Ei(x > 174.673) \approx \infty$. These were tested and return ±7.2D75 which are approximately the smallest negative or largest positive floating numbers for the IBM 360. No timing test was performed owing to the apparent lack of reliability of time accounting on the UNIVAC 1108 EXEC-8 system used here.

I am indebted to C. L. Lawson and W. J. Cody for helpful discussions.

References:

1. Clenshaw, C. W., Miller, G. F., and Woodger, M. Algorithms for special functions I. *Numer. Math. 4* (1963), 403–419.
2. IBM System/360 Scientific Subroutine Package (360A-CM-03X) Version III Programmer's Manual. 1968, pp. 368–369.
3. Clenshaw, C. W. Chebyshev series for mathematical functions. NPL Math. Tables, Vol. 5, Dept. of Scien. and Indus. Res., H.M.S.O., London, 1962.
4. Cody, W. J., and Thacher, H. C., Jr. Chebyshev approximations for the exponential integral $Ei(x)$. *Math. Comp. 23* (Apr. 1969), 289–303.
5. Hart, et al. *Computer Approximations.* Wiley, New York, 1968.

REMARK ON ALGORITHM 385 [S13] EXPONENTIAL INTEGRAL *Ei*(*x*) [Kathleen A. Paciorek, *Comm. ACM 13* (July 1970), 446–447]

K. A. Redish (Recd. 3 Aug. 1970)
Department of Applied Mathematics, Hamilton, McMaster University, Ontario, Canada

KEYWORDS AND PHRASES: ANSI Fortran standard
CR CATEGORIES: 4.0, 4.22

(a) This algorithm does not conform to the standard in that the DATA statements contain array names. Section 7.2.2 of ANSI Fortran standard [*Comm. ACM* 7 (Oct. 1964), 590–625] (1) states that the list(s) of a data statement contain "names of variables and array elements." It is therefore necessary to list the elements singly. (A more readable layout can be obtained in one of the following ways:

```
     DATA
  1 A(1)/−5.77215664901532863D−1/, A(2)/7.54164313663016620D−1/,
  2 A(3)/  1.29849232927373234D−1/, A(4)/2.40681355683977413D−2/,
  3 A(5)/  1.32084309209609371D−3/, A(6)/6.57739399753264501D−5/
```

or

```
     DATA        A(1)              ,              A(2)
  1       /−5.77215664901532863D−1 ,     7.54164313663016620D−1/,
  2                A(3)            ,              A(4)
  3       / 1.29849232927373234D−1 ,     2.40681355683977413D−2/,
  4                A(5)            ,              A(6)
  5       / 1.32084309209609371D−3 ,     6.57739399753264501D−5/
```

The latter example might well be broken into three separate data statements.)

(b) In the discussion of *Machine dependent features* it is noted, in particular, that references are made to the largest positive real number and (in effect) its natural logarithm. These references are buried in the code, at the statement numbered 80, 2 lines later, and 2 lines after the statement numbered 100. I feel that these should, at least, be defined by DATA statements at the head of the program. In fact, perhaps the time is now ripe for standard names and definitions of these and other environmental entities.

Remark on Algorithm 385 [S13]

Exponential Integral *Ei*(*x*)
[Kathleen Paciorek, *Comm. ACM 13* (July 1970), 446–447]

Michael J. Frisch [Recd. 27 Jan. 1971]
University Computer Center, University of Minnesota,
Minneapolis, MN 55455

Key Words and Phrases: ANSI Fortran standard
CR Categories: 4.0, 4.22

The following items were found during compilation of the algorithms written in Fortran published to date in Communications. The MNF compiler written at the University of Minnesota for CDC 6000 Series machines by Lawrence A. Liddiard and E. James Mundstock was used to check the validity of the algorithms.

Algorithm 385 does not conform to the standard in that the function name *DEI* appears in a type statement (Section 8.3.1). It should not appear there, and the function statement should be *DOUBLE PRECISION FUNCTION DEI* (*X*1). The third statement (*PRINT* 500) after the statement numbered 100 is not among the statements allowed in standard Fortran. A comment card separates the initial line from the continuation line in the statement numbered 200 contrary to Section 3.2.1.

ALGORITHM 386
GREATEST COMMON DIVISOR OF n INTEGERS AND MULTIPLIERS* [A1]

GORDON H. BRADLEY (Recd. 14 Oct. 1969, 28 Nov. 1969, and 26 Feb. 1970)
Administrative Sciences Department, Yale University, New Haven, CT 06520

* This research supported in part by funds from the Yale Computer Center.

KEY WORDS AND PHRASES: greatest common divisor, Euclidean algorithm, number theory, diophantine equations
CR CATEGORIES: 3.15, 5.10

DESCRIPTION:

The algorithm calculates the greatest common divisor, IGCD, of n integers A(i). Multipliers Z(i) are constructed so that

$$\text{IGCD} = \text{A}(1) \times \text{Z}(1) + \cdots + \text{A}(n) \times \text{Z}(n).$$

Details of the method and comparisons to other algorithms are given in [1].

The algorithm is a new version of the Euclidean algorithm for n integers. The algorithm first calculates *gcd*(A(1), A(2)), then *gcd*(*gcd*(A(1), A(2)), A(3)), etc. The $n - 1$ calculations of the greatest common divisor of two integers is accomplished by means of a modified version of the Blankinship algorithm which is described in [1]. The $n - 1$ sets of multipliers are then used to calculate the multipliers for the A(i).

If the $n - 1$ applications of the *gcd* algorithm for two integers requires a total of k iterations, then the algorithm requires $2(n - 1) + 2k$ multiplications, $k + n - 1$ divisions, and $2k$ additions. The number of arithmetic operations is less than indicated in [1] due to a modification noted below. In [1] the following bound on k is given.

THEOREM. *k is never greater than $n - 2$ plus five times the number of digits in A(1).*

COROLLARY. *k is less than $n - 1$ plus the logarithm of A(1) to the base 1.6.*

This bound can be achieved. The bound on k can be reduced by having A(1) be the smallest number (in absolute value) among the A(i).

If at some step of the algorithm the *gcd* becomes one, then the *gcd* calculations are terminated. There is a reduction in the number of arithmetic operations in this case.

If all input integers are zero, then output is zero *gcd* and all multipliers zero.

The multipliers constructed by the algorithm are, in general, not small numbers. A minimal set of multipliers described in [1] can be constructed by a slight modification of the FORTRAN program.

REFERENCES:

1. BRADLEY, G. H. Algorithm and bound for the greatest common divisor of n integers. *Comm. ACM* 13 (July 1970), 433–436.

ALGORITHM:

```
      SUBROUTINE GCDN
     *  (N,A,Z,IGCD)
C   N      NUMBER OF INTEGERS
C   A(I)   INPUT  ARRAY OF N INTEGERS, A(I) IS USED AS WORKING STORAGE,
C          INPUT IS DESTROYED.
C   Z(I)   OUTPUT ARRAY OF N MULTIPLIERS
C   IGCD   OUTPUT, GREATEST COMMON DIVISOR OF THE A(I) INTEGERS
C
      DIMENSION A(50),Z(50)
      INTEGER A,Z,C1,C2,Y1,Y2,Q
C FIND FIRST NON-ZERO INTEGER
      DO 1 M = 1,N
      IF(A(M).NE.0) GO TO 3
    1 Z(M) = 0
C ALL ZERO INPUT RESULTS IN ZERO GCD AND ALL ZERO MULTIPLIERS
      IGCD = 0
      RETURN
C IF LAST NUMBER IS THE ONLY NON-ZERO NUMBER, EXIT IMMEDIATELY.
    3 IF(M.NE.N) GO TO 4
      IGCD = A(M)
      Z(M) = 1
      RETURN
    4 MP1 = M + 1
      MP2 = M + 2
C CHECK THE SIGN OF A(M)
      ISIGN = 0
      IF(A(M).GE.0) GO TO 5
      ISIGN = 1
      A(M) = -A(M)
C CALCULATE GCD VIA N-1 APPLICATIONS OF THE GCD ALGORITHM FOR TWO
C INTEGERS. SAVE THE MULTIPLIERS.
    5 C1 = A(M)
      DO 30 I = MP1,N
      IF(A(I).NE.0) GO TO 7
      A(I) = 1
      Z(I) = 0
      GO TO 25
    7 Y1 = 1
      Y2 = 0
      C2 = IABS(A(I))
   10 Q = C2/C1
      C2 = C2 - Q*C1
C TESTING BEFORE COMPUTING Y2 AND BEFORE COMPUTING Y1 BELOW SAVES  N - 1
C ADDITIONS AND N - 1 MULTIPLICATIONS.
      IF(C2.EQ.0) GO TO 20
      Y2 = Y2 - Q*Y1
      Q = C1/C2
      C1 = C1 - Q*C2
      IF(C1.EQ.0) GO TO 15
      Y1 = Y1 - Q*Y2
      GO TO 10
   15 C1 = C2
      Y1 = Y2
   20 Z(I) = (C1 - Y1*A(M))/A(I)
      A(I) = Y1
      A(M) = C1
C TERMINATE GCD CALCULATIONS IF GCD EQUALS ONE.
   25 IF(C1.EQ.1) GO TO 60
   30 CONTINUE
   40 IGCD = A(M)
C CALCULATE MULTIPLIERS
      DO 50 J = MP2,I
      K = I - J + 2
      KK = K + 1
      Z(K) = Z(K)*A(KK)
   50 A(K) =A(K)*A(KK)
      Z(M) = A(MP1)
      IF(ISIGN.EQ.0) GO TO 100
      Z(M) = -Z(M)
  100 RETURN
C GCD FOUND, SET REMAINDER OF THE MULTIPLIERS EQUAL TO ZERO.
   60 IP1 = I + 1
      DO 65 J = IP1,N
   65 Z(J) = 0
      GO TO 40
      END
```

Certification of Algorithm 386 [A1]

Greatest Common Divisor of n Integers and Multipliers [Gordon H. Bradley, *Comm. ACM 13* (July 1970), 447]

Larry C. Ragland and Donald I. Good [Recd. 18 June 1971, 22 August 1972, and 6 November 1972]
Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712

Key Words and Phrases: proof of algorithms, greatest common divisor, Euclidean algorithm, inductive assertion method
CR Categories: 3.15, 4.42, 5.10, 5.24

Subroutine *GCDN*, Algorithm 386 as described in [1, 2], computes the greatest common divisor, *IGCD*, of n integers $A(1), \ldots, A(n)$ by using the Euclidean algorithm to compute first $gcd(A(1), A(2))$, then $gcd(gcd(A(1), A(2)), A(3))$, etc. It also computes integer multipliers $Z(1), \ldots, Z(n)$ such that $IGCD = \sum_{i=1}^{n} A(i)Z(i)$.

A formal proof that a modified version of *GCDN* performs these two tasks has been constructed and is available from the authors. The proof employs a slight variation of one of the inductive assertion method techniques described in [3, 4]. Eight points in the program were tagged with assertions and the verification conditions for the 20 resulting paths were constructed automatically and proved manually. The initial assertion used in the proof is

$$1 \leq N_0 \leq \text{dimension } (A) = \text{dimension } (Z)$$

and the final assertion is

$$IGCD = gcd(A_0(1), \ldots, A_0(N_0)) \qquad \text{and}$$

$$IGCD = \sum_{i=1}^{N_0} A_0(i)Z(i).$$

A variable with a zero subscript denotes the value of that variable at the time the subroutine is entered, and a variable without the zero subscript denotes the value of the variable when the subroutine terminates. A proof of termination is not included, but termination can be deduced from the bounds Bradley describes for the algorithm in [2].

Three modifications of the program were necessitated by errors in the original algorithm.

(a) The two statements following statement 3

```
IGCD = A(M)
Z(M) = 1
```

should be replaced by

```
IGCD = IABS(A(M))
Z(M) = A(M)/IGCD
```

so that a positive greatest common divisor will result when all elements of array A are zero except the last, and it is negative.

(b) The second statement after statement 40

```
K = I - J + 2
```

should be replaced by

```
K = I - J + MP1.
```

The statement replaced is correct only if the first element of array A is nonzero, in which case $MP1 = 2$.

(c) Statement 60

```
60 IP1 = I + 1
```

should be replaced by

```
60 IF(I.EQ.N)GO TO 40
   IP1 = I + 1.
```

This is necessary when the greatest common divisor becomes one on the last element of array A. If N_0 is strictly less than dimension (Z) then this last change may be omitted; however, this leads to the possibility of the value of the initial parameter of a *DO* statement being greater than the value of the terminal parameter. This problem is discussed below.

The proof of *GCDN* assumes that *DO* statements consist of the following four steps.

Step 1.
Assign the control variable the value of the initial parameter.

Step 2.
Execute the body of the *DO* statement.

Step 3.
If control reaches the terminal statement, execute the terminal statement and increment the control variable by the incrementation parameter.

Step 4.
If the value of the control variable is less than or equal to the value of the terminal parameter, go back to 2; otherwise the *DO* is satisfied and execution continues out of the statement.

This interpretation of the *DO* statement makes it necessary to insert the statement

```
I = N
```

following statement 30.

For implementations in which *DO* statements are not handled as described above, other program modifications may be necessary. For example, according to the Fortran standard [5], at Step 1 the value of the initial parameter must be less than or equal to the value of the terminal parameter and in Step 4, if the *DO* is satisfied, the control variable becomes undefined. In subroutine *GCDN*, the only *DO* statement in which the value of the initial parameter may be greater than the value of the terminal parameter is *DO* 50 *J* = *MP2,I*. The program will give the correct result whether this loop is executed once (as in the proof) or is bypassed; however, if a fatal error will result, then the statement

```
IF(MP2.GT.I)GO TO 51
```

should be inserted before the statement

```
DO 50 J = MP2,I
```

and the statement following statement 50 should be labeled 51. In many implementations the control variable remains defined at the last value used in execution when the *DO* is satisfied. In this case the statement $I = N$, which was inserted earlier, may be omitted. This statement is necessary if the control variable becomes undefined, or if the control variable remains defined at its last value used in execution plus the incrementation parameter (as in this proof).

References

1. Bradley, G.H. Algorithm 386, Greatest common divisor of n integers and multipliers. *Comm. ACM 13*, 7 (July 1970), 447–448.
2. Bradley, G.H. Algorithm and bound for the greatest common divisor of n integers. *Comm. ACM 13*, 7 (July 1970), 433–436.
3. Good, D.I. Toward a man-machine system for proving program correctness. Ph.D. Th., U. of Wisconsin, June 1970.
4. Elspas, B., Levitt, K.N., Waldinger, R. J., and Waksman, A. An assessment of techniques for proving program correctness *Computing Surveys 4*, 2 (June 1972), 97–147.
5. USA Standard X3.9–1966 Fortran. United States of America Standards Institute, New York, 1966.

ALGORITHM 387
FUNCTION MINIMIZATION AND LINEAR SEARCH [E4]

K. Fielding (Recd. 23 Sept. 1969)
Computing Centre, University of Essex, Wivenhoe Park, Colchester, Essex, England

KEY WORDS AND PHRASES: function minimization, relative minimum, quasi-Newton method
CR CATEGORIES: 5.15

[Editor's note. According to tests made by the referee this algorithm is slower than FLEPOMIN, Algorithm 251, *Comm. ACM 8* (Mar. 1965), 169–170. However, in two out of six tests FLEPOMIN failed and BROMIN did not fail to find a minimum.—L.D.F.]

```
procedure Bromin (n, iterations, number, maxiters, toliter, toler-
    ance, x, f, g, h, compute f, compute g, converged);
  value n, iterations, toliter, tolerance, maxiters;
  integer n, iterations, number, maxiters;
  real toliter, tolerance, f;
  array x, g, h; Boolean converged; procedure compute f,
    compute g;
comment This procedure minimizes a function using the method
  of Broyden [1]. The parameters are described as follows. n is
  the number of independent variables. iterations is an upper
  limit on the number of iterations allowed. On exit number is the
  actual number of iterations taken. maxiters is the maximum
  number of function evaluations allowed on each linear search.
  toliter is the convergence limit for Linmin 2. tolerance is used as
  the convergence limit. A solution is assumed to have been reached
  if g(x)g'(x) < tolerance. x[1:n] is an estimate of the solution. On
  exit it is the best estimate of the solution found. f is the current
  function value f(x). g[1:n] is the current gradient vector of f(x).
  h[1:n, 1:n] is the inverse Jacobian at the solution if number ≧ n
  and if converged = true on exit. compute f(x, f) is a procedure
  provided by the user to evaluate the function at any point.
  compute g(x, g) is a procedure provided by the user to evaluate
  the gradient vector at any point. converged is a Boolean vari-
  able used as follows:
    On entry converged = true implies that x, f, g, and h all have
  been assigned values, if converged = false however it is assumed
  that just x has been assigned a value and h will be set to a unit
  diagonal matrix.
    On exit converged = true means that a solution has been found,
  converged = false means that no solution has been found. How-
  ever x is set to the best point found so far while the function
  value, gradient vector, and estimated inverse Jacobian cor-
  responding to x are in f, g, and h.
    The procedure Linmin 2 (n, maxiters, toliter, x, f, compute f,
  t, p) is used to find a linear minimum on each iteration.
    Reference:
  1. Broyden, C. G. The convergence of a class of double-rank
       minimization algorithms. J. Inst. Math. Appl. (to appear);
begin
  integer i, j; real norm, t, ythy, pty, temp;
  array p, y, hy [1:n];
  if ¬ converged then
  begin
    comment Initialize g, f, h and converged;
    compute f(x, f); compute g(x, g);
    converged := true;
    for i := 1 step 1 until n do
    begin
      h[i, i] := 1.0;
      for j := i + 1 step 1 until n do
        h[i, j] := h(j, i) := 0.0
    end of loop on i to set up h
  end of initial set up
  start of main loop on number;
  for number := 1 step 1 until iterations do
  begin
    for i := 1 step 1 until n do
    begin
      comment Evaluate the search vector p;
      p[i] := 0.0;
      for j := 1 step 1 until n do
        p[i] := p[i] − h[i, j] × g[j]
    end of loop on i to evaluate p;
    Linmin 2 (n, maxiters, toliter, x, f, compute f, t, p);
    comment Finds the optimum value of t and the values of x
      and f associated with it;
    for i := 1 step 1 until n do
      y[i] := g[i];
    comment Use y as a temporary storage location for the old
      gradient before evaluating the new one as y = g new − g old;
    compute g(x, g);
    norm := 0.0;
    for i := 1 step 1 until n do
    begin
      norm := norm + g[i] ↑ 2;
      y[i] := g[i] − y[i]
    end of loop to calculate g'g and y;
    ythy := pty := 0;
    for i := 1 step 1 until n do
    begin
      hy[i] := 0;
      for j := 1 step 1 until n do
        hy[i] := hy[i] + h[i, j] × y[j];
      ythy := ythy + y[i] × hy[i];
      pty := pty + p[i] × y[i]
    end of loop to evaluate hy, p'y and y'hy;
    temp := ythy / pty + t;
    for i := 1 step 1 until n do
    begin
      h[i, i] := h[i, i] + ((p[i] × temp−2.0 × hy[i]) × p[i])/pty;
      for j := i + 1 step 1 until n do
        h[i, j] := h[j, i] := h[i, j] + ((p[i] × temp − hy[i])
          × p[j] − hy[j] × p[i])/pty
    end of loop to update the matrix h;
    if norm < tolerance then go to successful
  end of main loop on number;
  number := iterations;
  converged := false;
successful:
end of procedure Bromin;
procedure Linmin 2 (n, maxiters, toliter, x, f, compute f, t, p);
  value n, maxiters, toliter;
  integer n, maxiters; real toliter, f, t; array x, p; procedure
    compute f;
comment This procedure carries out a linear search over t. It
  considers f(x+p × t) as a function of t alone. f is evaluated for
  three points. It is now assumed that f(t) can be approximated
  by a quadratic. If this quadratic has a minimum, then this is
  taken as a better estimate of the minimum of f(t). If, however,
  the quadratic is concave, a step is taken in the direction of the
  best point so far. If the four points obtained form an increasing
```

or decreasing sequence with respect to t then the largest is rejected. If they do not, then they must bracket a local linear minimum and the three points retained are those that most closely enclose this minimum. This process is repeated until it is felt that a good estimate of t is available (see parameter *toliter*), or until some limit on the number of function evaluations is violated (see parameter *maxiters*). The parameters are described as follows. n is the number of variables. *maxiters* is the maximum number of function evaluations allowed in the linear search. *toliter* is the tolerance for minimization, exit if *abs*((*t* − *last t*)/*t*) < *toliter*. $x[1{:}n]$ is the array of independent variables. f contains the function value $f(x)$. *compute f*(*x*,*f*) is the user provided routine to evaluate the function values at any point. t contains the best value of the scalar used for the step length. $p[1{:}n]$ is the vector which gives the direction of the step. If *tf* is the final value of t then the actual step taken is $p \times tf$. This routine is based on the procedure *quadmin* by Broyden [2].

Reference:

2. Broyden, C. G. A class of methods for solving nonlinear simultaneous equations. *Math. Comp. 19* (1965), 577–593;

```
begin
  integer i, left, center, right, count;
  real alpha, beta, gamma, last t, ptp;
  array vt, phi [1:3];
  procedure reject (j);
    value j;  integer j;
  comment  This procedure replaces one of the old values of t
    and then sorts the remaining three in ascending order of t in
    the array vt;
  begin
    procedure interchange (i, j);
      integer i, j;
    comment  if vt[i] > vt[j] interchange i and j;
    begin
      integer k;
      if vt(i) > vt[j] then
      begin k := i;  i := j;  j := k end
    end of interchange
    start of reject;
    vt[j] := t;  phi[j] := f;
    interchange (center, right);
    interchange (left, center);
    interchange (center, right)
  end of reject;
  procedure basic
  comment  This procedure evaluates a new value for x and the
    corresponding value of f;
  begin
    for i ·= 1 step 1 until n do
      x[i] := x[i] + (t−last t) × p[i];
    last t := t;  compute f(x, f)
  end of basic
  start of Linmin 2 itself;
  comment  Initialize phi, vt, left, center and right;
  phi[1] := f;
  left := 1;  center := 2;  right := 3;
  last t := vt[1] := ptp := 0.0;
  for i := 1 step 1 until n do
    ptp := ptp + p[i] ↑ 2;
  ptp := 1.0/sqrt(ptp);
  comment  ptp is now used to limit the initial step;
  vt[2] := t := if ptp < 1.0 then ptp else 1.0;
  basic;
  phi[2] := f;  vt[3] := t := t × 2.0;
  basic;
  phi[3] := f;
  comment  Sets up first three values before entering main loop;
  for count := 3 step 1 until maxiters do
  begin
    alpha := vt[2] − vt[3];
    beta := vt[3] − vt[1];
    gamma := vt[1] − vt[2];
    alpha := −(phi[1]×alpha+phi[2]×beta+phi[3]
      × gamma)/(alpha × beta × gamma);
    beta := (phi[1] − phi[2])/gamma − alpha × (vt[1] + vt[2]);
    comment  If the quadratic through the three points is con-
      vex, t is chosen as the minimum of it. If it is concave, how-
      ever, t is chosen as a step in the direction of steepest descent;
    t := if alpha > 0.0 then − beta/(2.0 × alpha)
      else if phi [right] > phi [left]
      then 3.0 × vt [left] − 2.0 × vt [center]
      else 3.0 × vt [right] − 2.0 × vt [center];
    if abs((t − last t)/t) < toliter then
    begin
      t := last t;  go to exit
    end of exit where minimum has been found;
    basic;
    if t > vt [right]
      ∨ (t > vt [center] ∧ f < phi [center])
      ∨ (t > vt [left] ∧ t < vt [center] ∧ f> phi [center])
      then reject (left) else reject (right);
    comment  Choose which point to reject;
  end of main loop which used count as an index;
exit:
end of Linmin 2
```

ALGORITHM 388
RADEMACHER FUNCTION [S22]

H. HÜBNER
Forschungsinstitut des FTZ der Deutschen Bundespost,
Darmstadt
H. KREMER, K. O. LINN, AND W. SCHWERING (Recd. 16 Jan. 1970)
Institut für Allgemeine Nachrichtentechnik der Technischen Hochschule Darmstadt, Germany

KEY WORDS AND PHRASES: Rademacher function
CR CATEGORIES: 5.12, 5.13

```
integer procedure radfun(k, x);
  value k, x;  integer k;  real x;
```

comment The procedure *radfun* computes the Rademacher function $r_k(x)$ as defined in [1, 2, 3]. This definition is used in recent papers and differs from the original definition [4] by an opposite sign. The Rademacher functions $r_k(x)$ form an incomplete set of orthogonal, normalized, periodic square wave functions with period equal to one. They assume only the values +1 and −1. The Rademacher function $r_k(x)$ may be defined either by the formula

$$r_k(x) = sgn[sin(2\pi 2^k x)] \qquad (1)$$

or by the following algorithm:

Let x be in the interval $\frac{m}{2^{k+1}} \leq x < \frac{m+1}{2^{k+1}}$, $m = 0, \pm 1, \cdots$

then

$$r_k(x) = \begin{cases} +1 & \text{for } m \text{ even} \\ -1 & \text{for } m \text{ odd.} \end{cases} \qquad (2)$$

The index k must be a nonnegative integer and the argument x can be any real number in the range $-\infty \leq x \leq \infty$.
Equations (1) and (2) show that $r_k(x)$ is piecewise constant and has 2^{k+1} jump discontinuities in the interval $0 \leq x < 1$. The procedure *radfun* uses eq. (2) for computation.

REFERENCES:

1. PALEY, R. E. A remarkable series of orthogonal functions. *Proc. London Math. Soc. Ser. 2, 34* (1932), 241–279.
2. FINE, N. J. On the Walsh functions. *Trans. Amer. Math. Soc. 65* (1949), 372–414.
3. MORGENTHALER, G. W. On Walsh-Fourier series. *Trans. Amer. Math. Soc. 84* (1957), 472–507.
4. RADEMACHER, H. Einige Sätze von allgemeinen Orthogonalfunktionen. *Math. Ann. 87* (1922), 112–138;

```
begin
  integer r;
    x := x − entier(x);
    r := entier(x×2↑(k+1));
    radfun := if r/2 = r ÷ 2 then 1 else − 1
end
```

ALGORITHM 389
BINARY ORDERED WALSH FUNCTIONS [S22]

H. Hübner
Forschungsinstitut des FTZ der Deutschen Bundespost,
Darmstadt
H. Kremer, K. O. Linn, and W. Schwering (Recd.
16 Jan. 1970)
Institut für Allgemeine Nachrichtentechnik der Technischen Hochschule Darmstadt, Germany

KEY WORDS AND PHRASES: Walsh functions, binary ordered Walsh functions
CR CATEGORIES: 5.12, 5.13

```
integer procedure binwal (k, x);
  value k, x;  integer k;  real x;
```

comment The procedure *binwal* computes the binary ordered Walsh function $w_k(x)$ as defined in [1, 2, 3, 4]. These functions form a complete set of orthogonal, normalized rectangular functions which are periodic with period equal to one. They assume only the values +1 and −1. Using the Rademacher functions $r_k(x)$ [5], the function $w_k(x)$ may be defined in the following way:

Write k as a binary number

$$k = \sum_{i=0}^{m} a_i 2^i, \quad a_i \in (0, 1),$$

then

$$w_k(x) = \prod_{i=0}^{m} [r_i(x)]^{a_i}.$$

The functions are defined for k a nonnegative integer in the range $-\infty \leq x \leq \infty$.

In *binwal* the procedure *radfun* [5] is used.

References:

1. Paley, R. E. A remarkable series of orthogonal functions. *Proc. London Math. Soc. Ser. 2, 34* (1932), 241–279.
2. Fine, N. J. On the Walsh functions. *Trans. Amer. Math. Soc. 65* (1949), 372–414.
3. Morgenthaler, G. W. On Walsh-Fourier series. *Trans. Amer. Math. Soc. 84* (1957), 472–507.
4. Hammond, J. L., and Johnson, R. S. A review of orthogonal square-wave functions and their applications to linear networks. *J. Franklin Inst. 273* (1962), 211–225.
5. Hübner, H., Kremer, H., Linn, K. O., and Schwering, W. Algorithm 388, Rademacher function. *Comm ACM 13* (Aug. 1970), 510;

```
begin
  integer i, l, m, ww;
  l := k;  m := ww := 1;
  i := −1;
  for i := i + 1 while m ≤ l do
  begin
    if k/(m + m) ≠ k ÷ (m + m) then
      begin ww := ww × radfun(i, x);  k := k − m end;
    m := m + m
  end;
  binwal := ww
end
```

ALGORITHM 390
SEQUENCY ORDERED WALSH FUNCTIONS [S22]

H. Hübner
Forschungsinstitut des FTZ der Deutschen Bundespost, Darmstadt
H. Kremer, K. O. Linn, and W. Schwering (Recd. 16 Jan. 1970)
Institut für Allgemeine Nachrichtentechnik der Technischen Hochschule Darmstadt, Germany

KEY WORDS AND PHRASES: Walsh functions, sequency ordered Walsh functions
CR CATEGORIES: 5.12, 5.13

```
integer procedure seqwal(k, x);
  value k, x;  integer k;  real x;
```

comment The procedure *seqwal* computes the sequency ordered Walsh function $wal_k(x)$ as defined in [1, 2]. These functions form a complete set of orthogonal, normalized, periodic rectangular functions with period equal to one. They are closely related to the binary ordered Walsh functions $w_k(x)$ [3]. The set of $wal_k(x)$ consists of the same functions as the set of $w_k(x)$ but in another scheme of ordering. The set of $w_k(x)$ is ordered with regard to the binary decomposition of the index k, whereas the set $wal_k(x)$ is ordered according to the number of jump discontinuities in the open basic interval $0 < x < 1$ in the sense that $wal_k(x)$ has exactly k jumps. The relation between $wal_k(x)$ and $w_k(x)$ is given by $wal_k(x) = w_n(x)$ with $n = k \oplus (k \div 2)$, where $\oplus$ means the addition modulo 2 (binary addition without carry). The functions are defined for k a nonnegative integer in the range $-\infty \leq x \leq \infty$. In *seqwal* the procedure *binwal* [3] is used.

References:

1. Walsh, J. L. A closed set of normal orthogonal functions. *Amer. J. Math., Vol. 45* (1923), 5–24.
2. Harmuth, H. F. *Transmission of Information by Orthogonal Functions.* Springer-Verlag, New York, 1969.
3. Hübner, H., Kremer, H., Linn, K. O., and Schwering, W. Algorithm 389, Binary ordered Walsh functions. *Comm. ACM 13* (Aug. 1970), 511;

```
begin
  integer i, k2, l, m, m2, n, v1, v2;
  k2 := k ÷ 2;  l := k;  m := 1;  n := 0;
  i := 0;
  for i := i + 1 while m ≤ l do
  begin
    v1 := v2 := 0;  m2 := m + m;
    if k/m2 ≠ k ÷ m2 then
    begin k := k − m;  v1 := 1 end;
    if k2/m2 ≠ k2 ÷ m2 then
    begin k2 := k2 − m;  v2 := 1 end;
    if v1 ≠ v2 then n := n + m;
    m := m + m
  end;
  seqwal := binwal(n, x)
end
```

ALGORITHM 391
UNITARY SYMMETRIC POLYNOMIALS [Z]

JOHN MCKAY (Recd. 9 Mar. 1970)
Department of Mathematics, California Institute of Technology, Pasadena, CA 91109

KEY WORDS AND PHRASES: symmetric polynomials, unitary symmetric polynomials
CR CATEGORIES: 5.11, 5.30, 5.5

```
procedure unitary (a, x, n);
  array a, x;  integer n;
comment  With $x_i$ in $x[i]$, $i = 1, 2, \cdots, n$, on entry, the unitary
  symmetric polynomials $a_r = \sum x_{i_1}x_{i_2} \cdots x_{i_r}$ will be found in
  $a[r]$, $r = 1, 2, \cdots, n$ on exit.
```

It is suggested that this algorithm replace Algorithm 156 which is an $O(2^n)$ procedure for computing $\sum_{r=1}^{n} (-1)^{r-1}a_r$.

It is optimal in storage and requires $n(n-1)$ additions and $\frac{1}{2}n(n-1)$ multiplications. It has uses in the theory of symmetric functions since the unitary symmetric polynomials form a basis for the symmetric polynomials. These polynomials arise, too, in probability theory. In numerical analysis it may be of interest to compute the coefficients $(-1)^r a_r$ of the monic polynomial with roots $x_1, x_2, \cdots, x_n$ which is best done by altering the two lines

$$a[k] := a[k] + t \times a[k-1];$$
$$a[1] := a[1] + t$$

to

$$a[k] := a[k] - t \times a[k-1];$$
$$a[1] := a[1] - t;$$

```
begin
  integer i, k;  real t;
  for i := 1 step 1 until n do
  begin
    a[i] := 0;  t := x[i];
    for k := i step − 1 until 2 do
      a[k] := a[k] + t × a[k−1];
    a[1] := a[1] + t
  end
end unitary
```

Remark on Algorithm 391 [Z]

Unitary Symmetric Polynomials [John McKay, *Comm. ACM 13* (Aug. 1970, 512]

Günther F. Schrack (Recd. 9 Nov. 1970 and 11 Jan. 1971)
Departments of Electrical Engineering and Computer Science, University of British Columbia, Vancouver 8, B.C., Canada

Key Words and Phrases: symmetric polynomials, elementary symmetric polynomials, unitary symmetric polynomials, polynomial synthesis, reverse Horner scheme, reverse synthetic division, binomial coefficients
CR Categories: 5.11, 5.30, 5.5

To avoid using semicolons in the body of the **comment**, replace:

the two lines ...; **begin**

by

the plus signs to minus throughout; **begin**

Algorithm 391 has been tested on the IBM 360-67 with the OS Algol F compiler running under the Michigan Terminal System. A number of sets of real x_i, $i = 1, 2, \dots, n$ with $n = 1, 2, \dots, 10$ with various positive, negative, and zero elements were used, drawn from a collection of test polynomials. Both versions of the algorithm produced correct results.

Remarks.

1. The modified version produces the coefficients of the monic polynomial

$f(z) = a_0z^n + a_1z^{n-1} + \cdots + a_{n-1}z + a_n$,

i.e. as the leading coefficient $a_0 = 1$ is implied it must be supplied by the calling program. Alternatively, the insertion of

$a[0] := 1;$

immediately preceding the first **for** statement will supply that coefficient. In this case line 3 of the **comment** should be replaced by

$a[r]$, $r = 0, 1, \dots, n$ on exit.

2. *unitary* may be used for the generation of complex elementary symmetric functions or complex polynomials from complex x_i, provided all real parameters in the procedure are declared complex.
3. The number of additions is $\frac{1}{2}n(n+1)$, i.e. for $n > 3$ it is less than the number claimed.
4. Consider a polynomial with real zeros x_i only, and consider a deflation of that polynomial by Horner's scheme (i.e. synthetic division) by the linear factor $(x-x_1)$. Again using Horner's scheme, deflate the quotient by $(x-x_2)$. Repeat this procedure for all zeros and call the resulting table the repeated Horner scheme. *unitary* is in effect the repeated Horner scheme carried out in reverse order. Since the Horner scheme is optimal in the number of operations [1], so is *unitary*.
5. The second version of *unitary* with the modification suggested in Remark 1 may be used to calculate all binomial coefficients $\binom{n}{m}$, $m = 0, 1, \dots, n$ by setting $x_i = 1$, $i = 1, 2, \dots, n$. The algorithm then represents an *in situ* generation of Pascal's triangle with $a_i = \binom{k}{i}$, $k = 1, 2, \dots, n$, $i = 0, 1, \dots, k$. Because all x_i are unity, this can be programmed using additions as the only arithmetic operations. Then the accuracy of the binomial coefficients is limited only by the word length of the computer.

References

1. Pan, V.Ya. Methods of computing values of polynomials. *Russian Math. Surveys 21* (1966), 105–136.

ALGORITHM 392
SYSTEMS OF HYPERBOLIC P.D.E. [D3]

Robert R. Smith and Dennis McCall (Recd. 7 Jan. 1969 and 17 June 1969)

US Naval Electronics Laboratory Center, San Diego, CA 92152

KEY WORDS AND PHRASES: hyperbolic p.d.e., characteristic, extrapolation, second order p.d.e., quasilinear p.d.e.
CR CATEGORIES: 5.17

Description:

CHARAC solves the initial value problem for the quasilinear hyperbolic system of equations

$$\begin{aligned} A_1U_x + A_2U_y + A_3V_x + A_4V_y &= H_1 \\ B_1U_x + B_2U_y + B_3V_x + B_4V_y &= H_2 \end{aligned} \qquad (1)$$

in two independent variables X, Y and two unknown functions $U(X, Y)$, $V(X, Y)$, where $A_1 = A_1(X, Y, U, V), \cdots, H_2 = H_2(X, Y, U, V)$. Specified data X_i, Y_i, U_i, V_i $(i=1, \cdots, M)$ given along a noncharacteristic curve Γ are used to find U and V at characteristic grid points in the entire characteristic cone associated with the initial curve. Values in the opposite characteristic cone can be computed by specifying the initial data points X_i, Y_i, U_i, V_i in the opposite order (X_1, Y_1, U_1, V_1 becomes X_M, Y_M, U_M, V_M, etc.).

If the system (1) is hyperbolic, it can be reduced to a normal form containing directional derivatives along two characteristic directions. The derivation of this normal form is given in Forsythe and Wasow [1, p. 38].

For (1) the normal form is

$$\begin{aligned} \left(\frac{dY}{dX}\right)_i &= \sigma_i, \\ R_i = \left(\frac{\delta U}{\delta X}\right)_i + S_i\left(\frac{\delta V}{\delta X}\right)_i &= T_i, \end{aligned} \qquad i = 1, 2, \qquad (2)$$

where $(\delta/\delta X)_i$ is the directional derivative along the characteristic with slope σ_i. Let $A = A_1B_3 - A_3B_1$, $C = A_2B_4 - A_4B_2$, $B = \frac{1}{2}(A_1B_4 - A_4B_1 - A_3B_2 + A_2B_3)$. Then the coefficients in (2) are given by

$$\sigma_i(X, Y, U, V) = \frac{B - (-1)^i (B^2 - AC)^{\frac{1}{2}}}{A}$$

$$R_i(X, Y, U, V) = A_1(B_1\sigma_i - B_2) - B_1(A_1\sigma_i - A_2),$$

$$S_i(X, Y, U, V) = A_3(B_1\sigma_i - B_2) - B_3(A_1\sigma_i - A_2),$$

$$T_i(X, Y, U, V) = H_1(B_1\sigma_i - B_2) - H_2(A_1\sigma_i - A_2).$$

The system (1) is called hyperbolic if $B^2 - AC > 0$ and if $R_1S_2 - R_2S_1 \neq 0$.

The subroutine *CH VAR* (*XYUV*, *VAR*) computes the values $\sigma_1, \sigma_2, R_1, R_2, S_1, S_2, T_1, T_2$ from A_i, B_i, H_i evaluated at the values X, Y, U, V given in the array *XYUV*. (The subroutine *CH COEF* giving A_i, B_i, H_i must be provided by the user, see *Examples*.) The computed values are returned in the array *VAR* of length 8. If σ_i, R_i, S_i, T_i are known to the user, he may provide his own routine *CH VAR*.

The system (2) is discretized by Massau's method, which is described in Forsythe and Wasow [1]. Given two adjacent points on the initial curve Γ, the nonparallel characteristics through the points intersect at a third point adjacent to the curve Γ. The values X, Y, U, V at the third point are estimated by replacing the differential equations in (2) by simple difference equations. The subroutine *CH STEP* performs this discretization. By repeating this process for each pair of adjacent points on Γ, datum points are computed on a curve Γ' adjacent to Γ and inside the characteristic cone. If the initial curve has characteristic slope somewhere, the curve Γ' will not be adjacent to Γ but will cross it. The routine does not recognize this, but it is obvious from the output. Successively calling *CHARAC* generates a sequence of adjacent curves until the entire characteristic cone is filled in.

Extrapolation to the limit is applied to this discretization as follows: Compute the data on Γ' by using only every fourth initial datum point on Γ. Then use every other initial datum point to estimate the data on an intermediate curve and then on Γ'. Finally use every datum point. Thus three estimates are found with different step sizes h_0, $h_0/2$, and $h_0/4$. One can then extrapolate these estimates to $h = 0$ in an attempt to obtain a better estimate. Numerical results have indicated that extrapolation does indeed significantly improve the estimates. In fact the method with extrapolation has an error of $o(h^3)$ while Massau's method alone has an error of $o(h)$. The theoretical considerations of extrapolation are given by Bulirsch and Stoer [2], and applications to integration and ordinary differential equations are discussed. In general, extrapolation improves calculated results only if the exact solution is sufficiently differentiable. *CHARAC* can thus be expected to be an improvement over Massau's method only when the coefficients of the system (1) and the initial data are sufficiently differentiable. Note that the extrapolation requires $M = 4 \times N + 1$ for some integer N.

C*HARAC* is defined by

SUBROUTINE CHARAC (*DATA*, *M*, *IFAIL*).

In the parameter list of *CHARAC*, *M* is the number of datum points on the initial curve. *DATA* is dimensioned *DATA*(4, *M*) (where $M = 4 \times N + 1$ for some N) and the column *DATA*(*, *J*) contains the four datum values X_j, Y_j, U_j, V_j of the Jth datum point. Upon calling *CHARAC*, the data on an adjacent curve Γ' are computed and restored in *DATA* and $M = 4 \times N + 1$ is replaced by $M - 4 = 4 \times (N-1) + 1$. Hence *CHARAC* can immediately be called again with Γ' the initial curve. Continuing until $M = 1$ will yield the single datum point at the apex of the characteristic cone. (See *TEST CH* used to solve example.) *IFAIL* is a flag which is 0 if the call to *CHARAC* was successful. If *IFAIL* = 1 upon returning, then *CH VAR* detected that $B^2 - AC \leq 0$, so (1) was not hyperbolic. If *IFAIL* = 2 upon returning, then *CH STEP* detected that $\sigma_1 = \sigma_2$ or $R_1S_2 = R_2S_1$ within a relative tolerance of 10^{-6}; this tolerance parameter is represented by *EPS* in *CH STEP*. This indicates that (1) was not hyperbolic or that the characteristics are so close to parallel that the method fails.

The user must provide a routine

SUBROUTINE CH COEF (*COEFF*, *XYUV*)

which computes the coefficients of the system (1) for the values X, Y, U, V given sequentially in the list *XYUV* of length 4. The computed coefficients must be returned in the list *COEFF* of length 8 in the order $A_1, A_2, A_3, A_4, H_1, B_1, B_2, B_3, B_4, H_2$.

Example (I). Unsteady, one-dimensional Isentropic Flow. (See Jeffrey and Taniuti [3, p. 71].) The system of equations for the flow velocity $u(x, t)$ and the density $\rho(x, t)$ in terms of the space

ALGORITHM:

```
      SUBROUTINE TEST CH
      DIMENSION DATA (4,81)
      N = 20
C GENERATE INITIAL DATA.
      M = 4 * N + 1
      FM = 4.0 * FLOAT (N)
      DO 100  I = 1,M
      DATA(1,I) = FLOAT (I-1) / FM
      DATA(2,I) = 0.0
      DATA(3,I) = 0.0
      DATA(4,I) = 2.0 * EXP (DATA(1,I))
  100 CONTINUE
      IFAIL = 0
      WRITE (51, 900)
  900 FORMAT (1H1)
  200 DO 250  I = 1,M
      WRITE (51,910) DATA(1,I),
     *DATA(2,I), DATA(3,I), DATA(4,I)
  910 FORMAT (4H X =,E20.9,5X,4H Y =,
     *E20.9,5X,4H U =,E20.9,
     *5X,4H V =,E20.9)
  250 CONTINUE
      IF (M.LE.1)  GO TO 300
      IF(IFAIL.NE.0)  GO TO 300
      CALL CHARAC (DATA, M, IFAIL)
      WRITE (51, 900)
      GO TO 200
  300 CONTINUE
      WRITE (51, 920) M, IFAIL
  920 FORMAT (X,3HM =,I2,8H IFAIL =,I2)
      RETURN
      END
      SUBROUTINE CH COEF (COEFF, XYUV)
      DIMENSION COEFF(10), XYUV(4)
C COMPUTES COEFFICIENTS A1, A2, A3,
C A4, H1, B1, B2, B3, B4, H2 AND
C STORES THEM SEQUENTIALLY IN COEFF.
      COEFF(1) = 1.0 - XYUV(3)**2
      COEFF(2) = -XYUV(3) * XYUV(4)
      COEFF(3) = -XYUV(3) *XYUV(4)
      COEFF(4) = 1.0 - XYUV(4)**2
      COEFF(5) = -4.0 * XYUV(3) *
     *EXP (XYUV(1))**2
      COEFF(6) = 0.0
      COEFF(7) = 1.0
      COEFF(8) = -1.0
      COEFF(9) = 0.0
      COEFF(10) = 0.0
      RETURN
      END


      SUBROUTINE CHARAC (DATA, M, IFAIL)
      DIMENSION DATA(4, M),D0(4), D1(4),
     *D2(4), D3(4), D4(4)
      DIMENSION T1(4), T2(4), T3(4),
     *T4(4), T5(4), T6(4), T7(4), V1(8),
     *V2(8), V3(8), V4(8), V5(8), V6(8),
     *V7(8), V8(8), V9(8), V10(8)
      DIMENSION S1(4), S2(4), S3(4)
      INTEGER FAIL, FLAG
      COMMON/CHFAIL/FAIL, FLAG
C THIS ROUTINE ADVANCES ONE GRID STEP
C THE SOLUTION OF THE SYSTEM
C A1 * U(X) + A2 * U(Y)
C + A3 * V(X) + A4 * V(Y) = H1
C B1 * U(X) + B2 * U(Y)
C + B3 * V(X) + B4 * V(Y) = H2
C WHERE U(X) MEANS PARTIAL DERIV. OF
C U  W.R.T. X, ETC.,
C AND A1 = A1 (X, Y, U, V), ---.
C THE INITIAL DATA IS GIVEN IN THE
C MATRIX  DATA, EACH COLUMN OF FOUR
C ELEMENTS CONTAINING A VALUE
C X, Y, U, V.
C M IS THE NUMBER OF DATA POINTS
C ON THE INITIAL CURVE.
      FAIL = 0
      M = M - 4
      IF (M.LE.0)  RETURN
      DO 145 J = 1,4
      D1(J) = DATA(J,1)
      D2(J) = DATA(J,2)
      D3(J) = DATA(J,3)
  145 D4(J) = DATA(J,4)
      CALL CH VAR (D1, V2)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH VAR (D2, V3)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH VAR (D3, V4)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH VAR (D4, V5)
            IF (FAIL.NE.0)  GO TO 250
      FLAG = 0
      DY = D2(2) - D1(2)
      DX = D2(1) - D1(1)
      IF (DY)  148, 149, 150
  148 DY = -DY
      DX = -DX
               GO TO 150
  149 DY = 1.0
      DX = 1.0E30
  150 DX1 = DY / V2(1)
      DX2 = DY / V2(2)
      IF ((DX1.LT.DX).AND.(DX2.GE.DX))
     *GO TO 170
      IF ((DX2.LT.DX).AND.(DX1.GE.DX))
     *GO TO 175
      IF (DX2.GE.DX1)  GO TO 175
  170 FLAG = 1
  175 CONTINUE
      CALL CH STEP (D1, V2,
     *D3, V4, T1)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH VAR (T1, V6)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH STEP (D2, V3,
     *D4, V5, T6)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH VAR (T6, V10)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH STEP (D3, V4,
     *D4, V5, T2)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH VAR (T2, V7)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH STEP (D2, V3,
     *D3, V4, T4)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH VAR (T4, V9)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH STEP (T4, V9,
     *T2, V7, T3)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH VAR (T3, V8)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH STEP (D1, V2,
     *D2, V3, T5)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH VAR (T5, V1)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH STEP (T5, V1,
     *T4, V9, T4)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH VAR (T4, V1)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH STEP (T4, V1,
     *T3, V8, T4)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH VAR (T4, V9)
            IF (FAIL.NE.0)  GO TO 250
      DO 100 I = 1,M
      DO 200  J = 1,8
          V1(J) = V2(J)
          V2(J) = V3(J)
          V3(J) = V4(J)
          V4(J) = V5(J)
  200 CONTINUE
      DO 201 J = 1,4
      D0(J) = D1(J)
      D1(J) = D2(J)
      D2(J) = D3(J)
      D3(J) = D4(J)
  201 D4(J) = DATA(J,I+4)
      CALL CH VAR (D4, V5)
            IF (FAIL.NE.0)  GO TO 250

      CALL CH STEP (D0, V1,
     *D4, V5, S1)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH STEP (D2, V3,
     *D4, V5, T5)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH VAR (T5, V1)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH STEP (T1, V6,
     *T5, V1, S2)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH STEP (D3, V4,
     *D4, V5, T1)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH VAR (T1, V6)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH STEP (T2, V7,
     *T1, V6, T7)
            IF (FAIL.NE.0)  GO TO 250
      DO 210  J = 1,4
          T2(J) = T1(J)
          V7(J) = V6(J)
        V7(J+4) = V6(J+4)
          T1(J) = T6(J)
          V6(J) = V10(J)
        V6(J+4) = V10(J+4)
          T6(J) = T5(J)
         V10(J) = V1(J)
       V10(J+4) = V1(J+4)
  210 CONTINUE
      CALL CH VAR (T7, V1)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH STEP (T3, V8,
     *T7, V1, T5)
            IF (FAIL.NE.0)  GO TO 250
      DO 220  J = 1,4
          T3(J) = T7(J)
          V8(J) = V1(J)
        V8(J+4) = V1(J+4)
  220 CONTINUE
      CALL CH VAR (T5, V1)
            IF (FAIL.NE.0)  GO TO 250
      CALL CH STEP (T4, V9,
     *T5, V1, S3)
            IF (FAIL.NE.0)  GO TO 250
      DO 230  J = 1,4
          T4(J) = T5(J)
          V9(J) = V1(J)
        V9(J+4) = V1(J+4)
  230 CONTINUE
C EXTRAPOLATE'THE THREE
C SUCCESSIVE APPROXIMATIONS
      DO 300  J = 1,4
  300 DATA (J,I) = 0.3333333333 *
     *(8.0 * S3(J) - 6.0 * S2(J) + S1(J))
  100 CONTINUE
      IFAIL = 0
      RETURN
C ERROR EXIT.
  250 IFAIL = FAIL
      RETURN
      END
      SUBROUTINE CH VAR (XYUV, VAR)
      DIMENSION XYUV(4), VAR(8), T(10)
      INTEGER FAIL, FLAG
      COMMON/CHFAIL/FAIL, FLAG
C COMPUTES THE VALUES SIGMA1,SIGMA2,
C R1, R2, S1, S2, T1, T2
C (STORED IN THE LIST VAR)
C FROM THE COEFFICIENT FUNCTIONS
C AND THE VALUES  X, Y, U, V
C (IN THE LIST XYUV).
      CALL CH COEF (T, XYUV)
C COEFFICIENTS OF SYSTEM ARE STORED
C IN THE LIST (T).
      A = T(1) * T(8) - T(3) * T(6)
      B = 0.5 * (T(1) * T(9) - T(4) *
     *T(6) - T(3) * T(7) + T(2) * T(8) )
      C = T(2) * T(9) - T(4) * T(7)
      IF (A.NE.0.0)  GO TO 150
      IF (B.EQ.0.0)  GO TO 500
      VAR(1) = 1.0E15
      VAR(2) = 0.5 * C / B
      IF (B.GT.0.0)  GO TO 400
      VAR(1) = VAR(2)
      VAR(2) = 1.0E15
               GO TO 400
  150 D = B * B - A * C
      IF (D.LE.0.0)  GO TO 500
      D = SQRT (D)
      IF (B.LT.0.0)  GO TO 300
      VAR(1) = (B + D) / A
      VAR(2) = C / (A * VAR(1) )
               GO TO 400
  300 VAR(2) = (B - D) / A
      VAR(1) = C / (A * VAR(2) )
  400 DO 100  I = 1,2
      T(4) = T(1) * VAR(I) - T(2)
      T(9) = T(6) * VAR(I) - T(7)
      VAR(I+2) = T(1)*T(9) - T(6)*T(4)
      VAR(I+4) = T(3)*T(9) - T(8)*T(4)
      VAR(I+6) = T(5)*T(9) - T(10)*T(4)
  100 CONTINUE
      RETURN
C ERROR EXIT.
  500 FAIL =  1
      RETURN
      END
      SUBROUTINE CH STEP
     *(DI, V1, D2, V2, D3)
      DIMENSION D1(4), D2(4), D3(4),
     *V1(8), V2(8)
      INTEGER FAIL, FLAG
      COMMON/CHFAIL/FAIL, FLAG
C THIS ROUTINE COMPUTES THE VALUES
C OF X3, Y3, U3, V3 AT THE POINT
C DETERMINED BY THE INTERSECTION OF
C THE CHARACTERISTICS THROUGH
C X1,Y1 AND X2,Y2.
      EPS = 1.0E-5
      IF (FLAG.NE.0)  GO TO 150
  100 SIG = V1(2)
       R1 = V1(4)
       S1 = V1(6)
       T1 = V1(8)
               GO TO 180
  150 SIG = V1(1)
       R1 = V1(3)
       S1 = V1(5)
       T1 = V1(7)
  180 CONTINUE
      IF (FLAG.NE.0)  GO TO 250
  200 RHO = V2(1)
       R2 = V2(3)
       S2 = V2(5)
       T2 = V2(7)
               GO TO 280
  250 RHO = V2(2)
       R2 = V2(4)
       S2 = V2(6)
       T2 = V2(8)
  280 CONTINUE
C COMPUTE X3,Y3,U3,V3
      DEM1 = SIG - RHO
      IF (ABS(DEM1).LT.EPS*ABS(SIG))
     *         GO TO 900
      AA = D1(2) - SIG * D1(1)
      BB = D2(2) - RHO * D2(1)
      D32 = (SIG * BB - RHO * AA) / DEM1
      D31 = (BB - AA) / DEM1
      TA = T1 * (D31 - D1(1) )
     * + R1 * D1(3) + S1 * D1(4)
      TB = T2 * (D31 - D2(1) )
     * + R2 * D2(3) + S2 * D2(4)
      TC = R1 * S2
      TD = R2 * S1
      DEM2 = TC - TD
      TC = AMAX1 (ABS(TC), ABS(TD) )
      IF (ABS(DEM2).LE.EPS*TC)
     *         GO TO 900
      D3(3) = (TA*S2 - TB*S1) / DEM2
      D3(4) = (TB*R1 - TA*R2) / DEM2
      D3(2) = D32
      D3(1) = D31
      RETURN
  900 FAIL =  2
      RETURN
      END
```

variable x and time t are

$$\begin{aligned} \rho u_x + u\rho_x + \rho_t &= 0 \\ \rho u u_x + \rho u_t + a^2\rho_x &= 0. \end{aligned} \tag{3}$$

Assume the sound speed $a = 1$. Let the initial data given along the curve $t = 0, 0 \le x \le 1$ be $u(x, 0) = 0$ and $\rho(x, 0) = 1 + cx$ for some constant c.

Setting $t = y$, $u = U$, and $\rho = V$, (3) has the form of (1) with $A_1 = V$, $A_2 = 0$, $A_3 = U$, $A_4 = 1$, $H_1 = 0$, $B_1 = UV$, $B_2 = V$, $B_3 = 1$, $B_4 = 0$, $H_2 = 0$.

For $c = 1$ the problem is well conditioned. Solving this problem on a 10-digit machine using the 21 initial datum points $X_j = (j-1)/20$; $Y_j = 0$; $U_j = 0$; $V_j = 1 + X_j$; $j = 1, \cdots, 21$, the following values were computed for the apex of the characteristic cone (by calling *CHARAC* 5 times):

$$X = .4107503; \quad Y = .5099940;$$
$$U = -.3465748; \quad V = 1.4142185.$$

The correct values for the apex are

$$X = .4107581; \quad Y = .5099899;$$
$$U = -.3465736; \quad V = 1.4142136.$$

The maximum relative error is 1.9×10^{-5}. Using 41 initial datum points and calling *CHARAC* 10 times, the computed values for the apex were

$$X = .4107572; \quad Y = .5099904;$$
$$U = -.3465737; \quad V = 1.4142142.$$

The maximum relative error is 2.2×10^{-6}. Thus doubling the number of points decreases the error by a factor of about 8, as would be expected for a third order method.

The above problem was also solved for $c = 10$ using 21 initial datum points $X_j = (j-1)/20$; $Y_j = 0$; $U_j = 0$; $V_j = 1 + 10X_j$; $j = 1, \cdots, 21$. The computed values at the apex were

$$X = .0905; \quad Y = .6190;$$
$$U = -1.2028; \quad V = 3.3100;$$

while the correct values are

$$X = .0936; \quad Y = .6176;$$
$$U = -1.1990; \quad V = 3.3165.$$

Using 41 initial datum points the computed values are

$$X = .0930; \quad Y = .6178;$$
$$U = -1.1996; \quad V = 3.3158.$$

Doubling the number of points decreases the error by a factor of only 5. The high order of the truncation error is partially obscured by the rounding error, which is larger for $c = 10$ than for $c = 1$.

Example (II). Steady Two-dimensional Supersonic Flow. (See Jeffrey and Taniuti [3, p. 76].) The single second-order equation

$$(c^2-\varphi_x^2)\varphi_{xx} - 2\varphi_x\varphi_y\varphi_{xy} + (c^2-\varphi_y^2)\varphi_{yy} = H \tag{4}$$

is hyperbolic if $\varphi_x^2 + \varphi_y^2 > c^2$. Set $H = -4\varphi_x \, exp \, (2x)$, so that $\varphi(X, Y) = 2 \, exp \, (X) \, sin \, (Y)$ is a solution of (4). Then (4) is hyperbolic for $c = 1$ if $X > ln \, (0.5)$.

Letting $U = \varphi_x$, $V = \varphi_y$, (4) becomes

$$\begin{aligned} (1-U^2)U_x - UV(U_y+V_x) + (1-V^2)V_y &= -4U \, exp \, (2X) \\ U_y - V_x &= 0. \end{aligned} \tag{5}$$

Let the initial data given along $Y = 0, 0 \le X \le 1$ be $U(X, 0) = 0$, $V(X, 0) = 2 \, exp \, (X)$. Then throughout the cone the exact solution is $U(X, Y) = 2 \, exp \, (X) \, sin \, (Y)$, $V(X, Y) = 2 \, exp \, (X) \, cos \, (Y)$.

This problem was solved using 81 datum points on the initial curve $X_j = (j-1)/80$; $Y_j = 0$; $U_j = 0$; $V_j = 2 \, exp \, (U_j)$; $j = 1, \cdots, 81$. By calling *CHARAC* 20 times, the following values were computed for the apex of the characteristic cone:

$$X = 1.6130; \quad Y = 1.1576;$$
$$U = 9.1980; \quad V = 4.0184.$$

The correct values for the apex are

$$X = 1.6144; \quad Y = 1.1580;$$
$$U = 9.2057; \quad V = 4.0312.$$

Using 81 datum points on the initial curve but not applying extrapolation, the computed values were

$$X = 1.5889; \quad Y = 1.1418;$$
$$U = 9.0441; \quad V = 3.7319.$$

Thus extrapolation significantly improved the results.

By plotting the characteristic grid points in the X-Y plane, one sees that the characteristics become more parallel near the apex. Thus the above problem is ill conditioned. If the initial curve is chosen as $Y = 0$, $1 \le X \le 2$, the problem becomes so ill conditioned that the method fails for 81 datum points on the initial curve.

Example of use. In the following listing *TEST CH* sets up the initial data and makes the necessary calls to *CHARAC* to solve *Example* (II) for 81 initial datum points. *CH COEF* computes the coefficients $A_1 = 1 - U^2$, $A_2 = -UV$, $A_3 = - UV$, $A_4 = 1 - V^2$, $H_1 = -4U \, exp \, (2X)$, $B_1 = 0$, $B_2 = 1$, $B_3 = -1$, $B_4 = 0$, $H_2 = 0$ as determined from (5).

REFERENCES:

1. Forsythe, G. E., and W. R. Wasow. *Finite-Difference Methods for Partial Differential Equations.* Wiley, New York, 1960, p. 64.
2. Bulirsch, R., and J. Stoer. Fehlerabschätzungen und Extrapolation mit Rationalen Funktionen bei Verfahren vom Richardson-Typus. *Num. Math. 6* (1964), 413–427.
3. Jeffrey, A., and T. Taniuti. *Non-Linear Wave Propagation.* Academic Press, New York, 1964.

Remark on Algorithm 392 [D3]

Systems of Hyperbolic P.D.E.
[Robert R. Smith and Dennis McCall, *Comm. ACM 13* (Sept. 1970), 567–570]

Michael J. Frisch [Recd. 27 Jan. 1971]
University Computer Center, University of Minnesota,
Minneapolis, MN 55455

Key Words and Phrases: ANSI Fortran standard
CR Categories: 4.0, 4.22

The following items were found during compilation of the algorithms written in Fortran published to date in Communications. The MNF compiler written at the University of Minnesota for CDC 6000 Series machines by Lawrence A. Liddiard and E. James Mundstock was used to check the validity of the algorithms.

Algorithm 392 does not conform to the standard in subroutine *CHARAC* in which at six statements before the statement numbered 145, the variable dimension M of the array *DATA* is redefined during execution contrary to Section 7.2.1.1.2.

ALGORITHM 393
SPECIAL SERIES SUMMATION WITH ARBITRARY PRECISION [C6]

S. Kamal Abdali* (Recd. 23 June 1969 and 9 Mar. 1970)
University of Wisconsin, Department of Computer Sciences, Madison, WI 35706

* This work was done while the author was at the University of Montreal, Montreal, Canada.

KEY WORDS AND PHRASES: function evaluation, series summation, approximation
CR CATEGORIES: 5.12, 5.13

```
procedure series (places, terms, base, digit, sgn, numerator, de-
  nominator, num0, denom0); value places, terms, base; integer
  places, terms, base, sgn, num0, denom0; integer array digit;
  integer procedure numerator, denominator;
comment Programs for very precise summation of series are
  conventionally written in machine language and employ multi-
  precision routines to perform arithmetic on especially defined
  multiword registers. The present algorithm requires only integer
  arithmetic and can be implemented in any algebraic language.
  It is applicable to series in which the ratios of successive terms
  can be expressed as quotients of given integers or integer func-
  tions of term positions.
    The sum of a given series is computed to a given number of
  places, places, in a specified base for representation, base. The
  number of terms needed, terms, should be calculated outside the
  procedure. Procedures numerator and denominator are to be
  obtained from the fraction ith term/(i-1)-th term, expressed as
  a ratio of two integer functions of i. (That fraction should prefer-
  ably be reduced to its lowest terms.) num0 and denom0 are the
  integer numerator and denominator of the 0th term. The out-
  puts of the procedure are the sign of the result, sgn, the integer
  part, digit [0], and the digits of the fractional part, digit [1], ··· ,
  digit [places].
    For example, one way to compute sin 0.6 = .6 - .6^3/3! + .6^5/5!
  - ··· correct to 1000 decimal places is to call series with the
  parameter values: terms = 226, num0 = 3, denom0 = 5, (and
  since ith term/(i-1)th term = -.6^2/2i(2i+1)) numerator(i) =
  -9 and denominator(i) = 50i(2i+1). By taking base = 100000
  and places = 200, five decimal digits of the result will be obtained
  per word of the array digit.
    The use of a large base (and, consequently, smaller places)
  results in faster computation, as the number of operations is
  proportional to (places×terms) for large values of terms and
  places. However, the intermediate products (base×num[i]×
  coef[i]) (and coef[i] can almost equal denom[i]) should not ex-
  ceed the largest number representable by an integer variable.
  Also within this limit should be the product of base and the in-
  teger portion of the result;
begin
  integer i, j, k, l; integer array num[-1:terms], denom,
    coef[0:terms];
  comment Express the series by the expression
```

$$\frac{n_0}{d_0}\left(c_0+\frac{n_1}{d_1}\left(c_1+\cdots+\frac{n_t}{d_t}(c_t)\cdots\right)\right) \qquad (1)$$

```
    where n_i and d_i are positive and c_i are ±1. (For short, n, d, c
    and t in (1) stand for num, denom, coef and terms, respectively);
  num[-1] := 1; num[0] := abs(num0); denom[0] := abs-
  (denom0); coef[0] := sign(num0) × sign(denom0);
  for j := 1 step 1 until terms do
  begin
    k := numerator(j); l := denominator(j); num[j] := abs(k);
    denom[j] := abs(l); coef[j] := coef[j-1] × sign(k) × sign(l)
  end;
  comment Calculate digits one at a step by extracting the in-
    teger part of base × (1) and restoring the fractional part in
    form (1);
  for i := 1 step 1 until places do
  begin
    l := 0;
    for j := terms step -1 until 0 do
    begin
      k := num[j] × (coef[j]×base+l); l := k ÷ denom[j];
      coef[j] := k - l × denom[j]; num[j] := num[j-1]
    end j;
    digit[i] := l
  end i;
  comment Some digits may be negative or larger than base in
    absolute value. Process the array digit to obtain true base
    representation;
  l := 0;
  for i := places step -1 until 1 do
  begin
    k := digit[i] + l; l := k ÷ base; digit[i] := k - base × l;
    if digit[i] < 0 then
    begin digit[i] := digit[i] + base; l := l - 1 end
  end;
  digit[0] := l; sgn := sign(l);
  if l < 0 then
  begin
    digit[0] := -l - 1; digit[places] := digit[places] - 1;
    for i := 1 step 1 until places do digit[i] := base - 1 - digit[i]
  end
end series
```

Remark on Algorithm 393

Special Series Summation with Arbitrary Precision [C6]
[S. Kamal Abdali, *Comm. ACM 13* (Sept. 1970), 570]

Arthur H.J. Sale
Basser Department of Computer Science, University of Sydney, NSW 2006, Australia

Key Words and Phrases: function evaluation, series summation, approximation
CR Categories: 5.12, 5.13

Algorithm 393 has been tested on a number of different series, including those for e^x and *sin*(*x*) and the harmonic series, and in all cases it gave the expected results. Some remarks should however be made concerning this algorithm.

This algorithm is a slight generalization of a method first described in the reference given here in which it was used to produce an accurate approximation to the transcendental number *e*. As noted in that reference the digits computed when expanding the *e*-series are correct as produced, and need no subsequent processing. This technique is very well suited to this application.

As the author correctly states some types of series will allow negative digits to be computed, or digits which exceed the value of the chosen base. The series for *sin*(x) can give rise to the first case, for it contains negative as well as positive terms; the second case can arise if the remnant series is not always fractional (and will always occur if the value of the original series has an integer part). To illustrate this the first few terms of the harmonic series may be summed:

$\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} = 1.45000...$

which using a base of 10 produces the digits 14, 4, 10. This means that the answer returned by the algorithm is not necessarily correct to the number of places requested either in a truncated or rounded sense. This is particularly important if it is possible that the $(i + 1)$-th term is greater in magnitude than the ith term, for then the final remnant series (which is of course the truncation error) may have a large value.

The author too has not sufficiently emphasized the problem of integer overflow. Intermediate results produced can be quite large, and for example the evaluation of the above mentioned few terms of the harmonic series generated an intermediate value of 100 (with a base of 10). Reversing the order of the terms gave a worse result: a value of 378 was generated, which even exceeds the bound given by the author of the algorithm. The implications of this are that considerable care must be taken to choose a base that is not too large, and that the technique may be restricted in application by the size of common computer words. For example to evaluate *sin*(0.999) (given to three decimal places), using 100 terms and a base of 10, would appear to require an integer range of about 10^{17} by the author's bound, which is certainly beyond the capacity of a 32 bit machine.

To summarize, this technique is fairly specialized; it is not suitable for summing series whose values have large integer parts, and care must be taken in applying it to an arbitrary series.

References

1. Sale, A.H.J. The calculation of *e* to many significant digits. *Comput. J. 11* (Aug. 1968), 229–230.

ALGORITHM 394
DECISION TABLE TRANSLATION [H]

Robert B. Dial (Recd. 31 Oct 1969 and 8 May 1970)
Alan M. Voorhees and Associates, Inc., McLean, Virginia, and Department of Civil Engineering, University of Washington, Seattle, WA. 98105

KEY WORDS AND PHRASES: Decision table, decision table translation
CR CATEGORIES: 4.19

integer procedure *decitable*(*t, m, n, test, yes, no*); **value** *m, n*;
comment This algorithm converts the limited-entry decision table stored in the *m* by *n* matrix *t* into a machine processable test-and-branch code matrix returned in the column vectors *test, yes,* and *no.* The input decision table's format and terminology generally agree with that introduced in Pollack [1]. The rows of *t* represent the decision table's conditions, its columns, its rules. Each of its entries represents a Y (truth), or an N (falsity), or a — (indifference). The output code matrix tabulates a decision tree, which can be traced to ascertain efficiently which rule any given transaction satisfies. Intended for use by a computer, this code matrix can readily drive an interpretive routine, or it can easily be transformed into code in some specified language. An example of a test-and-branch code matrix appears below in Figure 2. Figure 1 is the input decision table which generates it, and Figure 3 is the decision tree it represents.

		Rule			
		R1	R2	R3	R4
Condition	C1	Y	N	—	—
	C2	N	—	Y	N
	C3	Y	—	—	N
	C4	—	N	Y	Y

Fig. 1. Decision Table

	Test-and-Branch		
i	*test* [*i*]	*yes*[*i*]	*no*[*i*[
1	4	2	5
2	2	−3	3
3	3	4	−4
4	1	−1	0
5	1	6	−2
6	3	7	0
7	2	0	−1

Fig. 2. Code Matrix

Each row of the code matrix in Figure 2 corresponds to a nonterminal, decision node in the tree in Figure 3. These row numbers have been posted alongside the nodes in Figure 3. The root node corresponds to row 1, and the first condition to be tested is C4, indicated by the 4 in *test*[1]. In general *test*[*i*] contains the condition (decision table row) number to be tested at node *i*. *yes*[*i*] and *no*[*i*] specify subsequent alternative actions selected on the basis of the result of testing condition *test*[*i*]. *yes*[*i*] is an integer telling what to do if condition *test*[*i*] is true. Its interpretation depends on its relationship to zero:

1. If *yes*[*i*] is positive, then the next thing to do is perform the test-and-branch given in row *yes*[*i*] of the code matrix. This is equivalent to moving down one ply in the decision tree via the "true arc" to enter another decision node.
2. If *yes*[*i*] is negative, no more testing is necessary; Rule *abs*(*yes*[*i*]) has been satisfied. This is equivalent to encountering terminal rule node in the decision tree. In typical applications, a procedure would be invoked to perform the actions corresponding to Rule *abs*(*yes*[*i*]).
3. If *yes*[*i*] is zero, then testing is complete; no rule can be satisfied. In this case a terminal node is reached which indicates that none of the decision table's rules is satisfied. The action(s) corresponding to the "Else-rule" would be invoked here.

The interpretation of *no*[*i*] is identical to that of *yes*[*i*], applying to the case where the result of testing Condition *test*[*i*] is false.

The algorithm's technique is due to Pollack, who explains it in fine tutorial manner [2]. Another excellent discussion is given by Press, who provides additional insights and refinements [3]. In brief, the procedure selects a row of the decision table and bifurcates the table into two decision subtables from which the selected row is excluded. One subtable contains only rules (columns) for which the selected row's condition may be true (Y or —). The other subtable contains only rules for which the condition may be false (N or —). This splitting is recursively applied to each subtable (which is at least one row smaller than the parent table) until a "degenerate" subtable results. If the degenerate subtable has no rows or is composed of only dashes, then a rule is satisfied and noted. If the degenerate subtable has no columns, then the Else-rule is in effect.

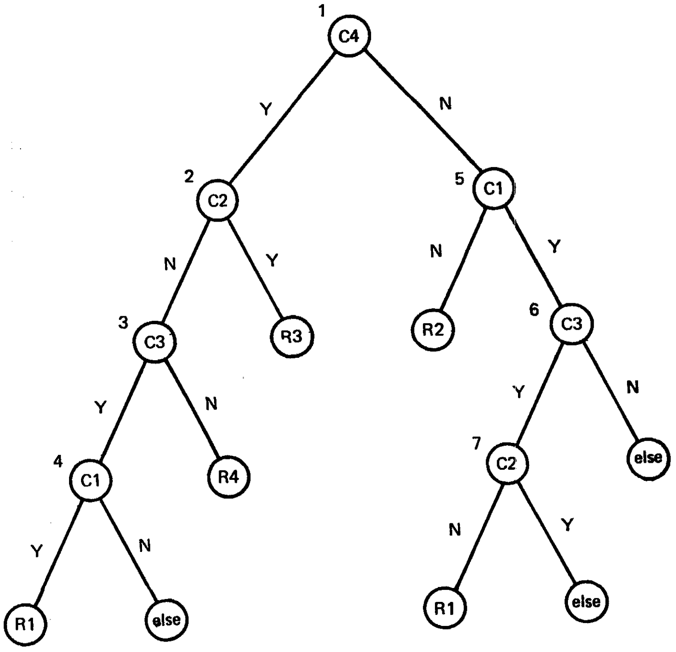


Fig. 3

In the Algol below, the author attempts to provide code which would allow a flexible and practical implementation. Computational efficiency is traded off for storage conservation and ease of modification. No local arrays are declared in recursive routines. The decision table's manipulation and subtable "creation" are effected by sorting the global row and column index arrays, *row* and *col*. The algorithm never modifies or reproduces any part of the original copy of the input decision table.

To facilitate user control of the desired attributes of the output decision tree, the routine which selects the condition row on which to split the table is made a separate procedure. To impose his own criterion for row selection, the user can easily modify or substitute code in the procedure *select*. His procedure generally depends on the kind of code matrix he wants. For example, if storage were a problem, he would want the shortest code matrix, i.e. a tree with fewest decision nodes. On the other hand, if execution time of the code matrix were of prime importance, he would want to minimize the expected number of executed decision nodes. In general, each of these criteria does not yield the same select procedure. The procedure below uses a criterion given in [2]. Others may be found in [2 and 3].

The syntax of Algol 60 does not allow strings such as "Y", "N", and "—" to be elements of an array such as the decision table matrix *t*. Thus in the code below, the local variables N, D, and Y contain the integers −1, 0 and 1 to represent respectively the characters "N", "—", and "Y". Accordingly, the input decision table must also follow these conventions, or the user must appropriately modify the three assignment statements which establish the value of these local variables.

The author thanks the referee and the editor for their valuable observations.

REFERENCES:

1. POLLACK, S. L. How to build and analyze decision tables. P-2829, Nov. 1963, RAND Corp., Santa Monica, Calif.
2. POLLACK, S. L. Conversion of limited entry decision tables to computer programs. RM-4030-PR, May 1964, RAND Corp., Santa Monica, Calif.
3. PRESS, LAURENCE I. Conversion of decision tables to computer programs. *Comm. ACM 8* (June 1965), 385–390.

```
begin
  own integer array row[1:m], col[1:n];
  own real array cc[1:n]; own integer N, D, Y, line;
  integer i;
  integer procedure select (t, rows, first, last);
  comment This procedure picks a row of the decision (sub)-
    table defined by the row indices row[1], row[2], ··· , row[rows]
    and the column indices col[first], col[first+1], ··· , col[last].
    The criterion is a minimal "dash count", with the difference
    between the number of Y's and the N's to be minimized in
    case of a tie. A short code matrix should result [2];
  value rows, first, last;
  begin
    integer i, j, imin, delta, deltamin; real dash, dmin;
    dmin := (last−first+1) × (2 ↑ rows); imin := 0;
    for j := first step 1 until last do
    begin
      comment Calculate column count;
      cc[col[j]] := 1; for i := 1 step 1 until rows do
        if t[row[i], col[j]] = D then cc[col[j]] := 2 × cc[col[j]]
        else imin := 1
    end;
    if imin ≠ 0 then for i := 1 step 1 until rows do
    begin
      comment Calculate dash count;
      dash := delta := 0; for j := first step 1 until last do
        if t[row[i], col[j]] = D then dash := dash + cc[col[j]]
        else delta :=
          delta + (if t[row[i], col[j]]=Y then 1 else −1);
      if dash < dmin ∨ (dash=dmin∧abs(delta)<deltamin) then
      begin
        comment Row i has the smallest dash count so far;
        imin := i; dmin := dash; deltamin := abs(delta)
      end
    end;
    select := imin
  end select;
  procedure left (t, row, first, last, key, lyp, ldp);
  comment This procedure creates the two subtables described
    above with respect to condition row by rearranging the column
    indices col[first], col[first+1], ··· , col[last] based on the con-
    tents of t[row, col[first]], ··· , t[row, col[last]]. Upon return,
    col[first] up to col[lyp−1] contain all the column indices j such
    that t[row, j] = Y. col[lyp] up to col[ldp−1] return the indices
    j such that t[row, j] = D, and col[ldp] up to col[last] have the
    indices such that t[row, j] = N. Thus the two subtables are
    defined by the indices col[first], ··· , col[ldp] and col[lyp], ··· :
    col[last]. left is executed twice for each external reference,
    First it places all the "Y" columns at the far left. Second it
    calls itself to push all the "—" columns to the right of the last
    "Y" column. The parameter key contains the code for "Y"
    or "—" to indicate which character is being matched;
  value row, first, last, key;
  begin
    integer i, j, temp;
    i := first; j := last;
    for i := i while i ≤ j do if t[row, col[i]] = key then i := i + 1
      else
    begin
      for j := j while t[row, col[j]] ≠ key ∧ i < j do j := j − 1;
      temp := col[i];  col[i] := col[j];  col[j] := temp;  j := j − 1
    end;
    lyp := i;  if key ≠ D then left (t, row, i, last, D, ldp, lyp)
  end left;
  integer procedure split (t, rows, first, last, test, yes, no)
  comment This procedure recursively bifurcates the nonde-
    generate decision subtable defined by the row indices row[1],
    ··· , row[rows] and the column indices col[first], ··· , col[last].
    The global parameter line determines the position of the
    code matrix into which split enters test-and-branch data.
    The procedure "creates" subtables from which the selected
    condition row is deleted by swapping the selected condition
    row index with the last row index, reducing the rows counter
    by 1, and having procedure left rearrange the column indices.
    If the input table has no rows, then split returns zero indicat-
    ing the Else-rule. If the table is entirely dashes or has no
    columns, then split returns the value −col[first], indicating a
    terminal, rule node. Otherwise split places the next condi-
    tion to be tested as a decision node into test[line+1] and
    calls itself for the corresponding subtables;
  value rows, first, last;
  begin
    integer mine, imin, lyp, ldp;
    mine := 0;  if first ≤ last then
    begin
      imin := select (t, rows, first, last);
      if imin = 0 then
      begin
        mine := −col[first];  if first ≠ last then
        begin
          outstring (1, 'Following rules are redundant:');
          for i := first step 1 until last do outinteger (1, col[i])
        end;
      end else
      begin
        mine := line := line + 1; test[mine] := row[imin];
        row[imin] := row[rows];
        left(t, test[mine], first, last, Y, lyp, ldp);
        yes[mine] := split (t, rows−1, first, ldp−1, test, yes, no);
        comment Restore column indices rearranged in
          recursion;
        left(t, test[mine], first, ldp−1, Y, lyp, ldp);
        no[mine] := split(t, rows−1, lyp, last, test, yes, no);
        row[imin] := test[mine]
      end
    end;
    split := mine
  end split;
  for i := 1 step 1 until m do row[i] := i;
  for i := 1 step 1 until n do, col[i] := i;
  N := −1;  D := 0;  Y := 1;  line := 0;
  i := split(t, m, l, n, test, yes, no);
  decitable := line;
  comment The value of decitable is the length of code matrix;
end decitable
```

Remark on Algorithm 394 [H]

Decision Table Translation [R.B. Dial, *Comm. ACM 13* (Sept. 1970), 570]

D.R.T. Marshall [Recd. 3 Mar. 1971]
Data Processing Department, University of Waterloo
Waterloo, Ontario, Canada

Key Words and Phrases: decision table, decision table translation
CR Categories: 4.19

The first comment of procedure *split* has the words "columns" and "row" transposed in sentences four/five. It should read "If the input tables has no columns, then *split* returns zero, ... If the table is entirely dashes or has no rows, then *split*, ...

The statement in the main procedure invoking the procedure *split* uses a variable "*l*", which is not defined.

This variable should be initialized to establish the "first" column in the array to be processed. This would, of course, normally be set to one.

The writer has programmed and executed the algorithm successfully in PL/I with the above noted changes.

ALGORITHM 395
STUDENT'S t-DISTRIBUTION [S14]

G. W. Hill (Recd. 17 Nov. 1969 and 23 Mar. 1970)
C.S.I.R.O., Division of Mathematical Statistics, Glen Osmond, South Australia

KEY WORDS AND PHRASES: Student's t-statistic, distribution function, approximation, asymptotic expansion
CR CATEGORIES: 5.12, 5.5

real procedure *student* (t, n, *normal*, *error*); **value** t, n; **real** t, n; **real procedure** *normal*, *error*;

comment *student* evaluates the two-tail probability $P(t \mid n)$ that t is exceeded in magnitude for Student's [1] t-distribution with n degrees of freedom. The procedure provides results accurate to 11 decimal places and 8 significant digits for integer values of n, with approximate continuation of the function through noninteger values of n (over 6 decimal places for $n > 4.3$).

The procedure *normal* (χ) returns the area under the standard normal frequency curve to the left of χ, so that a negative argument yields the lower-tail area. The user-supplied procedure, *error*(n), should produce a diagnostic warning and may **go to** a label, terminate, or return a distinctive value (zero or -1.0) as a signal of error to the calling program.

Student's series expansion of the probability integral is supplemented by a faster asymptotic approximation for large values of n and by a more precise "tail" series expansion for large values of t.

The value of χ, defined as the normal deviate at the same probability level as t, may be approximated by an asymptotic normalizing expansion of Cornish-Fisher type [2].

$$\chi = z + (z^3+3z)/b - (4z^7+33z^5+240z^3+855z)/10b^2$$
$$+(64z^{11}+788z^9+9801z^7+89775z^5+543375z^3+1788885z)/210b^3 - \cdots$$

where $z = (a \times \ln(1+t^2/n))^{\frac{1}{2}}$, $a = n - \frac{1}{2}$ and $b = 48a^2$ [3].
This is well approximated by the first three terms with the third term's divisor replaced by

$$10b(b+0.8z^4+100).$$

The *student* probability is double the normal single-tail area, corresponding to the deviate χ.

The maximum error in the probability result for all values of t is displayed as a function of n in Figure 1, for this approximation, for the first few terms of the asymptotic expansion and for Fisher's [4] fifth-order approximation used in Algorithm 321 [5] for $n \geq 30$.

For small n and moderate t the result is calculated as $P(t \mid n) = 1 - A(t \mid n)$ using Student's cosine series for $A(t \mid n)$, rearranging formulas 26.7.3 and 26.7.4 of the NBS Handbook [6] in nested form

$$A(t|n \text{ odd}) = \frac{2}{\pi}\left[arctan(y) + \frac{y}{b}\left\{1 + \frac{2}{3b}\left\{\cdots \frac{(n-5)}{(n-4)b} \cdot \left\{1 + \frac{(n-3)}{(n-2)b}\right\} \cdots\right\}\right\}\right]$$

$$A(t|n \text{ even}) = \frac{y}{\sqrt{(b)}}\left\{1 + \frac{1}{2b}\left\{\cdots \frac{(n-5)}{(n-4)b}\left\{1 + \frac{(n-3)}{(n-2)b}\right\} \cdots\right\}\right\},$$

where $y = \sqrt{(t^2/n)}$ and $b = 1 + t^2/n$. In the nested form, terms are treated in reverse order to the summation in Algorithm 321 and Algorithm 344 [7], reducing the number of operations required and reducing build up of roundoff error. Explicit decrementing of the "loop" parameter ensures that its final value remains defined on exit from the loop for use in an odd/even test.

Execution times for Fortran versions run on a CDC 3200 with programmed floating point are displayed in Figure 2, which indicates that nesting decreases the time for the cosine series method by about 30 percent and that it is appropriate to change over to the asymptotic method (using Algorithm 209 [8] for *normal*) when $n \geq 20$. Although this approximation would be accurate to more than 11 decimal places, the use of Algorithm 209 limits accuracy to about 9 decimals. This accuracy may be sufficient for many applications, in which case *student* may be abbreviated by deleting lines 15 and 27 through 35, removing

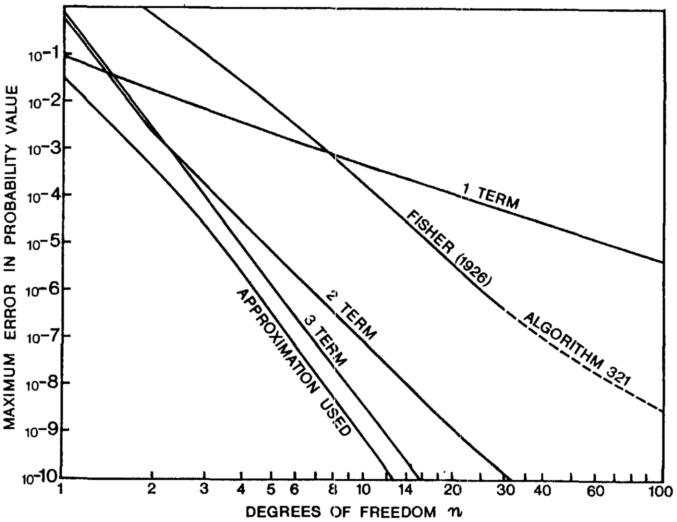


FIG. 1. Maximum error of approximations for "Student's" t-probability: 1, 2, and 3 term expansion, approximation with adjusted divisor, and Fisher's 5th order approximation

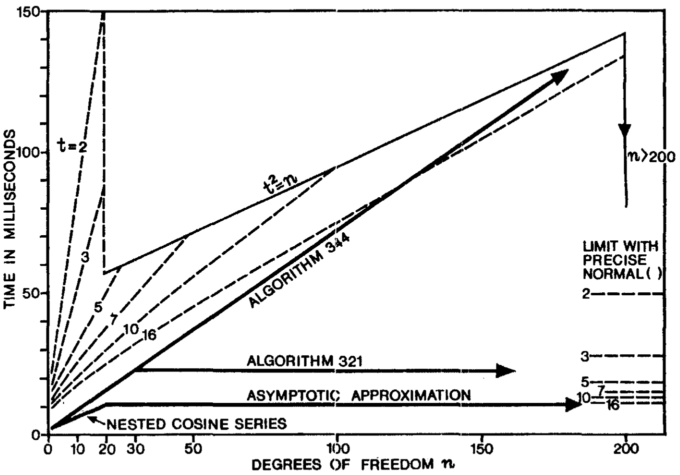


FIG. 2. Execution times (CDC3200 with programmed floating point). Broken lines: "tail" series for selected values of t (upper left); asymptotic method using precise *normal* (right)

the declaration and assignment of z from line 3, replacing line 5 by

$$\textbf{if } n > \mathit{entier}(n) \vee n \geq 20 \textbf{ then}$$

and replacing line 25 by

$$\mathit{student} := \textbf{if } a > 1.0 \textbf{ then } 0.0 \textbf{ else } 1.0 - a$$

The latter avoids spurious negative results due to roundoff error when a is near 1 for large values of t. The storage required for this abbreviated version was a little less than for Algorithm 344 and less than half that for Algorithm 321.

Applications such as production of tables or function inversion to obtain extreme quantiles may require greater precision at extreme probability levels than these methods provide. For the cosine series and the asymptotic approximation using a high precision procedure for *normal*, such as Algorithm 304 [9], the relative error in the result increases in magnitude as the result decreases to extremely small values, as illustrated in Figure 3.

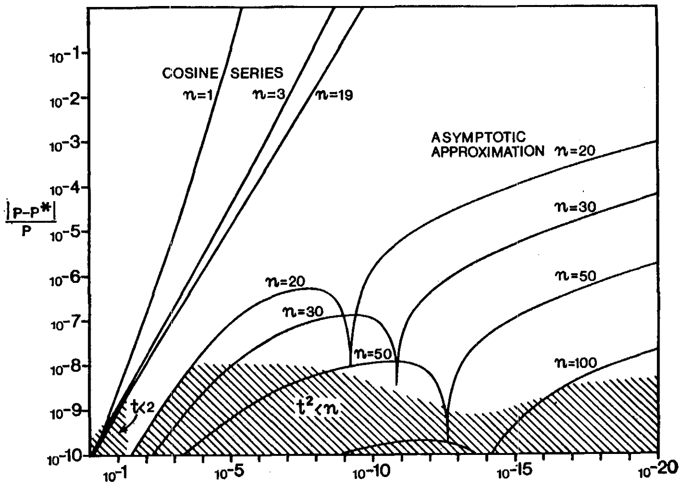


FIG. 3. Relative error, $|P - P^*|/P$, of approximation P^*; shaded region for restricted t values

For small P more precise results are obtained using a series expansion of $P(t \mid n)$ in terms of $w = 1/sqrt(1+t^2/n)$,

$$P(t|n) = C(n) \times w^n \left\{ \frac{1}{n} + \frac{1 \times w^2}{2(n+2)} + \frac{1 \times 3 \times w^4}{2 \times 4(n+4)} + \cdots \right\},$$

where $C(n) = \Gamma((n+1)/2)/(\sqrt{\pi} \times \Gamma(n/2))$. The series is summed till a negligible term occurs and then the factor $C(n) \times w^n$ is applied using the same repeated loop as the cosine series. Except for w near 1 when t is small, the truncation error is small, and accumulation of error in the repeated loop is moderate unless n is very large.

The cosine series method loses precision mainly in the subtraction $1 - A(t \mid n)$ as well as from the *sqrt* procedure and *arctan* when n is odd. In the worst case, $n = 19$, the error is kept below 3 decimals by changing to the tail series if $t > 2$, which ensures 8 significant digits in the result for the 36-bit (about 11 decimal) precision real variables for the processor used. As shown in Figure 3, change over from the asymptotic method to the tail series when $t^2 > n$ maintains about 8 significant digits in the result. For a machine of greater precision the use of more terms in the asymptotic series may be warranted, and the change over criteria would need adjustment to balance speeds and precision between the three methods.

Execution times for the tail series are shown as broken lines in Figure 2 for selected values of t: with bounds $t \geq 2$ for $n < 20$, $t^2 \leq n$ for $n \geq 20$ and with the limit $n < 200$ preventing excessive time for large t beyond a probability level near 10^{-40}. For the asymptotic method, using for *normal* a higher precision procedure based on Algorithm 304, the execution times for different values of the argument approach those shown at the right of Figure 2. Averaged over a range of arguments arising in practice, the provision for higher precision more than doubles the time required. In the case of Smirnov's [10] 6D tables of $S(t \mid n) = 1 - 0.5 \times P(t \mid n)$, retabulation to 10D, using the more precise procedure for *normal*, increased the time from about 7 minutes to 12 minutes, while introducing the tail series method to tabulate $P(t \mid n)$ over the same range to 8 significant digits increased the time further to about 16 minutes. Use of the asymptotic approximation enabled Smirnov's 6D tables of $\psi(t \mid 1000/\xi)$, which is an approximate continuation of $S(t \mid n)$ over noninteger values of $n = 1000/\xi$, to be extended to 10D for $\xi = 0(2)30$ in 5 minutes, and permits continuation to $\xi = 200$ with over 6D accuracy as indicated in Figure 1.

The preparation of diagrams by Murray C. Childs is gratefully acknowledged.

REFERENCES:

1. GOSSET, W. S. (Student). On the probable error of a mean. *Biometrika 6* (1908), 1.
2. HILL, G. W., AND DAVIS, A. W. Generalized asymptotic expansions of Cornish-Fisher type. *Ann. Math. Statist. 39*, 4(1968), 1264.
3. HILL, G. W. Progress results on asymptotic approximations for Student's t. Unpublished manuscript, Oct. 1969.
4. FISHER, R. A. Expansion of "Student's" integral in powers of n^{-1}. *Metron, 5* (1926), 109–112.
5. MORRIS, J. Algorithm 321, t-test. *Comm. ACM 11* (Feb. 1968), 115.
6. ABRAMOWITZ, M., AND STEGUN, I. A. (Eds.) Handbook of Mathematical Functions. Appl. Math. Ser. Vol. 55, Nat. Bur. Stand., US Govt. Printing Off. Washington, D.C., 1965, p. 948.
7. LEVINE, D. A. Algorithm 344, Student's t-distribution. *Comm. ACM 12* (Jan. 1969), 37.
8. IBBETSON, D. Algorithm 209, Gauss. *Comm. ACM 6* (Oct. 1963), 616.
9. HILL, I. D., AND JOYCE, S. A. Algorithm 304, Normal. *Comm. ACM 10* (June 1967), 374.
10. SMIRNOV, N. V. *Tables for the Distribution and Density Functions of t-Distribution.* Pergamon Press, New York, 1961;

```
if n < 1 then student := error(n) else
begin
  real a, b, y, z;  z := 1.0;
  t := t ↑ 2;  y := t/n;  b := 1.0 + y;
  if n > entier(n) ∨ n ≥ 20 ∧ t < n ∨ n > 200 then
  begin
    comment  Asymptotic series for large or noninteger n;
    if y > ₁₀−6 then y := ln(b);
    a := n − 0.5;  b := 48.0 × a ↑ 2;  y := a × y;
    y := (((((−0.4×y−3.3)×y−24.0)×y−85.5)/
      (0.8×y↑2+100.0+b)+y+3.0)/b+1.0)×sqrt(y);
    student := 2.0 × normal(−y);
  end
  else
  if n < 20 ∧ t < 4.0 then
  begin
    comment  Nested summation of "cosine" series;
    a := y := sqrt(y);  if n = 1 then a := 0.0;
loop:
    n := n − 2;  if n > 1 then
    begin a := (n−1)/(b×n) × a + y;  go to loop end;
    a := if n = 0 then a/sqrt(b)
      else (arctan(y)+a/b) × 0.63661977236;
    comment 2/π = 0.636619772367581343075351 ··· ;
    student := z − a
```

```
end
else
begin
  comment "tail" series expansion for large t-values;
  integer j;  a := sqrt(b);  y := a × n;  j := 0;
  for j := j + 2 while a ≠ z do
        begin
          z := a;  y := y × (j−1)/(b×j);  a := a + y/(n+j)
        end;
      n := n + 2;  z := y := 0.0;  a := −a;  go to loop
    end
  end
```

ACM Transactions on Mathematical Software, Vol. 5, No. 2, June 1979, Pages 238-239.

REMARK ON ALGORITHM 395

Student's *t*-distribution [S14]
[G.W. Hill, *Comm. ACM 13,* 10 (Oct. 1970), 617-619]

and

REMARK ON ALGORITHM 396

Student's Quantiles [S14]
[G.W. Hill, *Comm. ACM 13,* 10 (Oct. 1970), 619-620]

Mohamed el Lozy [Recd 9 June 1978]
Department of Nutrition, Harvard School of Public Health, 665 Huntington Ave., Boston, MA 02115

Both of these algorithms incorporate very accurate mathematical methods, but contain a source of loss of precision which is severe for the many processors with precision less than or not sufficiently greater than that claimed for the algorithms.

In Algorithm 395 the use of the asymptotic series involves the evaluation of $\ln(1 + t^2/n)$. For small $y = t^2/n$ and $b = 1 + y$, $\ln(b)$ is of the order of magnitude of y, so that the statement

if $y > 10^{-6}$ **then** $y := \ln(b)$

admits a loss of precision of up to 6 decimal digits. This loss will be especially marked on a machine with hexadecimal number representation, since the leading byte in $1 + y$ will be hexadecimal 1, or binary 0001, with a loss of a further 3 bits, in addition to the loss inherent in the addition. Where the processor's implementation of $\ln(b)$ for b near 1 effectively involves the Taylor series $(b - 1) - (b - 1)^2/2 + \ldots$, the replacement statement

if $b \neq 1$ **then** $y := y \times (\ln(b)/(b - 1))$;

as in IMSL's subroutine MDTD [1], counteracts the loss of precision in evaluating the logarithm as evidenced by column 3 of Table I. However, in the general case there are two solutions, the simplest of which is to evaluate Y = DLOG(1.0D0 + DBLE(Y)), using the variable Y (single precision) for t^2/n, as in the algorithm under discussion. An alternative method might be based on the use of single precision LOG(1.0 + Y) for "sufficiently large" Y, and a suitable number of terms of the Taylor expansion otherwise. In this case the optimal crossover point between the two methods of evaluation would be machine dependent and the coding would be longer, as exemplified for an analogous case in Algorithm 465 [2].

In Algorithm 396 the expression $\exp(x^2/n) - 1$ occurs, and here again substantial loss of precision can occur for small y, to use the algorithm's notation. Admitting a loss of precision of up to nearly 3 decimal digits, this algorithm shifts to a Taylor series expansion of $\exp(y) - 1$ for $y < 0.002$, but this choice is machine dependent and unsuitable for 32-bit machines. Here again I would opt for double precision evaluation of that one expression (storing the result in single precision) over the alternative Taylor series approach.

The Remark on Algorithm 396 was supported by the Fund for Research and Teaching, Department of Nutrition, Harvard School of Public Health.

```
end
else
begin
  comment "tail" series expansion for large t-values;
  integer j;  a := sqrt(b);  y := a × n;  j := 0;
  for j := j + 2 while a ≠ z do
    begin
      z := a;  y := y × (j−1)/(b×j);  a := a + y/(n+j)
    end;
    n := n + 2;  z := y := 0.0;  a := −a;  go to loop
  end
end
```

ACM Transactions on Mathematical Software, Vol. 5, No. 2, June 1979, Pages 238-239.

REMARK ON ALGORITHM 395

Student's t-distribution [S14]
[G.W. Hill, *Comm. ACM 13,* 10 (Oct. 1970), 617-619]

and

REMARK ON ALGORITHM 396

Student's Quantiles [S14]
[G.W. Hill, *Comm. ACM 13,* 10 (Oct. 1970), 619-620]

Mohamed el Lozy [Recd 9 June 1978]
Department of Nutrition, Harvard School of Public Health, 665 Huntington
Ave., Boston, MA 02115

Both of these algorithms incorporate very accurate mathematical methods, but contain a source of loss of precision which is severe for the many processors with precision less than or not sufficiently greater than that claimed for the algorithms.

In Algorithm 395 the use of the asymptotic series involves the evaluation of $\ln(1 + t^2/n)$. For small $y = t^2/n$ and $b = 1 + y$, $\ln(b)$ is of the order of magnitude of y, so that the statement

if $y > 10^{-6}$ **then** $y := \ln(b)$

admits a loss of precision of up to 6 decimal digits. This loss will be especially marked on a machine with hexadecimal number representation, since the leading byte in $1 + y$ will be hexadecimal 1, or binary 0001, with a loss of a further 3 bits, in addition to the loss inherent in the addition. Where the processor's implementation of $\ln(b)$ for b near 1 effectively involves the Taylor series $(b - 1) - (b - 1)^2/2 + \ldots$, the replacement statement

if $b \neq 1$ **then** $y := y \times (\ln(b)/(b - 1))$;

as in IMSL's subroutine MDTD [1], counteracts the loss of precision in evaluating the logarithm as evidenced by column 3 of Table I. However, in the general case there are two solutions, the simplest of which is to evaluate Y = DLOG(1.0D0 + DBLE(Y)), using the variable Y (single precision) for t^2/n, as in the algorithm under discussion. An alternative method might be based on the use of single precision LOG(1.0 + Y) for "sufficiently large" Y, and a suitable number of terms of the Taylor expansion otherwise. In this case the optimal crossover point between the two methods of evaluation would be machine dependent and the coding would be longer, as exemplified for an analogous case in Algorithm 465 [2].

In Algorithm 396 the expression $\exp(x^2/n) - 1$ occurs, and here again substantial loss of precision can occur for small y, to use the algorithm's notation. Admitting a loss of precision of up to nearly 3 decimal digits, this algorithm shifts to a Taylor series expansion of $\exp(y) - 1$ for $y < 0.002$, but this choice is machine dependent and unsuitable for 32-bit machines. Here again I would opt for double precision evaluation of that one expression (storing the result in single precision) over the alternative Taylor series approach.

The Remark on Algorithm 396 was supported by the Fund for Research and Teaching, Department of Nutrition, Harvard School of Public Health.

Table I. Relative Errors in the Calculation of $\ln(1 + t^2/n)$ and $\exp(x^2/n) - 1$ by the Methods of Algorithms 395 and 396, for $x = t = 2$ and Various Values of n

	$\ln(1 + t^2/n)$			$\exp(x^2/n) - 1$	
n	PDP	IBM	IMSL/IBM	PDP	IBM
20	0.245E−6	0.654E−6	0.0	0.538E−6	0.242E−5
40	0.313E−6	0.500E−5	0.0	0.708E−6	0.567E−5
80	0.137E−5	0.149E−4	0.299E−6	0.203E−5	0.118E−4
160	0.754E−6	0.151E−4	0.0	0.177E−5	0.311E−4
320	0.817E−5	0.150E−4	0.0	0.281E−5	0.349E−4
640	0.688E−5	0.909E−4	0.0	0.187E−4	0.178E−4
1280	0.201E−4	0.244E−3	0.298E−6	0.153E−4	0.282E−3
2560	0.224E−5	0.244E−3	0.149E−6	0.372E−6	0.447E−6
5120	0.700E−4	0.244E−3	0.0	0.745E−7	0.298E−6
10240	0.104E−3	0.164E−2	0.0	0.745E−7	0.0

Table I shows the relative errors of single precision evaluation of these two expressions for t (or x) equal to 2 and for various values of n, using the first two terms of the Taylor series for the exponential for $y < 0.002$ as in the algorithm, as well as the IMSL "fix." The computations were done on an IBM 370/168 running under OS/MVT and on a PDP 11/70 running under UNIX. Though both machines have a mantissa of 24 bits, the results on the PDP are far better than those on the 370, presumably due to the hexadecimal normalization of the latter machine.

REFERENCES

1. *Library 1 Reference Manual, Vol. 2.* Int. Math. Stat. Libraries, 3rd ed., 1973.
2. HILL, G.W. Algorithm 465. Student's *t* frequency. *Comm. ACM 16,* 11 (Nov. 1973), 690.

REMARK ON ALGORITHM 395

Student's *t*-Distribution [S14]
[G. W. Hill, *Commun. ACM 13,* 10 (Oct. 1970), 617-618.]

G. W. Hill [Received 6 December 1978; revised 7 July 1979; accepted 6 August 1979]
Division of Mineral Chemistry, CSIRO, Port Melbourne, Australia 3207.

The precision loss noted in [1], in the evaluation of $\ln(1 + t^2/n)$ for Algorithm 395, exceeds the margin of precision of the 36-bit processor over the eight significant decimal digits target mentioned in the algorithm. A suitable correction for this case is the replacement (recall that $y = t^2/n$ and $b = 1 + t^2/n$) of line 8 of the procedure body by

```
if y > 0.01 then y := ln(b)
else y := ((−y × 0.75 + 1.0) × y/3.0 − 0.5) × y × y + y;
```

However, when extended precision is required [2, 4], a number of details of the algorithm must be changed. A more generally applicable replacement of line 8 imitates a technique in Algorithm 465 [3].

```
z := t := y; if y > cmax then y := ln(b)
else
for a := 2.0, a + 1.0 while y ≠ b do
begin z := − z × t; b = y; y := z/a + y end;
```

For small y (<*cmax* say) the precision lost in evaluating $\ln(1 + y)$ corresponds to a relative error about ϵ/y, where ϵ denotes the relative magnitude of processor roundoff. The alternative summation of the logarithmic series until the Rth term

ACM Transactions on Mathematical Software, Vol. 7, No. 2, June 1981, Pages 247-249.

is negligible, $(y^R/(R+1) \leq \epsilon)$, accumulates roundoff error resulting in an average relative error of about $\epsilon\sqrt{R}$. The maximum of these relative errors is minimized, as in Algorithm 465, by choosing $cmax = R^{-1/2}$, where R is determined for a p-bit precision processor by an approximate criterion for neglecting the Rth term; $cmax^R/R \approx \epsilon = 2^{-p}$, or equivalently, $R/2 + 1 \approx 2^p$. For $p = 36$ the solutions $R = 16$ and $cmax = 0.25$ imply an approximate relative error about 4ϵ in the result. For precision as extended as $p = 96$, $cmax \doteq 0.168$ holds this precision loss to about one decimal digit.

For each combination of actual parameter values, Algorithm 395 applies criteria to select whether to use Student's cosine series, the asymptotic normal approximation, or the "tail" series, in order to achieve 8S (significant decimal digits) without excessive loss of speed for the 10.8S processor used. For an extended precision version the criteria must be changed to balance precision against speed characteristics of the processor used. In the case of double precision to about 29S of a CDC 6000–7000 series processor; a target precision of 25S allows for precision loss up to four decimal digits, such as occurs in the subtraction of almost equal quantities, $P(t/n) = 1 - A(t/n)$, to obtain small tail probabilities using Student's cosine series for $A(t/n)$. The effect of this and other causes of precision loss is illustrated in Figure 3 of Algorithm 395.

Greater precision is achieved in the case of extreme probability levels and large n values by the use of the asymptotic normal approximation. To improve precision for larger n, it is efficient to extend the normal approximation up to the sixth term of the series [2] in terms of $z = [(n - \frac{1}{2})\ln(1 + t^2/n)]^{1/2}$ and $b = 48(n - \frac{1}{2})^2$.

$$\begin{aligned}\chi = z &+ (z^3 + 3z)/b - (4z^7 + 33z^5 + 240z^3 + 855z)/10b^2 \\ &+ (64z^{11} + 788z^9 + 9801z^7 + 89775z^5 + 543375z^3 + 1788885z)/210b^3 \\ &- (1152z^{15} + 18896z^{13} + 329496z^{11} + 4698585z^9 + 52027920z^7 \\ &\quad + 424303110z^5 + 2349874800z^3 + 7412830425z)/4200b^4 \\ &\quad + (12288z^{19} + 251776z^{17} + 5645776z^{15} + 108788520z^{13} \\ &\quad + 1738275417z^{11} + 22499221635z^9 + 229192224030z^7 \\ &\quad + 1754611114410z^5 + 9309549058425z^3 + 28756631378475z)/46200b^5 - \cdots\end{aligned}$$

To achieve at least 25S for $n > 100$, the sixth term's divisor is replaced by

$$46200b^4\,(b + 0.43595z^4 + 2z^2 + 537),$$

which accounts for a substantial portion of the omitted next terms, in a fashion similar to the effect displayed in Figure 1 of Algorithm 395, which also illustrates "diminishing returns" in precision gain from additional terms of the series. However, the consequent increase in computing time is moderated by the fact that two-thirds of the arithmetic operations arise in evaluating the fifth and sixth terms, for which single-precision arithmetic and representation of coefficients prove sufficient.

For large enough values of $z^4/b = [\ln(1 + t^2/n)]^2/48$, the asymptotic approximation becomes poor or even divergent, so that for such large values of $1 + t^2/n$ the tail series in powers of $w^2 = 1/(1 + t^2/n)$ is used and converges rapidly with little accumulation of rounding error. The factor $\Gamma((n+1)/2)/(\sqrt{\pi} \times \Gamma(n/2)) \times w^n$ may be evaluated using the same repeated loop as for the cosine series, or by using Algorithm 465 to evaluate the frequency function $f(t\,|\,n)$ as a factor for the equivalent tail series expansion,

$$P(t\,|\,n) = 2f(t\,|\,n) \times \frac{\sqrt{n}}{\sqrt{w}}\left[\frac{1}{n} + \frac{1 \times w^2}{2(n+2)} + \frac{1 \times 3 \times w^4}{2 \times 4(n+4)} + \cdots\right].$$

This can improve speed for large n and, since Algorithm 465 is valid for noninteger n, permits continuation of the probability integral over noninteger values of n down to $n = 1$ with considerable precision for $t^2 > n$; that is $w^2 < \frac{1}{2}$.

In neither form does the series converge well for w near 1; and for small t or large n the time required for evaluation, the accumulated roundoff error, and the truncation error can increase to unacceptable levels.

Where the domains of validity of the three methods overlap, correspondence between results of two methods can be used as a basis for determining the error level of the third. Where two methods achieve precision exceeding the target, counts of instructions or timing tests may be used to select the faster. Increase or decrease of the target precision is found to have a marked effect on computing time so that some compromise trade-off of precision against speed is required according to the particular processor used and the intended application. Reasonable speed of execution with precision at least to 23*S*, but generally 25*S* or more, is achieved for the CDC 6000–7000 series processor by replacing line 5 by (recall that t represents t^2, the square of the actual parameter value)

if $n > entier(n) \vee n > 1000 \vee n \geq 100 \wedge t < 0.1 \times n - 5$ **then**

to select evaluation by the six-term asymptotic approximation. Replacement of line 15 by

if $n < 100 \wedge t < 16$ **then**

selects the cosine series method for smaller values of n and $t < 4$; the **else** clause evaluates the double-precision tail expansion to obtain a sufficiently precise result for smaller probability levels. For continuation extension as outlined in the preceding paragraph, the replacements of lines 5 and 15 are

if $n > 1000 \vee n \geq 100 \wedge t < 0.1 \times n - 5$ **then**
if $n = entier(n) \wedge n < 100 \wedge t < 16$ **then**

Some margin of precision loss from the full precision level of the processor is unavoidable due to accumulated roundoff error and is traded off further to achieve an acceptable speed of execution. With this reservation the methods of Algorithm 395 can be extended to provide higher precision results, as evidenced by their use in evaluating quantiles to 20*D* [2].

REFERENCES

1. El Lozy, M. Remark on Algorithm 395. Student's t distribution. *ACM Trans. Math. Softw. 5*, 2 (June 1979), 238–239.
2. Hill, G.W. Reference Table: "Student's" t-distribution quantiles to 20*D*. *Tech. Paper No. 35*, Div. Math. Statist., CSIRO, Australia, 1972, 24pp.
3. Hill, G.W. Algorithm 465. Student's t frequency. *Commun. ACM 16*, 11 (Nov. 1973), 690.
4. Ling, R.F. A study of the accuracy of some approximations to t, χ^2 and F tail probabilities. *J. Amer. Statist. Assoc. 73*, 362 (1978), 274–283.

Table I. Relative Errors in the Calculation of $\ln(1 + t^2/n)$ and $\exp(x^2/n) - 1$ by the Methods of Algorithms 395 and 396, for $x = t = 2$ and Various Values of n

	$\ln(1 + t^2/n)$			$\exp(x^2/n) - 1$	
n	PDP	IBM	IMSL/IBM	PDP	IBM
20	0.245E−6	0.654E−6	0.0	0.538E−6	0.242E−5
40	0.313E−6	0.500E−5	0.0	0.708E−6	0.567E−5
80	0.137E−5	0.149E−4	0.299E−6	0.203E−5	0.118E−4
160	0.754E−6	0.151E−4	0.0	0.177E−5	0.311E−4
320	0.817E−5	0.150E−4	0.0	0.281E−5	0.349E−4
640	0.688E−5	0.909E−4	0.0	0.187E−4	0.178E−4
1280	0.201E−4	0.244E−3	0.298E−6	0.153E−4	0.282E−3
2560	0.224E−5	0.244E−3	0.149E−6	0.372E−6	0.447E−6
5120	0.700E−4	0.244E−3	0.0	0.745E−7	0.298E−6
10240	0.104E−3	0.164E−2	0.0	0.745E−7	0.0

Table I shows the relative errors of single precision evaluation of these two expressions for t (or x) equal to 2 and for various values of n, using the first two terms of the Taylor series for the exponential for $y < 0.002$ as in the algorithm, as well as the IMSL "fix." The computations were done on an IBM 370/168 running under OS/MVT and on a PDP 11/70 running under UNIX. Though both machines have a mantissa of 24 bits, the results on the PDP are far better than those on the 370, presumably due to the hexadecimal normalization of the latter machine.

REFERENCES

1. *Library 1 Reference Manual, Vol. 2.* Int. Math. Stat. Libraries, 3rd ed., 1973.
2. Hill, G.W. Algorithm 465. Student's *t* frequency. *Comm. ACM 16,* 11 (Nov. 1973), 690.

ALGORITHM 396
STUDENT'S *t*-QUANTILES [S14]

G. W. Hill (Recd. 6 Jan. 1970 and 18 May 1970)
C.S.I.R.O., Division of Mathematical Statistics, Glen Osmond, South Australia

KEY WORDS AND PHRASES: Student's *t*-statistic, quantile, asymptotic approximation
CR CATEGORIES: 5.12, 5.5

real procedure *t quantile* (*P*, *n*, *normdev*, *error*);
value *P*, *n*; **real** *P*, *n*; **real procedure** *normdev*, *error*;
comment This algorithm evaluates the positive quantile at the (two-tail) probability level *P*, for Student's *t*-distribution with *n* degrees of freedom. The quantile function is an inverse of the two-tail

$$P(t|n) = 2\,\frac{\Gamma(\frac{1}{2}n+\frac{1}{2})}{\sqrt{(\pi n)}\,\Gamma(\frac{1}{2}n)}\int_t^\infty \frac{du}{(1+u^2/n)^{(\frac{1}{2}n+\frac{1}{2})}}$$

which is approximated in Algorithm 395 [1] by series whose inverses are used in this algorithm for *t* quantiles. Test calculations to 36-bit precision indicate that the result is correct to at least 6 significant digits, even for the analytic continuation through noninteger values of $n > 5$.

The procedure *normdev*(*p*) is assumed to return a negative normal deviate at the lower tail probability level *p*, e.g. -2.32 for $p = 0.01$. The user-supplied procedure for *error*(*n*) should give a diagnostic warning that the value of *P* or *n* is invalid and may **go to** a label, terminate, or return a distinctive value as an error signal to the calling program.

For $n = 1$ and $n = 2$ the exact result of integration is readily inverted to yield $t = cot(P\times\pi/2)$ and $t^2 = 2/(P(2-P))-2$, respectively. For larger *n* an asymptotic inverse expansion about normal deviates is applicable, while for smaller values of *P* a second series expansion is used to achieve sufficient precision. Both approximations have been adjusted to enhance precision for *n* as low as 3.

Both methods involve an expansion of the factor

$$d/n = \tfrac{1}{2}\sqrt{\pi}\,\Gamma(\tfrac{1}{2}n)/\Gamma(\tfrac{1}{2}n + \tfrac{1}{2})$$

in terms of $a = 1/(n-\frac{1}{2})$ and $b=48/a^2$

$$d/n = \sqrt{(a\pi/2)}(1-3/b+94.5/b^2-9058.5/b^3+\cdots)\ [2].$$

A three term approximation uses $b(b+c)$ instead of b^2 as a divisor, where the coefficients in

$$c = 96.36 - 16a - 98a^2 + 20700a^3/b,$$

have been fitted to ensure 8 significant digits in *d* for *n* as low as 3.

The inverse asymptotic expansion of Cornish-Fisher type relates a function $y(t) = \sqrt{[(n-\frac{1}{2})\ln(1+t^2/n)]}$ to the normal deviate χ at the corresponding probability level, $P/2$:

$$\begin{aligned}y=\chi - (\chi^3+3\chi)/b + (4\chi^7+63\chi^5+360\chi^3+945\chi)/10b^2\\ - (64\chi^{11}+1628\chi^9+19881\chi^7+145719\chi^5+694575\chi^3\\ +1902285\chi)/210b^3 + \cdots\ [2],\end{aligned}$$

whence $t = \sqrt{[n\times(exp(a\times y^2)-1)]}$. For a three term approximation the third term's divisor is replaced by

$$10b \times (b+c-2\chi-7\chi^2-5\chi^3+0.05\times d\times\chi^4),$$

whose coefficients have been fitted to reduce the error for small *n* and for larger *n* and χ. For $n < 5$, *c* is increased by $0.3(n-4.5)(\chi+0.6)$ to further reduce error in an interval of *P* not well covered by the following approximation.

For small *P*, where t^2/n is large, the integrand may be expanded in terms of $w^2 = 1/(1+t^2/n)$ and integrated term by term to yield

$$P = \frac{nw^n}{d}\left\{\frac{1}{n} + \frac{w^2}{2(n+2)} + \frac{1\times 3w^4}{2\times 4(n+4)} + \cdots\right\},$$

which may be inverted to express t^2/n in terms of $y = (P\times d)^{2/n}$

$$\frac{t^2}{n} = \frac{1}{y} + \frac{n+1}{n+2}\left\{-1 + \frac{y}{2(n+4)} + \frac{n\times y^2}{3(n+2)(n+6)} + \frac{n(n+3)(2n^2+9n-2)y^3}{8(n+2)^2(n+4)^2(n+8)} + \cdots\right\}.$$

Since the ratio of successive terms is nearly $n \times y/(n+6)$ for small *n*, replacement of the term in y^2 by $y/[3(n+2)\{(n+6)/(n\times y)-1.0\}]$ provides an approximate allowance for subsequent terms in the series, which is empirically improved by replacing the -1.0 by $-0.822 - 0.089 \times d$.

As *n* and *P* increase, the errors for the asymptotic approximation decrease, whereas errors for the second series increase, so that for each value of *n* the error curves intersect at a value of *P* above which the asymptotic approximation is better and below which the second series should be used. By adjusting the two approximations the error level at these intersections has been balanced at about the seventh significant digit for $n \geq 3$ and $P > 10^{-24}$. The value of *y* at these points is about $a + 0.05$ and this fact provides a convenient criterion for selecting which approximation to use: the asymptotic series if *y* exceeds $a + 0.05$, otherwise the second series.

Although better approximations could be obtained by use of more terms in each series, greater precision can be achieved by using the result of this algorithm as a starting value for iterative inversion of $P(t \mid n)$, whose value and derivative can be computed with considerable precision using recurrence relations as in Algorithm 395.

A comparison of results from this algorithm against values obtained by inverting the function provided by Algorithm 395 indicates a precision of over 6 significant digits for $10^{-24} \leq P \leq 0.9, n \geq 1$. At the conventional tabulation points in $0.001 \leq P \leq 0.9$ results for $n = 1$, $n = 2$, and $n > 10$ checked to 8 significant digits.

Previously published tables [3, 4, 5] provide 3 or 4 decimal place check values, some of which are found to be slightly in error. Thus for $n = 2$, $P = 0.001$, *t* is given as 31.598 by Fisher and Yates and by Federighi, 31.5991 by Smirnov, and 31.5990546 by this procedure, while for $n = 1$, $P = 0.001$ the value 636.6096 given by Smirnov conflicts with Fisher and Yates, Federighi (636.619) and this procedure (636.61925). Other errors in the last few digits in Smirnov's table for low values of *n* and *P* include 10.2129 for $n = 3$, $P = 0.002$, which should be 10.2145, and 4.7812 for $n = 9$, $P = 0.001$, which should be 4.7809.

t quantile may be used to obtain percentiles at values of *P* and *n* not provided in existing tables or for extending their accuracy. Such tables are customarily used for assessing the significance of a sample value for *t*, but for automatic computation the probability level is more effectively determined as $P(t \mid n)$ using a direct procedure such as Algorithm 395.

Pseudorandom *t*-values may be generated for sampling applications by using uniformly distributed pseudorandom numbers for *P*, and in this case *normdev* may be a **real procedure** returning pseudorandom normal deviates which are independent of *P*.

REFERENCES:

1. HILL, G. W. Algorithm 395, Student's *t*-distribution *Comm. ACM 13* (Oct. 1970), 617-618.
2. HILL, G. W. Progress results on asymptotic approximations for Student's *t*. Unpublished manuscript, Oct. 1969.
3. FISHER, R. A., AND YATES, F. *Statistical Tables for Biological Agricultural and Medical Research*. Oliver and Boyd, London, 1963.
4. SMIRNOV, N. V. *Tables for the Distribution and Density Functions of t-Distribution*. Pergamon Press, New York, 1961.
5. FEDERIGHI, E. T. Extended tables of the percentage points of Student's *t*-distribution. *J. Amer. Stat. Assoc. 54* (1959), 683-688;

```
if n < 1 ∨ P > 1.0 ∨ P ≤ 0.0 then t quantile := error(n)
else if n = 2 then t quantile := sqrt(2.0/(P×(2.0−P))−2.0)
else
begin
  real half pi;  half pi := 1.5707963268;
  if n = 1 then
  begin P := P × half pi;  t quantile := cos(P)/sin(P) end
  else
  begin
    real a, b, c, d, x, y;
    a := 1.0/(n−0.5);  b := 48.0/a ↑ 2;
    c := ((20700×a/b−98)×a−16) × a + 96.36;
    d := ((94.5/(b+c)−3.0)/b+1.0) × sqrt(a×half pi) × n;
    x := d × P;  y := x ↑ (2.0/n);
    if y > 0.05 + a then
    begin
      comment  Asymptotic inverse expansion about normal;
      x := normdev(P×0.5);  y := x ↑ 2;
      if n < 5 then c := c + 0.3 × (n−4.5) × (x+0.6);
      c := (((0.05×d×x−5.0)×x−7.0)×x−2.0) × x + b + c;
      y  :=  (((((0.4×y+6.3)×y+36.0)×y+94.5)/c−y−3.0)/b+
        1.0) × x;
      y := a × y ↑ 2;
      y := if y>0.002 then exp(y) − 1.0 else 0.5 × y ↑ 2 + y
    end
    else y := ((1.0/(((n+6.0)/(n×y)−0.089×d−0.822)×
      (n+2.0)×3.0)+0.5/(n+4.0))×y−1.0) ×
        (n+1.0)/(n+2.0) + 1.0/y;
    t quantile := sqrt(n×y)
  end
end Student's t-quantile
```

ACM Transactions on Mathematical Software, Vol. 5, No. 2, June 1979, Pages 238-239.

REMARK ON ALGORITHM 395

Student's *t*-distribution [S14]
[G.W. Hill, *Comm. ACM 13,* 10 (Oct. 1970), 617-619]

and

REMARK ON ALGORITHM 396

Student's Quantiles [S14]
[G.W. Hill, *Comm. ACM 13,* 10 (Oct. 1970), 619-620]

Mohamed el Lozy [Recd 9 June 1978]
Department of Nutrition, Harvard School of Public Health, 665 Huntington Ave., Boston, MA 02115

Both of these algorithms incorporate very accurate mathematical methods, but contain a source of loss of precision which is severe for the many processors with precision less than or not sufficiently greater than that claimed for the algorithms.

In Algorithm 395 the use of the asymptotic series involves the evaluation of $\ln(1 + t^2/n)$. For small $y = t^2/n$ and $b = 1 + y$, $\ln(b)$ is of the order of magnitude of *y*, so that the statement

if $y > 10^{-6}$ **then** $y := \ln(b)$

admits a loss of precision of up to 6 decimal digits. This loss will be especially marked on a machine with hexadecimal number representation, since the leading byte in $1 + y$ will be hexadecimal 1, or binary 0001, with a loss of a further 3 bits, in addition to the loss inherent in the addition. Where the processor's implemen-

The Remark on Algorithm 396 was supported by the Fund for Research and Teaching, Department of Nutrition, Harvard School of Public Health.

Table I. Relative Errors in the Calculation of $\ln(1 + t^2/n)$ and $\exp(x^2/n) - 1$ by the Methods of Algorithms 395 and 396, for $x = t = 2$ and Various Values of n

	$\ln(1 + t^2/n)$			$\exp(x^2/n) - 1$	
n	PDP	IBM	IMSL/IBM	PDP	IBM
20	0.245E−6	0.654E−6	0.0	0.538E−6	0.242E−5
40	0.313E−6	0.500E−5	0.0	0.708E−6	0.567E−5
80	0.137E−5	0.149E−4	0.299E−6	0.203E−5	0.118E−4
160	0.754E−6	0.151E−4	0.0	0.177E−5	0.311E−4
320	0.817E−5	0.150E−4	0.0	0.281E−5	0.349E−4
640	0.688E−5	0.909E−4	0.0	0.187E−4	0.178E−4
1280	0.201E−4	0.244E−3	0.298E−6	0.153E−4	0.282E−3
2560	0.224E−5	0.244E−3	0.149E−6	0.372E−6	0.447E−6
5120	0.700E−4	0.244E−3	0.0	0.745E−7	0.298E−6
10240	0.104E−3	0.164E−2	0.0	0.745E−7	0.0

tation of $\ln(b)$ for b near 1 effectively involves the Taylor series $(b - 1) - (b - 1)^2/2 + \ldots$, the replacement statement

if $b \neq 1$ **then** $y := y \times (\ln(b)/(b - 1))$;

as in IMSL's subroutine MDTD [1], counteracts the loss of precision in evaluating the logarithm as evidenced by column 3 of Table I. However, in the general case there are two solutions, the simplest of which is to evaluate Y = DLOG(1.0D0 + DBLE(Y)), using the variable Y (single precision) for t^2/n, as in the algorithm under discussion. An alternative method might be based on the use of single precision LOG(1.0 + Y) for "sufficiently large" Y, and a suitable number of terms of the Taylor expansion otherwise. In this case the optimal crossover point between the two methods of evaluation would be machine dependent and the coding would be longer, as exemplified for an analogous case in Algorithm 465 [2].

In Algorithm 396 the expression $\exp(x^2/n) - 1$ occurs, and here again substantial loss of precision can occur for small y, to use the algorithm's notation. Admitting a loss of precision of up to nearly 3 decimal digits, this algorithm shifts to a Taylor series expansion of $\exp(y) - 1$ for $y < 0.002$, but this choice is machine dependent and unsuitable for 32-bit machines. Here again I would opt for double precision evaluation of that one expression (storing the result in single precision) over the alternative Taylor series approach.

Table I shows the relative errors of single precision evaluation of these two expressions for t (or x) equal to 2 and for various values of n, using the first two terms of the Taylor series for the exponential for $y < 0.002$ as in the algorithm, as well as the IMSL "fix." The computations were done on an IBM 370/168 running under OS/MVT and on a PDP 11/70 running under UNIX. Though both machines have a mantissa of 24 bits, the results on the PDP are far better than those on the 370, presumably due to the hexadecimal normalization of the latter machine.

REFERENCES

1. *Library 1 Reference Manual, Vol. 2.* Int. Math. Stat. Libraries, 3rd ed., 1973.
2. Hill, G.W. Algorithm 465. Student's *t* frequency. *Comm. ACM 16,* 11 (Nov. 1973), 690.

Table I. Relative Errors in the Calculation of $\ln(1 + t^2/n)$ and $\exp(x^2/n) - 1$ by the Methods of Algorithms 395 and 396, for $x = t = 2$ and Various Values of n

	$\ln(1 + t^2/n)$			$\exp(x^2/n) - 1$	
n	PDP	IBM	IMSL/IBM	PDP	IBM
20	0.245E−6	0.654E−6	0.0	0.538E−6	0.242E−5
40	0.313E−6	0.500E−5	0.0	0.708E−6	0.567E−5
80	0.137E−5	0.149E−4	0.299E−6	0.203E−5	0.118E−4
160	0.754E−6	0.151E−4	0.0	0.177E−5	0.311E−4
320	0.817E−5	0.150E−4	0.0	0.281E−5	0.349E−4
640	0.688E−5	0.909E−4	0.0	0.187E−4	0.178E−4
1280	0.201E−4	0.244E−3	0.298E−6	0.153E−4	0.282E−3
2560	0.224E−5	0.244E−3	0.149E−6	0.372E−6	0.447E−6
5120	0.700E−4	0.244E−3	0.0	0.745E−7	0.298E−6
10240	0.104E−3	0.164E−2	0.0	0.745E−7	0.0

tation of $\ln(b)$ for b near 1 effectively involves the Taylor series $(b - 1) - (b - 1)^2/2 + \ldots$, the replacement statement

if $b \neq 1$ **then** $y := y \times (\ln(b)/(b - 1))$;

as in IMSL's subroutine MDTD [1], counteracts the loss of precision in evaluating the logarithm as evidenced by column 3 of Table I. However, in the general case there are two solutions, the simplest of which is to evaluate $Y =$ DLOG(1.0D0 + DBLE(Y)), using the variable Y (single precision) for t^2/n, as in the algorithm under discussion. An alternative method might be based on the use of single precision LOG(1.0 + Y) for "sufficiently large" Y, and a suitable number of terms of the Taylor expansion otherwise. In this case the optimal crossover point between the two methods of evaluation would be machine dependent and the coding would be longer, as exemplified for an analogous case in Algorithm 465 [2].

In Algorithm 396 the expression $\exp(x^2/n) - 1$ occurs, and here again substantial loss of precision can occur for small y, to use the algorithm's notation. Admitting a loss of precision of up to nearly 3 decimal digits, this algorithm shifts to a Taylor series expansion of $\exp(y) - 1$ for $y < 0.002$, but this choice is machine dependent and unsuitable for 32-bit machines. Here again I would opt for double precision evaluation of that one expression (storing the result in single precision) over the alternative Taylor series approach.

Table I shows the relative errors of single precision evaluation of these two expressions for t (or x) equal to 2 and for various values of n, using the first two terms of the Taylor series for the exponential for $y < 0.002$ as in the algorithm, as well as the IMSL "fix." The computations were done on an IBM 370/168 running under OS/MVT and on a PDP 11/70 running under UNIX. Though both machines have a mantissa of 24 bits, the results on the PDP are far better than those on the 370, presumably due to the hexadecimal normalization of the latter machine.

REFERENCES

1. *Library 1 Reference Manual, Vol. 2.* Int. Math. Stat. Libraries, 3rd ed., 1973.
2. Hill, G.W. Algorithm 465. Student's t frequency. *Comm. ACM 16,* 11 (Nov. 1973), 690.

REMARK ON ALGORITHM 396

Student's *t*-Quantiles [S14]
[G. W. Hill, *Commun. ACM 13,* 10 (Oct. 1970), 619–620.]

G. W. Hill [Received 6 December 1978; revised 7 July 1979; accepted 6 August 1979]
Division of Mineral Chemistry, CSIRO, Port Melbourne, Australia 3207.

The precision in excess of six decimal digits, claimed for quantiles evaluated using Algorithm 396 on a 36-bit precision processor, cannot be achieved for a processor precision of six hexadecimal digits. As noted in [1], the statement

$y :=$ **if** $y > 0.002$ **then** $exp(y) - 1.0$ **else** $0.5 \times y \uparrow 2 + y$

should be replaced by its implied extension

```
y := if y > 0.1 then exp(y) − 1.0
     else ((y + 4.0) × y + 12.0) × y × y/24.0 + y
```

The relative error of this truncated Taylor series is less than that recorded for $exp(y > 0.1) - 1$ in el Lozy's tests [1] on an IBM 370/168.

For extended precision quantiles an initial approximation by Algorithm 396, for example, $t_0 :=$ *t quantile* (*P, n, normdev, error*), may be used as argument in an extended precision version of Algorithm 395 [2] to evaluate the two-tail probability integral $P(t_0 \mid n)$. The difference of this result from the target probability level may be divided by twice the frequency $f(t_0 \mid n)$, evaluated using Algorithm 465 [3], to obtain the first-order correction for t_0,

$$z = \frac{\frac{1}{2}(P(t_0 \mid n) - P)}{f(t_0 \mid n)}.$$

Rather than iterative inversion $t_{r+1} = t_r + z(t_r \mid n)$, as suggested in the commentary of Algorithm 396, it is more efficient to avoid repeated evaluation of the probability integral and frequency function by using the Taylor series expansion [5]

$$t = t_0 + z + \frac{\psi z^2}{2!} + \frac{(2\psi^2 + \psi')z^3}{3!} + \cdots,$$

where

$$\psi = \frac{-\partial}{\partial t_0}[\ln f(t_0 \mid n)] = \frac{(n+1)t_0}{n + t_0^2},$$

$$\psi' = \frac{\partial \psi}{\partial t_0} = \frac{(n+1)(n - t_0^2)}{(n + t_0^2)^2},$$

and the coefficient c_r of $z^r/r!$ is determined from

$$c_{r+1} = \left(r\psi + \frac{\partial}{\partial t_0}\right)c_r, \qquad c_0 \equiv 1.$$

The relative error of the series, truncated to order z^s, is approximately $\psi^s t_0^s \epsilon^{s+1}/(s+1)$, where $\epsilon = z/t_0$ is the relative error of the initial approximation. Using Algorithm 396, for which $|\epsilon| < 10^{-6}$, the first few terms of the series provide considerable precision in the result.

For processor "double precision" of 14 hexadecimal (16–17 decimal) digits, such as that of the IBM 360/370 series, the first three terms are sufficient:

$$t := (n+1) \times t_0 \times z \times z \times \frac{0.5}{t_0 \times t_0 + n} + z + t_0;$$

provided that both the precision of $P(t_0 \mid n)$ and the sum of precisions of t_0 and $f(t_0 \mid n)$ at least equal a level appropriate for 14 hexadecimal precision, such as 14 decimals to allow for precision loss in evaluating $P(t_0 \mid n)$. For 96-bit double precision of the CDC 6000 series processor, allowing two or three decimal digit precision loss in $P(t_0 \mid n)$, the series to z^3 is sufficient for precision in excess of 25 decimal digits, except for extreme probability levels beyond 10^{-20} and large n (>50), for which the term in z^4 ensures 25–26 decimals.

It is faster to use single-precision rather than double-precision operations in evaluating higher order terms of the Taylor series, such as the first term in the statement displayed above and the terms in z^3 and z^4 in the fifth-order case. This approach has been validated by a FORTRAN implementation to double precision for the CDC 6400 and 7600 for tabulation of Student's t-quantiles rounded off to 20D [4].

REFERENCES

1. El Lozy, M. Remark on Algorithm 395. Student's *t* distribution. *ACM Trans. Math. Softw. 5*, 2 (June 1979), 238–239.
2. Hill, G.W. Algorithm 395. Student's *t* distribution. *Commun. ACM 13*, 10 (Oct. 1970), 617–619.
3. Hill, G.W. Algorithm 465. Student's *t* frequency. *Commun. ACM 16*, 11 (Nov. 1973), 690.
4. Hill, G.W. Reference Table: "Student's" *t*-distribution quantiles to 20*D*. *Tech. Paper No. 35*, Div. Math. Statist., CSIRO, Australia, 1972, 24pp.
5. Hill, G.W., and Davis, A.W. Generalized asymptotic expansions of Cornish-Fisher type. *Ann. Math. Statist. 39* (1968), 1264–1273.

ALGORITHM 397
AN INTEGER PROGRAMMING PROBLEM [H]

S. K. Chang and A. Gill (Recd. 16 Feb. 1970 and 11 May 1970)
Electronics Research Laboratory and Department of Electrical Engineering and Computer Sciences, University of California,* Berkeley, CA 94720

* Research sponsored by the Air Force Office of Scientific Research Office of Aerospace Research, United States Air Force, AFOSR Grant AF-AFOSR-639-67 and the National Science Foundation, Grant GK2277.

KEY WORDS AND PHRASES: integer programming, change-making problem
CR CATEGORIES: 5.41

```
procedure MINDIST(C, M, SENSE, W, RESULT);
  value C, M;  integer C, M;  Boolean SENSE;
  integer array W, RESULT;
```

comment This algorithm solves an integer programming problem described in [1]. Given is a fixed weight vector $w = (w_1, w_2, \cdots, w_m)$, where the w_i are nonnegative integers, where m is a positive integer, and where

$$1 = w_1 < w_2 < \cdots < w_m$$

For any nonnegative integer c (representing cost), an m-distribution of c relative to w is an m-tuple $(a_1, a_2, \cdots, a_m)$ such that the a_i are nonnegative integers, and such that $\sum_{i=1}^{m} a_i w_i = c$. The m-distribution $(a_1, a_2, \cdots, a_m)$ is minimal if, for any m-distribution $(b_1, b_2, \cdots, b_m)$ of c relative to w, we have $\sum_{i=1}^{m} a_i \leq \sum_{i=1}^{m} b_i$. The m-distribution $(a_1, a_2, \cdots, a_m)$ is standard if it is obtainable as follows:

$$c_m = c$$

$$c_i = c_{i+1} - a_{i+1} \times w_{i+1} \qquad (i=m-1, m-2, \cdots, 1)$$

$$a_i = c_i/w_i \qquad (i=m, m-1, \cdots, 1)$$

(where all divisions are integer divisions).

If *MINDIST*(*C*, *M*, *SENSE*, *W*, *RESULT*) is called with a nonnegative integer *C*, a positive integer *M*, and an array *W* = (*W*[1], *W*[2], $\cdots$, *W*[*M*]), then the resulting array *RESULT* = (*RESULT*[1], *RESULT*[2], $\cdots$, *RESULT*[*M*]) is a minimal *M*-distribution of *C* relative to *W*. If, before calling *MINDIST*, *SENSE* is set to **true**, then *MINDIST* retains *SENSE* as **true** if and only if *RESULT* is also a standard *M*-distribution of *C* relative to *W*.

Reference:

1. Chang, S. K., and Gill, A. Algorithmic solution of the change-making problem. *J. ACM 17* (Jan. 1970) 113–122;

```
begin
  integer I, J, R, Q, SUM, SUN;
  integer array A[1:M], B[1:M];
  if M = 1 then
  begin
    RESULT[1] := C;
EXIT1 :
    go to EXIT
  end
  Q := C/W[M];
  if (Q×W[M]) > C then Q := Q − 1;
  R := C − W[M] × Q;
  if M = 2 then
  begin
    RESULT[1] := R;  RESULT[2] := Q;
EXIT2 .
    go to EXIT
  end;
  J := 0;
LOOP:
  MINDIST (R+J×W[M], M−1, SENSE, W, B);
  if J ≠ 0 then go to NOT ZERO;
BETA:
  for I := 1 step 1 until M−1 do A[I] := B[I];
  A[M] := 0;
GAMMA:
  if J = Q then
  begin
    for I := 1 step 1 until M do RESULT[I] := A[I];
EXIT3:
    go to EXIT
  end;
  SUM := 0;
  for I := 1 step 1 until M do SUM := SUM + A[I];
  if (W[M]×SUM−R−J×W[M])/(W[M]−W[M−1]) ≤ 0 then
  begin
    for I := 1 step 1 until M − 1 do RESULT[I] := A[I];
    RESULT[M] := A[M] + Q − J;
EXIT4:
    go to EXIT
  end;
  J := J + 1;
  go to LOOP;
NOT ZERO:
  SUM := 0;  SUN := 0;
  for I := 1 step 1 until M do SUM := SUM + A[I];
  for I := 1 step 1 until M − 1 do SUN := SUN + B[I];
  if SUM ≤ SUN then
  begin A[M] := A[M] + 1; go to GAMMA end;
  SENSE := false;
  go to BETA;
EXIT:
end PROCEDURE MINDIST
```

Remark on Algorithm 397 [H]

An Integer Programming Problem [S.K. Chang and A. Gill, *Comm. ACM 13* (Oct. 1970), 620–621]

Stephen C. Johnson and Brian W. Kernighan (Recd. 15 Sept. 1971)
Bell Laboratories, Murray Hill, NJ 07974

Editor's note: The first correction was also noted by K.W. Coull of the University of Alberta.—L.D.F.

Key Words and Phrases: integer programming, change-making problem

CR Categories: 5.41

The published algorithm contains two substantial errors.

1. Five lines after the label *EXIT*3, the line

if $(W[M] \times SUM-R-J \times W[M])/(W[M]-W[M-1]) \leq 0$ **then**

should be replaced by

if $(W[M-1] \times SUM-R-J \times W[M]) < (W[M]-W[M-1])$ **then**

The use of $W[M-1]$ instead of $W[M]$ corrects an error which also appears in the J. ACM article [1] upon which Algorithm 397 is based.

2. Four lines after the label *NOTZERO*, the line

if $SUM \leq SUN$ **then**

must be replaced by

if $SUM < SUN$ **then**

When this change is made, the algorithm correctly solves the test case described in [1], although producing a different answer than was published there.

The algorithm would be clarified if, three and four lines after the label *EXIT*1, the statements

$Q := C/W[M];$

if $(Q \times W[M]) > C$ **then** $Q := Q - 1;$

were replaced by

$Q := C \div W[M];$

References

1. Chang, S.K., and Gill, A. Algorithmic solution of the change-making problem. *J. ACM 17* (Jan. 1970), 113–122.

ALGORITHM 398
TABLELESS DATE CONVERSION* [Z]

Richard A. Stone (Recd. 2 Jan. 1970 and 6 April 1970)
Western Electric Company, P.O. Box 900,
Princeton, NJ 08540

* Patent applied for.

KEY WORDS AND PHRASES: date, calendar
CR CATEGORIES: 5.9

```
procedure calendar(y, n, m, d);
  value y, n;  integer y, n, m, d, t;
comment  calendar is called with the year in y and the day of the
  year in n. The month number is returned in m, and the day of the
  month is returned in d. The first section of the procedure changes
  the dates so that February has 30 days. The second section uses
  the fact that 30.55 (m+2) − 91 passes through the number of
  days preceeding each month.
    Error detection: m will be in the range 1–12 if and only if n
  is in the correct range;
begin
  t := if (y ÷ 4)*4 = y then 1 else 0;
  comment  The following statement is unnecessary
    if it is known that 1900 < y < 2100;
  t := if (y÷400)*400 = y ∨ (y÷100)*100 ≠ y then t else 0;
  d := n + (if n > (59+t) then 2 − t else 0);
  m := ((d+91)*100) ÷ 3055;
  d := (d+91) − (m*3055) ÷ 100;
  m := m − 2
end calendar
```

Remark on Algorithm 398 [Z]

Tableless Date Conversion [Richard A. Stone, *Comm. ACM 13* (Oct. 1970), 621]

J. Douglas Robertson [Recd. 16 Dec. 1970 and 30 Mar. 1971]
200 Oakcrest Drive F-161, Lafayette, LA 70501

Key Words and Phrases: date, calendar, Fortran statement function, arithmetic statement function
CR Categories: 3.15, 4.9, 5.9

As a companion to Algorithm 398, I offer a relatively compact algorithm for calculating the day of the year on which a particular date falls given the year, month, and day of the month. The algorithm is written below as a Fortran arithmetic statement function, where *I* is the year; *J* is the month, (1 = Jan, . . . , 12 = Dec); and *K* is the day of the month.

```
IDAY(I,J,K) = 3055*(J+2)/100−(J+10)/13*2−91
              + (1−(I−I/4*4+3)/4+(I−I/100*100+99)/100
              − (I−I/400*400+399)/400)*(J+10)/13+K
```

The above, along with Stone's Algorithm 398, Robert G. Tantzen's Algorithm 199 [2], and the two algorithms by H.F. Fliegel and T.C. Van Flandern [1] constitute a comprehensive set of algorithms for processing calendar dates. A useful addition to this set would be an algorithm for Zeller's Congruence (calculates the day of the week on which a particular date falls) as described in [3]. It appears below as a Fortran arithmetic statement function, where *I* is the year; *J* is the month, (1 = Jan, . . . , 12 = Dec); and *K* is the day of the month.

```
IZLR(I,J,K) = MOD((13*(J+10−(J+10)/13*12)−1)/5+K+77
              + 5*(I+(J−14)/12−(I+(J−14)/12)/100*100)/4
              +  (I+(J−14)/12)/400−(I+(J−14)/12)/100*2,7)
```

References

1. Fliegel, H.F., and Van Flandern, T.C. A machine algorithm for processing calendar dates. *Comm. ACM 11* (Oct. 1968), 657.
2. Tantzen, Robert G. Conversions between calendar date and Julian day number, Algorithm 199. *Comm. ACM 6* (Aug. 1963), 444.
3. Uspensky, J.V., and Heaslet, M.A. *Elementary Number Theory.* McGraw-Hill, New York, 1939, p. 206.

ALGORITHM 399
SPANNING TREE [H]

Jouko J. Seppänen (Recd. 6 Jan. 1970 and 8 May 1970)
Computing Center, Helsinki University of Technology,
Otaniemi, Finland

KEY WORDS AND PHRASES: graph, tree, spanning tree
CR CATEGORIES: 5.32

```
procedure spanning tree(v, e, I, J, p, T);
  value v, e;  integer v, e, p;  integer array I, J, T;
```

comment This procedure grows a spanning tree T for a given undirected loop-free graph $G = (N, E)$ of v vertices and e edges. If G is disconnected a spanning forest will be grown.

The edges $(I[k], J[k]) \in E$ for $k = 1, 2, \cdots, e$ are assumed to be stored in the arrays $I[1:e]$ and $J[1:e]$. At each stage of the algorithm one edge is considered whereby one of four possible conditions will arise. If neither of the vertices is included in a tree, this edge is taken as a new tree and its vertices numbered by an incremented component number c. If one vertex is in a tree, the edge will be grown to this tree. If the two vertices are in different trees, these will be grafted into a single tree by renumbering the vertices of the other component. Finally, if both vertices are in the same tree, the edge completes a fundamental cycle of the graph with respect to the spanning tree and consequently will not be considered further. At the end, the indices of the edges in the spanning tree are stored in the array $T[1:v-p]$ where p is the number of trees in the forest. The procedure can also be used to find a minimal spanning tree by sorting the edges into ascending order before calling the procedure.

The main loop in the procedure is executed e times. For cases where the ratio e/v is high it could be worthwhile to introduce an additional variable, say d, in the program, for keeping a count of the number of edges included in T. When d has attained the value of $v - 1$ the algorithm could terminate.

References:

1. Berge, C., and Ghouila-Houri, A. *Programmes, Jeux et Reseaux de Transport.* Dunod, Paris, 1962, pp. 179–182.
2. Berge, C., and Ghouila-Houri, A. *Programming, Games and Transportation Networks.* Methuen, London, and Wiley, New York, 1965, pp. 177–180.
3. Kruskal, J. B., Jr. On the shortest spanning subtree of a graph and the travelling salesman problem. *Proc. Amer. Math. Soc. 7* (1956) 48–50.
4. Obruca, A. Algorithm 1. Mintree. *Computer Bull.* (Sept. 1964) 67.
5. Knuth, D. E. *The Art of Computer Programming, Vol I Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1968. pp. 370–371;

```
begin
  integer i, j, k, c, n, r;
  integer array V[1:v];
  c := n := 0;
  for k := 1 step 1 until v do V[k] := 0;
  for k := 1 step 1 until e do
  begin
    i := I[k];  j := J[k];
    if V[i] = 0 then
    begin
      T[k−n] := k;
      if V[j] = 0 then V[i] := V[j] := c := c + 1
      else
      V[i] := V[j]
    end
    else if V[j] = 0 then
    begin
      T[k−n] := k;  V[j] := V[i]
    end
    else if V[i] ≠ V[j] then
    begin
      T[k−n] := k;  i := V[i];  j := V[j];
      for r := 1 step 1 until v do
        if V[r] = j then V[r] := i
    end graft
    else n := n + 1
  end edge;
  p := v − e + n
end spanning tree
```

ALGORITHM 400
MODIFIED HAVIE INTEGRATION [D1]

George C. Wallick (Recd. 26 Jan. 1970 and 25 Apr. 1970)
Mobil Research and Development Corporation, Field Research Laboratory, P.O. Box 900, Dallas, TX 75221

KEY WORDS AND PHRASES: numerical integration, Havie integration, Romberg quadrature, modified Romberg-quadrature, trapezoid values, rectangle values
CR CATEGORIES: 5.16

Description:

The Havie integration method for the approximate evaluation of the definite integral

$$I = \int_A^B F(x)\, dx \tag{1}$$

as implemented in ACM Algorithm 257 [4] is based upon the parallel generation of the Romberg table of trapezoidal T_i^k values [1] and the table of rectangular R_i^k values also used by Krasun and Prager [3]. At each step in the development of the tables the difference $| T_i^k - R_i^k |$ is examined. If $| T_i^k - R_i^k | \le \epsilon$ the process is said to have converged and the algorithm returns a value of

$$T_i^{k+1} = \tfrac{1}{2}(T_i^k + R_i^k). \tag{2}$$

For some $F(X)$, e.g. $F(X) = e^{-X^2}$ and $F(X) = 2/(2+\sin 10\pi X)$, the R_i^k, T_i^k pairs converge more rapidly than the Romberg sequence of T_i^k values. (This is the same class of $F(X)$ for which a simple nonadaptive Simpsons Rule algorithm [5] is competitive with the Havie algorithm.) For other $F(X)$, the Havie algorithm is slightly less efficient than the Romberg algorithm.

Like Romberg quadrature, Havie integration requires the evaluation of the rectangular values

$$R_o^k = \frac{B-A}{2^k} \sum_{j=1}^{2^k} F\left[A + (j-\tfrac{1}{2})\frac{B-A}{2^k}\right]. \tag{3}$$

Rutishauser [6] recognized that this repeated addition of small terms to a large partial sum can lead to serious roundoff error. He suggested a procedure for the evaluation of the R_o^k which significantly reduces this error. The method, used by Fairweather [2] in a modified Romberg algorithm, leads to a significant improvement in accuracy for large orders of extrapolation.

In the modified Havie integration algorithm HRVINT the R_o^k are evaluated using a 3-level version of the Rutishauser procedure. The arguments X of the generating function $F(X)$ are evaluated as in eq. (3) rather than by accumulative addition as in Algorithm 257.

In the argument list for HRVINT, F is the name of the generating function FUNCTION $F(X)$ which returns a value of $F(X)$ corresponding to a specified value of X, A, and B represent the lower and upper limits of integration, and MAX is the maximum order of extrapolation to be permitted, MAX $\le$ 16. Values of MAX $>$ 16 are interpreted as MAX $=$ 16; the value of MAX is not changed by the subprogram. Computation is terminated when

$$| T_i^k - R_i^k | \le \text{ACC}*\, | T_i^k |$$

or when the order of extrapolation MFIN = MAX. Here ACC is a measure of the desired relative accuracy, ACC $>$ 0. Upon exit HRVINT is the approximate value of the integral, FAC is a measure of the final relative accuracy achieved

$$\text{FAC} = | T_i^k - R_i^k | / | T_i^k |$$

and MFIN is the order of extrapolation.

Test case. HRVINT was tested in Fortran IV on a CDC 6400 computer using single-precision floating point arithmetic (14+

TABLE I. A Comparison of the Havie and Modified Havie Algorithms

$$I = \int_A^B F(X)\, dX$$

(*m* = *Extrapolation Order, m* ≤ *16; N.S.F.* = *Number of Significant Figures*)

				Numerical Evaluation						
					Havie			*Modified Havie*		
$F(X)$	A	B	*Correct value (digits 10–16)*	*Specified relative accuracy*	*I (digits 10–14)*	*m*	*N.S.F.*	*I (digits 10–14)*	*m*	*N.S.F.*
e^{-x^2}	0.0	5.0	45139 55	10^{-1}–10^{-2}	46726	3	10	46726	3	10
				10^{-3}–10^{-10}	45039	4	11	45039	4	11
				10^{-11}	45110	5	12	45111	5	12
				10^{-12}	45128	6	12	45131	6	12
				10^{-13}	45134	6	12	45137	6	13
				10^{-14}	39757	16	9	45137	7	13
				10^{-15}	39757	16	9	45136	10	13
$\ln x$	1.0	10.0	29940 46	10^{-9}	29845	8	11	29846	8	11
				10^{-10}	29937	8	13	29939	8	13
				10^{-11}–10^{-12}	29937	9	13	29940	9	14
				10^{-13}	29937	9	13	29940	10	14
				10^{-14}	29556	16	11	29940	10	14
$(1+x)^{-1}$	0.0	1.0	55994 53	10^{-9}	56353	6	11	56354	6	11
				10^{-10}	55996	6	13	55997	6	13
				10^{-11}	55990	6	13	55991	6	13
				10^{-12}	55988	7	12	55991	7	13
				10^{-13}	55987	8	12	55991	7	13
				10^{-14}–10^{-15}	53242	16	10	55991	9	13
$(1+x^4)^{-1}$	0.0	1.0	33991 10	10^{-6}–10^{-7}	35633	5	10	35634	5	10
				10^{-8}–10^{-10}	33993	6	13	33995	6	13
				10^{-11}–10^{-12}	33984	7	12	33989	7	13
				10^{-13}	30854	16	10	33987	7	13
				10^{-14}–10^{-15}	30854	16	10	33988	9	13
x^{-3}	0.01	1.1	68595 04	10^{-8}	71022	13	10	71529	13	10
				10^{-9}	68136	13	11	68647	13	11
				10^{-10}	68076	13	10	68589	13	12
				10^{-11}	64508	16	10	68590	14	12
				10^{-12}–10^{-13}	64508	16	10	68589	14	12
				10^{-14}–10^{-15}	64508	16	10	68584	16	12
x^{-4}	0.01	1.1	89506 64	10^{-8}	89368	13	11	89694	13	11
				10^{-9}	89199	13	11	89526	13	12
				10^{-10}	88857	14	10	89503	14	13
				10^{-11}–10^{-12}	86878	16	10	89502	14	13
				10^{-13}	86878	16	10	89502	15	13
				10^{-14}–10^{-15}	86878	16	10	89499	16	12
x^{-5}	0.01	1.1	29246 64	10^{-8}	29556	13	11	29767	13	10
				10^{-9}–10^{-10}	28828	14	11	29247	13	14
				10^{-11}	27557	16	10	29245	14	13
				10^{-12}–10^{-13}	27557	16	10	29244	15	13
				10^{-14}	27557	16	10	29244	16	13
				10^{-15}	27557	16	10	29242	16	13

decimal digits). Corresponding integral values were also obtained using a Fortran version of the standard Havie Algorithm 257. The results of these tests are summarized in Table I.

For modest accuracy requirements, the two algorithms are seen to be equivalent. For both algorithms the maximum accuracy achievable is limited by truncation and roundoff error. Since the Rutishauser modification serves to reduce the magnitude of such errors, the modified Havie algorithm can, in many cases, return optimum integral values that are from 1 to 2 significant figures more accurate than those returned by Algorithm 257.

In the routine use of the algorithms it is possible to specify an accuracy requirement that cannot be satisfied. When this condition obtains, the algorithms are forced to proceed to the maximum permitted extrapolation order. With Algorithm 257 error accumulation accompanying such an overspecification can lead to a serious decline in evaluation accuracy. With the modified Havie algorithm HRVINT this loss is minimized and in most cases virtually eliminated.

Acknowledgment. The author wishes to thank Mobil Research and Development Corporation for permission to publish this information.

REFERENCES:

1. BAUER, F. L. Algorithm 60, Romberg integration. *Comm. ACM 4* (June 1961), 255.
2. FAIRWEATHER, G. Algorithm 351, Modified Romberg quadrature. *Comm. ACM 12* (June 1969), 324-325.
3. KRASUN, A. M., AND PRAGER, W. Remark on Romberg quadrature. *Comm. ACM 8* (Apr. 1965), 236-237.
4. KUBIK, R. N. Algorithm 257, Havie integrator. *Comm. ACM 8* (June 1965), 381.
5. PERLIS, A. J., AND SAMELSON, K. Preliminary report—international algebraic language. *Comm. ACM 1* (Dec. 1958), 8-22.
6. RUTISHAUSER, H. Description of Algol 60. In *Handbook for Automatic Computation, Vol. 1.* Springer-Verlag, New York, 1967, Part a, pp. 105-106.

ALGORITHM:

```
      FUNCTION HRVINT(F,A,B,MAX,ACC,FAC,MFIN)
C HAVIE INTEGRATION WITH AN EXPANDED RUTISHAUSER-
C TYPE SUMMATION PROCEDURE
      DIMENSION T(17),U(17),TPREV(17),UPREV(17)
C TEST FOR MAX GREATER THAN 16
      MUX=MAX
      IF(MAX-16)10,10,5
    5 MUX=16
C INITIALIZATION
   10 ENPT=0.5*(F(A)+F(B))
      SUMT=0.0
      MFIN=1
      N=1
      H=B-A
      SH=H
C BEGIN REPETITIVE LOOP FROM ORDER 1 TO ORDER MAX
   15 T(1)=H*(ENPT+SUMT)
      SUM=0.
      NN=N+N
      FN=NN
      EM=SH/FN
C BEGIN RUTISHAUSER EVALUATION OF RECTANGULAR SUMS
C INITIALIZATION
      IF(NN-16)20,20,25
   20 NZ=NN
      GO TO 30
   25 NZ=16
      IF(NN-256)30,30,35
   30 NA=NN
      GO TO 40
   35 NA=256
      IF(NN-4096)40,40.45
   40 NB=NN
      GO TO 50
   45 NB=4096
C DEVELOPMENT OF RECTANGULAR SUMS
   50 DO 70 KC=1,NN,4096
         SUMB=0.
         KK=KC+NB-1
         DO 65 KB=KC,KK,256
            SUMA=0.
            KKK=KB+NA-1
            DO 60 KA=KB,KKK,16
               SUMZ=0.
               KFR=KA+NZ-1
               DO 55 KZ=KA,KFR,2
                  ZKZ=KZ
   55          SUMZ=SUMZ+F(A+ZKZ*EM)
   60       SUMA=SUMZ+SUMA
   65    SUMB=SUMA+SUMB
   70 SUM=SUMB+SUM
C END OF RUTISHAUSER PROCEDURE
      U(1)=H*SUM
      K=1
C BEGIN EXTRAPOLATION LOOP
   75 FAC=ABS(T(K)-U(K))
      IF(T(K))80,85,80
C TEST FOR RELATIVE ACCURACY
   80 IF(FAC-ABS(ACC*T(K)))90,90,100
C TEST FOR ABSOLUTE ACCURACY WHEN T(K)=0
   85 IF(FAC-ABS(ACC))95,95,100
   90 FAC=FAC/ABS(T(K))
C INTEGRAL EVALUATION BEFORE EXIT
   95 HRVINT=0.5*(T(K)+U(K))
      RETURN
  100 IF(K-MFIN)105,115,115
  105 AK=K+K
      D=2.**AK
      DMA=D-1.0
C BEGIN EXTRAPOLATION
      T(K+1)=(D*T(K)-TPREV(K))/DMA
      TPREV(K)=T(K)
      U(K+1)=(D*U(K)-UPREV(K))/DMA
      UPREV(K)=U(K)
C END EXTRAPOLATION
      K=K+1
      IF(K-MUX)75,110,110
C END EXTRAPOLATION LOOP
  110 FAC=ABS(T(K)-U(K))
      IF(T(K))90,95,90
C ORDER IS INCREASED BY ONE
  115 H=0.5*H
      SUMT=SUMT+SUM
      TPREV(K)=T(K)
      UPREV(K)=U(K)
      MFIN=MFIN+1
      N=NN
      GO TO 15
C RETURN FOR NEXT ORDER EXTRAPOLATION
      END
```

Remark on Algorithm 400 [D1]

Modified Håvie Integration

[George C. Wallick, *Comm. ACM 13* (Oct. 1970), 622-624]

Robert Piessens [Recd. 17 Apr. 1973]
Applied Mathematics and Programming Division, University of Leuven, B-3030 Heverlee, Belgium

Recently, Casaletto et al. [1] tested a number of automatic integrators by calculating 50 test integrals with different specified tolerances. We shall refer to these integrals as #1, #2, . . . , #50. (A list can be found in [1] or [2].) One of the aims of their tests was to give a summary of the number of failures (when the computed value was not within the requested tolerance) and overflows (when an upper bound on the number of integrand evaluations prevented the specified accuracy from being reached) of each integrator. We have examined some other recently published integrators in a similar way. Our study reveals that *HRVINT* fails more frequently than the other integrators. For example, for the specified relative accuracy $ACC = 10^{-3}$, *HVRINT* fails on #26, #31, #34, #45, and #47, and for $ACC = 10^{-4}$, on #20, #26, #31, #32, #34, #45, and #47. It is worth while to note that #20 and #32 are integrals with very smooth integrand.

Most failures can be avoided by changing the statement labeled 75 to

75 IF (MFIN−2) 100, 100, 76
76 FAC = ABS (T (K)−U(K))

Indeed, with this alteration failures occur only on #47 (for both accuracies $ACC = 10^{-3}$ and 10^{-4}).

References

1. Casaletto, J., Pickett, M., and Rice, J. A comparison of some numerical integration programs. SIGNUM Newsletter 4, 3(1969), 30-40.
2. Gentleman, W.A. Implementing Clenshaw-Curtis quadrature, I. Methodology and experience. *Comm. ACM 15* (May 1972), 337-342.

ALGORITHM 401
AN IMPROVED ALGORITHM TO PRODUCE COMPLEX PRIMES [A1]

Paul Bratley (Recd. 25 Feb. 1970)
Département d'informatique, Université de Montréal,
C.P. 6128, Montréal 101, Quebec, Canada

KEY WORDS AND PHRASES: number theory, prime numbers, complex numbers
CR CATEGORIES: 5.39

```
integer procedure cprimes(m, PR, PI);
  value m; integer m; integer array PR, PI;
comment The procedure generates the complex prime numbers
  located in the one-eighth plane defined by 0 ≤ y < x. Any prime
  found in that area has seven more associated primes: −x + yi,
  ±x − yi, ±y ± xi. These associated primes must be generated
  externally to cprimes. The first complex prime generated by
  cprimes is 1 + i, which exceptionally lies on x = y and has only
  three associated primes.
    The algorithm generates a list of complex primes in order of
  increasing modulus: the parameter m of the call is the highest
  modulus to be included in the list and should satisfy m > 2.
  PR and PI will contain respectively the real and imaginary
  parts of the generated list, with PR ≥ PI ≥ 0 for each prime.
  The value of the procedure is the number of primes generated.
    Algorithm 311 [1], sieve 2, is used to generate the rational
  primes less than m². Then it is known (see, for instance [2])
  that a rational prime p of the form p = 4n + 1 can be expressed
  as p = a² + b², and factorized as (a+bi)(a−bi) in the complex
  plane, where a + bi and a − bi are complex primes. For our
  present purpose we choose a > b and include only a + bi in the
  list. A rational prime p of the form p = 4n + 3 remains prime
  in the complex plane, so we include p + 0i in the list if p < m.
  Finally, the complex prime 1 + i may be thqught of as one of
  the factors of the remaining rational prime 2 = (1+i)(1−i).
    Although this algorithm and Algorithm 372 [3] are not directly
  comparable, since they produce the list of complex primes in a
  different order, the accompanying remark suggests that the
  present algorithm is often to be preferred.
    References:
  1. Chartres, B. A. Algorithm 311, Prime number generator 2.
     Comm. ACM 10 (Sept. 1967), 570.
  2. Hardy, G. H., and E. M. Wright. An Introduction to the
     Theory of Numbers, 4th ed. Clarendon Press, Oxford, 1965,
     Chs XII and XV.
  3. Dunham, K. B. Algorithm 372, An Algorithm to produce
     complex primes, CSIEVE. Comm. ACM 13 (Jan. 1970),
     52-53;
begin
  integer a, b, c, d, e, i, j, p, q;
  integer array P2[1:0.7×m↑2/ln(m)],
    P3[1:1.4×m/ln(m)];
  e := sieve 2(m↑2, P2);
  PR[1] := PI[1] := a := c := 1;
  b := 0;
  for d := 2 step 1 until e do
  begin
    p := P2[d]; q := p − 1;
    if (q÷4) × 4 ≠ q then
    begin
      if p ≤ m then
      begin b := b + 1; P3[b] := p end
    end
    else
    begin
L1:
      if a ≤ b then
      begin
        if P3[a] ↑ 2 < p then
        begin
          c := c + 1; PR[c] := P3[a];
          a := a + 1; PI[c] := 0;
          go to L1
        end
      end;
      q := entier(sqrt(p/2)+1);
      for i := q step 1 until p do
      begin
        j := sqrt(p−i↑2);
        if i ↑ 2 + j ↑ 2 = p then go to L2
      end
      comment Note that the jump to L2 is always made before
        the cycle is terminated;
L2:
      c := c + 1; PR[c] := i; PI[c] := j
    end
  end;
L3:
  if a ≤ b then
  begin
    c := c + 1; PR[c] := P3[a];
    a := a + 1; PI[c] := 0;
    go to L3
  end;
  cprimes := c
end cprimes
```

REMARKS ON ALGORITHM 372 [A1]
AN ALGORITHM TO PRODUCE COMPLEX PRIMES, CSIEVE [K. B. Dunham. *Comm. ACM 13* (Jan. 1970), 52–53]
ALGORITHM 401 [A1]
AN IMPROVED ALGORITHM TO PRODUCE COMPLEX PRIMES [P. Bratley. *Comm. ACM 13* (Nov. 1970), 693]

Paul Bratley (Recd. 25 Feb. 1970)
Département d'informatiqué, Universite de Montréal,
C.P. 6128, Montréal 101, Quebec, Canada

KEY WORDS AND PHRASES: number theory, prime numbers, complex numbers
CR CATEGORIES: 5.39

Algorithm 372 was run on the CDC 6400 at the University of Montreal. The variable *i* is undefined if the **for**-loop at label *A* is completed. The statement

$$i := j + 1;$$

should be added immediately before label B. Algol purists may also care to remove redundant semicolons after **go to** A and **go to** B, and the redundant parentheses in one **if**-statement. With these changes the algorithm produced correct results for several values of m.

The comment in Algorithm 372 is slightly inaccurate. The first prime generated by the algorithm is $1 + i$, which does not have $PR > PI$, and which has not seven but three associated primes.

It is not possible to compare the speeds of Algorithm 372 and Algorithm 401 directly since they generate primes in a different order. However, the following test was run. A value of m was chosen, and Algorithm 401 was used to list all the complex primes with modulus less than m. The time taken and the number of primes produced were noted. Then Algorithm 372 was used to produce an equal number of primes, the time taken again being noted. Times observed are shown in Table I.

TABLE I

Limit on modulus	*Algorithm 401 produced this number of primes*	*Time taken (secs)*	*Time taken by Algorithm 372 to produce the same number of primes (secs)*	*Ratio of times taken*
25	60	0.278	0.331	1.2
50	189	1.577	2.140	1.4
75	373	4.217	7.602	1.8
100	623	8.618	20.214	2.4
150	1266	23.732	79.481	3.4

The conclusion from the figures in Table I is that if the speed with which the complex primes are generated is of paramount importance then Algorithm 401 should be preferred to Algorithm 372.

As written Algorithm 401 will use more memory than Algorithm 372 since it is convenient and perspicuous to use *sieve*2 in an unmodified form, which makes it necessary to store temporarily all the rational primes less than m^2. However, if space is tight then *sieve*2 can easily be modified so as to generate rational primes one at a time on successive calls, and in this way the use of the long array $P2$ can be avoided. If this modification is made Algorithm 401 will in fact use less store than Algorithm 372, which wastefully stores many useless values in PM. It is also to be noticed that the factors 0.7 and 1.4 occurring in the declarations of $P2$ and $P3$ may be diminished for large m: all that is necessary is that $P2$ should be long enough to hold the rational primes less than m^2, and that $P3$ should be long enough to hold the rational primes which are not greater than m and which are of the form $4n + 3$. Some space may be saved similarly in *sieve*2, which is called from Algorithm 401.

ALGORITHM 402 INCREASING THE EFFICIENCY OF QUICKSORT* [M1]

M. H. van Emden (Recd. 15 Dec. 1969 and 7 July 1970)
Mathematical Centre, Amsterdam, The Netherlands

* The algorithm is related to a paper with the same title and by the same author, which was published in *Comm. ACM 13* (Sept. 1970), 563–567.

KEY WORDS AND PHRASES: sorting, quicksort
CR CATEGORIES: 5.31, 3.73, 5.6, 4.49

```
procedure qsort(a, l1, ul);
  value l1, ul;  integer l1, ul;  array a;
comment  This procedure sorts the elements a[l1], a[l1+1], ··· ,
  a[ul] into nondescending order. It is based on the idea described
  in [1]. A comparison of this procedure with another procedure,
  called sortvec, obtained by combining C. A. R. Hoare's quicksort
  [2] and R. S. Scowen's quickersort [3], in such a way as to be
  optimal for the Algol 60 system in use on the Electrologica X-8
  computer at the Mathematical Centre is shown below. Here
  "repetitions" denotes the number of times the sorting of a
  sequence of that "length" is repeated; "average time" is the
  time in seconds averaged over the repetitions; "gain" is the
  difference in time relative to time taken by sortvec.
```

procedure	*length*	*repetitions*	*average time*	*gain*
sortvec	30	23	.09	
qsort	30	23	.06	+.37
sortvec	300	16	1.25	
qsort	300	16	1.03	+.17
sortvec	3000	9	17.43	
qsort	3000	9	15.25	+.13
sortvec	30000	2	232.46	
qsort	30000	2	197.96	+.15

References:

1. van Emden, M. H. Increasing the efficiency of quicksort. *Comm. ACM 13* (Sept. 1970), 563–567.
2. Hoare, C. A. R. Algorithm 64, quicksort. *Comm. ACM 4* (July 1961), 321–322.
3. Scowen, R. S. Algorithm 271, quickersort. *Comm. ACM 8* (Nov. 1965), 669;

```
begin
  integer p, q, ix, iz;
  real x, xx, y, zz, z;
  procedure sort;
  begin
    integer l, u;
    l := l1;  u := ul;
part:
    p := l;  q := u;  x := a[p];  z := a[q];
    if x > z then
    begin y := x;  a[p] := x := z;  a[q] := z := y end;
    if u − l > 1 then
    begin
      xx := x;  ix := p;  zz := z;  iz := q;
left:
      for p := p + 1 while p < q do
      begin
        x := a[p];
        if x ≥ xx then go to right
      end;
      p := q − 1; go to out;
right:
      for q := q − 1 while q > p do
      begin
        z := a[q];
        if z ≤ zz then go to dist
      end;
      q := p;  p := p − 1;  z := x;  x := a[p];
dist:
      if x > z then
      begin
        y := x;  a[p] := x := z;
        a[q] := z := y
      end;
      if x > xx then
      begin xx := x;  ix := p end;
      if z < zz then
      begin zz := z;  iz := q end;
      go to left;
out:
      if p ≠ ix ∧ x ≠ xx then
      begin a[p] := xx;  a[ix] := x end;
      if q ≠ iz ∧ z ≠ zz then
      begin a[q] := zz;  a[iz] := z end;
      if u − q > p − l then
      begin l1 := l;  ul := p − 1;  l := q + 1 end
      else
      begin ul := u;  l1 := q + 1;  u := p − 1 end;
      if ul > l1 then sort;
      if u > l then go to part
    end
  end of sort;
  if ul > l1 then sort
end of qsort
```

Remark on Algorithm 402 [M1]

Increasing the Efficiency of Quicksort [M.H. Van Emden, *Comm. ACM 13* (Nov. 1970), 693–694]

Robert E. Wheeler [Recd. 6 July 1971]
E.I. du Pont de Nemours and Company,
Wilmington, DE 19899

It will happen during execution of this algorithm that sequences will be encountered which are already in nondescending order and which should not be further sorted. Changes to the algorithm which accomplish this are indicated below. For a Fortran version of this algorithm running on a Univac 1108, these changes decreased running time by 1.25 percent when sorting random arrays of length 500 and by 2.7 percent when sorting random arrays of length 50.

Line	*Change to:*
2	**integer** p, q, ix, iz, i, j;
9	$p := l$; $q := u$; $x := a[p]$; $z := a[q]$; $i := 0$; $j := q - p - 1$;
36	**begin** $xx := x$; $i := i + 1$; $ix := p$ **end**;
38	**begin** $zz := z$; $i := i + 1$; $iz := q$ **end**;
48.5	**if** $i \neq j$ **begin**
50.5	**end**;

ALGORITHM 403
CIRCULAR INTEGER PARTITIONING [A1]

M. W. Coleman and M. S. Taylor (Recd. 30 June 1970)
Aberdeen Proving Ground, MD 21005

KEYWORDS AND PHRASES: partitions, combinatorics, statistical design of experiments
CR CATEGORIES: 5.39, 5.5

Description:

The partition, when expressed as a K-tuple $(X_1, \cdots, X_K)$, may be thought of as a K-digit number in the base V number system. The procedure *CIRPI* then functions as a counter which generates successive K-digit numbers in the base V number system. However, since all K-digit numbers do not correspond to circular partitions, it is possible to have the procedure generate only a subset of K-tuples for consideration, using the following criteria:

(a) The digits are constrained to sum to V, consequently, the K digits are not independent. Thus the procedure need only operate on the $K - 1$ most significant digits, the least significant digit being an easily computable function of the other $K - 1$ digits.

(b) Since the numbers are sequentially increasing, a given number is a cyclic permutation of a previously generated number if a cyclic rotation of its digits produces a number with a smaller value. Thus the most significant digit, X_1, provides an effective minimum value for any of the digits.

(c) Given that the digits must sum to V and the minimum value for any digit is X_1, the value $V - X_1 * (K - 1)$ provides an effective maximum for any digit.

(d) Since the maximum and minimum values depend on the most significant digit, X_1, the procedure is finished when X_1 has increased to the point where the minimum digit size exceeds the maximum digit size, i.e. when $X_1 > V - X_1 * (K - 1)$. This easily reduces to $X_1 > V/K$, providing an easy method for terminating the K-tuple generation as early as possible.

Therefore, the procedure efficiently generates the totality of circular partitions since it can greatly restrict the number of K-tuples that must be considered.

References:

1. David, H. A., and F. W. Wolock. Cyclic designs. *Annals of Math. Stat. 36* (1965), 1526–1534.
2. Niven, I., *Mathematics of Choice.* Random House, New York, 1965, ch. 6.

Algorithm:

```
      SUBROUTINE CIRPI (V, K, X)
C
C   THIS SUBROUTINE GENERATES ALL K-TUPLES SUCH THAT.....
C A) THE SUM OF THE K ELEMENTS OF THF K-TUPLE IS V,
C B) EACH OF THE ELEMENTS IS AN INTEGER GREATER THAN 0, AND
C C) NO K-TUPLE IS A CYCLIC PFRMUTATION OF ANY OTHER K-TUPLE.
C THE K-TUPLE IS STORED IN THE ARRAY X, WITH ONE ELEMENT
C PER ARRAY ELEMENT.  EACH K-TUPLE IS PROCESSED BY THE USER
C (USING THE SUBROUTINE 'PROCES') BEFORE THE NEXT K-TUPLE IS
C GENERATED.  THE SUBROUTINF 'PROCES' MUST NOT CHANGE THE
C CONTENTS OF THE ARRAY X.
C
      INTEGER X(K), V, V1, V2, C, SUM
        V1 = V-K+1
        V2 = V/K
        K1 = K-1
        K2 = K-2
        SUM = K1
C
C   INITIALIZE THE ARRAY X WITH THE FIRST K-TUPLE.
C
      DO 100 I = 1, K1
        X(I) = 1
  100 CONTINUE
        GO TO 115
C
C   GENERATE THE NEXT K-TUPLE WHICH SATISFIES THE GIVEN
C CONDITIONS, A) - C).
C
  110   C = 1
        SUM = X(1)
      DO 113 I = 1, K2
        I1 = K-I
        X(I1) = X(I1)+C
        IF (X(I1) .LT. V1) GO TO 111
          X(I1) = X(1)
          GO TO 112
  111   C = 0
  112   SUM = SUM+X(I1)
  113 CONTINUE
        IF (C .EQ. 0) GO TO 115
          X(1) = X(1)+1
          IF (X(1) .GT. V2) RETURN
      DO 114 I1 = 2, K1
        X(I1) = X(1)
  114 CONTINUE
        SUM = X(1)*K1
        V1 = V-SUM
  115   SUM = V-SUM
        IF (SUM .LT. X(1)) GO TO 110
          X(K) = SUM
C
C   CHECK TO SEE IF THF K-TUPLE IS A CYCLIC PERMUTATION OF
C ANY PREVIOUSLY GENERATED K-TUPLES.  IF IT IS, GENERATE THE
C NEXT CANDIDATE, OTHERWISE, CALL THE SUBROUTINE 'PROCES' TO
C PROCFSS THE K-TUPLE BEFORE GFNERATING THE NEXT ONE.
C
  120 DO 122 I = 2, K
        IF (X(I) .GT. X(1)) GO TO 122
        IF (X(I) .LT. X(1)) GO TO 110
        I1 = I+1
        DO 121 I2 = 2, K
          IF (I1 .GT. K) I1 = I1-K
          IF (X(I1) .GT. X(I2)) GO TO 122
          IF (X(I1) .LT. X(I2)) GO TO 110
          I1 = I1+1
  121   CONTINUE
        GO TO 130
  122 CONTINUE
  130 CALL PROCES (X, K)
        GO TO 110
      END
```

ALGORITHM 404
COMPLEX GAMMA FUNCTION [S14]

C. W. LUCAS JR.* AND C. W. TERRILL (Recd. 13 Feb. 1970 and 19 June 1970)

Physics Department, College of William and Mary in Virginia, Williamsburg, VA 23185

*William and Mary Predoctoral Fellow. This work was partly supported by the National Aeronautics and Space Agency, Contract NGL 47-006008.

KEY WORDS AND PHRASES: gamma function, poles of gamma function, Stirling's asymptotic series, recursion formula, reflection formula
CR CATEGORIES: 5.12

DESCRIPTION:

CGAMMA evaluates in single precision the gamma function for complex arguments. The method of evaluation is similar to the one employed by A. M. S. Filho and G. Schwachheim in evaluating the gamma function with arbitrary precision for real arguments [1]. First the real part of the argument of the gamma function is increased by some integer M, if necessary, so that Stirling's asymptotic series for the logarithm of the gamma function may be used with high precision and a small number of terms. Then the recursion formula for the gamma function

$$\Gamma(Z) = \Gamma(Z+1)/Z$$

is used to step down to the original gamma function.

The conditions on the value of $T = Z + M$ used in Stirling's asymptotic series are:

1. $Real(T) > 10$
2. $Arg(T) = arctan(Imaginary(T)/Real(T)) \leq \pi/4$

This second condition ensures that the error incurred in using Stirling's asymptotic series with a finite number of terms is less than the value of the next term in the series [2].

The only condition on the argument Z is that it must not be too close to a pole of the gamma function, i.e. $Z = 0, -1, -2, \cdots$. A rough empirical relation was found between the number of significant figures obtained by Stirling's asymptotic series and the distance δ in the complex plane from Z to the nearest pole by approaching the poles at 0 and -1 from several directions. If $\delta = 10^{-n}$ (n an integer ≥ 3) this relation is $(minimum\ number\ of\ significant\ figures) = 7 - n$. With $\delta = 10^{-4}$, for instance, Stirling's asymptotic series gives three or more significant figures depending on the direction of Z from the pole. The upper limit on the size of Z for which *CGAMMA* will work is a function of the computer system. For the IBM 360 system where the largest size number that can be handled is about 10^{75} the upper limit for real Z is about ± 57, for Z on the line $Imaginary\ (Z) = \pm Real(Z)$ it is $(63 \pm 63i)$, for $Real(Z) > 0$ and $(-32 \pm 32i)$ for $Real(Z) < 0$, and for Z on the imaginary axis it is $\pm 107i$.

CGAMMA has been tested in several ways. The reflection formula

$$\Gamma(Z)\Gamma(1-Z) = \frac{\pi}{sin(\pi Z)}$$

and the relation

$$\Gamma(n+1) = n! \quad (n \text{ integer})$$

have been employed as checks. Also $log(gamma(Z))$ has been compared with tabulated valued in reference [2] for a number of values of Z. These tests lead us to conclude that *CGAMMA* gives four to five significant figures for Z outside disks of radius $\delta = 10^{-3}$ centered on the poles. If the subroutine is written in double precision, we have found that about eight more significant figures will be obtained everywhere for an IBM 360 system, and near the poles

$$(minimum\ number\ of\ significant\ figures) = 15 - n$$

where $\delta = 10^{-n}$. The range of the subroutine remains the same.

Acknowledgment. The authors wish to express their gratitude to Dr. Morton Eckhause of the William and Mary Physics Department for his continuing advice and to Shirley McCallum and Samuel Pettus Hoyle of the College of William and Mary Computer Center who frequently reduced the programming difficulties.

REFERENCES:

1. FILHO, ANTONINA MACHADO SOUZA AND SCHWACHHEIM, GEORGES. Algorithm 309, Gamma function of arbitrary precision. *Comm. ACM 10* (Aug. 1967), 511.
2. US Dep. of Commerce, Amer. Nat. Stand. Inst. Table of the gamma function for complex arguments. Clearinghouse, Springfield, VA 22151 (1954), p. VIII.

ALGORITHM:

[*Warning.* System dependent constants are used in assigning values to IOUT, PI, TOL, SUM—L.D.F.]

```
      FUNCTION CGAMMA(Z)
      COMPLEX Z,ZM,T,TT,SUM,TERM,DEN,CGAMMA,PI,A
      DIMENSION C(12)
      LOGICAL REFLEK
C SET IOUT FOR PROPER OUTPUT CHANNEL OF COMPUTER SYSTEM FOR
C ERROR MESSAGES
      IOUT = 3
      PI = (3.141593,0.0)
      X = REAL(Z)
      Y = AIMAG(Z)
C TOL = LIMIT OF PRECISION OF COMPUTER SYSTEM IN SINGLE PRECISION
      TOL = 1.0E-7
      REFLEK = .TRUE.
C DETERMINE WHETHER Z IS TOO CLOSE TO A POLE
C CHECK WHETHER TOO CLOSE TO ORIGIN
      IF(X.GE.TOL) GO TO 20
C FIND THE NEAREST POLE AND COMPUTE DISTANCE TO IT
      XDIST = X-INT(X-.5)
      ZM = CMPLX(XDIST,Y)
      IF(CABS(ZM).GE.TOL) GO TO 10
C IF Z IS TOO CLOSE TO A POLE, PRINT ERROR MESSAGE AND RETURN
C WITH CGAMMA = (1.E7,0.0E0)
      WRITE(IOUT,900) Z
      CGAMMA = (1.E7,0.E0)
      RETURN
C FOR REAL(Z) NEGATIVE EMPLOY THE REFLECTION FORMULA
C                GAMMA(Z) = PI/(SIN(PI*Z)*GAMMA(1-Z))
C AND COMPUTE GAMMA(1-Z).  NOTE REFLEK IS A TAG TO INDICATE THAT
C THIS RELATION MUST BE USED LATER.
10    IF(X.GE.0.0) GO TO 20
      REFLEK = .FALSE.
      Z = (1.0,0.0)-Z
      X = 1.0-X
      Y = -Y
C IF Z IS NOT TOO CLOSE TO A POLE, MAKE REAL(Z)>10 AND ARG(Z)<PI/4
20    M = 0
40    IF(X.GE.10.) GO TO 50
      X = X + 1.0
      M = M + 1
      GO TO 40
50    IF(ABS(Y).LT.X) GO TO 60
      X = X + 1.0
      M = M + 1
      GO TO 50
60    T = CMPLX(X,Y)
      TT = T*T
      DEN = T
C COEFFICIENTS IN STIRLING'S APPROXIMATION FOR LN(GAMMA(T))
      C(1) = 1./12.
      C(2) = -1./360.
      C(3) = 1./1260.
      C(4) = -1./1680.
      C(5) = 1./1188.
      C(6) = -691./360360.
      C(7) = 1./156.
      C(8) = -3617./122400.
      C(9) = 43867./244188.
      C(10) = -174611./125400.
      C(11) = 77683./5796.
      SUM = (T-(.5,0.0))*CLOG(T)-T+CMPLX(.5*ALOG(2.*3.14159),0.0)
      J = 1
70    TERM = C(J)/DEN
C TEST REAL AND IMAGINARY PARTS OF LN(GAMMA(Z)) SEPARATELY FOR
C CONVERGENCE.  IF Z IS REAL SKIP IMAGINARY PART OF CHECK.
      IF(ABS(REAL(TERM)/REAL(SUM)).GE.TOL) GO TO 80
      IF(Y.EQ.0.0) GO TO 100
      IF(ABS(AIMAG(TERM)/AIMAG(SUM)).LT.TOL) GO TO 100
80    SUM = SUM + TERM
      J = J + 1
      DEN = DEN*TT
C TEST FOR NONCONVERGENCE
      IF(J.EQ.12) GO TO 90
      GO TO 70
C STIRLING'S SERIES DID NOT CONVERGE.  PRINT ERROR MESSAGE AND
C PROCEDE.
90    WRITE(IOUT,910) Z
C RECURSION RELATION USED TO OBTAIN LN(GAMMA(Z))
C              LN(GAMMA(Z)) = LN(GAMMA(Z+M)/(Z*(Z+1)*...*(Z+M-1)))
C                           = LN(GAMMA(Z+M)-LN(Z)-LN(Z+1)-...-LN(Z+M-1)
100   IF(M.EQ.0) GO TO 120
      DO 110 I = 1,M
      A = CMPLX(I*1.-1.,0.0)
110   SUM = SUM-CLOG(Z+A)
C CHECK TO SEE IF REFLECTION FORMULA SHOULD BE USED
120   IF(REFLEK) GO TO 130
      SUM = CLOG(PI/CSIN(PI*Z))-SUM
      Z = (1.0,0.0) -Z
130   CGAMMA = CEXP(SUM)
      RETURN
900   FORMAT(1X,2F14.7,10X,49HARGUMENT OF GAMMA FUNCTION IS TOO CLOSE TO
     1 A POLE)
910   FORMAT(44H ERROR - STIRLING'S SERIES HAS NOT CONVERGED/14X,4HZ = ,
     12F14.7)
      END
```

Certification and Remark on Algorithm 404 [S14]

Complex Gamma Function [C.W. Lucas Jr. and C.W. Terril, *Comm. ACM 14* (Jan. 1971), 48]

G. Andrejková and J. Vinař, Computing Center, Šafarik University, Košice, Czechoslovakia

The following changes were made in the algorithm:

a. The function subroutine heading was changed to read

COMPLEX FUNCTION CGAMMA(Z)

in accordance with the standard.

b. The convergence tests following statement number 70 involve the computation of the quantity *REAL(TERM)/REAL(SUM)*. This can lead to overflow if *Z* is real and near to a pole. For these reasons the two statements were replaced by

*IF (ABS(REAL(TERM)) .GE. TOL*ABS(REAL(SUM))) GO TO* 80

and

*IF (ABS(AIMAG(TERM)) .GE. TOL*ABS(AIMAG(SUM))) GO TO* 100

c. For similar reasons the statement

*SUM = CLOG(PI/CSIN(PI*Z))−SUM*

was replaced by

*SUM = CLOG(PI)−CLOG(CSIN(PI*Z))−SUM*

With these modifications the algorithm was translated on MINSK 22M using the FEL Fortran compiler (with seven significant digits in single precision and 15 in double precision) and ran satisfactorily.

The following tests were performed:

a. The logarithms of *CGAMMA(Z)* for $z = x+iy$ with $x = 1.0(0.1)10.0$ and $y = 0.0(0.1)3.0$ were checked against the values given in [1]. An overall accuracy of five to six digits was observed. The imaginary part frequently had one more accurate digit than the real part.

b. The behavior in the vicinity of poles was tested by computing the values of *CGAMMA(Z)* in eight evenly spaced points on circles of decreasing diameter. The value of $1.E-7$ for the minimum diameter was found adequate.

c. The values of *CGAMMA(Z)* were computed for $z = x+iy$ with

1. $x = 0.0(1.0)23.0$, $y = 0.0$
2. $x = 0.0$, $y = 0.0(1.0)26.0$
3. $x = y = 0.0(1.0)25.0$
4. $x = -y = 0.0(1.0)25.0$
5. $-x = y = 0.0(1.0)12.0$
6. $-x = -y = 0.0(1.0)12.0$

in all cases the final value is the last for which the program did not run into overflow or, in the last two cases, try to take a logarithm of too small a number.

References

1. Table of gamma function for complex arguments. National Bureau of Standards, Applied Math. Series 34, August 1954.

Algorithm 405 Roots of Matrix Pencils: The Generalized Eigenvalue Problem [F2]

Alice M. Dell, Roman L. Weil, Gerald L. Thompson*

(Recd. 25 May 1970 and 12 Oct. 1970)

Committee on Information Sciences, University of Chicago, Chicago, IL 60637, Graduate School of Business and Committee on Information Sciences, University of Chicago, Chicago, IL 60637, and Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA 15213

* This work was supported by the US National Aeronautics and Space Administration, by the National Science Foundation under Grant GS-2703 to the University of Chicago, and by the US Office of Naval Research under Grant NONR 760(24) NR 047-048 to Carnegie-Mellon University. Computations were done on the University of Chicago's Maniac III computer and were supported by the US Atomic Energy Commission under grants AT (11-1)-614 and AT (11-1)-2094.

KEY WORDS AND PHRASES: eigenvalues, matrix roots, pencil roots
CR CATEGORIES: 5.1, 5.3

procedure *PENCIL*(*A*, *B*, *m*, *n*, *LAMDA*, *Sp*, *Par*, *Tol*);
  value *A*, *B*; **real array** *A*, *B*, *LAMDA*; **integer** *m*,*n*, *Sp*, *Par*; **real** *Tol*;

comment *PENCIL* finds the generalized eigenvalues *LAMDA* which solve $x(A-\lambda B) = 0$ and $(A-\lambda B)y = 0$ and simultaneously reduce the rank of $(A-\lambda B)$, where A and B are m by n matrices, see [1, 3, 4]. *PENCIL* converts the m by n problem of finding the rank-reducing numbers of $(A-\lambda B)$ into an ordinary r by r eigenvalue problem by a sequence of elementary transformations. The theory is developed in [3] and [4]. These techniques are to be thought of as a combinatorial solution to an unsolved general problem. Our techniques may be numerically unsound for ill-conditioned problems. There are at most $k = min(m, n)$ such generalized eigenvalues. *Sp* is the number of generalized eigenvalues found. The real parts of the roots are stored in *LAMDA*(1, *j*), and the imaginary parts in *LAMDA*(2, *j*), $j = 1 \dots Sp$. *LAMDA* is declared external to this procedure and should be dimensioned [1:2, 1:*k*]. The procedure sets the parameter *Par*: *Par* = 0 indicates there are no roots, otherwise *Par* = +1. The tolerance value *Tol* governs the accuracy of the pivoting routine used in the procedure *REDUCE*. *REDUCE* is a procedure applied to a matrix X of rank r to find matrices $P1$ and $P2$ so that

$$P1 \times X \times P2 = \begin{bmatrix} I_r & 0 \\ 0 & 0 \end{bmatrix}.$$

The input parameters of *PENCIL* must be *A*, *B*, *m*, *n*, and *Tol*. The following supplementary procedures are required: *REDUCE*, *SWAP*, *Matmul*, *EIG*. The purpose of each of these procedures is explained in the head comment of each. A routine for finding eigenvalues of a square matrix should be supplied by the user to be called from procedure *EIG*;

comment Examples. We show several examples of the generalized eigenvalue problem and how the procedure *PENCIL* processed these examples for input into the user-supplied eigenvalue routine, here named *EIGENVALUES*, which is called in *PENCIL* by *EIG*. The format for the examples is (a) the original A and B matrices are shown, (b) the derived A_r matrix (to four digits) whose eigenvalues are the rank-reducing numbers of $(A-\lambda B)$ which is input to *EIG* and then to *EIGENVALUES* is shown, and (c) any pertinent comments about that example are made.

$$A = \begin{bmatrix} 10 & 2 & 3 & 1 & 1 \\ 2 & 12 & 1 & 2 & 1 \\ 3 & 1 & 11 & 1 & -1 \\ 1 & 2 & 1 & 9 & 1 \\ 1 & 1 & -1 & 1 & 15 \end{bmatrix} \quad B = \begin{bmatrix} 12 & 1 & -1 & 2 & 1 \\ 1 & 14 & 1 & -1 & 1 \\ -1 & 1 & 16 & -1 & 1 \\ 2 & -1 & -1 & 12 & -1 \\ 1 & 1 & 1 & -1 & 11 \end{bmatrix}$$

$$A_5 = \begin{bmatrix} .7220 & .0248 & .0871 & .2358 & -.1426 \\ .0160 & .8663 & .1882 & .0636 & .0015 \\ .0781 & .2339 & .7967 & -.0357 & .2016 \\ .3107 & .0554 & -.0699 & .8551 & -.0791 \\ -.1792 & .0261 & .1447 & -.0173 & 1.4020 \end{bmatrix} \tag{1}$$

This example contains no complications since both A and B are square of full rank. A_5 is effectively $B^{-1}A$. The example is the first shown in [2, Sec. 7].

$$A = \begin{bmatrix} 2 & 3 & 1 \\ 3 & 5 & 2 \\ 3 & 4 & 2 \end{bmatrix} \quad B = \begin{bmatrix} 2 & -1 & 1 \\ 2 & 2 & 2 \\ 2 & -1 & 1 \end{bmatrix} \tag{2}$$

This example, from unpublished notes by J. H. Wilkinson, calls *PENCIL* recursively. A and B above are transformed to the one by one matrices

$$A' = [.23077] \quad B' = [1.1538]$$

for re-entry to *PENCIL*. The final output from *PENCIL* is the derived matrix $A_1 = [.2]$.

$$A = \begin{bmatrix} 2 & 3 & 2 \\ 3 & 5 & 2 \\ 2 & 2 & 2 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{3}$$

Here $m - r - q = n - r - q = 0$ so that there were no recursive calls of *PENCIL*. On exit from *PENCIL* $A_2 = \begin{bmatrix} 0 & 1 \\ 1 & 3 \end{bmatrix}$.

$$A = \begin{bmatrix} 2 & 1 & 2 & 1 \\ 2 & 1 & 2 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 1 & -1 & 1 & 1 \\ 1 & -1 & 1 & 1 \end{bmatrix} \tag{4}$$

This system has no roots which reduce the rank $(A-\lambda B)$. The failure is an example of Theorem 2.3(a) of [4] when both $E12$ and $E21$ exist.

$$A = \begin{bmatrix} 2 & 2 \\ 1 & 1 \\ 2 & 2 \\ 1 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 1 \\ -1 & -1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \tag{5}$$

This system has no roots which reduce the rank $(A-\lambda B)$. The failure is an example of Theorem 2.3(b) of [4] where both $E12$ and $E21$ exist.

$$A = \begin{bmatrix} -1 & 0 & 2 & 4 & 1 & -1 \\ -1 & 1 & 3 & 6 & 1 & -1 \\ 0 & 1 & 1 & 6 & 0 & 0 \\ 1 & 2 & 4 & 8 & 1 & 1 \end{bmatrix} \quad B = \begin{bmatrix} -1 & -1 & 0 & 0 & 1 & 1 \\ 0 & 1 & -1 & 1 & -1 & 0 \\ 1 & 0 & 1 & -1 & -1 & -1 \\ 2 & -1 & 3 & -3 & 1 & -2 \end{bmatrix} \tag{6}$$

This system has no roots which reduce the rank of $(A-\lambda B)$, but that fact is not discovered by *PENCIL* until a recursive call is made on

$$A' = [.2353 \quad .2353 \quad 0] \quad B' = [.4706 \quad 1 \quad -.3529].$$

The failure is an example of Theorem 2.3(a) of [4] when $E21$ is degenerate.

$$A = \begin{bmatrix} -1 & -1 & 0 & 1 \\ 0 & 1 & 1 & 2 \\ 2 & 3 & 1 & 4 \\ 4 & 6 & 6 & 8 \\ 1 & 1 & 0 & 1 \\ -1 & -1 & 0 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} -1 & 0 & 1 & 2 \\ -1 & 1 & 0 & -1 \\ 0 & -1 & 1 & 3 \\ 0 & 1 & -1 & -3 \\ 1 & -1 & 0 & 1 \\ 1 & 0 & -1 & -2 \end{bmatrix} \tag{7}$$

This system has no roots which reduce the rank of $(A-\lambda B)$, but that fact is not discovered by *PENCIL* until a recursive call is made on

$$A' = \begin{bmatrix} .1176 \\ .2353 \end{bmatrix} \qquad B' = \begin{bmatrix} .2353 \\ 1 \end{bmatrix}.$$

Except for the entry in the fifth row and third column of B, this example is the transpose of example (6). The failure is an example of Theorem 2.3(b) of [4] when $E12$ is degenerate.

$$A = \begin{bmatrix} 2 & 3 & 0 & 0 \\ 3 & 5 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{8}$$

$$A_2 = \begin{bmatrix} 2 & 3 \\ 3 & 5 \end{bmatrix}$$

Examples (8), (9), and (10) all reduce to the same derived eigenproblem. Each, however, tests paths to different exits from *PENCIL*. Here $s + t = 0$.

$$A = \begin{bmatrix} 2 & 3 & 0 & 0 \\ 3 & 5 & 0 & 0 \\ 0 & 0 & 2 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{9}$$

$$A_2 = \begin{bmatrix} 2 & 3 \\ 3 & 5 \end{bmatrix}$$

See comment at example (8). Here $s = 0$, i.e. $E21$ is degenerate and t is found to be zero.

$$A = \begin{bmatrix} 2 & 3 & 0 \\ 3 & 5 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{10}$$

$$A_2 = \begin{bmatrix} 2 & 3 \\ 3 & 5 \end{bmatrix}$$

See comment at example (8). Here $t = 0$, i.e. $E12$ is degenerate and s is found to be zero.

$$A = \begin{bmatrix} 2 & -1 & 1 \\ 2 & 2 & 2 \\ 2 & -1 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 2 & 3 & 1 \\ 3 & 5 & 2 \\ 3 & 4 & 2 \end{bmatrix} \tag{11}$$

$$A_3 = \begin{bmatrix} 3 & 0 & 1 \\ -7 & 2 & -1 \\ 4 & -2 & 0 \end{bmatrix}$$

In all other examples $rank(A) > rank(B)$. Here $rank(A) < rank(B)$. The derived eigenproblem has one nonzero root, 5, and two zero roots. Note that this problem is the same as example (2) except that A and B are interchanged. Ordinarily interchanging the roles of A and B yields the reciprocals of the eigenvalues. When one problem has zero eigenvalues, the interchanged problem has no corresponding reciprocal eigenvalue. Thus in example (2) we find only one solution, the reciprocal of 5;

comment Here we relate our work to that reported in the literature. Gantmacher [1, Chap. XII] has shown that every $m \times n$ matrix of the form $(A-\lambda B)$ can be transformed by elementary row and column operations to a canonical form. (We call this form the Gantmacher Normal Form, G.N.F.) That is, there exist nonsingular $m \times m$ matrix P and $n \times n$ matrix Q so that $P(A-\lambda B)Q$ has a quasi-diagonal form which is the direct sum of as many as $(p+q+r+2)$ blocks as follows:

$$\left\{ \overset{g}{h[0]}, L_{\epsilon_{g+1}}, \cdots, L_{\epsilon_{g+p}}, \right.$$
$$\left. \overset{T}{L}_{\eta_{h+1}}, \cdots, \overset{T}{L}_{\eta_{h+q}}, N_{\mu_1}, \cdots, N_{\mu_r}, (J-\lambda \mathrm{I}) \right\}.$$
$$g$$

All other entries in the G.N.F. are zero. The block $\overset{g}{h[0]}$ has h rows and g columns and all its elements are zero. The block $L_{\epsilon_{g+i}}$ has ϵ_{g+i} rows and $(\epsilon_{g+i} + 1)$ columns with structure

$$L_\epsilon = \begin{array}{c} \epsilon + 1 \\ \begin{bmatrix} \lambda & 1 & & & 0 \\ & \lambda & 1 & & \\ & & \cdot & \cdot & \\ & & & \cdot & \cdot \\ & & & \cdot & \cdot \\ 0 & & & \lambda & 1 \end{bmatrix} \epsilon. \end{array}$$

The block $\overset{T}{L}_{\eta_{h+j}}$ has $(\eta_{h+j}+1)$ rows and η_{h+j} columns with structure

$$L_\eta{}^T = \begin{array}{c} \eta \\ \begin{bmatrix} \lambda & & & 0 \\ 1 & \lambda & & \\ & 1 & \cdot & \\ & & \cdot & \cdot \\ & & \cdot & \cdot \\ & & & \cdot\ \lambda \\ 0 & & & 1 \end{bmatrix} \eta + 1. \end{array}$$

The square block N_{μ_k} is μ_k by μ_k with structure

$$N_\mu = \begin{array}{c} \mu \\ \begin{bmatrix} 1 & \lambda & & 0 \\ & 1 & \cdot & \\ & & \cdot & \cdot \\ & & \cdot & \cdot \\ & & & \cdot\ \lambda \\ 0 & & & 1 \end{bmatrix} \mu. \end{array}$$

The final block $(J-\lambda I)$ is an ordinary square eigensystem in Jordan normal form.

For a given matrix $(A-\lambda B)$ let

$$w = \max_i \left\{ \epsilon_i, \eta_i, entier\left(\frac{\mu_i - 1}{2}\right) \right\}$$

where ϵ_i, η_i, μ_i are defined from the G.N.F. Then *PENCIL* applied to $(A-\lambda B)$ will require no more than w recursive calls to derive the reduced eigenproblem. We have run many examples with various combinations of L, L^T, and N blocks to test our procedures. Since the L, L^T, and N blocks contribute no solutions, these examples are uninteresting to reproduce here. If the G.N.F. of the original problem contains only L, L^T, and N blocks, there are no solutions. If the G.N.F. contains $(J-\lambda I)$ as well, the output of our procedures for *EIG* is the matrix whose Jordan normal form is J.

References:

1. Gantmacher, F. R. *The Theory of Matrices, II.* Chelsea Pub. Co., New York, 1959, pp. 35–40.
2. Martin, R. S., and Wilkinson, J. H. Reduction of the symmetric eigenproblem $Ax = \lambda Bx$ and related problems to standard form. *Num. Math. 11* (1968), 99–110.
3. Thompson, G. L., and Weil, R. L. Reducing the rank of $(A-\lambda B)$. *Proc. AMS 26*, 4 (Dec. 1970), 548–554.
4. Thompson, G. L., and Weil, R. L. The roots of matrix pencils $(Ay=\lambda By)$: existence, calculations, and relations to game theory. Center for Mathematical Studies in Business and Economics, Rep. 6936, U. Chicago, Aug. 1969 [*Linear Alg. Appl.* (to appear)];

```
begin
  integer q, r, s, t, i, j, k, limc, limr;
  begin
    array P1[1:m, 1:m], P2[1:n, 1:n];
    Par := +1;  Sp := 0;
    k := if m < n then m else n;
    for i := 1 step 1 until 2 do
      for j := 1 step 1 until k do LAMDA[i, j] := 0;
    REDUCE(P1, B, P2, m, n, r, Tol);
    if r = 0 then
    begin Par := 0;  go to Endp;  end;
    Matmul(P1, A, A, m, m, n);  Matmul(A, P2, A, m, n, n);
    Matmul(P1, B, B, m, m, n);  Matmul(B, P2, B, m, n, n);
    comment NOTE:; The last two matrix multiplications to-
      gether, by definition of P1 and P2, change B to

            $\begin{bmatrix} I_r & 0 \\ 0 & 0 \end{bmatrix}$.

    To avoid the multiplications at this point, an r by r identity
      matrix B can be generated directly. Note that r is deter-
      mined by the immediately preceding call of REDUCE;
  end;
  comment At this stage

    $A := P1 \times A \times P2, \quad B := P1 \times B \times P2 = \begin{bmatrix} I_r & 0 \\ 0 & 0 \end{bmatrix}$.

    B is "reduced" and the corresponding operations have been
    performed on A;
  if ((n−r) = 0 ∧ (m−r)=0) then
  begin
    EIG(LAMDA, A, r);  go to Endp;
    comment Calculations for examples (1) and (11) exit here,
      and for example (2) cease here after one recursive call. See
      discussion in the "Examples" comment;
  end;
  limc := if (n−r) = 0 then 1 else n − r;
  limr := if (m−r) = 0 then 1 else m − r;
  begin
    array C12[1:r, 1:limc], C21[1:limr, 1:r], C22[1:limr, 1:limc],
      P1 [1:limr, 1:limr], P2[1:limc, 1:limc];
    if (n−r) ≠ 0 then
    begin
      for i := 1 step 1 until r do
      for j := r + 1 step 1 until n do
      C12[i, j−r] := A[i, j];
    end;
    if (m−r) ≠ 0 then
    begin
      for i := r + 1 step 1 until m do
      begin
        for j := 1 step 1 until r do C21[i−r, j] := A[i, j];
        if (n−r) ≠ 0 then
        begin
          for j := r + 1 step 1 until n do C22[i−r, j−r] := A[i, j];
        end
      end
    end;
    comment A has now been partitioned and the parts are
      referred to below as

            $\begin{array}{ccc} r & n-r & \\ \left[\begin{matrix} A \\ C21 \end{matrix}\right. & \left.\begin{matrix} C12 \\ C22 \end{matrix}\right] & \begin{matrix} r \\ n-r; \end{matrix} \end{array}$

    if ((n−r) = 0 ∨ (m−r)=0) then
    begin q := 0;  go to Mul;  end;
    REDUCE(P1, C22, P2, limr, limc, q, Tol);
    Matmul(C12, P2, C12, r, limc, limc);  Matmul(P1, C21, C21,
      limr, limr, r);
    Matmul(P1, C22, C22, limr, limr, limc);  Matmul(C22, P2, C22,
      limr, limc, limc);
    comment See "Note" comment above to generate C22
      directly without matrix multiplications;
    comment C22 has been "reduced"ʳ and the requisite opera-
      tions have been performed on C12 and C21. That is

            $C22 := P1 \times C22 \times P2 = \begin{bmatrix} I_q & 0 \\ 0 & 0 \end{bmatrix}$,

      C12 := P1 × C12, and C21 := C21 × P2. Thus A now looks
        like

            $\begin{array}{cc} & \begin{array}{ccc} r & & n-r \end{array} \\ r & \begin{bmatrix} A_r & \cdot & C12 & \\ \cdot & \cdot\ \cdot & \cdot\ \cdot & \cdot \\ & \cdot\ I_q & & 0 \\ C21 & \cdot & & \\ & \cdot\ 0 & & 0 \end{bmatrix} m-r; \end{array}$

    if q = 0 then go to Mul;
    begin
      array D21[1:r, 1:r];
      Matmul (C12, C21, D21, r, q, r);
      for i := 1 step 1 until r do
        for j := 1 step 1 until r do
          A[i, j] := A[i, j] − D21[i, j];
    end;
  Dstep:
    if ((m−r−q)=0 ∧ (n−r−q)=0) then
    begin
      EIG(LAMDA, A, r);  go to Endp;
      comment Calculations for example (3) cease here. See
        discussion in the "Examples" comment;
    end;
  Mul:
    limr := if (m−r−q) = 0 then 1 else m − r − q;
    limc := if (n−r−q) = 0 then 1 else n − r − q;
    begin
      array E12[1:r, 1:limc], E21[1:limr, 1:r];
      if (n−r−q) ≠ 0 then
      begin
        for i := 1 step 1 until r do
          for j := q + 1 step 1 until n − r do
            E12[i, j−q] := C12[i, j];
      end;
      if (m−r−q) ≠ 0 then
      begin
        for i := q + 1 step 1 until m − r do
          for j := 1 step 1 until r do
            E21[i−q, j] := C21[i, j];
      end;
      comment The columns of C12 above I_q and the rows of C21
        to the left of I_q are annihilated. The remaining submatrices
        are now called E12 and E21, respectively.

            $A = \begin{bmatrix} A_r & 0 & E12 \\ 0 & I_q & 0 \\ E21 & 0 & 0 \end{bmatrix}$;

      begin
        array P1, P4[1:r, 1:r], P2[1:limc, 1:limc],
          P3[1:limr, 1:limr];
        if (n−r−q) ≠ 0 then REDUCE(P1, E12, P2, r, limc, t,
          Tol
        else); t := 0;
        if (m−r−q) ≠ 0
          then REDUCE(P3, E21, P4, limr, r, s, Tol);
        else s := 0;
        if ((r=t) ∨ (r=s)) then
        begin
          comment Set parameter for no solutions;
```

```
            par := 0;   go to Endp;
            comment  Calculations for examples (4–7) (after one
              recursive call for (6) and (7)) cease here. See discus-
              sion in the "Examples" comment;
          end;
          if (s+t) = 0 then
          begin
            EIG(LAMDA, A, r);   go to Endp;
            comment  Calculations for examples (8–10) cease here.
              See discussion in "Examples" comment;
          end;
          if (n-r-q) ≠ 0 then
          begin
            Matmul(P1, A, A, r, r, r);   Matmul(P1, B, B, r, r, r);
          end;
           if (m-r-q) ≠ 0 then
          begin
            Matmul(A, P4, A, r, r, r);   Matmul(B, P4, B, r, r, r);
            comment  E12 and E21 have been "reduced" and the
              requisite operations have been performed on B. That
              is
```

$$E12 := P1 \times E12 \times P2 = \begin{bmatrix} I_t & 0 \\ 0 & 0 \end{bmatrix},$$

$$E21 := P3 \times E21 \times P4 = \begin{bmatrix} I_s & 0 \\ 0 & 0 \end{bmatrix},$$

and $B_r := P1 \times I_r \times P4$. Thus A looks like

$$\begin{array}{cccc|l} r & q & t & (n-r-q-t) & \\ & \cdot\ \cdot & & & \\ & & I_t & 0 & \\ & \cdot\ \cdot & & & \\ A_r & 0 & & & r \\ & \cdot\ \cdot & & & \\ & & 0 & 0 & \\ \cdots & \cdots & \cdots & \cdots & \\ 0 & I_q & 0 & & q \\ \cdots & \cdots & \cdots & \cdots & \\ I_s & 0 & & & s \\ & \cdot\ \cdot & & & \\ & 0 & 0 & & \\ & \cdot\ \cdot & & & \\ 0 & 0 & & & \\ & \cdot\ \cdot & & & (m-r-q-s) \end{array}$$

and B looks like

$$\left[\begin{array}{ccccc} B_r & 0 & 0 & & 0 \\ 0 & 0 & 0 & & 0 \\ 0 & 0 & 0 & & 0 \\ 0 & 0 & 0 & & 0 \end{array}\right]\begin{array}{l} r \\ q \\ s \\ (m-r-q-s) \end{array}$$

```
          end
        end
      end
    end;
  comment  The columns of A_r above I_s and the rows of A_r to the
    left of I_t are annihilated, and the remaining (r-s) × (r-t)
    submatrix is called G. The corresponding r - s rows and r - t
    columns of B are called H. The following statements build
    the matrices G and H;
  begin
    array G, H[1:r-t, 1:r-s];
    for i := t + 1 step 1 until r do
      for j := s + 1 step 1 until r do
      begin
        G[i-t, j -s] := A[i, j];   H[i-t, j-s] := B[i, j]
      end;
    PENCIL(G, H, r-t, r-s, LAMDA, Sp, Par, Tol);
  end;
Endp:
end PENCIL;
```

```
procedure  REDUCE(I1, X, I2, m, n, dex, Tol);   value X;
  real array X, I1, I2;   real Tol;   integer m, n, dex
comment  REDUCE applied to an m by n matrix X of rank dex
  finds an m by m matrix I1 and an n by n matrix I2 such that
```

$$I1 \times X \times I2 = \begin{bmatrix} I_{dex} & 0 \\ 0 & 0 \end{bmatrix}.$$

```
  The rank is found by REDUCE and returned in dex. Gaussian
  elimination with complete pivoting is used until the (dex + 1)st
  pivot element would be less than Tol, a parameter to be supplied
  by the user to PENCIL. This procedure is supplied to make
  the PENCIL routine complete. Users concerned with increased
  numerical accuracy should write their own routines paying
  attention to multiple precision, and ill-conditioning. Note that
  X is called by value and is not altered. When preserving X is
  not important, PENCIL can be made to run faster by elimi-
  nating value X in REDUCE and the matrix multiplications in
  PENCIL that directly follow the calls to REDUCE;
begin
  integer i, j, k, l, lim, p, q;   real div;
  real array CVEC, TEMP[1:n, 1:n], I3[1:m, 1:m];
  integer array rvec[1:m], mvec[1:n];
  if m > n then lim := n else lim := m;
  dex := 0;
  for i := 1 step 1 until m do
  begin
    I1[i, i] := 1;   rvec[i] := i;
    for j := 1 step 1 until m do if i ≠ j then I1[i, j] := 0;
  end;
  for i := 1 step 1 until n do
  begin
    mvec[i] := i;
    for j := 1 step 1 until n do
    begin
      if i ≠ j then I2[i, j] := TEMP[i, j] := CVE1C[i, j] := 0
        else I2[i, j] := TEMP[i, j] := CVEC[i, j] :=
    end
  end;
Rowop:
  for i := 1 step 1 until m do
  for j := 1 step 1 until m do
    if i = j then I3[i, j] := 1 else I3[i, j] := 0;
  dex := dex + 1;
  if dex ≤ lim then
  begin
    SEARCH(X, dex, k, l, m, n, rvec, mvec);
    comment X(k, l) is the pivot element;
    ISWAP(rvec[dex], rvec[k]);  ISWAP(mvec[dex], mvec[l]);,
    for i := 1 step 1 until n do SWAP(CVEC[i, dex], CVEC[i l]);
    if abs(X[rvec[dex], mvec[dex]]) < Tol then
    begin dex := 0;  go to Endr  end;
    for i := 1 step 1 until m do if i ≠ dex then
    begin
      div := X[rvec[i], mvec[dex]]/X[rvec[dex], mvec[dex]]);
      I3[i, k] := -div;
      for j := dex step 1 until n do
        X[rvec[i], mvec[j]]
          := X[rvec[i], mvec[j]] - (div×X[rvec[dex], mvec[j]]);
    end;
    I3[dex, k] := 1.0/X[rvec[dex], mvec[dex]];
    for j := dex step 1 until n do
      if j ≠ dex then
        X[rvec[dex], mvec[j]]
          := X[rvec[dex], mvec[j]]/X[rvec[dex], mvec[dex]];
    X[rvec[dex], mvec[dex]] := 1.0;
    if k ≠ dex then SWAP(I3[dex, dex], I3[k, dex]);
    Matmul(I3, I1, I1, m, m, m);
    for i := dex + 1 step 1 until m do
      for j := dex + 1 step 1 until n do
```

```
        if abs (X[rvec[i], mvec[j]]) > Tol then go to Rowop;
  end;
  if m < n ∨ dex < lim then
  begin
    integer p, q;   real mul;
    for i := 1 step 1 until dex do
      for j := dex + 1 step 1 until n do
        if abs(X[rvec[i], mvec[j]]) > Tol then
        begin
          mul := TEMP[i, j] := −X[rvec[i], mvec[j]];
          for p := 1 step 1 until m do
            X[rvec[p], mvec[j]]
              := X[rvec[p], mvec[j]] + (mul×X[rvec[p], mvec[i]]);
          Matmul(I2, TEMP, I2, n, n, n);
          for p := 1 step 1 until n do
            for q := 1 step 1 until n do
              if p ≠ q then TEMP[p, q] := 0
                else TEMP[p, q] := 1;
        end
  end;
  Matmul(CVEC, I2, I2, n, n, n);
Endr:
end REDUCE;
procedure SWAP(r, s);   real r, s;
comment  SWAP interchanges real variables r and s;
begin
  real temp;
  temp := r;   r := s;   s := temp;
end SWAP;
procedure ISWAP(r, s) ;   integer r, s;
comment  ISWAP interchanges integer variables r and s;
begin
  integer temp;
  temp := r;   r := s;   s := temp;
end ISWAP;
procedure Matmul(X, Y, Z, u, v, w);
  real array X, Y, Z;   integer u, v, w;
comment  Matmul causes the matrix product X times Y to be
  stored in matrix Z. X is u by v, Y is v by w, and Z is u by w.
  For improved accuracy inner products should be accumulated
  using double precision arithmetic;
begin
  integer i, j, k;   real array TEMP[1:u, 1:w];
  for i := 1 step 1 until u do
    for j := 1 step 1 until w do
    begin
      TEMP[i, j] := 0
      for k := 1 step 1 until v do
        TEMP[i, j] := TEMP[i, j] + X[i, k] × Y[k, j];
    end;
    for i := 1 step 1 until u do
      for j := 1 step 1 until w do Z[i, j] := TEMP[i, j];
end Matmul;
procedure SEARCH(Y, Lim, k, l, m, n, veci, vecj);
  array Y;   integer Lim, k, l, m, n;   integer array veci, vecj;
comment  SEARCH finds the largest element in the m by n
  array Y starting at Y[Lim, Lim], searching the remaining sub-
  array. Vectors veci and vecj record the row and column swaps
  which have occurred previous to the call of SEARCH. k and l
  are the row and column indices, respectively, for the largest
  element in the array searched.
begin integer i, j;
  k := l := Lim;
  for i := Lim step 1 until m do
    for j := Lim step 1 until n do
    begin
      if abs(Y[veci[i], vecj[j]]) > abs(Y[veci[k], vecj[l]]) then
      begin k := i;   l := j;   end
    end
end SEARCH;
procedure EIG(LAMDA, X, r);
  real array LAMDA, X;   integer r;
comment  EIG calls a procedure which finds the eigenvalues of
  the r by r matrix X and stores them in the 2 by r matrix LAMDA,
  real parts in LAMDA[1, j], imaginary parts in LAMDA[2, j];
begin
  EIGENVALUES (X, LAMDA, r);
  Sp := r;
end EIG;
```

Remark on Algorithm 405 [F2]

Roots of Matrix Pencils: The Generalized Eigenvalue Problem [A.M. Dell, R.L. Weil, and G.L. Thompson, *Comm. ACM 14*, (Feb. 1971), 113–117]

Richard M. Heiberger [Recd. 19 May 1971, 29 July 1971, and 8 Sept. 1971]
Department of Statistics, Harvard University*

Key Words and Phrases: eigenvalues, matrix roots, pencil roots
CR Categories: 5.1, 5.3

Algorithm 405 calculates rank-reducing numbers which are similar to, but not identical to, generalized eigenvalues. An eigenvalue of A with respect to B, as defined in this Remark, satisfies the equations

$$x^{T}(A - \lambda B) = 0, \qquad (A - \lambda B)y = 0 \tag{1}$$

for appropriately dimensioned vectors x and y. A rank-reducing number λ_0, as defined by Thompson and Weil [3], further satisfies

$$\text{Rank }(A - \lambda_0 B) < \text{Rank }(A - \lambda B) \tag{2}$$

for some value $\lambda \neq \lambda_0$. The distinction is meaningful only if the matrices A and B are of less than full rank.

The definition (1) is the simplest generalization of the ordinary eigenvalue problem in that the only new concept is the replacement of an identity matrix with an arbitrary matrix B. This form of the problem arises in many physical contexts, usually with A and B square symmetric, and B positive definite (see [4] for examples). Dell, Weil, and Thompson find that in their context the additional condition (2) is desirable since rank-reducing numbers are always discrete, finite in number, and related to a Jordan-like canonical form.

In order to insure that all eigenvalues, as defined here by (1), are discrete, one further condition than given in Algorithm 405 must be tested. It is necessary that

$$\text{Rank }(A - \lambda B) = \min\,(m, n) \tag{3}$$

for at least one value of λ. In the special case that $m = n$ (square matrices) the condition (3) is equivalent to

$$\det\,(A - \lambda B) \neq 0 \tag{4}$$

for at least one value of λ. When this condition is violated, the spectrum of eigenvalues is continuous; that is, for every complex number λ there exist vectors x and y such that (1) is satisfied. Discrete rank-reducing numbers may exist even when the rank condition (3) is violated. Example 8 accompanying Algorithm 405 does not satisfy condition (3) and therefore does not have discrete eigenvalues although it does have discrete rank-reducing numbers.

The procedure *PENCIL* is similar to the algorithm developed by Fix and Heiberger [1] for the generalized eigenvalue problem when A and B are Hermitian matrices. We showed that the spectrum

* This work was supported by the Cambridge Project through the Department of Statistics, Harvard University. Author's current address; Department of Statistics, Iowa State University, Ames, IA 50010.

of $Ax - \lambda Bx = 0$ consists of stable and unstable eigenvalues, which undergo, respectively, small and large changes in response to small changes in A and B. We proved that our algorithm isolates and accurately computes the eigenspace associated with the stable eigenvalues. We did not attempt to extend our proof to non-Hermitian and rectangular matrices, for which Algorithm 405 may also be used. Our proof explicitly does not apply to rank-reducing numbers unless the rank condition (3) is satisfied. Instead it suggests that the computed solution may be inaccurate, as the first example in [1] shows. We programmed in APL [2] and Fortran (unpublished).

The following changes to Algorithm 405 will modify it to calculate either eigenvalues or rank-reducing numbers at the user's option. The user of rank-reducing numbers will be warned if the rank condition (3) is not satisfied, and there may be numerical inaccuracy in his solution.

Page 113, column 1. Replace procedure heading with:

```
    procedure PENCIL (A,B,m,n,LAMDA,Sp,Par,Tol,eigrrn);
```

Page 113, column 1, preceding first **comment** insert:

```
    integer array eigrrn;
```

Page 113, column 1, first **comment**. Replace the sentence:

The input parameters of *PENCIL* must be *A*, *B*, *m*, *n*, and *Tol*.
with the following:

The input parameter *eigrrn*[1] is used to direct the program to calculate either eigenvalues or rank-reducing numbers. If *eigrrn*[1] = 0, then eigenvalues will be calculated. If *eigrrn*[1] = 1, then rank-reducing numbers will be calculated. The parameter *eigrrn*[2] must be set to 0 as an input parameter. As an output parameter *eigrrn*[2] indicates whether the rank condition (3) is satisfied. If *eigrrn*[2] = 0, the condition is satisfied. If *eigrrn*[2] = 1, the condition is violated. When the rank condition is violated and eigenvalues are being calculated, the parameter *Par* is set to 0 indicating no roots, and the procedure is terminated. When the rank condition is violated and rank-reducing numbers are being calculated, the procedure continues calculations as at present, but the user is warned that there may be numerical inaccuracy in the solution. The input parameters of *PENCIL* must be *A*, *B*, *m*, *n*, *Tol*, *eigrrn*[1], and *eigrrn*[2].

Page 116, column 1, preceding line −11. Insert:

```
if ((n−r−q−t≠0) ∧ (m−r−q−s≠0)) then
begin
comment Set parameter for continuous spectrum;
    eigrrn[2] := 1;
if eigrrn[1] = 0 then
    begin
    comment Set parameter for no solution;
      Par := 0;   go to Endp;
    end;
    comment Beware of possible numerical inaccuracy;
end;
```

Page 116, column 1, line −4. Replace with:

```
    PENCIL(G,H,r−t,r−s,LAMDA,Sp,Par,Tol,eigrrn);
```

There are several typographical errors. The following lines should read as given below.

Page 115, column 2, lines −8 and−7:

```
    Tol)
    else t := 0;
```

Page 115, column 2, line −5:

```
    REDUCE(P3,E21,P4,limr,r,s,Tol)
```

Page 116, column 1, line 1:

```
    Par := 0;   go to Endp;
```

I would also suggest that the following **value** parts be added for more efficient execution.

```
Procedure PENCIL                 Procedure Matmul
    value A,B,m,n,Tol;               value u,v,w;
Procedure REDUCE                 Procedure SEARCH
    value X,m,n,Tol;                 value Lim,m,n,veci,vecj;
```

References

1. Fix, G., and Heiberger, R.M. An algorithm for the Ill-conditioned generalized eigenvalue problems. *SIAM J. Numer. Anal.* (Mar. 1972), 78–88.

2. Heiberger, R.M. APL functions for data analysis and statistics. Res. Rep. CP-5, Dep. of Statistics, Harvard U., 1971.

3. Thompson, G.L., and Weil, R.L. Reducing the rank of $(A - \lambda B)$. Proc. Amer. Math. Soc. 26, 4 (Dec. 1970), 548–54.

4. Wilkinson, J.H. *The Algebraic Eigenvalue Problem.* Oxford U. Press, Oxford, 1965.

Algorithm 406

Exact Solution of Linear Equations Using Residue Arithmetic [F4]

Jo Ann Howell (Recd. 23 Mar. 1970 and 2 July 1970)
The University of Texas at Austin, Center for Numerical Analysis, Austin, TX 78712

Key Words and Phrases: residue arithmetic, symmetric residue, modulus, mixed-radix representation, symmetric mixed-radix representation, mixed-radix conversion, prime number, linear equations, Gaussian elimination, matrix inversion, determinant, adjoint matrix, ill-condition

CR Categories: 3.15, 5.14

Description

Purpose. The subroutine *EXACT* solves the matrix equation $AX = B$ for X, where A is an N by N integer matrix, B is an N by M integer matrix, and X is an N by M real matrix. Residue arithmetic is used to obtain the exact solution, consisting of the rational components of X, i.e. $det(A)$ and the elements of $Y = A^{adj}B$, and the rounded solution, computed as the quotient of the rational components and stored in the array X. The subroutine can be used to solve systems of linear algebraic equations, to invert matrices, and to compute determinants and adjoint matrices.

Method. A methd similaor to the one described in [1, 2, and 3] is used to solve a system of linear algebraic equations $AX = B$, using residue arithmetic. However, since there are differences we shall describe them here. In [1] the concept of *residue modulo m* refers to the least nonnegative remainder of the integer x after division by m. The definition here, on the other hand, is preferable as a matter of computational convenience reflected by the definition of the *FORTRAN MOD* function.

Definition. Given any integer x and any modulus m, if

(i) $r \equiv x \ (mod\ m)$,
(ii) $|r| < m$, and
(iii) $sgn(r) = sgn(x)$,

then we write

$r = |x|_m$

and say r is a residue of x modulo m.

It is easily shown that this residue is also unique.

Since our definition of residue here differs from that in [1], we must point out that each of the theorems in [1], relative to the nonnegative residue system, has an analog in the residue system defined here. However, in some of the analogous theorems it may be necessary to use the congruence symbol $\equiv$ in place of the equality symbol $=$. Thus, Algorithm I in [1] and [2] and Algorithm II in [3] can be completely described using our definition of residue.

We should point out that there are related discussions in [5], [6], and [7].

The subroutine *EXACT* uses Algorithm II to solve $AX = B$ using the residue system described here. First, the following preliminary calculations are carried out by the program before solving the system of equations. (i) The number, *IS*, of moduli required to obtain a solution is predicted by subroutine *LOGBND*, as described in [2]. The program computes

$$BOUND = log\left(2\left[\prod_{i=1}^{N}\left(\sum_{j=1}^{N} a_{ij}^{2}\right)^{\frac{1}{2}} \prod_{l=1}^{M}\prod_{k=1}^{N} |b_{kl}|\right]\right), \quad (|b_{kl}| \neq 0)$$

and *IS* chosen so that

$$\begin{aligned} BOUND &\leq SUMLOG \\ &= log(MM(1)) + \cdots + log(MM(IS)). \end{aligned}$$

where the $MM(I)$ are the stored moduli. (ii) The elements of A and B are reduced modulo $MM(I), I = 1, \cdots, IS$.

The subroutine *SOLVE* solves the residue system

$AX \equiv B\ (mod\ MM(I)), \quad I = 1, \cdots, IS$

for the residue representations (see [1]) of d and the elements of Y,

$d \sim \{|d|_{MM(1)}, |d|_{MM(2)}, \cdots, |d|_{MM(IS)}\}$

and

$y_{ij} \sim \{|y_{ij}|_{MM(1)}, |y_{ij}|_{MM(2)}, \cdots, |y_{ij}|_{MM(IS)}\}$,

where

$d = det(A)$

and

$Y = A^{adj}B$.

The computation is performed by means of Gaussian elimination for residue arithmetic [1] using the residue system described here. Then, the residue representations for d and the elements of Y are converted to their symmetric residue representations (see [3]),

$d \sim \{/d/_{MM(1)}, /d/_{MM(2)}, \cdots, /d/_{MM(IS)}\}$

and

$y_{ij} \sim \{/y_{ij}/_{MM(1)}, /y_{ij}/_{MM(2)}, \cdots, /y_{ij}/_{MM(IS)}\}$.

Next, subroutine *MXRADX* converts the symmetric residue representations for d and the elements of Y to their corresponding symmetric mixed-radix representations [3],

$d \sim \langle \beta_1, \beta_2, \cdots, \beta_{IS} \rangle$

and

$y_{ij} \sim \langle \alpha_{ij_1}, \alpha_{ij_2}, \cdots, \alpha_{ij_{IS}} \rangle$.

The conversion is accomplished by means of a mixed-radix conversion process described in [3].

From their symmetric mixed-radix representations, d and the elements of Y are directly obtainable, as follows:

$$d = \beta_1 + \beta_2 MM(1) + \beta_3 \prod_{k=1}^{2} MM(k) + \cdots + \beta_{IS-1} \prod_{k=1}^{IS-2} MM(k) + \beta_{IS} \prod_{k=1}^{IS-1} MM(k)$$

and

$$y_{ij} = \alpha_{ij_1} + \alpha_{ij_2} MM(1) + \alpha_{ij_3} \prod_{k=1}^{2} MM(k) + \cdots + \alpha_{ij_{IS-1}} \prod_{k=1}^{IS-2} MM(k) + \alpha_{ij_{IS}} \prod_{k=1}^{IS-1} MM(k).$$

Since each of these quantities may overflow a fixed-point word, they are stored as "multilength" numbers. In other words, d and each of the elements of Y are stored in several words, with *NDIGIT* digits in each word. On return from *EXACT*, these multilength numbers are stored in *MULTL*, with the elements of Y (stored columnwise) in the first $M*N$ rows of *MULTL*, and d in the $(M*N + 1)$th row of *MULTL*. The lowest order digits are in the first column, and the highest order digits are in column *LCOUNT*. Thus, the exact solution of $AX = B$, consisting of the elements of Y (stored columnwise) and the determinant of A, may be printed out as follows (assuming $NDIGIT \leq 7$):

```
      WRITE (1, 10)
10    FORMAT(24H MULTILENGTH DIGITS OF Y/)
      MTN = M*N
      MN1 = MTN+1
      L1 = LCOUNT+1
      DO 20 I = 1, MTN
20    WRITE (1, 30)(MULTL(I, L1-J), J=1, LCOUNT)
30    FORMAT (1X, 10I8)
      WRITE (1, 40)
40    FORMAT (//17H DETERMINANT OF A/)
      WRITE (1, 30)(MULTL(MN1, L1-J), J=1, LCOUNT)
```

Program Call. Subroutine *EXACT* is completely self-contained (composed of eight subroutines *EXACT*, *SOLVE*, *MXRADX*, *MLTLTH*, *CHECK*, *INVERS*, *RESIDU*, and *LOGBND*), and the calling sequence, which has 22 parameters, is

CALL EXACT (A, N, IN, B, M, IM, IMPIN, IMINI, NDIGIT, KPRIME, NOPRIM, NO2, X, DET, IER, MULTL, LCOUNT, ATEMP, MM, RY, W, V)

Communication to *EXACT* is solely through the parameter list which is described in comments at the beginning of the subroutine *EXACT*.

Cautions to User. 1. The user should test *IER* before attempting to print results. An error code of 1 may arise if

(a) $|det(A)| > \prod_{I=1}^{r} KPRIME(I)$,

where r is the number of primes, $KPRIME(J)$, for which $det(A) \not\equiv 0\ (mod(KPRIME(J)))$,

(b) $\max_{i,j} |y_{ij}| > \prod_{I=1}^{r} KPRIME(I)$,

where r is defined as in (a),

(c) $KPRIME(I)$ is not a prime (for some I),

(d) $KPRIME(I) = KPRIME(J), \ J \neq I$.

2. This algorithm is of limited use due to the fact that A and B must be integral, due to the limitations given in 1(a)–1(d) above, and due to the algorithm's inherent slowness. It is not intended as a substitute for other well-established procedures for solving systems of linear algebraic equations. However, it may be useful in obtaining the exact solution of an ill-conditioned system of equations which has integral coefficients or a system which has rational coefficients which can be scaled to make it integral. In fact, as Knuth [8, p. 256] states, this method is "substantially faster than anv other known method for obtaining exact solutions."

Test Results. Subroutine *EXACT* was tested on a CDC 6600 computer on which the maximum size of integer variables which can be used in arithmetic operations is 48 bits (~14 digits). The maximum size of real variables is 48 bits with an 11-bit exponent. The results are summarized below. The following parameters were used as input for both test cases:

IN = 10		*NDIGIT* = 7	
IM = 10		*NOPRIM* = 10	
IMPIN = 20		*NO2* = 20	
IMINI = 101			

$$KPRIME = \begin{bmatrix} 10000019 \\ 10000079 \\ 10000103 \\ 10000121 \\ 10000139 \\ 10000141 \\ 10000169 \\ 10000189 \\ 10000223 \\ 10000229 \end{bmatrix}.$$

(i) Input to *EXACT*:

$N = 10 \qquad M = 1$

$$A = \begin{bmatrix} 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 10^7 \\ 9 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ 8 & 8 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ 7 & 7 & 7 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \\ 6 & 6 & 6 & 6 & 6 & 5 & 4 & 3 & 2 & 1 \\ 5 & 5 & 5 & 5 & 5 & 5 & 4 & 3 & 2 & 1 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 3 & 2 & 1 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 & 2 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

$B = e_{10}$.

Output from *EXACT*:

$$X = \begin{bmatrix} -1.9999998E+07 \\ 1.9999998E+07 \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \\ 0. \\ -1.0000000E+00 \\ 2.0000000E+00 \end{bmatrix}$$

$DET = 1.0000000E+00$

MULTILENGTH DIGITS FOR Y

```
-1     -9999998
 1      9999998
 0            0
 0            0
 0            0
 0            0
 0            0
 0            0
 0           -1
 0            2
```

MULTILENGTH DIGITS FOR DETERMINANT A

```
 0            1
```

(ii) Input to *EXACT*[4]:

$N = 5 \quad M = 5$

$$A = \begin{bmatrix} 5 & 300 & -2100 & 4200 & -2520 \\ -60 & -2400 & 18900 & -40320 & 25200 \\ 210 & 6300 & -52920 & 117600 & -75600 \\ -280 & -6720 & 58800 & -134400 & 88200 \\ 126 & 2520 & -22680 & 52920 & -35280 \end{bmatrix}$$

$B = I_5$.

Output from *EXACT*:

$$X = \begin{bmatrix} 1.000000000000\text{E}{+}00 & 1.000000000000\text{E}{+}00 & 1.000000000000\text{E}{+}00 & 1.000000000000\text{E}{+}00 & 1.000000000000\text{E}{+}00 \\ 5.000000000000\text{E}{-}01 & 3.333333333333\text{E}{-}01 & 2.500000000000\text{E}{-}01 & 2.000000000000\text{E}{-}01 & 1.666666666667\text{E}{-}01 \\ 3.333333333333\text{E}{-}01 & 2.500000000000\text{E}{-}01 & 2.000000000000\text{E}{-}01 & 1.666666666667\text{E}{-}01 & 1.428571428571\text{E}{-}01 \\ 2.500000000000\text{E}{-}01 & 2.000000000000\text{E}{-}01 & 1.666666666667\text{E}{-}01 & 1.428571428571\text{E}{-}01 & 1.250000000000\text{E}{-}01 \\ 2.000000000000\text{E}{-}01 & 1.666666666667\text{E}{-}01 & 1.428571428571\text{E}{-}01 & 1.250000000000\text{E}{-}01 & 1.111111111111\text{E}{-}01 \end{bmatrix}$$

$DET = 5.3343360000\text{E}{+}10$

MULTILENGTH DIGITS FOR Y

```
5334    3360000
2667    1680000
1778    1120000
1333    5840000
1066    8672000
5334    3360000
1778    1120000
1333    5840000
1066    8672000
 889     560000
5334    3360000
1333    5840000
1066    8672000
 889     560000
 762     480000
5334    3360000
1066    8672000
 889     560000
 762     480000
 666    7920000
5334    3360000
 889     560000
 762     480000
 666    7920000
 592    7040000
```

MULTILENGTH DIGITS FOR DETERMINANT A

```
5334    3360000
```

Acknowledgments. The author wishes to thank Dr. Robert T. Gregory for his encouragement. This work was supported in part by grants from the Army Research Office (Durham) (Grant DA-ARO(D)-31-124-G1050), and the National Science Foundation (NSF Grant GP8442), at the University of Texas at Austin.

References

1. Howell, J. A. and Gregory, R. T. An algorithm for solving linear algebraic equations using residue arithmetic I. *BIT 9*, 3 (1969), 200–224.
2. Howell, J. A. and Gregory, R. T. An algorithm for solving linear algebraic equations using residue arithmetic II. *BIT 9*, 4 (1969), 324–337.
3. Howell, J. A. and Gregory, R. T. Solving linear equations using residue arithmetic-algorithm II. *BIT 10*, 1 (1970), 23–37.
4. Lotkin, M. A set of test matrices. *MTAC 9* (1955), 153–161.
5. Borosh, I. and Fraenkel, A. S. Exact solutions of linear equations with rational coefficients by congruence techniques. *Math. Comp. 20* (1966), 107–112.
6. Newman, M. Solving equations exactly, *J. Research NBS 17B*, 4 (1967), 171–179.
7. Takahasi, H. and Ishibashi, Y. A new method for exact calculations by a digital computer. *Information Processing in Japan 1*, (1961), 28–42.
8. Knuth, D. E. *The Art of Computer Programming, vol. 2.* Addison-Wesley, Reading, Mass., 1969.

Algorithm

```
      SUBROUTINE EXACT(A,N,IN,B,M,IM,IMPIN,IMIN1,NDIGIT,
     1KPRIME,NOPRIM,NO2,X,DET,IER,MULTL,LCOUNT,ATEMP,MM,
     2RY,W,V)
      DIMENSICN A(IN,IN),B(IN,IM),X(IN,IM),ATEMP(IN,IMPIN),
     1MULTL(IMIN1,NOPRIM),MM(NOPRIM),RY(IMIN1),
     2KPRIME(NOPRIM),W(NO2),V(NO2)
      INTEGER A,B,ATEMP,PP,W,V
      COMMON/MLEN/IB,PP,NZ,IS,IFLAG,IQUIT,NORES
C
C         THIS SUBROUTINE SOLVES THE MATRIX EQUATION AX=B
C         FOR X AND FOR THE EXACT SOLUTION, Y=A(ADJ)*B
C         AND DET A. RESIDUE ARITHMETIC IS USED TO OBTAIN
C         THE SOLUTION.
C
C          A  IS THE N BY N COEFFICIENT MATRIX AND MUST BE
C             CF TYPE INTEGER.
C          N  IS THE ORDER OF THE MATRIX A (N GREATER THAN 1).
C         IN  IS A DIMENSION PARAMETER WHICH DEFINES THE
C             CIMENSION OF A.  IT MUST BE EQUAL TO OR GREATER
C             THAN N.
C          B  IS THE N BY M MATRIX OF THE RIGHT-HAND SIDE AND
C             MUST BE OF TYPE INTEGER.
C          M  IS THE NUMBER OF COLUMNS OF B AND X (M GREATER
C             THAN 0).
C         IM  IS A DIMENSION PARAMETER WHICH DEFINES THE
C             SECOND DIMENSION OF THE 2-DIMENSIONAL ARRAYS
C             B AND X.  IT MUST BE EQUAL TO OR GREATER THAN M.
C      IMPIN  IS A DIMENSION PARAMETER WHICH IS IM + IN.
C      IMIN1  IS A DIMENSION PARAMETER WHICH IS IM * IN + 1.
C     NDIGIT  IS THE NUMBER OF DIGITS STORED IN EACH WORD
C             CURING MULTILENGTH ARITHMETIC OPERATIONS.  IT IS
C             MACHINE DEPENDENT AND MUST BE CHOSEN SO THAT
C             10 ** (2 * NDIGIT) IS LESS THAN OR EQUAL TO THE
C             LARGEST REPRESENTABLE INTEGER FOR THE COMPUTER
C             BEING USED.
C     KPRIME  IS THE LINEAR ARRAY OF NOPRIM MODULI.  THE
C             MODULI MUST BE PRIMES, CHOSED AS LARGE AS
C             POSSIBLE AND SO THAT KPRIME(I) * KPRIME(J) DOES
C             NOT OVERFLOW AN INTEGER WORD, FOR ALL I AND J.
C     NOPRIM  IS A DIMENSION PARAMETER WHICH DENOTES THE
C             NUMBER OF PRIMES (MODULI) STORED IN KPRIME.
C        NO2  IS A DIMENSION PARAMETER WHICH IS 2*NOPRIM.
C          X  IS THE N BY M FLOATING-POINT MATRIX WITH THE
C             M SOLUTION VECTORS AS COLUMNS.  IT IS THE
C             ROUNDED QUOTIENT OF THE RATIONAL COMPONENTS
C             CF X.
```

```
C        DET  IS THE FLOATING POINT DETERMINANT OF A.
C        IER  IS AN ERROR CODE WHICH IS
C               0 IF THE SYSTEM IS SOLVED SATISFACTORILY,
C               1 IF THERE ARE NOT ENOUGH MODULI AVAILABLE
C                 TO SOLVE THE SYSTEM,
C               2 IF THE COEFFICIENT MATRIX IS SINGULAR
C                 MOLULO EACH OF THE NOPRIM MODULI (IN WHICH
C                 CASE X AND DET ARE NOT COMPUTED),
C               3 IF ONE OR MORE OF THE INPUT INTEGER
C                 ARGUMENTS IS INCORRECT (I.E. N,M,IN,IM,
C                 IMPIN,IMIN1,NO2).
C      MULTL  IS THE MATRIX IN WHICH THE MULTILENGTH DIGITS
C             CF Y(I,J) AND DET A ARE STORED.  THE ELEMENTS
C             CF Y ARE STORED BY COLUMNS IN THE FIRST M * N
C             ROWS OF MULTL, AND DET A IS STORED IN THE
C             (M * N + 1)TH ROW.  LOW ORDER DIGITS ARE IN
C             COLUMN ONE OF MULTL, AND HIGHEST ORDER DIGITS
C             ARE IN COLUMN LCOUNT.  IT SHOULD BE DIMENSIONED
C             IMIN1 BY NOPRIM.
C     LCOUNT  IS THE CULUMN NUMBER IN MULTL WHICH CONTAINS
C             THE HIGHEST ORDER MULTILENGTH DIGITS.
C      ATEMP  IS THE IN BY IMPIN MATRIX OF TYPE INTEGER USED
C             BY EXACT TO HOLD THE AUGMENTED MATRIX (A,B)
C             IN RESIDUE FORM.
C         MM  IS THE LINEAR ARRAY USED BY SOLVE TO HOLD THE
C             MODULI WHICH WERE USED TO SOLVE THE SYSTEM OF
C             EQUATIONS.  IT SHOULD BE DIMENSIONED THE SAME
C             AS KPRIME.
C         RY  IS THE LINEAR ARRAY USED BY EXACT TO STORE THE
C             FLOATING-POINT ELEMENTS OF Y AND THE FLOATING-
C             POINT DETERMINANT OF A.  ITS DIMENSION SHOULD
C             BE IMIN1.
C          W  IS THE LINEAR ARRAY OF TYPE INTEGER USED RY
C             MLTLTH TO HOLD A MULTILENGTH NUMBER WHILE
C             PERFORMING MULTILENGTH ARITHMETIC OPERATIONS
C             CN IT.  IT SHOULD BE DIMENSIONED NO2.
C          V  IS THE LINEAR ARRAY OF TYPE INTEGER USED BY
C             CHECK FOR COMPARING THE VALUES OF TWO MULTILENGTH
C             NUMBERS.  IT SHOULD BE DIMENSIONED THE SAME AS W.
C
C
C CHECK INPUT PARAMETERS FOR CONSISTENCY
      IF(N .LE. 1 .OR. N .GT. IN) GO TO 80
      IF(M .LE. 0 .OR. M .GT. IM) GO TO 80
      IF(IMPIN .NE. IM+IN) GO TO 80
      IF(IMIN1 .NE. IM*IN+1) GO TO 80
      IF(NO2 .NE. 2*NOPRIM) GO TO 80
      NORES=M*N+1
      IB=10**NDIGIT
      SUMLOG=C.
C NZ IS THE NUMBER OF PRIMES FOR WHICH
C THE RESIDUE SYSTEM IS SINGULAR
      NZ=0
C IF IQUIT IS NOT EQUAL TO 0, THEN A IS SINGULAR
C MODULO EACH OF THE STORED PRIMES
      IQUIT=0
C IS WILL COUNT THE NUMBER OF PRIMES
C USED SUCCESSFULLY
      IS=1
C ICOUNT WILL COUNT THE NUMBER OF PRIMES TRIED
      ICOUNT=1
C COMPUTE A LCWER BOUND ON THE NUMBER
C OF REQUIRED MODULI
      CALL LOGBND(A,N,IN,B,M,IM,BOUND)
C COMPUTE RESIDUE OF A AND B
C AND STORE BCTH IN ATEMP
   10 PP=KPRIME(ICOUNT)
      P=PP
      DO 20 I=1,N
        DO 20 J=1,N
   20     ATEMP(I,J)=MOD(A(I,J),PP)
      DO 30 I=1,N
        DO 30 J=1,M
          JJ=N+J
   30     ATEMP(I,JJ)=MOD(B(I,J),PP)
      IFLAG=0
C SOLVE THE RESIDUE SYSTEM AX=B (MOD PP)
C FOR Y=A(ADJ)*B (MOD PP) AND DET (MOD PP)
C AND STORE RESULTS IN MULTL
      CALL SOLVE(ATEMP,MULTL,N,IN,MM,M,IMPIN,IMIN1,NOPRIM)
C IF IQUIT IS NOT EQUAL TO 0, THEN THE SYSTEM IS
C SINGULAR MOCULO EACH OF THE STORED PRIMES,
C AND HENCE, CANNOT BE SOLVED BY THIS PROGRAM.
C RETURN AN ERROR CODE OF 2.
      IF(IQUIT .EQ. 0) GO TO 40
      IER=2
      RETURN
C IF IFLAG IS NOT EQUAL TO 0, THEN A IS SINGULAR
C MODULO KPRIME(ICOUNT).  CHOOSE ANOTHER PRIME,
C I.E. KPRIME(ICOUNT+1), AND TRY TO SOLVE
C THE SYSTEM AGAIN.
   40 IF(IFLAG .NE. 0) GO TO 50
      SUMLOG=SUMLOG+ALOG(P)
C TEST TO SEE IF THE REQUIRED NUMBER
C OF PRIMES HAVE BEEN USED
      IF(SUMLCG .GE. BOUND) GO TO 60
      IS=IS+1
   50 ICOUNT=ICOUNT+1
C IF ALL PRIMES HAVE BEEN TRIED
C AND STILL ANOTHER IS REQUIRED,
C COMBINE RESULTS AND CHECK SOLUTION
      IF(ICOUNT .LE. NOPRIM) GO TO 10
      IS=IS-1
C COMBINE RESULTS BY CONVERSION
C TO SYMMETRIC MIXED-RADIX
   60 CALL MXRADX(MULTL,MM,RY,LCOUNT,NDIGIT,IMIN1,NOPRIM,
     1NO2,W)
C CHECK SOLUTION BY COMPUTING AY AND DB
      CALL CHECK(A,N,IN,B,M,IM,IER,MULTL,IMIN1,NOPRIM,
     1NO2,W,V)
      IF(IER .EQ. 1) RETURN
C COMPUTE THE SOLUTION X = (1/DET)*Y
      DET=RY(NORES)
      INDEX=0
      DO 70 J=1,M
        DO 70 I=1,N
          INDEX=INDEX+1
   70     X(I,J)=RY(INDEX)/DET
      RETURN
C RETURN ERROR CODE OF 3 FOR INCONSISTENT
C INPUT PARAMETERS
   80 IER=3
      RETURN
      END
      SUBROUTINE SOLVE(ATEMP,MULTL,N,IN,MM,M,IMPIN,IMIN1,
     1NOPRIM)
      DIMENSICN MM(NOPRIM),MULTL(IMIN1,NOPRIM),ATEMP(IN,IMPIN)
      INTEGER ATEMP,PP,RESIDU
      COMMON/MLEN/IB,PP,NZ,IS,IFLAG,IQUIT,NORES
C THIS SUBROUTINE SOLVES THE RESIDUE SYSTEM
C AX=B (MOD PP) FOR Y (MOD PP) AND DET (MOD PP).
      IDET=1
C FIND A PIVOTAL ELEMENT RELATIVELY PRIME TO PP
      MPN=M+N
      DO 80 J=1,N
        DO 10 I=J,N
          IF(MOD(ATEMP(I,J),PP) .NE. 0) GC TO 20
          IF(I .EQ. N) GO TO 100
   10     CONTINUE
C PERMUTE ROWS I AND J
   20   IF(I .EQ. J) GO TC 40
        IDET=-IDET
        DO 30 JJ=J,MPN
          ITEMP=ATEMP(J,JJ)
          ATEMP(J,JJ)=ATEMP(I,JJ)
   30     ATEMP(I,JJ)=ITEMP
C ACCUMULATE CETERMINANT
   40   IDET=IDET*ATEMP(J,J)
        IDET=MOD(IDET,PP)
C FIND INVERSE OF PIVOTAL ELEMENT
        IX=INVERS(ATEMP(J,J),PP)
C MULTIPLY ROW J BY INVERSE OF PIVOTAL ELEMENT
        DO 50 JJ=J,MPN
          ITEMP=ATEMP(J,JJ)*IX
   50     ATEMP(J,JJ)=MOD(ITEMP,PP)
C REPLACE LTH ROW BY LTH ROW-JTH ROW, (L NOT EQUAL J)
        DO 70 L=1,N
          IF(L .EQ. J) GO TO 70
          IK=ATEMP(L,J)
          DO 60 JJ=J,MPN
            ITEMP=ATEMP(J,JJ)*IK
            ITEMP=MOD(ITEMP,PP)
            ITEMP=ATEMP(L,JJ)-ITEMP
            ATEMP(L,JJ)=MOD(ITEMP,PP)
   60       CCNTINUE
   70     CONTINUE
   80   CONTINUE
C STORE SYMMETRIC RESIDUE DIGITS IN MULTL,
C AND MODULUS IN MM
      N1=N+1
      INDEX=0
      DO 90 J=N1,MPN
        DO 90 I=1,N
          INDEX=INDEX+1
          ITEMP=ATEMP(I,J)*IDET
   90     MULTL(INDEX,IS)=RESIDU(ITEMP,PP)
      MULTL(NCRES,IS)=RESIDU(IDET,PP)
      MM(IS)=PP
      RETURN
  100 NZ=NZ+1
      IFLAG=1
C TEST TO SEE IF ALL PRIMES HAVE FAILED
      IF(NZ .GT. NOPRIM-1) IQUIT=1
      RETURN
      END
      SUBROUTINE MXRADX(MULTL,MM,RY,LCOUNT,NDIGIT,IMIN1,
     1NOPRIM,NO2,W)
      DIMENSICN MM(NOPRIM),MULTL(IMIN1,NOPRIM),RY(IMIN1),
     1W(NO2)
      INTEGER RESIDU,W,PP
      DOUBLE PRECISION ACC,ACC1,ACC2,TEX
      COMMON/MLEN/IB,PP,NZ,IS,IFLAG,IQUIT,NORES
C SUBROUTINE MXRADX COMPUTES THE SYMMETRIC
C MIXED-RADIX DIGITS CF Y AND DET A FROM
C THEIR SYMMETRIC RESIDUE DIGITS.
      IF(IS .EQ. 1) GO TO 150
C COMPUTE SYMMETRIC MIXED-RADIX DIGITS
C AND STORE THEM IN MULTL
      DO 10 I=2,IS
        KK=I-1
        DO 10 J=I,IS
          IX=INVERS(MM(KK),MM(J))
          DO 10 K=1,NORES
            ITEMP=MULTL(K,J)-MULTL(K,I-1)
            ITEMP=ITEMP*IX
   10       MULTL(K,J)=RESIDU(ITEMP,MM(J))
C COMPUTE Y AND D FROM
C THEIR SYMMETRIC MIXED-RADIX DIGITS
C USING MULTILENGTH ARITHMETIC
      LCOUNT=C
   20 DO 140 I=1,NORES
        W(1)=1
        DO 30 K=2,IS
   30     W(K)=0
C COMPUTE Y(I)=(...(MULTL(I,IS)*MM(IS-1)+
C MULTL(I,IS-1))*MM(IS-2)+...+MULTL(I,2))
C MM(1)+MULTL(I,1)
        W(1)=W(1)*MULTL(I,IS)*MM(IS-1)
        CALL MLTLTH(NO2,W)
        J=IS
   40   J=J-1
        IF(J .LE. 1) GO TO 60
        W(1)=W(1)+MULTL(I,J)
        CALL MLTLTH(NO2,W)
        DO 50 K=1,IS
   50     W(K)=W(K)*MM(J-1)
        CALL MLTLTH(NO2,W)
        GO TO 40
   60   W(1)=W(1)+MULTL(I,J)
        CALL MLTLTH(NO2,W)
C STORE MULTILENGTH DIGITS OF Y(I)
C IN MULTL(I,J),J=1,IS
C STORE MULTILENGTH DIGITS OF DET A
C IN MULTL(NORES,J),J=1,IS
        DO 70 J=1,IS
   70     MULTL(I,J)=W(J)
C COMPUTE Y(I) IN FLOATING-PT. FROM MULTILENGTH DIGITS
        K=IS
   80   IF(W(K) .NE. 0) GO TO 90
        IF(K .EQ. 1) GO TO 100
        K=K-1
        GO TO 80
   90   IF(K .LE. 1) GO TC 100
        ACC=W(K)*IB+W(K-1)
        TEX=NCIGIT*(K-2)
        GO TO 110
  100   RY(I)=W(1)
        GO TO 130
  110   IF(K .LE. 2) GO TO 120
        ACC1=W(K-2)
        ACC2=10.D0**NDIGIT
        ACC=ACC+ACC1/ACC2
  120   ACC1=1.0D+1**TEX
        RY(I)=ACC*ACC1
  130   IF(K .LE. LCOUNT) GO TU 140
        LCOUNT=K
  140   CONTINUE
      RETURN
  150 DO 160 I=1,NORES
  160   RY(I)=MULTL(I,1)
      RETURN
      END
      SUBROUTINE MLTLTH(NC2,W)
      DIMENSICN W(NO2)
      INTEGER W,PP
      CCMMON/MLEN/IB,PP,NZ,IS,IFLAG,IQUIT,NORES
      IF(IS .EQ. 1) RETURN
      L=IS-1
C DISTRIBUTE THE DIGITS IN W SO THAT
C EACH ELEMENT OF W CONTAINS NCIGIT DIGITS
      DO 10 K=1,L
        W(K+1)=W(K)/IB+W(K+1)
   10   W(K)=-W(K)/IB*IB+W(K)
      K=IS
C ALL THE ELEMENTS OF W SHOULD HAVE THE SAME SIGN.
   20 IF(W(K))60,30,40
   30 IF(K .EC. 1) RETURN
      K=K-1
      GO TO 2C
   40 DO 50 K=1,L
        IF(W(K) .GE. 0) GO TO 50
        W(K)=W(K)+IB
        W(K+1)=W(K+1)-1
   50   CONTINUE
      RETURN
   60 DO 70 K=1,L
        IF(W(K) .LE. 0) GO TO 70
        W(K)=W(K)-IB
        W(K+1)=W(K+1)+1
   70   CONTINUE
      RETURN
      END
      SUBROUTINE CHECK(A,N,IN,B,M,IM,IER,MULTL,IMIN1,
     1NOPRIM,NO2,W,V)
      DIMENSICN V(NO2),MULTL(IMIN1,NOPRIM),A(IN,IN),
     1B(IN,IM),W(NO2)
      INTEGER W,V,A,B,PP
      COMMON/MLEN/IB,PP,NZ,IS,IFLAG,IQUIT,NORES
C SUBROUTINE CHECK CHECKS THE SOLUTION BY COMPUTING
C A*Y AND (DET A)*B AND COMPARING THE RESULTS.
C Y IS STORED BY COLUMNS IN MULTL
C DET IS STORED IN MULTL(NORES,I),I=1,IS
      LL=IS
      KK=IS+1
      DO 90 I=1,N
        INDEX=0
        DO 90 L=1,M
C MULTIPLY ROW I OF A BY COLUMN L OF Y
          DO 10 K=1,NO2
   10       W(K)=0
          IS=KK
          DO 40 J=1,N
            INDEX=INDEX+1
            JJ=A(I,J)/IB
            II=-JJ*IB+A(I,J)
            IF(LL .EQ. 1) GO TC 30
            DC 20 K=2,LL
              W(K)=W(K)+MULTL(INDEX,K)*II+MULTL(INDEX,K-1)*JJ
              CALL MLTLTH(NO2,W)
   20         CONTINUE
   30       W(1)=II*MULTL(INDEX,1)+W(1)
            W(KK)=JJ*MULTL(INDEX,LL)+W(KK)
            CALL MLTLTH(NC2,W)
            IF(IABS(W(IS)) .LT. IB) GO TO 40
            IF(IS .GE. NO2) GO TO 40
            IS=IS+1
            W(IS)=W(IS-1)/IB
            W(IS-1)=-W(IS)*IB+W(IS-1)
   40       CCNTINUE
C STORE THE PRODUCT IN V
          DO 50 K=1,IS
   50       V(K)=W(K)
C MULTIPLY B(I,L) BY CET AND STORE IN W
          JJ=B(I,L)/IB
          II=-JJ*IB+B(I,L)
          IF(LL .EQ. 1) GC TO 70
          DO 60 K=2,LL
   60       W(K)=MULTL(NORES,K)*II+MULTL(NORES,K-1)*JJ
   70     W(1)=II*MULTL(NCRES,1)
          W(KK)=JJ*MULTL(NORES,LL)+W(KK)
          CALL MLTLTH(NO2,W)
C TEST EQUALITY OF W AND V
          DO 80 J=1,IS
            IF(W(J) .NE. V(J)) GO TO 100
   80       CCNTINUE
   90       CONTINUE
C IF SOLUTION CHECKS, RETURN IER=C
C ELSE, RETURN IER=1
      IER=0
      IS=LL
      RETURN
  100 IER=1
      RETURN
      END
      FUNCTION INVERS(K,M)
C INVERS COMPLTES AN INVERSE OF K (MOD M)
C BY THE EUCLIDEAN ALGCRITHN
      I=K
      L=M
      J=1
      INVERS=C
   10 KK = I/L
      NN=MOD(I,L)
      IF(NN .EQ. 0) GO TO 20
      I=L
      L=NN
      NN=-KK*INVERS+J
      J=INVERS
      INVERS=NN
      GO TO 1C
   20 IF(L .GE. 0) GC TO 30
      INVERS=-INVERS
C RETURN A POSITIVE VALUE
   30 IF(INVERS .GE. C) RETURN
      INVERS=M+INVERS
      RETURN
      END
      INTEGER FUNCTICN RESIDU(K,M)
      RESIDU=MOD(K,M)
C THE FUNCTION RESIDU COMPUTES THE SYMMETRIC
C RESIDUE OF K (MCD M)
C I.E.  -M/2 LESS THAN RESIDU LESS THAN M/2
      IF(RESICU)10,20,30
   10 IF(2*RESIDU+M .GE. 0) RETURN
      RESIDU=RESIDU+M
   20 RETURN
   30 IF(-2*RESIDU+M .GE. 0) RETURN
      RESIDU=RESIDU-M
      RETURN
      END
      SUBROUTINE LOGBND(A,N,IN,B,M,IM,BOUND)
      DIMENSICN A(IN,IN),B(IN,IM)
      INTEGER A,B
C BOUND IS A LOWER BOUND FOR THE
C LOG OF THE PRODUCT OF THE MODULI
      BOUND=0.
      DO 20 I=1,N
        ALPHA=0.
        DO 10 J=1,N
          TEMP=A(I,J)
          TEMP=TEMP*TEMP
   10     ALPHA=ALPHA+TEMP
   20   BOUND=BOUND+ALOG(ALPHA)
      BOUND=BCUND/2.
      DO 30 J=1,M
        DO 30 I=1,N
          ALPHA=IABS(B(I,J))
          IF(ALPHA .EQ. 0.) GO TO 30
          BOUND=BOUND+ALOG(ALPHA)
   30     CONTINUE
   40 BOUND=BCUND+ALOG(2.)
      RETURN
      END
```

Remark on Algorithm 406 [F4]

Exact Solution of Linear Equations Using Residue Arithmetic [Jo Ann Howell, *Comm. ACM 14* (Mar. 1971), 180–184]

Jo Ann Howell [Rec'd 6/10/71]
Department of Computer Science, Yale University,
New Haven, CT 06520

The following statement should be added to subroutine *MXRADX* before the last *RETURN* statement (after statement 160):

LCOUNT = 1

Without this statement, *LCOUNT* is undefined whenever *IS* = 1.

Algorithm 407

DIFSUB for Solution of Ordinary Differential Equations [D2]

C.W. Gear [Recd. 29 Dec. 1969 and 10 April 1970]
Department of Computer Science, University of Illinois,
Urbana, IL 61801

Key Words and Phrases: differential equations, stiff differential equations
CR Categories: 5.17

Description

This subroutine integrates a set of up to N ordinary differential equations one step of length H, where H may be specified by the user, but is controlled by the subroutine to control the estimated error within a specified tolerance, if possible.

A multistep predictor corrector method is used whose order is automatically chosen by the subroutine as the integration proceeds. Either an Adams' method or methods suitable for stiff equations can be selected. The starting procedure is automatic and the information retained by the program about previous steps is stored in such a way as to make the interpolation to a nonmesh point straightforward. (See the description of the parameter Y in the subroutine.) The methods used are described from a mathematical point of view in the papers referenced in [1].

The programs may call on up to three subroutines. They are

DIFFUN(T, Y, DY)
PEDERV(T, Y, PW, M)
MATINV(PW, N, M, J)

The first, *DIFFUN*, must be provided always, and it must evaluate the derivatives of the dependent variables Y with respect to the independent variable T and place the results in DY. Y is dimensioned 8 by N, and the function values are in $Y(1, I)$ for I = 1 to N.

MATINV must be provided if stiff methods are requested. It should invert the matrix PW which is of size N by N. The (I, J) element of PW is stored in position $PW(\mathrm{I}+M^*(J-1))$, that is, PW is dimensioned as an M by M array. (The value of M used by *DIFSUB* is equal to the value of N used on the first call to *DIFSUB* when the user supplied parameter *JSTART* is 0.) If stiff methods are not used, *MATINV* is never called, so it is sufficient to provide a dummy subroutine to satisfy the loader if Adams' methods are used. The parameter J in *MATINV* should be set to $a + 1$ on return if the inversion is successful, -1 if the matrix is thought to be nearly singular.

If large systems of stiff equations are to be integrated, the inversion should be done in two stages. The call to *MATINV* after statement 300 should be replaced by a call of an LU factorization program; e.g. subroutine *DECOMP* [2, p. 68]. The set of statements

```
      DO400  I = 1, N
               ⋮
400   SAVE(9, I) = D
```

should be replaced by a call to the second stage of a Gaussian elimination program; e.g. subroutine *SOLVE* [2, p. 69]. The net result must be to solve the N by N linear system $PW{*}X = Y$ where the array Y is in *SAVE* $(I, 1)$, $I = N5 + 1$ to $N5 + N$ and the unknown array X is to be returned to *SAVE* $(9, I)$, $I = 1$ to N.

Tests have indicated that *MATINV* is called about ten times less frequently than the code represented by the above DO loop is executed. The cost of the change would be the overhead of the call to *SOLVE* which is independent of N; the saving due to the change would be about $5N^3/6$ multiplications and overhead operations each time that *DECOMP* is called instead of *MATINV*. The break point will depend on the computer and compiler used, but the change will lead to time saving on most computers when N exceeds about 5.

PERDERV is another optional subroutine called only if the method flag *MF* is set to 1. (See the description of the parameters.) If it is not used it can be replaced by a dummy subroutine to satisfy the loader. When used, it should compute the partial derivatives of the differential equations with respect to the dependent variables. The partial of the Ith equation with respect to the Jth variable should be stored in $PW(I+M^*(J-1))$. For example, if the two equations

$$y_1' = y_1 y_2 t^2$$

$$y_2' = -y_2^2 + 6y_1$$

were being solved by method type 1, *PEDERV* should compute as follows:

$PW(1) = Y(1, 2){*}T{**}2$
$PW(2) = 6.0$
$PW(1+M) = Y(1,1){*}T{**}2$
$PW(2+M) = -2.0{*}Y(1,2)$

If the first value of N used in a call to *DIFSUB* was 2, then the left hand sides of the last two assignment statements could better be written $PW(3)$ and $PW(4)$ for speed.

The *DOUBLE PRECISION* statement may be removed if a single precision version is required. If it is left in, all variables beginning with the letters A to H and Q to Z are double-precision floating-point, those beginning with P are single-precision floating-point. (In particular the matrix PW is computed and inverted in

Table I

```
METHOD TYPE  0

 ERROR RQ   PRESENT ERROR MAXIMUM ERRPR NO. STEPS FN EVALS MAT INVS   AVERAGE STEP         CURRENT TIME
0.100D-03     0.2806D-06     0.2806D-06      42      120        0      0.8751D-04          0.0105012848
0.100D-03     0.3361D-06     0.3361D-06     211      623        0      0.1607D-03          0.1001121513
0.100D-03 INTEGRATION ABANDONED WHEN NO. OF FN EVALUATIONS EXCEEDED 5000 AT    T =          0.8512205128
0.100D-04     0.1033D-07     0.1033D-07      50      143        0      0.7043D-04          0.0100714571
0.100D-04     0.5175D-06     0.5175D-06     218      645        0      0.1554D-03          0.1002143720
0.100D-04 INTEGRATION ABANDONED WHEN NO. OF FN EVALUATIONS EXCEEDED 5000 AT    T =          0.8434239890
0.100D-05     0.1863D-07     0.1863D-07      60      178        0      0.5724D-04          0.0101889615
0.100D-05     0.4525D-06     0.4525D-06     182      548        0      0.1833D-03          0.1004531167
0.100D-05 INTEGRATION ABANDONED WHEN NO. OF FN EVALUATIONS EXCEEDED 5000 AT    T =          0.9301080121
0.100D-06     0.6061D-08     0.6061D-08      77      228        0      0.4450D-04          0.0101451645
0.100D-06     0.4608D-07     0.4608D-07     203      574        0      0.1747D-03          0.1002597074
0.100D-06     0.4419D-07     0.4608D-07    1576     4588        0      0.2180D-03          1.0001159874
0.100D-06 INTEGRATION ABANDONED WHEN NO. OF FN EVALUATIONS EXCEEDED 5000 AT    T =          1.0686178261
0.100D-07     0.1550D-08     0.1550D-08      85      269        0      0.3772D-04          0.0101462928
0.100D-07     0.5784D-08     0.5784D-08     223      649        0      0.1542D-03          0.1001053613
0.100D-07     0.3162D-07     0.3162D-07    1396     3997        0      0.2502D-03          1.0000859952
0.100D-07 INTEGRATION ABANDONED WHEN NO. OF FN EVALUATIONS EXCEEDED 5000 AT    T =          1.1678642168
0.100D-08     0.2378D-09     0.2378D-09     106      323        0      0.3120D-04          0.0100780205
0.100D-08     0.3412D-09     0.3412D-09     249      710        0      0.1415D-03          0.1004798094
0.100D-08     0.6680D-10     0.3412D-09    1421     3923        0      0.2552D-03          1.0011817994
0.100D-08 INTEGRATION ABANDONED WHEN NO. OF FN EVALUATIONS EXCEEDED 5000 AT    T =          1.2618059881
0.100D-09     0.3961D-10     0.3961D-10     130      392        0      0.2576D-04          0.0100975698
0.100D-09     0.5042D-10     0.5042D-10     284      841        0      0.1199D-03          0.1008770225
0.100D-09     0.2036D-09     0.2036D-09    1420     4028        0      0.2483D-03          1.0000863103
0.100D-09 INTEGRATION ABANDONED WHEN NO. OF FN EVALUATIONS EXCEEDED 5000 AT    T =          1.2731856392
```

Table II

```
METHOD TYPE  1

 ERROR RQ   PRESENT ERROR MAXIMUM ERRPR NO. STEPS FN EVALS MAT INVS   AVERAGE STEP         CURRENT TIME
0.100D-03     0.3533D-04     0.3533D-04      39       87        6      0.1237D-03          0.0107625419
0.100D-03     0.7552D-04     0.7552D-04      66      155       10      0.6548D-03          0.1014987087
0.100D-03     0.7956D-04     0.7956D-04      98      231       13      0.4401D-02          1.0166403388
0.100D-03     0.9021D-04     0.9021D-04     126      303       16      0.3500D-01         10.6057160621
0.100D-03     0.1113D-03     0.1113D-03     146      346       20      0.3138D 00        108.5682945165
0.100D-03     0.3447D-04     0.1113D-03     157      380       24      0.2786D 01       1058.6211483326
0.100D-04     0.2202D-06     0.2202D-06      53      121        6      0.8564D-04          0.0103624070
0.100D-04     0.1369D-04     0.1369D-04     103      243       12      0.4156D-03          0.1009973518
0.100D-04     0.2649D-05     0.1369D-04     155      366       17      0.2748D-02          1.0056549946
0.100D-04     0.2096D-04     0.2096D-04     192      453       21      0.2275D-01         10.3073364831
0.100D-04     0.1209D-04     0.2096D-04     220      518       24      0.1957D 00        101.3505658414
0.100D-04     0.1671D-06     0.2096D-04     242      564       27      0.1995D 01       1125.4277070076
0.100D-05     0.9100D-07     0.9100D-07      70      179        7      0.5723D-04          0.0102436701
0.100D-05     0.2667D-05     0.2667D-05     110      262       12      0.4003D-03          0.1048869068
0.100D-05     0.2208D-05     0.2667D-05     168      405       15      0.2499D-02          1.0122667324
0.100D-05     0.2870D-05     0.2870D-05     216      523       20      0.1914D-01         10.0110785897
0.100D-05     0.2984D-05     0.2984D-05     252      616       25      0.1664D 00        102.4771283917
0.100D-05     0.1199D-05     0.2984D-05     283      693       29      0.1480D 01       1025.7769259724
0.100D-06     0.3261D-07     0.3261D-07      86      192        6      0.5245D-04          0.0100712902
0.100D-06     0.3073D-07     0.3261D-07     145      341       12      0.3049D-03          0.1039585812
0.100D-06     0.7833D-07     0.7833D-07     219      530       17      0.1921D-02          1.0178773065
0.100D-06     0.5889D-07     0.7833D-07     281      672       22      0.1539D-01         10.3413882438
0.100D-06     0.1096D-07     0.7833D-07     333      792       27      0.1297D 00        102.7375071042
0.100D-06     0.2588D-06     0.2588D-06     393      930       32      0.1097D 01       1020.6688599969
0.100D-07     0.4569D-08     0.4569D-08     113      254        6      0.3995D-04          0.0101467705
0.100D-07     0.5468D-08     0.5468D-08     171      409       11      0.2445D-03          0.1000020340
0.100D-07     0.1440D-07     0.1440D-07     269      654       17      0.1553D-02          1.0156394503
0.100D-07     0.1381D-07     0.1440D-07     350      837       23      0.1211D-01         10.1342196229
0.100D-07     0.1394D-08     0.1440D-07     413      978       29      0.1030D 00        100.7578631819
0.100D-07     0.7589D-09     0.1440D-07     473     1114       34      0.9005D 00       1003.1153401613
0.100D-08     0.6486D-09     0.6486D-09     148      327        6      0.3115D-04          0.0101868333
0.100D-08     0.1618D-08     0.1618D-08     218      529       10      0.1940D-03          0.1026173559
0.100D-08     0.5132D-08     0.5132D-08     345      849       13      0.1193D-02          1.0126228430
0.100D-08     0.5325D-08     0.5325D-08     447     1101       19      0.9167D-02         10.0934159546
0.100D-08     0.3716D-08     0.5325D-08     528     1297       24      0.7771D-01        100.7893655954
0.100D-08     0.2200D-08     0.5325D-08     593     1447       29      0.7008D 00       1014.0734315269
0.100D-09     0.7502D-10     0.7502D-10     184      422        6      0.2394D-04          0.0101011830
0.100D-09     0.2977D-09     0.2977D-09     273      666       11      0.1520D-03          0.1012469806
0.100D-09     0.5416D-09     0.5416D-09     439     1094       12      0.9156D-03          1.0016548361
0.100D-09     0.9022D-09     0.9022D-09     575     1417       19      0.7100D-02         10.0607268383
0.100D-09     0.6331D-09     0.9022D-09     681     1657       25      0.6118D-01        101.3735674223
0.100D-09     0.3515D-09     0.9022D-09     765     1856       32      0.5477D 00       1016.5477264277
```

single-precision to save space and time. Its accuracy only affects the rate of convergence of the method slightly.) All variables beginning with letters *I* to *N* are integers.

Because this program computes its own indices in the temporary storage array *SAVE* provided by the user in the call sequence, use of an optimizing compiler will reduce execution time considerably. (A version in which several more arrays of temporary storage must be provided in the call sequence has been compared with this. It uses these arrays to avoid computing indices, and consequently runs about 10 percent faster than this version on an IBM 360/91 using Fortran H, OPT = 2. However it is not as convenient for the user.)

Generally the problem should be scaled so that the square of any values of the solution that are to be considered nonzero when multiplied by the test constant *EPS* discussed below remain within the range of numbers representable in floating-point.

The following test problem proposed by F. T. Krogh (private communication) was run. Let *U* be the unitary matrix given by

$$U = \tfrac{1}{2}\begin{bmatrix} -1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 \end{bmatrix}.$$

Let B be the diagonal matrix

$$B = \begin{bmatrix} \beta_1 & 0 & 0 & 0 \\ 0 & \beta_2 & 0 & 0 \\ 0 & 0 & \beta_3 & 0 \\ 0 & 0 & 0 & \beta_4 \end{bmatrix}.$$

The differential equation

$$y' = Uz - UBUy$$

is integrated from $t = 0$ to $t = 10{,}000$ where

$$y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}, \quad z = \begin{bmatrix} w_1^2 \\ w_2^2 \\ w_3^2 \\ w_4^2 \end{bmatrix}, \quad w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} = Uy \text{ with } y(0) = \begin{bmatrix} -1 \\ -1 \\ -1 \\ -1 \end{bmatrix}.$$

The solution is

$$y = U \begin{bmatrix} \beta_1/(1 - (1+\beta_1)e^{\beta_1 t}) \\ \beta_2/(1 - (1+\beta_2)e^{\beta_2 t}) \\ \beta_3/(1 - (1+\beta_3)e^{\beta_3 t}) \\ \beta_4/(1 - (1+\beta_4)e^{\beta_4 t}) \end{bmatrix}.$$

Table III

```
METHOD TYPE  2

  ERROR RQ   PRESENT ERROR MAXIMUM ERRPR NO. STEPS FN EVALS MAT INVS    AVERAGE STEP          CURRENT TIME
 0.100D-03     0.3533D-04     0.3533D-04        39      111        6     0.9696D-04          0.0107625419
 0.100D-03     0.7552D-04     0.7552D-04        66      195       10     0.5205D-03          0.1014987087
 0.100D-03     0.7956D-04     0.7956D-04        98      283       13     0.3592D-02          1.0166402801
 0.100D-03     0.9020D-04     0.9020D-04       126      367       16     0.2890D-01         10.6050403654
 0.100D-03     0.1113D-03     0.1113D-03       146      426       20     0.2548D 00        108.5636816976
 0.100D-03     0.7901D-04     0.1113D-03       157      477       24     0.2391D 01       1140.4008035531
 0.100D-04     0.2202D-06     0.2202D-06        53      145        6     0.7146D-04          0.0103624070
 0.100D-04     0.1369D-04     0.1369D-04       103      291       12     0.3471D-03          0.1009973518
 0.100D-04     0.2649D-05     0.1369D-04       155      434       17     0.2317D-02          1.0056550430
 0.100D-04     0.2082D-04     0.2082D-04       192      537       21     0.1919D-01         10.3073282637
 0.100D-04     0.1207D-04     0.2082D-04       220      614       24     0.1654D 00        101.5463099866
 0.100D-04     0.1695D-05     0.2082D-04       242      677       28     0.1665D 01       1127.4919557261
 0.100D-05     0.9100D-07     0.9100D-07        70      207        7     0.4949D-04          0.0102436701
 0.100D-05     0.2667D-05     0.2667D-05       110      310       12     0.3383D-03          0.1048869068
 0.100D-05     0.2208D-05     0.2667D-05       168      465       15     0.2177D-02          1.0122667324
 0.100D-05     0.2870D-05     0.2870D-05       216      603       20     0.1660D-01         10.0110655660
 0.100D-05     0.2984D-05     0.2984D-05       252      716       25     0.1431D 00        102.4764579999
 0.100D-05     0.1201D-05     0.2984D-05       283      809       29     0.1268D 01       1025.9894923429
 0.100D-06     0.3261D-07     0.3261D-07        86      216        6     0.4663D-04          0.0100712902
 0.100D-06     0.3073D-07     0.3261D-07       145      389       12     0.2672D-03          0.1039585812
 0.100D-06     0.7833D-07     0.7833D-07       219      598       17     0.1702D-02          1.0178780838
 0.100D-06     0.5888D-07     0.7833D-07       281      760       22     0.1361D-01         10.3414927392
 0.100D-06     0.1096D-07     0.7833D-07       333      900       27     0.1142D 00        102.7387715393
 0.100D-06     0.2586D-06     0.2586D-06       393     1057       32     0.9655D 00       1020.5539661127
 0.100D-07     0.4569D-08     0.4569D-08       113      278        6     0.3650D-04          0.0101467707
 0.100D-07     0.5468D-08     0.5468D-08       171      453       11     0.2208D-03          0.1000020357
 0.100D-07     0.1440D-07     0.1440D-07       269      722       17     0.1407D-02          1.0156381221
 0.100D-07     0.1318D-07     0.1440D-07       350      927       23     0.1093D-01         10.1342171393
 0.100D-07     0.3246D-07     0.3246D-07       415     1092       30     0.9190D-01        100.3545926508
 0.100D-07     0.3758D-07     0.3758D-07       467     1232       35     0.8128D 00       1001.4240875968
 0.100D-08     0.6486D-09     0.6486D-09       148      351        6     0.2902D-04          0.0101868321
 0.100D-08     0.1618D-08     0.1618D-08       218      569       10     0.1803D-03          0.1026174257
 0.100D-08     0.5131D-08     0.5131D-08       345      901       13     0.1124D-02          1.0126198694
 0.100D-08     0.5324D-08     0.5324D-08       447     1177       19     0.8575D-02         10.0927076064
 0.100D-08     0.3716D-08     0.5324D-08       528     1393       24     0.7235D-01        100.7810184380
 0.100D-08     0.2195D-08     0.5324D-08       593     1563       29     0.6485D 00       1013.5805314175
 0.100D-09     0.7502D-10     0.7502D-10       184      446        6     0.2265D-04          0.0101011813
 0.100D-09     0.2977D-09     0.2977D-09       273      710       11     0.1426D-03          0.1012478756
 0.100D-09     0.5468D-09     0.5468D-09       439     1140       12     0.8792D-03          1.0022354698
 0.100D-09     0.8977D-09     0.8977D-09       575     1499       19     0.6745D-02         10.1108046311
 0.100D-09     0.6283D-09     0.8977D-09       681     1788       27     0.5688D-01        101.6984171224
 0.100D-09     0.3537D-09     0.8977D-09       765     2005       33     0.5102D 00       1023.0377820492
```

Tables I–III show the results for $\beta_1 = 1000$, $\beta_2 = 800$, $\beta_3 = -10$, $\beta_4 = .001$. The columns show the requested error (*EPS*), the error at the time of printing in the least accurate component, the maximum such error to date, the number of steps, number of calls to *DIFFUN* (i.e. function evaluations), number of calls to *MATINV*, average step size and the current value of *T*. The initial step was set to 10^{-4} and printing occurred at the first step to pass 10^i for $i = -2, -1, 0, 1, 2$, and 3. The three different methods were used ($MF = 0$, 1, and 2), but the integration was stopped if the number of function evaluations exceeded 5000, as it did with Adams' methods for this stiff problem. The problem was run for $EPS = 10^{-i}$ for $i = 4, 5, \cdots, 10$. (Warning; this problem is critically stable. If an error in excess of about 10^{-3} occurs, the solution of the perturbed problem may have a pole.) It should be noted that the results will depend slightly on the precision of the machine and the characteristics of the library program used for *MATINV*.

References

1. Gear, C. W. The automatic integration of ordinary differential equations. *Comm. ACM 14* (Mar. 1971), 176–179.
2. Forsythe, G. and Moler, C. *Computer Solution of Linear Algebraic Systems.* Prentice Hall, Englewood Cliffs, N. J., 1967.

Algorithm

```
      SUBROUTINE DIFSUB(N,T,Y,SAVE,H,HMIN,HMAX,EPS,MF,YMAX,ERROR,KFLAG,
     1                    JSTART,MAXDER,PW)
      DOUBLE PRECISION A,D,E,H,R,T,Y,R1,R2,BND,EPS,EUP,EDWN,ENQ1
     1 ,ENQ2,ENQ3,HMAX,HMIN,HNEW,HOLD,SAVE,TOLD,YMAX,ERROR,RACUM
C
C  THE PARAMETERS TO THE SUBROUTINE DIFSUB HAVE
C  THE FOLLOWING MEANINGS..
C
C    N         THE NUMBER OF FIRST ORDER DIFFERENTIAL EQUATIONS.  N
C                MAY BE DECREASED ON LATER CALLS IF THE NUMBER OF
C                ACTIVE EQUATIONS REDUCES, BUT IT MUST NOT BE
C                INCREASED WITHOUT CALLING WITH JSTART = 0.
C    T         THE INDEPENDENT VARIABLE.
C    Y         AN 8 BY N ARRAY CONTAINING THE DEPENDENT VARIABLES AND
C                THEIR SCALED DERIVATIVES.  Y(J+1,I) CONTAINS
C                THE J-TH DERIVATIVE OF Y(I) SCALED BY
C                H**J/FACTORIAL(J) WHERE H IS THE CURRENT
C                STEP SIZE. ONLY Y(1,I) NEED BE PROVIDED BY
C                THE CALLING PROGRAM ON THE FIRST ENTRY.
C                  IF IT IS DESIRED TO INTERPOLATE TO NON MESH POINTS
C                THESE VALUES CAN BE USED.  IF THE CURRENT STEP SIZE
C                IS H AND THE VALUE AT T + E IS NEFDED, FORM
C                S = E/H, AND THEN COMPUTE
C                              NQ
C                  Y(I)(T+E) =  SUM  Y(J+1,I)*S**J
C                              J=0
C    SAVE      A BLOCK OF AT LEAST 12*N FLOATING POINT LOCATIONS
C                USED BY THE SUBROUTINES.
C    H         THE STEP SIZE TO BE ATTEMPTED ON THE NEXT STEP.
C                H MAY BE ADJUSTED UP OR DOWN BY THE PROGRAM
C                IN ORDER TO ACHIEVE AN ECONOMICAL INTEGRATION.
C                HOWEVER, IF THE H PROVIDED BY THE USER DOES
C                NOT CAUSE A LARGER ERROR THAN REQUESTED, IT
C                WILL BE USED.  TO SAVE COMPUTER TIME, THE USER IS
C                ADVISED TO USE A FAIRLY SMALL STEP FOR THE FIRST
C                CALL.  IT WILL BE AUTOMATICALLY INCREASED LATER.
C    HMIN      THE MINIMUM STEP SIZE THAT WILL BE USED FOR THE
C                INTEGRATION.  NOTE THAT ON STARTING THIS MUST
C                MUCH SMALLER THAN THE AVERAGE H EXPECTED SINCE
C                A FIRST ORDER METHOD IS USED INITIALLY.
C    HMAX      THE MAXIMUM SIZE TO WHICH THE STEP WILL BE INCREASED
C    EPS       THE ERROR TEST CONSTANT.  SINGLE STEP ERROR ESTIMATES
C                DIVIDED BY  YMAX(I)  MUST BE LESS THAN THIS
C                IN THE EUCLIDEAN NORM. THE STEP AND/OR ORDER IS
C                ADJUSTED TO ACHIEVE THIS.
C    MF        THE METHOD INDICATOR. THE FOLLOWING ARE ALLOWED..
C                       0   AN ADAMS PREDICTOR CORRECTOR IS USED.
C                       1   A MULTI-STEP METHOD SUITABLE FOR STIFF
C                             SYSTEMS IS USED. IT WILL ALSO WORK FOR
C                             NON STIFF SYSTEMS.  HOWEVER THE USER
C                             MUST PROVIDE A SUBROUTINE PEDERV WHICH
C                             EVALUATES THE PARTIAL DERIVATIVES OF
C                             THE DIFFERENTIAL EQUATIONS WITH RESPECT
C                             TO THE Y'S.   THIS IS DONE BY CALL
C                             PEDERV(T,Y,PW,M). PW IS AN N BY N ARRAY
C                             WHICH MUST BE SET TO THE PARTIAL OF
C                             THE I-TH EQUATION WITH RESPECT
C                             TO THE J DEPENDENT VARIABLE IN PW(I,J).
C                             PW IS ACTUALLY STORED IN AN M BY M
C                             ARRAY WHERE M IS THE VALUE OF N USED ON
C                             THE FIRST CALL TO THIS PROGRAM.
C                       2   THE SAME AS CASE 1, EXCEPT THAT THIS
C                             SUBROUTINE COMPUTES THE PARTIAL
C                             DERIVATIVES BY NUMERICAL DIFFERENCING
C                             OF THE DERIVATIVES. HENCE PEDERV IS
C                             NOT CALLED.
C    YMAX      AN ARRAY OF  N LOCATIONS WHICH CONTAINS THE MAXIMUM
C                OF EACH Y SEEN SO FAR.  IT SHOULD NORMALLY BE SET TO
C                1 IN EACH COMPONENT BEFORE THE FIRST ENTRY. (SEE THE
C                DESCRIPTION OF EPS.)
C    ERROR     AN ARRAY OF  N ELEMENTS WHICH CONTAINS THE ESTIMATED
C                ONE STEP ERROR IN EACH COMPONENT.
```

```
C    KFLAG   A COMPLETION CODE WITH THE FOLLOWING MEANINGS..
C                      +1  THE STEP WAS SUCCESFUL.
C                      -1  THE STEP WAS TAKEN WITH H = HMIN, BUT THE
C                            REQUESTED ERROR WAS NOT ACHIEVED.
C                      -2  THE MAXIMUM ORDER SPECIFIED WAS FOUND TO
C                            BE TOO LARGE.
C                      -3  CORRECTOR CONVERGENCE COULD NOT BE
C                            ACHIEVED FOR H .GT. HMIN.
C                      -4  THE REQUESTED ERROR IS SMALLER T.AN CAN
C                            BE HANDLED FOR THIS PROBLEM.
C    JSTART  AN INPUT INDICATOR WITH THE FOLLOWING MEANINGS..
C                      -1  REPEAT THE LAST STEP WITH A NEW H
C                       0  PERFORM THE FIRST STEP.  THE FIRST STEP
C                            MUST BE DONE WITH THIS VALUE OF JSTART
C                            SO THAT THE SUBROUTINE CAN INITIALIZE
C                            ITSELF.
C                      +1  TAKE A NEW STEP CONTINUING FROM THE LAST.
C              JSTART IS SET TO NQ, THE CURRENT ORDER OF THE METHOD
C              AT EXIT.  NQ IS ALSO THE ORDER OF THE MAXIMUM
C              DERIVATIVE AVAILABLE.
C    MAXDER  THE MAXIMUM DERIVATIVE THAT SHOULD BE USED IN THE
C              METHOD.  SINCE THE ORDER IS EQUAL TO THE HIGHEST
C              DERIVATIVE USED, THIS RESTRICTS THE ORDER. IT MUST
C              BE LESS THAN 8 OR 7 FOR ADAMS OR STIFF METHODS
C              RESPECTIVELY.
C    PW      A BLOCK OF AT LEAST N**2 FLOATING POINT LOCATIONS.
C
      DIMENSION Y(8,N),YMAX(N),SAVE(10,N),ERROR(N),PW(N),
     1          A(8),PERTST(7,2,3)
C
C  THE COEFFICIENTS IN PERTST ARE USED IN SELECTING THE STEP AND
C  ORDER, THEREFORE ONLY ABOUT ONE PERCENT ACCURACY IS NEEDED.
C
      DATA PERTST /2.0,4.5,7.333,10.42,13.7,17.15,1.0,
     1              2.0,12.0,24.0,37.89,53.33,70.08,87.97.
     1              3.0,6.0,9.167,12.5,15.98,1.0,1.0,
     1              12.0,24.0,37.89,53.33,70.08,87.97,1.0,
     1              1.,1.,0.5,0.1667,0.04133,0.008267,1.0,
     1              1.0,1.0,2.0,1.0,.3157,.07407,.0139/
      DATA A(2) / -1.0/
      IRET = 1
      KFLAG = 1
      IF (JSTART.LE.0) GO TO 140
C
C  BEGIN BY SAVING INFORMATION FOR POSSIBLE RESTARTS AND CHANGING
C  H BY THE FACTOR R IF THE CALLER HAS CHANGED H.  ALL VARIABLES
C  DEPENDENT ON H MUST ALSO BE CHANGED.
C  E IS A COMPARISON FOR ERRORS OF THE CURRENT ORDER NQ. EUP IS
C  TO TEST FOR INCREASING THE ORDER, EDWN FOR DECREASING THE ORDER.
C  HNEW IS THE STEP SIZE THAT WAS USED ON THE LAST CALL.
C
 100  DO 110 I =.1,N
        DO 110 J = 1,K
 110      SAVE(J,I) = Y(J,I)
      HOLD = HNEW
      IF (H.EQ.HOLD) GO TO 130
 120  RACUM = H/HOLD
      IRET1 = 1
      GO TO 750
 130  NQOLD = NQ
      TOLD = T
      RACUM = 1.0
      IF (JSTART.GT.0) GO TO 250
      GO TO 170
 140  IF (JSTART.EQ.-1) GO TO 160
C
C  ON THE FIRST CALL, THE ORDER IS SET TO 1 AND THE INITIAL
C  DERIVATIVES ARE CALCULATED.
C
      NQ = 1
      N3 = N
      N1 = N*10
      N2 = N1 + 1
      N4 = N**2
      N5 = N1 + N
      N6 = N5 + 1
      CALL DIFFUN(T,Y,SAVE(N2,1))
      DO 150 I = 1,N
        N11 = N1 + I
 150    Y(2,I) = SAVE(N11,1)*H
      HNEW = H
      K = 2
      GO TO 100
C
C  REPEAT LAST STEP BY RESTORING SAVED INFORMATION.
C
 160  IF (NQ.EQ.NOOLD) JSTART = 1
      T = TOLD
      NQ = NQOLD
      K = NQ + 1
      GO TO 120
C
C  SET THE COEFFICIENTS THAT DETERMINE THE ORDER AND THE METHOD
C  TYPE.  CHECK FOR EXCESSIVE ORDER.  THE LAST TWO STATEMENTS OF
C  THIS SECTION  SET IWEVAL .GT.0 IF PW IS TO BE RE-EVALUATED
C  BECAUSE OF THE ORDER CHANGE, AND THEN REPEAT THE INTEGRATION
C  STEP IF IT HAS NOT YET BEEN DONE (IRET = 1) OR SKIP TO A FINAL
C  SCALING BEFORE EXIT IF IT HAS BEEN COMPLETED (IRET = 2).
C
 170  IF (MF.EQ.0) GO TO 180
      IF (NQ.GT.6) GO TO 190
      GO TO (221,222,223,224,225,226),NQ
 180  IF (NQ.GT.7) GO TO 190
      GO TO (211,212,213,214,215,216,217),NQ
 190  KFLAG = -2
      RETURN
C
C  THE FOLLOWING COEFFICIENTS SHOULD BE DEFINED TO THE MAXIMUM
C  ACCURACY PERMITTED BY THE MACHINE.  THEY ARE IN THE ORDER USED..
C
C  -1
C  -1/2,-1/2
C  -5/12,-3/4,-1/6
C  -3/8,-11/12,-1/3,-1/24
C  -251/720,-25/24,-35/72,-5/48,-1/120
C  -95/288,-137/120,-5/8,-17/96,-1/40,-1/720
C  -19087/60480,-49/40,-203/270,-49/192,-7/144,-7/1440,-1/5040
C
C  -1
C  -2/3,-1/3
C  -6/11,-6/11,-1/11
C  -12/25,-7/10,-1/5,-1/50
C  -120/274,-225/274,-85/274,-15/274,-1/274
C  -180/441,-58/63,-15/36,-25/252,-3/252,-1/1764
C
 211  A(1) = -1.0
      GO TO 230
 212  A(1) = -0.500000000
      A(3) = -0.500000000
      GO TO 230
 213  A(1) = -0.4166666666666667
      A(3) = -0.750000000
      A(4) = -0.1666666666666667
      GO TO 230
 214  A(1) = -0.375000000
      A(3) = -0.9166666666666667
      A(4) = -0.3333333333333333
      A(5) = -0.04166666666666667
      GO TO 230
 215  A(1) = -0.3486111111111111
      A(3) = -1.041666666666667
      A(4) = -0.4861111111111111
      A(5) = -0.1041666666666667
      A(6) = -0.008333333333333333
      GO TO 230
 216  A(1) = -0.3298611111111111
      A(3) = -1.141666666666667
      A(4) = -0.625000000
      A(5) = -0.1770833333333333
      A(6) = -0.0250000000
      A(7) = -0.001388888888888889
      GO TO 230
 217  A(1) = -0.3155919312169312
      A(3) = -1.235000000
      A(4) = -0.7518518518518519
      A(5) = -0.2552083333333333
      A(6) = -0.04861111111111111
      A(7) = -0.004861111111111111
      A(8) = -0.0001984126984126984
      GO TO 230
 221  A(1) = -1.000000000
      GO TO 230
 222  A(1) = -0.6666666666666667
      A(3) = -0.3333333333333333
      GO TO 230
 223  A(1) = - 0.5454545454545455
      A(3) = A(1)
      A(4) = -0.09090909090909091
      GO TO 230
 224  A(1) = -0.480000000
      A(3) = -0.700000000
      A(4) = -0.200000000
      A(5) = -0.020000000
      GO TO 230
 225  A(1) = -0.437956204379562
      A(3) = -0.8211678832116788
      A(4) = -0.3102189781021898
      A(5) = -0.05474452554744526
      A(6) = -0.003649635036496350 4
      GO TO 230
 226  A(1) = -0.4081632653061225
      A(3) = -0.9206349206349206
      A(4) = -0.4166666666666667
      A(5) = -0.0992063492063492
      A(6) = -0.0119047619047619
      A(7) = -0.0005668934240362 82
 230  K = NQ+1
      IDOUB = K
      MTYP = (4 - MF)/2
      ENQ2 = .5/FLOAT(NQ + 1)
      ENQ3 = .5/FLOAT(NQ + 2)
      ENQ1 = 0.5/FLOAT(NQ)
      PEPSH = EPS
      EUP = (PERTST(NQ,MTYP,2)*PEPSH)**2
      E = (PERTST(NQ,MTYP,1)*PEPSH)**2
      EDWN =(PERTST(NQ,MTYP,3)*PEPSH)**2
      IF (EDWN.EQ.0) GO TO 780
      BND = EPS*ENQ3/DFLOAT(N)
 240  IWEVAL = MF
      GO TO ( 250 , 680 ),IRET
C
C  THIS SECTION COMPUTES THE PREDICTED VALUES BY EFFECTIVELY
C  MULTIPLYING THE SAVED INFORMATION BY THE PASCAL TRIANGLE
C  MATRIX.
C
 250  T = T + H
      DO 260 J = 2,K
        DO 260 J1 = J,K
          J2 = K - J1 + J - 1
          DO 260 I = 1,N
 260        Y(J2,I) = Y(J2,I) + Y(J2+1,I)
C
C     UP TO 3 CORRECTOR ITERATIONS ARE TAKEN.  CONVERGENCE IS TESTED
C     BY REQUIRING CHANGES TO BE LESS THAN BND WHICH IS DEPENDENT ON
C     THE ERROR TEST CONSTANT.
C        THE SUM OF THE CORRECTIONS IS ACCUMULATED IN THE ARRAY
C     ERROR(I).  IT IS EQUAL TO THE K-TH DERIVATIVE OF Y MULTIPLIED
C     BY  H**K/(FACTORIAL(K-1)*A(K)), AND IS THEREFORE PROPORTIONAL
C     TO THE ACTUAL ERRORS TO THE LOWEST POWER OF H PRESENT. (H**K)
C
      DO 270 I = 1,N
 270    ERROR(I) = 0.0
      DO 430 L = 1,3
          CALL DIFFUN (T,Y,SAVE(N2,1))
C
C     IF THERE HAS BEEN A CHANGE OF ORDER OR THERE HAS BEEN TROUBLE
C     WITH CONVERGENCE, PW IS RE-EVALUATED PRIOR TO STARTING THE
C     CORRECTOR ITERATION IN THE CASE OF STIFF METHODS.  IWEVAL IS
```

```
C     THEN SET TO -1 AS AN INDICATOR THAT IT HAS BEEN DONE.
C
      IF (IWEVAL.LT.1) GO TO 350
        IF (MF.EQ.2) GO TO 310
        CALL PEDERV(T,Y,PW,N3)
        R = A(1)*H
        DO 280 I = 1,N4
 280      PW(I) = PW(I)*R
 290    N11 = N3 + 1
        N12 = N*N11 - N3
        DO 300 I = 1,N12,N11
 300      PW(I) = 1.0 + PW(I)
        IWEVAL = -1
        CALL MATINV(PW,N,N3,J1)
        IF (J1.GT.0) GO TO 350
        GO TO 440
 310    DO 320 I = 1,N
 320      SAVE(9,I) = Y(1,I)
        DO 340 J = 1,N
          R = EPS*DMAX1(EPS,DABS(SAVE(9,J)))
          Y(1,J) = Y(1,J) + R
          D = A(1)*H/R
          CALL DIFFUN(T,Y,SAVE(N6,1))
          DO 330 I = 1,N
            N11 = I + (J-1)*N3
            N12 = N5 + I
            N13 = N1 + I
 330        PW(N11) = (SAVE(N12,1) - SAVE(N13,1))*D
 340      Y(1,J) = SAVE(9,J)
        GO TO 290
 350      IF (MF.NE.0) GO TO 370
          DO 360 I = 1,N
            N11 = N1 + I
 360        SAVE(9,I) = Y(2,I) - SAVE(N11,1)*H
          GO TO 410
 370      DO 380 I = 1,N
            N11 = N5 + I
            N12 = N1 + I
 380        SAVE(N11,1) = Y(2,I) - SAVE(N12,1)*H
          DO 400 I = 1,N
            D = 0.0
            DO 390 J = 1,N
              N11 = I + (J-1)*N3
              N12 = N5 + J
 390          D = D + PW(N11)*SAVE(N12,1)
 400        SAVE(9,I) = D
 410      NT = N
        DO 420 I = 1,N
          Y(1,I) = Y(1,I) + A(1)*SAVE(9,I)
          Y(2,I) = Y(2,I) - SAVE(9,I)
          FRROR(I) = ERROR(I) + SAVE(9,I)
          IF  (DABS(SAVE(9,I)).LE.(BND*YMAX(I))) NT = NT - 1
 420      CONTINUE
        IF (NT.LE.0) GO TO 490
 430    CONTINUE
C
C   THE CORRECTOR ITERATION FAILED TO CONVERGE IN 3 TRIES.  VARIOUS
C   POSSIBILITIES ARE CHECKED FOR.  IF H IS ALREADY HMIN AND
C   THIS IS EITHER ADAMS METHOD OR THE STIFF METHOD IN WHICH THE
C   MATRIX PW HAS ALREADY BEEN RE-EVALUATED, A NO CONVERGENCE EXIT
C   IS TAKEN. OTHERWISE THE MATRIX PW IS RE-EVALUATED AND/OR THE
C   STEP IS REDUCED TO TRY AND GET CONVERGENCE.
C
 440  T = T - H
      IF ((H.LE.(HMIN*1.00001)).AND.((IWEVAL - MTYP).LT.-1)) GO TO 460
      IF ((MF.EQ.0).OR.(IWEVAL.NE.0)) RACUM = RACUM*0.2500
      IWEVAL = MF
      IRET1 = 2
      GO TO 750
 460  KFLAG = -3
 470  DO 480 I = 1,N
        DO 480 J = 1,K
 480      Y(J,I) = SAVE(J,I)
      H = HOLD
      NQ = NQOLD
      JSTART = NQ
      RETURN
C
C   THE CORRECTOR CONVERGED AND CONTROL IS PASSED TO STATEMENT 520
C   IF THE ERROR TEST IS O.K.,  AND TO 540 OTHERWISE.
C   IF THE STEP IS O.K. IT IS ACCEPTED. IF IDOUB HAS BEEN REDUCED
C   TO ONE,  A TEST IS MADE TO SEE IF THE STEP CAN BE INCREASED
C   AT THE CURRENT ORDER OR BY GOING TO ONE HIGHER OR ONE LOWER.
C   SUCH A CHANGE IS ONLY MADE IF THE STEP CAN BE INCREASED BY AT
C   LEAST 1.1.  IF NO CHANGE IS POSSIBLE IDOUB IS SET TO 10 TO
C   PREVENT FUTHER TESTING FOR 10 STEPS.
C   IF A CHANGE IS POSSIBLE, IT IS MADE AND IDOUB IS SET TO
C   NQ + 1     TO PREVENT FURTHER TESING FOR THAT NUMBER OF STEPS.
C   IF THE ERROR WAS TOO LARGE, THE OPTIMUM STEP SIZE FOR THIS OR
C   LOWER ORDER IS COMPUTED, AND THE STEP RETRIED.  IF IT SHOULD
C   FAIL TWICE MORE IT IS AN INDICATION THAT THE DERIVATIVES THAT
C   HAVE ACCUMULATED IN THE Y ARRAY HAVE ERRORS OF THE WRONG ORDER
C   SO THE FIRST DERIVATIVES ARE RECOMPUTED AND THE ORDER IS SET
C   TO 1.
C
 490  D = 0.0
      DO 500 I = 1,N
 500    D = D + (ERROR(I)/YMAX(I))**2
      IWEVAL = 0
      IF (D.GT.E) GO TO 540
      IF (K.LT.3) GO TO 520
      DO 510 J = 3,K
        DO 510 I = 1,N
 510        Y(J,I) = Y(J,I) + A(J)*ERROR(I)
 520  KFLAG = +1
      HNEW = H
      IF (IDOUB.LE.1) GO TO 550
      IDOUB = IDOUB - 1
      IF (IDOUB.GT.1) GO TO 700
      DO 530 I = 1,N
 530    SAVE(10,I) = ERROR(I)
      GO TO 700
 540  KFLAG = KFLAG - 2
      IF (H.LE.(HMIN*1.00001)) GO TO 740
      T = TOLD
      IF (KFLAG.LE.-5) GO TO 720
 550  PR2 = (D/E)**ENQ2*1.2
      PR3 = 1.E+20
      IF ((NQ.GE.MAXDER).OR.(KFLAG.LE.-1)) GO TO 570
      D = 0.0
      DO 560 I = 1,N
 560    D = D + ((ERROR(I) - SAVE(10,I))/YMAX(I))**2
      PR3 = (D/EUP)**ENQ3*1.4
 570  PR1 = 1.E+20
      IF (NQ.LE.1) GO TO 590
      D = 0.0
      DO 580 I. = 1,N
 580    D = D + (Y(K,I)/YMAX(I))**2
      PR1 = (D/EDWN)**ENQ1*1.3
 590  CONTINUE
      IF (PR2.LE.PR3) GO TO 650
      IF (PR3.LT.PR1) GO TO 660
 600  R = 1.0/AMAX1(PR1,1.E-4)
      NEWQ = NQ - 1
 610  IDOUB = 10
      IF ((KFLAG.EQ.1).AND.(R.LT.(1.1))) GO TO 700
      IF (NEWQ.LE.NQ) GO TO 630
      DO 620 I = 1,N
 620    Y(NEWQ+1,I) = ERROR(I)*A(K)/DFLOAT(K)
 630  K = NEWQ + 1
      IF (KFLAG.EQ.1) GO TO 670
      RACUM = RACUM*R
      IRET1 = 3
      GO TO 750
 640  IF (NEWQ.EQ.NQ) GO TO 250
      NQ = NEWQ
      GO TO 170
 650  IF (PR2.GT.PR1) GO TO 600
      NEWQ = NQ
      R = 1.0/AMAX1(PR2,1.E-4)
      GO TO 610
 660  R = 1.0/AMAX1(PR3,1.E-4)
      NEWQ = NQ + 1
      GO TO 610
 670  IRET = 2
      R = DMIN1(R,HMAX/DABS(H))
      H = H*R
      HNEW = H
      IF (NQ.EQ.NEWQ) GO TO 680
      NQ = NEWQ
      GO TO 170
 680  R1 = 1.0
      DO 690 J = 2,K
        R1 = R1*R
        DO 690 I = 1,N
 690      Y(J,I) = Y(J,I)*R1
      IDOUB = K
 700  DO 710 I = 1,N
 710    YMAX(I) = DMAX1(YMAX(I),DABS(Y(1,I)))
      JSTART = NQ
      RETURN
 720  IF (NQ.EQ.1) GO TO 780
      CALL DIFFUN (T,Y,SAVE(N2,1))
      R = H/HOLD
      DO 730 I = 1,N
        Y(1,I) = SAVE(1,I)
        N11 = N1 + I
        SAVE(2,I) = HOLD*SAVE(N11,1)
 730    Y(2,I) = SAVE(2,I)*R
      NQ = 1
      KFLAG = 1
      GO TO 170
 740  KFLAG = -1
      HNEW = H
      JSTART = NQ
      RETURN
C
C   THIS SECTION SCALES ALL VARIABLES CONNECTED WITH H AND RETURNS
C   TO THE ENTERING SECTION.
C
 750  RACUM = DMAX1(DABS(HMIN/HOLD),RACUM)
      RACUM = DMIN1(RACUM,DABS(HMAX/HOLD))
      R1 = 1.0
      DO 760 J = 2,K
        R1 = R1*RACUM
        DO 760 I = 1,N
 760      Y(J,I) = SAVE(J,I)*R1
      H = HOLD*RACUM
      DO 770 I = 1,N
 770    Y(1,I) = SAVE(1,I)
      IDOUB = K
      GO TO ( 130 , 250 , 640 ),IRET1
 780  KFLAG = -4
      GO TO 470
      END
```

Certification of Algorithm 407 [D2]

DIFSUB for Solution of Ordinary Differential Equations [C.W. Gear, *Comm. ACM 14* (Mar. 1971), 185–190]

Paul J. Nikolai [Recd. 1 Mar. 1972, 21 July 1972]
Aerospace Research Laboratories, Wright-Patterson AFB, OH 45433

The program used for this certification was keypunched directly from the printed Fortran algorithm [2]. The algorithm was implemented on a CDC 6600 computer using Fortran Extended, Version 3.0, Level 261A, OPT (Optimization Level) = 1. The *DOUBLE PRECISION* statement was deleted, and the built-in or intrinsic double precision function references were replaced by their single precision equivalents. Thus about 14 decimal digits (48 binary digits) were retained in the computations. An apparent bug in Fortran Extended required changing the statement

*N*4 = *N***2

following statement 140 to the equivalent statement

*N*4 = *N***N*.

The test problem given in [2] was coded, compiled, and executed to prepare three tables analogous to those given with the problem. The results are available from the present writer. In addition to the computed error [1, eq. (16)] returned by *DIFSUB*, the tables include the corresponding true error obtained by computing the Euclidean norm of the difference between the dependent variable vector returned by *DIFSUB* and that computed directly from the known solution of the test equation normalized by the infinity norm of the latter. The number of steps and average step size reflect these items over the appropriate printing interval and are not cumulative as the corresponding values apparently are in the tables with [2]. *H* was set initially to 10^{-4}, and *MAXDER* was set to 4. The tables compare quite favorably for the larger values of the requested error, the discrepancies over the smaller values being attributable to the drop in precision from 16 decimal digits on the IBM 360/91 to roughly 14 on the CDC 6600. The results from the stiff methods are truly impressive.

Several inconsistencies become apparent, unfortunately, if one should choose the value of *H*, the current step size, to be negative. For negative values of *H* the *IF* statement following statement 440, the *IF* statement following 540, and the arithmetic expression for *R* following 670 do not work correctly. We recommend replacing *H* and *HMIN* by *ABS*(*H*) and *ABS*(*HMIN*) in the *IF* statements and *HMAX*/*ABS*(*H*) by *ABS*(*HMAX*/*H*) in the expression for *R*.

DIFSUB with the above modifications has been incorporated into a general program for solving linear two-point boundary value problems for ordinary differential equations by the method of projections [3]. Past experience with the method of projections indicates that stiff equations arise often in applications. We currently feel that *DIFSUB* is our best hope for handling these problems.

References

1. Gear, C.W. The automatic integration of ordinary differential equations. *Comm. ACM 14* (Mar. 1971), 176–179.

2. Gear, C.W. Algorithm 407, DIFSUB for solution of ordinary differential equations. *Comm. ACM 14* (Mar. 1971), 185–190.

3. Guderley, Karl G., and Nikolai, Paul J. Reduction of two-point boundary value problems in a vector space to initial value problems by projection, *Numer. Math. 8* (1966), 270–289.

Algorithm 408
A Sparse Matrix Package (Part I) [F4]

John Michael McNamee (Recd. 26 Nov. 1969 and
15 July 1970)
York University, Downsview, Ontario, Canada

Key Words and Phrases: matrix, sparse matrix, matrix manipulation

CR Categories: 5.14

Description

It is frequently necessary to manipulate large sparse matrices, for example in electrical network problems. In such cases much time and memory space can be saved if only the nonzero elements are stored. A set of Fortran subroutines has been written for performing various operations on sparse matrices stored in compact form *in core*. Core storage requirement is reduced for any square matrix less than 66 percent dense. These subroutines have been tested on an IBM 360/50 using a "WATFOR" compiler.

Method of Storage. The nonzero elements are stored row-by-row (in one case column-by-column) in a single-dimensioned real array (A, say) while entries in an associated single-dimensioned integer array (M, say) contain the column indices of the corresponding elements. In addition the M-array contains certain control information.

The control information and column indices are packed into the M-array as indicated in Table I. By the "right half" of an integer word is meant the four least significant decimal digits, while the "left half" means the next four digits. Thus it is assumed that the computer word length is sufficient to contain at least an eight decimal digit integer (i.e. 28 bits including sign).

There should be no gaps in the M-array; thus, if the number of rows is odd, the first column index will appear in the right half of the word which contains "number of elements in last row" in its left half.

The total number of words needed in the M-array will be {4 + (number of rows) + (number of nonzero elements) + 1}/2 [rounded *down* to nearest integer].

Note that the number of rows or columns may be as high as 9999, while the number of elements stored may be $10^8 - 1$. (This is more than can fit into the core of any existing computer.)

As an example consider the matrix:

$$\begin{bmatrix} 1 & 0 & 2 \\ 2 & 3 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

The A-array would be as follows:

I:	1	2	3	4	5
$A(I)$:	1.	2.	2.	3.	1.

while the M-array would be:

I:	1	2	3	4	5	6	7
$M(I)$:	40003	5	20002	00001	10003	10002	20000

As a second example consider a 100 × 100 matrix having an average of three nonzero elements per row (as might arise in an electrical network problem). The A-array requires 300 words and the M-array $(4 + 100 + 300 + 1)/2 = 202$, for a total of 502. This is just over 5 percent of the area required to store the matrix in full.

Thirdly, consider a 100 × 100 matrix having an average of 66 nonzero elements per row. This requires a total of $6600 + (4 + 100 + 6600 + 1)/2 = 9952$ words, just short of the 10000 needed for full storage. Thus it is economical to use the sparse method of storage for square matrices having up to 66 percent nonzero elements. Time is also saved up to a certain degree of "nonsparseness."

List of Subroutines. The subroutines described here are listed in Table II.

Notes on the Subroutines

1. Using *RDSPMX* a sparse matrix may be input on cards as follows. The nonzero elements only are entered row-by-row in order of ascending column number with a sentinel (which may be any

Table I. Storage of Control Information and Column Indices

Word Number	Left Half	Right Half
$M(1)$	Number of rows	Number of columns
$M(2)$	←Number of elements stored→	
$M(3)$	Number of nonzero elements in row 1	Number of nonzero elements in row 2
$M(4)$	Number of nonzero elements in row 3	etc. . .
⋮		⋮
$M(I)$	⋮	Number of nonzero elements in last row
$M(I+1)$	Column index of first element stored	Column index of second element stored
$M(I+2)$	Column index of third element stored	etc. . .
⋮		⋮
$M(J)$	⋮	Column index of last element stored

number) after the end of each row. After each element, its column index is entered, the end-of-row sentinel having an index of the form 90000 + I where I is the row number. At the end of the whole matrix there is an additional sentinel (any number) with an index 99999.

The elements and column indices are entered four per card in the format 4 (E15.8, I5); i.e.

Columns		
	1–15	first element in E15.8 format
	16–20	first column index in I5 format
	21–35	second element
	36–40	second column index
	41–55	third element
	56–60	third column index
	61–75	fourth element
	76–80	fourth column index
		etc.

The elements are preceded by a control card containing in $I5$ format:

Columns		
	1–5	number of rows in A
	6–10	number of columns in A
	11–15	number of nonzero elements in A

For example the matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 0 & 5 \\ 0 & 0 & 0 \end{bmatrix}$$

would be entered thus:

	Col. 5 ↓	Col. 10 ↓	Col. 15 ↓					
Card #1:	3	3	5					
Cols.	1–15	20	21–35	40	41–55	60	61–75	76–80
Card							(say)	
2	1.0E00	1	2.0E00	2	3.0E00	3	0.0E00	90001
3	4.0E00	1	5.0E00	3	0.0E00	90002	0.0E00	90003
4	0.0E00	99999						

Note the third row must have an end-of-row sentinel.

The subroutine checks that this information agrees with the number of rows and elements actually entered, and that no column index exceeds the number of columns as stated. It also checks that column indices within a given row are entered in ascending order.

Table II. List of Sparse Matrix Subroutines. (X, MX) means "matrix with elements stored in X and control information and column indices stored in MX."

Name and Parameters	Function	Result stored in	See note number
RDSPMX(A,M, NA,NM)	Read from cards in non-packed form	(A,M)	1
ADSPMX(A, MA,B,MB,C, MC,NA,NM)	Add (A,MA) and (B,MB)	(C,MC)	2
MUSPMX(A, MA,B,MB,C, MC,NA,NM)	Postmultiply (A, MA) by the transpose of (B, MB)	(C,MC)	3
TRSPMX(A,M, AT,MT,NA, NM,IP,NP)	Transpose (A, M)	(AT,MT)	11
PERROW(A,M, AP,MP,IP,NA, NM,NP)	Permute rows of (A,M) according to permutation in IP	(AP,MP)	4(a), 5
PERCOL(A,M, AP, MP,IP,AT, MT,NA,NM, NP)	Permute columns of (A,M) according to permutation in IP	(AP,MP)	4(b), 5
ARSPMX(A,M, AN,MN,R,IR, IT,NA,NM)	Add R times row IR of (A, M) to row IT	(AN,MN)	
ACSPMX(A,M, AN,MN,R,IR, IT,NA,NM)	Add R times column IR of (A,M) to column IT	(AN,MN)	
MRSPMX(A,M, AN,MN,R,IR, NA,NM)	Multiply row IR of (A,M) by the scalar R	(AN,MN)	
MCSPMX(A,M, AN,MN,R,IC, NA,NM)	Multiply column IC of (A, M) by the scalar R	(AN,MN)	
ERSPMX(A,M, AN,MN,IR,JR, J,NA,NM,NP)	Exchange rows IR and JR of (A,M)	(AN,MN)	6
ECSPMX(A,M, AN,MN,IR,JR, J,NA,NM,NP)	Exchange columns IR and JR of (A,M)	(AN,MN)	6
MVSPMX(A,M, AN,MN,NA, NM)	Move (A,M)	(AN,MN)	
SMSPMX(A,M, AN,MN,S,NA, NM)	Multiply all elements of (A,M) by the scalar S	(AN,MN)	
RVSPMX(A,M, IR,V,N,NA, NM)	Extract row IR of (A,M)	V	7
CVSPMX(A,M, IC,V,AT,MT,N, NA NM,IP)	Extract column IC of (A,MA)	V	7, 11
INSPMX(A,M, N,NA,NM)	Read from back-up storage (Fortran unit N) in packed form	(A,M)	8
OTSPMX(A,M, N,NA,NM)	Write (A,M) onto back-up storage in packed form	Fortran unit N	8
WRSPMX(A,M, TIT,NA,NM)	Print (A,M) in edited form. TIT (10) contains 10 four-letter words describing (A,M).	Printer	9

Reading is via unit *IN*, which is set to 5. This may be changed by the user if necessary.

2. A subroutine to subtract (*B,MB*) from (*A,MA*) may be obtained by making the following minor changes to *ADSPMX*:

(i) Replace first line by *SUBROUTINE SUSPMX(A,MA,B,MB, C,MC,NA,NM)*

(ii) Replace statement number 9 by $T = A(JA) - B(JB)$

(iii) Replace 1st line after statement number 10, and also 2nd line after statement number 12, by $C(JA) = -B(JB)$

(iv) In statements 2, 4, 17 & 19 replace "... *ADSPMX* ..." by "... *SUSPMX* ..."

3. *MUSPMX* requires (*B,MB*) to be stored column-by-column. If it is not in this form the user must first call *TRSPMX*.

4(a). In *PERROW* old row *IP(I)* becomes new row *I*. *NP* is dimension of *IP* (equals number of rows in *A*).

4(b). In *PERCOL* old column *I* becomes new column *IP(I)*. *NP* is dimension of *IP* (equal number of columns). *AT,MT* are used internally.

5. The subroutine *ANTIP* (see ancillary subprograms below) may be used to invert the permutation *IP*.

6. *J* is used internally. *NP* is number of rows (for *ERSPMX*) or number of columns (for *ECSPMX*). It is the dimension of *J*.

7. The row (or column) extracted from (*A,M*) by *RVSPMX* (or *CVSPMX*) is stored in full in *V*; i.e. zero elements are included. *N* is dimension of *V* (equal number of columns or rows in *A*).

8. It is often possible to write more efficient subroutines for transfer to/from mass storage devices, using machine coding or special subroutines available on individual computer systems.

9. *WRSPMX* produces a printout of the nonzero elements of (*A,M*), row-by-row, five elements per line. Each element is followed by its column index. Each row is preceded by the heading "row number *I*". *TIT* is printed at top of each page.

10. In all the subroutines *NA,NM* are the dimensions of *A,M*, respectively. Checks are made that these limits are not exceeded.

11. The array *IP(NP)* in *TRSPMX* or *IP(N)* in *CVSPMX* is used internally.

12. All on-line writes are on unit *LP*, set to six at start of each subroutine. The user may change this number.

Ancillary Subprograms

(i) *FUNCTION IND(M,I,NM)* is used to extract the *I*th half-word from the array *M*. All the subroutines listed in Table II use this except *RDSPMX, INSPMX*.

(ii) *SUBROUTINE IPK(K,M,I,NM)* is used to pack *K* into the *I*th half-word of array *M*. All the subroutines listed in Table II use this except *MVSPMX, SMSPMX, RVSPMX, CVSPMX, INSPMX, OTSPMX, WRSPMX*. *NM* is the dimension of *M*.

(iii) *ANTIP(IP,AP,N)* may be used to invert a permutation array *IP* of *N* elements, storing the result in *AP*. For example suppose *IP* is (3,1,2), then *AP* will be (2,3,1). This may be useful in conjunction with *PERROW* and *PERCOL*. Note also that some subroutines call on others: namely, *ERSPMX* calls *PERROW*, *ECSPMX* calls *PERCOL*, *CVSPMX* calls *TRSPMX*, and *RVSPMX*.

Possible Alterations

(i) On machines having word lengths of 36 bits or more (such as IBM 7000 series), an integer contains over ten decimal digits. Hence by a slight change to *IND* or *IPK* a five digit integer may be stored in each half-word of the *M*-array. (No change to the main subroutines is required.) Thus matrices with up to 99999 rows or columns can be stored. At the cost of extra storage and changes to the main subroutines a similar effect can be obtained on the IBM 360 by using a full-word for each column index (then *IND* and *IPK* are not needed).

(ii) If the program does not have to handle matrices with more than 999 rows or columns, a further saving of space can be made by packing three (more on some machines) indices into each word of the *M*-array. This requires changes to most of the subroutines as well as to *IND* and *IPK*; e.g. in *MRSPMX* and *MCSPMX* second line before statement number 2 would be changed to $I1 = (5 + NRA + NEA)/3$.

(iii) On the IBM 360 the same effect as packing two column indices per word can be obtained more easily by declaring the *M*-array to be half-length (two bytes per word), and using one (half-length) word per index. Then subprograms *IND* and *IPK* are no longer required. This requires considerable changes to all the subroutines, but may save time.

Further Extensions. It is hoped to present subroutines for solving sparse systems of linear equations, and (perhaps) for solving eigen-problems of sparse matrices at a future date.

Algorithm:

```
      FUNCTION IND(M,I,NM)
C     ********************
C UNPACKS I*TH COLUMN INDEX.ARRAY M CONTAINS TWO 4-DIGIT
C INDICES PER WORD, LOWER INDEX IN UPPER FOUR DIGITS.
C
      DIMENSICN M(NM)
C J*TH WORD OF M CONTAINS I*TH INDEX.
      J       = (I+1)/2
C L IS 0 IF I EVEN, 1 IF I ODD.
      L       = I-(I/2)*2
C KT CONTAINS UPPER 4 DIGITS OF M(J).
      KT      = M(J)/10000
      IF (L) 1,1,2
    1 IND     = M(J)-KT*10000
      RETURN
    2 IND     = KT
      RETURN
      END
      SUBROUTINE IPK(K,M,I,NM)
C     ************************
C PACKS K (I*TH COLUMN INDEX) IN ARRAY M, WHICH WILL
C CONTAIN TWO 4-DIGIT INDICES PER WORD, LOWER INDEX
C UPPER 4 DIGITS.
C
      DIMENSICN M(NM)
      J       = (I+1)/2
      L       = I-(I/2)*2
      IF (L) 1,1,2
    1 M(J)    = M(J)+K
      RETURN
    2 M(J)    = M(J)+K*10000
      RETURN
      END
C ************************************************************
      SUBROUTINE ACSPMX(A,M,AN,MN,R,IR,IT,NA,NM)
C     ******************************************
C
C ADDS R TIMES COL IR TO COL IT OF MATRIX STORED IN A,
C PLACING RESULT IN AN.  M,MN CONTAIN CONTROL DATA AND COL
C INDICES FOR A,AN.
C NA IS DIMENSION OF A,AN. NM IS DIMENSION OF M,MN.
      REAL A,AN,R,AR
      INTEGER M,MN,IR,IT,NA,NM,I,NRA,NCA,L,JF,NIR,NIRA,J2,K,
     * K1,IFL,LP,J
      DIMENSICN A(NA),M(NM),AN(NA),MN(NM)
C LP IS UNIT NUMBER OF LINE PRINTER.
      LP      = 6
C CHECK THAT PARAMETERS WITHIN RANGE.
      IF (R.EC.0.0) WRITE (LP,14)
      IF (IR.LE.0.OR.IT.LE.0) GO TO 15
C CLEAR MN.
      DO 1 I = 1,NM
    1  MN(I) = 0
C CHECK THAT AN DOES NOT OVER-WRITE A.
      IF (M(1).EQ.0) GO TO 17
C UNPACK AND TRANSFER CONTROL DATA. NRA,NCA ARE NUMBERS OF
C ROWS,COLS IN A.
      NRA     = IND(M,1,NM)
      NCA     = IND(M,2,NM)
      MN(1)   = M(1)
C CHECK PARAMETERS WITHIN RANGE.
      IF (IR.GT.NCA.OR.IT.GT.NCA) GO TO 15
C L COUNTS ELEMENTS OF AN.
      L       = 1
      JF      = 4+NRA
C J COUNTS ELEMENTS TO END OF ROW (I-1).
      J       = 0
C I COUNTS ROWS OF A.
      DO 13 I  = 1,NRA
C NIR IS NUMBER IN NEW ROW.
       NIR      = 0
C NIRA IS NUMBER IN ROW I OF A.
       NIRA     = IND(M,4+I,NM)
       IF (NIRA.EQ.0) GO TO 12
```

```
C J2 COUNTS ELEMENTS TO END OF CURRENT ROW.
      J2      = J+NIRA
C J1 COUNTS ELEMENTS UP TO FIRST ONE IN CURRENT ROW.
      J1      = J+1
C PICK OUT ELEMENT IN COLUMN IR.
      AR      = 0.
      DO 2 K  = J1,J2
       K1     = IND(M,K+JF,NM)
       IF (K1.NE.IR) GO TO 2
       AR     = A(K)
    2 CONTINUE
C PICK OUT AND ALTER IF NECESSARY ELEMENT IN COL IT.
C TRANSFER REST OF ROW TO AN,MN. IFL SET TO 1 WHEN ELEMENT
C IN COL IT FCUND OR CREATED.
      IFL      = 0
      K        = J1-1
    3 K       = K+1
      K1      = IND(M,K+JF,NM)
      A1      = A(K)
      IF (K1.GE.IT) GO TO 7
C CHECK IF ARRAYS FULL.
    4 IF (L.LE.NA.AND.JF+L.LE.2*NM) GO TO 6
      WRITE (LP,5)
    5 FORMAT(21H IN ACSPMX ARRAY FULL)
      CALL EXIT
C COLUMN IT NCT YET REACHED.
    6 AN(L)   = A1
      CALL IPK(K1,MN,JF+L,NM)
      L       = L+1
      NIR     = NIR+1
      GO TO 10
    7 IF (K1.GT.IT) GO TO 8
C K1 EQUALS IT, I.E. THERE IS A NON-ZERO ELEMENT IN COL. IT
C OF ROW I.
      IFL     = 1
      A1      = AR*R+A(K)
      IF (A1.NE.0.0) GO TO 4
      GO TO 10
    8 IF (IFL.NE.0) GO TO 9
C K1 GREATER THAN IT AND ELEMENT IN COL IT HAS NOT YET
C BEEN FOUND, THUS COL IT HAS A ZERO ELEMENT.
      IFL     = 1
      K       = K-1
      A1      = AR*R
C A NEW ELEMENT IN COL IT IS CREATED IF A1 NOT ZERO.
      K1      = IT
      IF (A1.NE.0.0) GO TO 4
      GO TO 10
C K GREATER THAN IT AND ELEMENT IN COL IT ALREADY FOUND OR
C CREATED, JUST TRANSFER TO NEW ARRAY.
    9 A1      = A(K)
      GO TO 4
   10 IF (K.LT.J2) GO TO 3
      IF (IFL.NE.0.OR.AR.EQ.0.0) GO TO 12
      IF (L.LE.NA.AND.JF+L.LE.2*NM) GO TO 11
      WRITE (LP,5)
      CALL EXIT
   11 AN(L) = AR*R
      CALL IPK(IT,MN,JF+L,NM)
      L      = L+1
      NIR    = NIR+1
C END OF ROW.
   12 CALL IPK(NIR,MN,4+I,NM)
      J      = J2
   13 CONTINUE
C END OF LAST ROW.
     MN(2)  = L-1
     RETURN
C ERROR MESSAGES.
   14 FORMAT(20H IN ACSPMX R IS ZERO)
   15 WRITE (LP,16)
   16 FORMAT(32H IN ACSPMX IR OR IT OUT OF RANGE)
     CALL EXIT
   17 WRITE (LP,18)
   18 FORMAT(35H IN ACSPMX OUTPUT OVER-WRITES INPUT,
    * 34H OR INPUT HAS NO ROWS AND COLUMNS.)
     CALL EXIT
     END
C ************************************************************
     SUBROUTINE ADSPMX(A,MA,B,MB,C,MC,NA,NM)
C    ****************************************
C
C ADD TWO SPARSE MATRICES.
C
C A,B,C CONTAIN ELEMENTS OF FIRST,SECOND AND SUM MATRICES.
C MA,MB,MC, CCNTAIN CONTROL DATA AND COL. INDICES FOR A,B,C.
C NA IS DIMENSION OF A,B,C. NM IS DIMENSION OF MA,MB,MC.
C
     REAL A,B,C
     INTEGER MA,MB,MC,LP,NRA,NCA,NRB,NCB,JC,KA,KB,JB,KF,
    * I,KA1,JA,J1,J2,NOLD,NA,NM
     DIMENSICN A(NA),MA(NM),B(NA),MB(NM),C(NA),MC(NM)
C LP IS UNIT NUMBER OF LINE PRINTER.
     LP      = 6
C CLEAR MC.
     DO 1 I = 1,NM
   1  MC(I) = 0
C CHECK THAT C DOES NOT OVER-WRITE A OR B.
     IF (MA(1).EQ.0.OR.MB(1).EQ.0) GO TO 18
C UNPACK CONTROL DATA.  NRA,NCA ARE NUMBER OF ROWS,COLUMNS
C IN A.  NRB, NCB ARE ROWS, COLUMNS IN B.
     NRA     = IND(MA,1,NM)
     NCA     = IND(MA,2,NM)
     NRB     = IND(MB,1,NM)
     NCB     = IND(MB,2,NM)
C TEST FOR COPPATIBILITY.
     IF (NRA.EQ.NRB) GO TO 3
     WRITE (LP,2) NRA,NRB
   2 FORMAT(32H IN ADSPMX NUMBER OF ROWS IN A (,I4,
    * 37H) DCES NOT EQUAL NUMBER OF ROWS IN B(,I4,2H).)
     CALL EXIT
   3 IF (NCA.EQ.NCB) GO TO 5
     WRITE (LP,4) NCA,NCB
   4 FORMAT(31H IN ADSPMX NUMBER OF COLS IN A(,I4,
    * 36H) DCES NOT EQUAL NUM. OF COLS. IN B(,I4,2H).)
     CALL EXIT
C JC COUNTS ELEMENTS OF C.
   5 JC      = 1
C KA,KB ARE NUMBERS IN FIRST I ROWS OF A,B.
      KA      = 0
      KB      = 0
C KF IS NUMBER OF CONTROL DATA IN A,B OR C.
      KF      = 4+NRA
C JB COUNTS ELEMENTS OF B.
      JB      = 1
C I COUNTS ROWS OF A,B,C.
      DO 15 I=1,NRA
       KB       = KB+IND(MB,4+I,NM)
C NIRA IS NUMBER IN ROW I OF A.
       NIRA     = IND(MA,4+I,NM)
       IF (NIRA.EQ.0) GO TO 12
       KA1      = KA+1
       KA       = KA+NIRA
C JA COUNTS ELEMENTS OF A.
       DO 11 JA= KA1,KA
    6   J1       = IND(MA,JA+KF,NM)
C AT END OF B-ROW TRANSFER REST OF A-ROW.
        IF (JB.GT.KB) GO TO 7
        J2       = IND(MB,JB+KF,NM)
        IF (J1-J2) 7,9,10
C IF A-INDEX LESS THAN B-INDEX TRANSFER A-ELEMENT TO C.
    7   IF (JC.GT.NA) GO TO 16
        C(JC)   = A(JA)
    8   IF (JC+KF.GT.2*NM) GO TO 16
        CALL IPK(J1,MC,JC+KF,NM)
        JC       = JC+1
        GO TO 11
C IF A-INDEX EQUALS B-INDEX ADD ELEMENTS ,PLACE SUM IN C.
    9   T        = A(JA)+B(JB)
C IGNORE SUM ELEMENT IF ZERO.
        IF (T.EQ.0.0) GO TO 11
        IF (JC.GT.NA) GO TO 16
        C(JC)   = T
        JB       = JB+1
        GO TO 8
C IF A-INDEX GREATER THAN B-INDEX TRANSFER B-ELEMENT TO C.
   10   IF (JC.GT.NA) GO TO 16
        C(JC)   = B(JB)
        IF (JC+KF.GT.2*NM) GO TO 16
        CALL IPK(J2,MC,JC+KF,NM)
        JB       = JB+1
        JC       = JC+1
        GO TO 6
   11  CONTINUE
C END OF ROW CF A.  TRANSFER REST OF ROW OF B.
   12  IF (JB.GT.KB) GO TO 13
       IF (JC.GT.NA) GO TO 16
       C(JC) = B(JB)
       J2     = IND(MB,JB+KF,NM)
       IF (JC+KF.GT.2*NM) GO TO 16
       CALL IPK(J2,MC,JC+KF,NM)
       JC     = JC+1
       JB      = JB+1
       GO TO 12
   13  IF (I.GT.1) GO TO 14
       NOLD  = JC-1
C NIRC IS NUMBER IN ROW I OF C.
       NIRC  = JC-1
       GO TO 15
   14  NIRC   = JC-1-NOLD
       NOLD  = JC-1
   15  CALL IPK(NIRC,MC,4+I,NM)
C
C LAST ROW.   STORE CONTROL DATA IN MC.
      CALL IPK(NRA,MC,1,NM)
      CALL IPK(NCA,MC,2,NM)
      MC(2)  = JC-1
      RETURN
C ERROR MESSAGES.
   16 WRITE (LP,17)
   17 FORMAT(41H IN ADSPMX SPACE FOR SUM MATRIX EXCEEDED.)
      CALL EXIT
   18 WRITE (LP,19)
   19 FORMAT(33H IN ADSPMX SUM OVER-WRITES INPUT,
    * 34H OR INPUT HAS NO ROWS AND COLUMNS.)
      CALL EXIT
      END
C ****************************************************************
      SUBROUTINE ANTIP(N,P,AP)
C     *************************
C
C INVERT PERMLTATION IN P, PLACING RESULT IN AP.
C N IS NUMBER OF ELEMENTS IN P,AP.
C
      INTEGER P(N),AP(N)
C LP IS UNIT NUMBER OF LINE PRINTER.
      LP      = 6
C CHECK THAT CUTPUT DOES NOT OVER-WRITE INPUT.
      AP(1)   = 0
      IF (P(1).EQ.0) GO TO 4
      DO 3 I = 1,N
       J      = P(I)
       IF (J) 1,1,3
    1  WRITE (LP,2) I
    2  FORMAT(37H IN ANTIP PERMUTATION CONTAINS A NON-,
    * 28HPOSITIVE NUMBER IN POSITION ,I5)
       CALL EXIT
    3  AP(J) = I
      RETURN
C ERROR MESSAGES.
    4 WRITE (LP,5)
    5 FORMAT(42H IN ANTIP OUTPUT OVER-WRITES INPUT OR P(1),
    * 8H IS ZERO)
      CALL EXIT
      END
C ****************************************************************
      SUBROUTINE ARSPMX(A,M,AN,MN,R,IR,IT,NA,NM)
C     *****************************************
C
C ADD R TIMES ROW IR OF SPARSE MATRIX TO ROW IT.
C
C A,M CONTAIN ELEMENTS, COLUMN INDICES OF INPUT MATRIX.
C AN,MN CONTAIN ELEMENTS, COLUMN INDICES OF NEW MATRIX.
C NA IS DIMENSION OF A,AN.  NM IS DIMENSION OF M,MN.
```

```
C
      REAL A,AN,R
      INTEGER M,MN,IR,IT,I,NRA,NCA,NEA,NIRA,JR,I1,JT,J,JF,
     * J1,JT2,JN,J2,KR,KT,JT1,IT1,K,NA,NM,JTO,LP
      DIMENSICN A(NA),M(NM),AN(NA),MN(NM)
C LP IS UNIT NUMBER OF LINE PRINTER.
      LP      = 6
C CHECK PARAMETERS WITHIN RANGE.
      IF (IR.LE.0.OR.IT.LE.0) GO TO 23
C CLEAR MN.
      DO 1 I = 1,NM
    1  MN(I) = 0
C CHECK THAT AN DOES NOT OVER-WRITE A.
      IF (M(1).EQ.0) GO TO 25
C UNPACK CONTROL DATA, STORE IN MN.
      NRA     = IND(M,1,NM)
      NEA     = M(2)
      MN(1)   = M(1)
      DO 2 I = 1,NRA
       IF (I.EQ.IT) GO TO 2
       K      = IND(M,4+I,NM)
       CALL IPK(K,MN,4+I,NM)
    2  CONTINUE
C CHECK PARAMETERS WITHIN RANGE.
      IF (IR.GT.NRA.OR.IT.GT.NRA) GO TO 23
C JR,JT ARE NLMBERS OF ELEMENTS BELOW ROWS IR,IT.
      JR      = 0
      IF (IR.LE.1) GO TO 4
      IR1     = IR-1
      DO 3 I = 1,IR1
    3  JR     = JR+IND(M,4+I,NM)
    4 JT      = 0
      JF      = 4+NRA
      IF (IT.LE.1) GO TO 7
      IT1     = IT-1
      DO 5 I = 1,IT1
    5  JT     = JT+IND(M,4+I,NM)
C TRANSFER ELEMENTS BELOW ROW IT.
      DO 6 I = 1,JT
       AN(I) = A(I)
       J      = IND(M,JF+I,NM)
    6  CALL IPK(J,MN,JF+I,NM)
    7 JTO     = JT
C AUD R TIMES ROW IR TO ROW IT.
      JT2     = JT+IND(M,4+IT,NM)
      JT      = JT+1
C JN COUNTS ELEMENTS OF NEW MATRIX.
      JN      = JT
C NIRR IS NUMBER OF ELEMENTS IN ROW IR OF A.
      NIRR    = IND(M,4+IR,NM)
      IF (NIRR.EQ.0) GO TO 14
      J1      = JR+1
      J2      = JR+NIRR
      DO 13 I= J1,J2
C CHECK ARRAY LIMIT.
       IF (JN.LE.NA.AND.(JN+JF).LE.2*NM) GO TO 8
       WRITE (LP,21)
       CALL EXIT
    8  KR     = IND(M,JF+I,NM)
    9  IF (JT.GT.JT2) GO TO 12
       KT     = IND(M,JF+JT,NM)
       IF (KT.GE.KR) GO TO 10
       AN(JN)=A(JT)
       CALL IPK(KT,MN,JN+JF,NM)
       JN     = JN+1
       JT     = JT+1
       GO TO 9
   10  IF (KT.GT.KR) GO TO 12
       S      = A(JT)+R*A(I)
       IF (S.EQ.0.0) GO TO 11
       AN(JN)=S
       CALL IPK(KT,MN,JN+JF,NM)
       JN     = JN+1
   11  JT     = JT+1
       GO TO 13
   12  S      = R*A(I)
       IF (S.EQ.0.0) GO TO 13
       AN(JN)=S
       CALL IPK(KR,MN,JN+JF,NM)
       JN     = JN+1
   13 CONTINUE
C TRANSFER REST OF ROW IT.
   14 IF (JT.GT.JT2) GO TO 16
      IF (JN.LE.NA.AND.(JN+JF).LE.2*NM) GO TO 15
      WRITE (LP,21)
      CALL EXIT
   15 AN(JN) = A(JT)
      KT      = IND(M,JF+JT,NM)
      CALL IPK(KT,MN,JF+JN,NM)
      JN      = JN+1
      JT      = JT+1
      GO TO 14
C JT1 IS NUMBER IN ROW IT OF AN.
   16 JT1     = JN-1-JTO
      CALL IPK(JT1,MN,4+IT,NM)
C STORE ROWS ABOVE IT.
      IF (IT.EQ.NRA) GO TO 20
      IT1     = IT+1
      DO 19 I=IT1,NRA
C K IS NUMBER IN ROW I.
       K      = IND(M,4+I,NM)
       IF (K.EQ.0) GO TO 19
       JT     = JT2
       JT2    = JT+K
       JT1    = JT+1
       DO 18 J = JT1,JT2
        IF (JN.LE.NA.AND.(JN+JF).LE.2*NM) GO TO 17
        WRITE (LP,21)
        CALL EXIT
   17   AN(JN) = A(J)
        K       = IND(M,J+JF,NM)
        CALL IPK(K,MN,JN+JF,NM)
   18   JN      = JN+1
   19 CONTINUE
   20 MN(2)   = JN-1
      RETURN
C ERROR MESSAGES.
   21 FORMAT(21HOARRAY FULL IN ARSPMX)
   22 FORMAT(20H IN ARSPMX R IS ZERO)
   23 WRITE (LP,24)
   24 FORMAT(33H IN ARSPMX IR OR IT OUT OF RANGE.)
      CALL EXIT
   25 WRITE (LP,26)
   26 FORMAT(35H IN ARSPMX OUTPUT OVER-WRITES INPUT,
     * 34H OR INPUT HAS NO ROWS AND COLUMNS.)
      CALL EXIT
      END
C *************************************************************
      SUBROUTINE CVSPMX(A,M,IC,V,AT,MT,N,NA,NM,IP)
C     *******************************************
C EXTRACTS COL IC OF SPARSE MATRIX IN A,STORING RESULT IN V
C IN EXTENDED FORM, I.E. ALL ELEMENTS INCLUDING ZEROS ARE
C REPRESENTED.
C M CONTAINS COLUMN INDICES OF A.
C AT,MT ARE USED INTERNALLY.
C N IS DIMENSION OF V. NA IS DIMENSION OF A,AT.
C NM IS DIMENSION OF M,MT.
C IP US USED INTERNALLY BY TRSPMX.
C
      REAL A,AT,V
      INTEGER M,IC,MT,N,NA,NM,IP
      DIMENSICN A(NA),M(NM),AT(NA),MT(NM),V(N),IP(N)
      CALL TRSPMX(A,M,AT,MT,NA,NM,IP,N)
      CALL RVSPMX(AT,MT,IC,V,N,NA,NM)
      RETURN
      END
C *************************************************************
      SUBROUTINE ECSPMX(A,M,AN,MN,IR,JR,J,AT,MT,NA,NM,NP)
C     ****************************************************
C
C RESULT IN AN.
C EXCHANGE COLUMNS IR,JR OF SPARSE MATRIX IN A, STORING
C M,MN CONTAIN COLUMN INDICES OF A,AN. J IS USED INTERNALLY.
C AT,MT ARE USED BY PERCOL.
C NA IS DIMENSION OF A,AN. NM IS DIMENSION OF M,MN.
C NP IS DIMENSION OF J,AT,MT.
C
      REAL A,AN,AT
      INTEGER M,MN,IR,JR,J,NCA,NA,NM,NP,MT
      DIMENSICN A(NA),M(NM),AN(NA),
     * MN(NM),J(NP),AT(NP),MT(NP)
C SET UP PERMLTATION ARRAY J WITH IR,JR INTERCHANGED.
C NCA IS NUMBER OF COLUMNS IN A.
      NCA     = IND(M,2,NM)
      DO 1 I = 1,NCA
    1  J(I)   = I
      J(IR)   = JR
      J(JR)   = IR
C PERMUTE COLS OF A.
      CALL PERCOL(A,M,AN,MN,J,AT,MT,NA,NM,NP)
      RETURN
      END
C *************************************************************
      SUBROUTINE ERSPMX(A,M,AN,MN,IR,JR,J,NA,NM,NP)
C     *********************************************
C
C EXCHANGE ROWS IR,JR OF SPARSE MATRIX IN A,STORING RESULT
C IN AN. M, MN CONTAIN COLUMN INDICES OF A, AN.
C NA IS DIMENSION OF A,AN.  NM IS DIMENSION OF M,MN.
C NP IS DIMENSION OF J,WHICH IS USED INTERNALLY.
C
      REAL A,AN
      INTEGER M,MN,IR,JR,NRA,NA,NM,J,NP
      DIMENSICN A(NA),M(NM),AN(NA),MN(NM),J(NP)
C SET UP PERMUTATION ARRAY WITH IR,JR INTERCHANGED AND ALL
C  OTHER INTEGERS IN NATURAL ORDER.
C NRA IS NUMBER OF ROWS IN A.
      NRA     = IND(M,1,NM)
      DO 1 I = 1,NRA
    1  J(I)   = I
      J(IR)   = JR
      J(JR)   = IR
C PERMUTE ROWS OF A.
      CALL PERROW(A,M,AN,MN,J,NA,NM,NP)
      RETURN
C
      END
C *************************************************************
      SUBROUTINE OTSPMX(A,M,N,NA,NM)
C     ******************************
C WRITE SPARSE MATRIX IN A,M ON FORTRAN UNIT N (MASS STORAGE
C DEVICE).
C NA,NM ARE DIMENSIONS OF A,M.
C
      REAL A
      INTEGER M,N,NEA,NRA,NEM,NA,NM
      DIMENSICN A(NA),M(NM)
C NEA IS NUMBER OF ELEMENTS IN A.
      NEA     = M(2)
C NRA IS NUMBER OF ROWS IN A.
      NRA     = IND(M,1,NM)
C NEM IS NUMBER OF WORDS IN M.
      NEM     = (5+NRA+NEA)/2
      WRITE (N) NEM
      WRITE (N) (M(I),I=1,NEM)
      WRITE (N) (A(I),I=1,NEA)
      REWIND N
      RETURN
      END
C *************************************************************
      SUBROUTINE INSPMX(A,M,N,NA,NM)
C     ******************************
C READ SPARSE MATRIX FROM FORTRAN UNIT N (MASS STORAGE
C DEVICE).  STORE IN A, WITH COLUMN INDEX ARRAY IN M.
C NA,NM ARE DIMENSIONS OF A,M.
C
      REAL A
      INTEGER M,N,NEM,NEA
      DIMENSICN A(NA),M(NM)
C
C NEM IS NUMBER OF WORDS IN M.
      READ (N) NEM
      READ (N) (M(I),I=1,NEM)
```

```
C NEA IS NUMBER OF ELEMENTS IN A.
      NEA     = M(2)
      READ (N) (A(I),I=1,NEA)
      REWIND N
      RETURN
      END
C     ******************************************************
      SUBROUTINE MCSPMX(A,M,AN,MN,R,IC,NA,NM)
C     ***************************************
C
C MULTIPLIES COL IC OF SPARSE MATRIX IN A BY R, STORING
C RESULT IN AN.  M,MN CONTAIN COLUMN INDICES OF A,AN.
C NA IS DIMENSION OF A,AN. NM IS DIMENSION OF M,MN.
C
      INTEGER M,MN,IC,NRA,NCA,NEA,I1,I,J,JF,NIRA,J2,J1,K,IT,
     * NA,NM,L,NC
      DIMENSICN A(NA),M(NM),AN(NA),MN(NM)
      REAL A,AN,R
C LP IS UNIT NUMBER OF LINE PRINTER.
      LP      = 6
C CLEAR MN.
      DO 1 I = 1,NM
    1  MN(I) = 0
C CHECK THAT CUTPUT DOES NOT OVER-WRITE INPUT.
      IF (M(1).EQ.0) GO TO 9
C UNPACK AND TRANSFER CONTROL DATA. NRA,NCA,NEA ARE NUMBERS
C  OF ROWS, CCLS, ELEMENTS IN A.
      NRA     = IND(M,1,NM)
      NCA     = IND(M,2,NM)
      NEA     = M(2)
      DO 2 I = 1,2
    2  MN(I) = M(I)
C CHECK IC WITHIN RANGE.
      IF (IC.GT.NCA.OR.IC.LE.0) GO TO 7
C J COUNTS ELEMENTS TO END OF ROW (I-1) OF A.
      J       = 0
      JF      = 4+NRA
C L COUNTS ELEMENTS OF NEW MATRIX.
      L       = 1
C I COUNTS ROW OF A.
      DO 5 I= 1,NRA
C NIRA IS NUMBER OF ELEMENTS IN ROW I OF A.
       NIRA   = IND(M,4+I,NM)
C NIRAN IS NUMBER OF ELEMENTS IN ROW OF AN.
      NIRAN = NIRA
       IF (NIRA.EQ.0) GO TO 5
C J2 COUNTS ELEMENTS TO END OF ROW I OF A.
       J2     = J+NIRA
C J1 COUNTS ELEMENTS UP TO FIRST ONE IN ROW I OF A.
       J1     = J+1
C PROCESS ROW I OF A.
       DO 4 K  = J1,J2
        IT     = IND(M,JF+K,NM)
        IF (IT.EQ.IC) GO TO 3
C TRANSFER COLUMNS OTHER THAN IC.
        AN(L)  = A(K)
        I1     = IND(M,JF+K,NM)
        CALL IPK(I1,MN,JF+L,NM)
        L      = L+1
        GO TO 4
C MULTIPLY COL IC BY R.
    3   IF (R.EQ.0.0) NIRAN= NIRA-1
        IF (R.EQ.0.0) GO TO 4
        AN(L)  = R*A(K)
        I1     = IND(M,JF+K,NM)
        CALL IPK(I1,MN,JF+L,NM)
        L      = L+1
    4  CONTINUE
C END OF ROW I.
       J      = J2
       CALL IPK(NIRAN,MN,4+I,NM)
    5 CONTINUE
C END OF LAST ROW.
      IF (R.NE.0.0) GO TO 6
      M(2)    = L-1
    6 RETURN
C ERROR MESSAGES.
    7 WRITE (LP,8)
    8 FORMAT(26H IN MCSPMX IC OUT OF RANGE)
      CALL EXIT
    9 WRITE (LP,10)
   10 FORMAT(35H IN MCSPMX OUTPUT OVER-WRITES INPUT,
     * 34H OR INPUT HAS NO ROWS AND COLUMNS.)
      CALL EXIT
      END
C     ******************************************************
      SUBROUTINE MRSPMX(A,M,AN,MN,R,IR,NA,NM)
C     ***************************************
C
C MULTIPLIES ROW IR OF SPARSE MATRIX A BY R, STORING
C RESULT IN AN.  M,MN CONTAIN COLUMN INDICES OF A,AN.
C NA IS DIMENSION OF A,AN.  NM IS DIMENSION OF M,MN.
      REAL A,AN,R
      INTEGER M,MN,IR,NM,NRA,NEA,I1,I,
     * J,NIRA,J2,J1,K,NA,L,JF
      DIMENSICN A(NA),M(NM),AN(NA),MN(NM)
C LP IS UNIT NUMBER OF LINE PRINTER.
      LP      = 6
C CLEAR MN.
      DO 1 I = 1,NM
    1  MN(I) =0
C CHECK THAT CUTPUT DOES NOT OVER-WRITE INPUT.
      IF (M(1).EQ.0) GO TO 9
C UNPACK AND TRANSFER CONTROL DATA.
      NRA     = IND(M,1,NM)
      NEA     = M(2)
      DO 2 I = 1,2
    2  MN(I) = M(I)
C CHECK THAT IR IS WITHIN RANGE.
      IF (IR.GT.NRA.OR.IR.LT.1) GO TO 11
C J COUNTS ELEMENTS TO END OF ROW (I-1) OF A.
      J       = 0
C L COUNTS ELEMENTS OF NEW MATRIX.
      L       = 1
      JF      = 4+NRA
C I COUNTS ROWS OF A.
      DO 7 I = 1,NRA
C NIRA IS NUMBER IN ROW I OF A.
       NIRA   = IND(M,4+I,NM)
       CALL IPK(NIRA,MN,4+I,NM)
       IF (NIRA.EQ.0) GO TO 7
C J2 COUNTS ELEMENTS TO END OF ROW I OF A.
       J2     = J+NIRA
C J1 COUNTS ELEMENTS TO FIRST ONE IN ROW I OF A.
       J1     = J+1
       IF (I.EQ.IR) GO TO 4
C TRANSFER ROWS OTHER THAN I.
       DO 3 K  = J1,J2
        I1     = IND(M,JF+K,NM)
        CALL IPK(I1,MN,JF+L,NM)
        AN(L)  = A(K)
    3   L      = L+1
       GO TO 6
C MULTIPLY ROW IR BY R.
    4  DO 5 K  = J1,J2
        IF (R.EQ.0.0) GO TO 5
        I1     = IND(M,JF+K,NM)
        CALL IPK(I1,MN,JF+L,NM)
        AN(L)  = R*A(K)
        L      = L+1
    5   CONTINUE
    6  J       = J2
C END OF ROW I.
    7 CONTINUE
C END OF LAST ROW.
      IF (R.NE.0.0) GO TO 8
      M(2)    = L-1
      CALL IPK(0,MN,4+IR,NM)
    8 RETURN
C ERROR MESSAGES.
    9 WRITE (LP,10)
   10 FORMAT(36H IN MRSPMX OUTPUT OVER-WRITES INPUT ,
     * 34H OR INPUT HAS NO ROWS AND COLUMNS.)
      CALL EXIT
   11 WRITE (LP,12)
   12 FORMAT(26H IN MRSPMX IR OUT OF RANGE)
      CALL EXIT
      END
C ************************************************************
      SUBROUTINE MUSPMX(A,MA,B,MB,C,MC,NA,NM)
C     ***************************************
C
C MULTIPLY TWC SPARSE MATRICES.
C
C B MUST BE STORED BY COLUMNS, I.E. WE FORM C =
C A*(B TRANSPCSED).
C A,B,C CONTAIN FIRST,SECOND AND PRODUCT MATRICES RESPECT-
C IVELY.
C NA,MB,MC CONTAIN COLUMN INDICES OF FIRST SECOND AND
C PRODUCT MATRICES RESPECTIVELY.
C NA IS DIMENSION OF A,B,C.  NM IS DIMENSION OF MA,MB,MC.
C
      REAL A,B,C,S
      INTEGER MA,MB,MC,NRA,NCA,NRB,NCB,LC,KA,KAF,KBF,KB,KA1,
     * KB1,JB,JAM,JBM,J1,J2,LCM,NEC,K,IT,I,J,NA,NM,LP
      DIMENSICN A(NA),B(NA),C(NA),MA(NM),MB(NM),MC(NM)
C LP IS UNIT NUMBER OF LINE PRINTER.
      LP      = 6
C CLEAR MC.
      DO 1 I = 1,NM
    1  MC(I) = 0
C CHECK THAT C DOES NOT OVER-WRITE A OR B.
      IF (MA(1).EQ.0.OR.MB(1).EQ.0) GO TO 16
C UNPACK CONTROL INFORMATION.  NRA IS NUMBER OF ROWS IN A.
      NRA      = IND(MA,1,NM)
C NCA IS NUMBER OF COLS IN A.
      NCA      = IND(MA,2,NM)
C NRB,NCB ARE NUMBER OF ROWS AND COLUMNS IN B.
      NRB      = IND(MB,1,NM)
      NCB      = IND(MB,2,NM)
C TEST FOR COMPATIBILITY.
      IF (NCA.EQ.NRB) GO TO 3
      WRITE (LP,2)
    2 FORMAT(31H A AND B INCOMPATIBLE IN MUSPMX)
      CALL EXIT
C LC IS NUMBER OF ELEMENTS IN C.
    3 LC       = 1
C KAF,KBF ARE NUMBERS OF CONTROL DATA IN MA,MB.
      KAF      = 4+NRA
      KBF      = 4+NRB
C KA,KB ARE NUMBERS OF ELEMENTS IN  FIRST I ROWS OF A,B.
      KA       = 0
C NEC IS NUMBER OF ELEMENTS IN C.
      NEC     = 0
      DO 15 I = 1,NRA
       KB      = 0
C NIRA IS NUMBER IN ROW I OF A.
       NIRA   = IND(MA,4+I,NM)
```

```
C NIRC IS NUMBER IN ROW I OF C.
      NIRC   = 0
      IF (NIRA.EQ.0) GO TO 15
      KA1     = KA+1
      KA      = KA+NIRA
      DO 14 J = 1,NCB
C NIRB IS NUMBER IN ROW I OF B.
       NIRB    = IND(MB,4+J,NM)
       IF (NIRB.EQ.0) GO TO 14
       KB1     = KB+1
       KB      = KB+NIRB
C S WILL CONTAIN I,J ELEMENT OF C.
       S       = 0.
C JB COUNTS ELEMENTS IN B.
       JB      = KB1
       DO 8 JA=KA1,KA
        JAM     = JA+KAF
    5   JBM     = JB+KBF
        J1      = IND(MA,JAM,NM)
        J2      = IND(MB,JBM,NM)
        IF (J1-J2) 8,6,7
    6   S       = S+A(JA)*B(JB)
        IF (JB.EQ.KB) GO TO 9
        JB      = JB+1
        GO TC 8
    7   IF (JB.EQ.KB) GO TO 9
        JB      = JB+1
        GO TC 5
    8  CONTINUE
C IF ELEMENT ZERO DO NOT STORE.
    9  IF (S.EQ.0.0) GO TO 14
       IF (LC.LE.NA) GO TO 11
       WRITE  (LP,10)
   10  FORMAT(17HOSIZE OF PRODUCT ,
     *  25HMATRIX EXCEEDED IN MUSPMX)
       CALL EXIT
C STORE ELEMENT AND INDEX IN C,MC.
   11  C(LC)  = S
       LCM     = LC+KAF
       IF (LCM.LE.2*NM) GO TO 13
       WRITE (LP,12)
   12  FORMAT(40H SIZE OF INDEX MATRIX EXCEEDED IN MUSPMX)
       CALL EXIT
   13  CALL IPK(J,MC,LCM,NM)
       LC      = LC+1
       NIRC    = NIRC+1
   14 CONTINLE
      NEC    = NEC+NIRC
      CALL IPK(NIRC,MC,4+I,NM)
   15 CONTINUE
C STORE CONTRCL DATA IN MC.
      CALL IPK(NRA,MC,1,NM)
      CALL IPK(NCB,MC,2,NM)
      MC(2)  = NEC
      RETURN
C ERROR MESSAGE.
   16 WRITE (LP,17)
   17 FORMAT(36H IN MUSPMX PRODUCT OVER-WRITES INPUT,
     * 34H OR INPUT HAS NO ROWS AND COLUMNS.)
      CALL EXIT
      END
C **********************************************************
      SUBROUTINE MVSPMX(A,M,AN,MN,NA,NM)
C     **********************************
C
C MOVE SPARSE MATRIX IN A TO AN.
C M,MN CONTAIN COLUMN INDICES FOR A,AN.
C NA IS DIMENSION OF A,AN.  NM IS DIMENSION OF M,MN.
C
      REAL A,AN
      INTEGER M,MN,NEA,I,NRA,N,NA,NM
      DIMENSICN A(NA),M(NM),AN(NA),MN(NM)
C NEA IS NUMBER OF ELEMENTS IN A.
      NEA     = M(2)
C MOVE A.
      DO 1 I = 1,NEA
    1  AN(I) = A(I)
C NRA IS NUMBER OF ROWS IN A.
      NRA     = IND(M,1,NM)
C N IS NUMBER OF WORDS IN M.
      N       = (5+NRA+NEA)/2
C MOVE M.
      DO 2 I = 1,N
    2  MN(I) = M(I)
      RETURN
      END
C     **********************************
      SUBROUTINE PERCOL(A,M,AP,MP,IP,AT,MT,NA,NM,NP)
C     **********************************************
C PERMUTE COLUMNS OF A SPARSE MATRIX STORED IN A.
C M CONTAINS COLUMN INDICES OF A.
C AP,MP WILL CONTAIN ELEMENTS AND COLUMN INDICES OF RESULT.
C IP CONTAINS PERMUTATION--THAT IS OLD COLUMN I BECOMES NEW
C COLUMN IP(I).
C AT,MT WILL CONTAIN ROW OF A,M.
C NA IS DIMENSION OF A,AP. NM IS DIMENSION OF M,MP.
C NP IS DIMENSION OF AT,MT,IP.
C
      REAL A,AP,AT,A1
      INTEGER M,MP,IP,NR,NC,I,I1,NIR,K,
     *L,N,J,J1,LJ,N1,IFL,M1,NA,NM,NP,LP
      DIMENSICN A(NA),M(NM),AP(NA),
     * MP(NM),IP(NP),AT(NP),MT(NP)
C
C LP IS UNIT NUMBER OF LINE PRINTER.
  LP      = 6
C CLEAR MP.
      DO 1 I = 1,NM
    1  MP(I) = 0
C CHECK THAT CUTPUT DOES NOT OVER-WRITE INPUT.
      IF (M(1).EQ.0) GO TO 15
C UNPACK CONTROL INFORMATION.
      NR       = IND(M,1,NM)
      NC       = IND(M,2,NM)
C CHECK THAT IP(K) IS WITHIN RANGE.
      DO 2 K = 1,NC
       IF (IP(K).LE.0.OR.IP(K).GT.NC) GO TO 13
    2  CONTINUE
C TRANSFER CONTROL DATA TO MP.
      I2       = 4+NR
      DO 3 I   = 1,I2
       K       = IND(M,I,NM)
    3  CALL IPK(K,MP,I,NM)
C L COUNTS ELEMENTS ALREADY PERMUTED.
      L        = 0
C I IS ROW COUNTER.
      DO 12 I = 1,NR
       N       = IND(M,4+I,NM)
       IF (N.EQ.0) GO TO 12
C STORE ROW I IN AT WITH COLUMN INDICES IN MT.
       DO 4 J  = 1,N
        J1     = 4+NR+L+J
        K      = IND(M,J1,NM)
        MT(J)  = IP(K)
        LJ     = L+J
    4   AT(J)  = A(LJ)
       IF (N.EQ.1) GO TO 10
       N1      = N-1
C IFL WILL REMAIN 0 WHEN SORTING OF ROW I COMPLETE.
    5  IFL     = 0
C SORT ELEMENTS OF ROW I IN ORDER OF INCREASING COLUMN INDEX
       DO 9 J  = 1,N1
        IF (MT(J)-MT(J+1)) 9,6,8
C ERROR MESSAGE.
    6   WRITE (LP,7)
    7   FORMAT(26H IN PERCOL 2 INDICES EQUAL)
        CALL EXIT
    8   M1      = MT(J)
        MT(J)   = MT(J+1)
        MT(J+1)= M1
        A1      = AT(J)
        AT(J)   = AT(J+1)
        AT(J+1)= A1
        IFL     = 1
    9  CONTINLE
       IF (IFL.EQ.1) GO TO 5
C TRANSFER ROW I.
   10  DO 11 J = 1,N
        LJ      = L+J
        AP(LJ) = AT(J)
        J1      = LJ+4+NR
        K       = MT(J)
   11   CALL IPK(K,MP,J1,NM)
   12  L       = L+N
      RETURN
C ERROR MESSAGES.
   13 WRITE (LP,14)
   14 FORMAT(43H IN PERCOL PERM CONTAINS INDEX OUT OF RANGE)
      CALL EXIT
   15 WRITE (LP,16)
   16 FORMAT(35H IN PERCOL OUTPUT OVER-WRITES INPUT,
     * 34H OR INPUT HAS NO ROWS AND COLUMNS.)
      CALL EXIT
      END
C ***********************************************************
      SUBROUTINE PERROW(A,M,AP,MP,IP,NA,NM,NP)
C     ********************************************
C PERMUTE ROWS OF A SPARSE MATRIX STORED IN A.
C M CONTAINS COLUMN INDICES OF INPUT MATRIX A.
C AP CONTAINS ELEMENTS OF OUTPUT MATRIX .
C MP CONTAINS COLUMN INDICES OF OUTPUT MATRIX.
C IP CONTAINS PERMUTATION--I.E. OLD ROW IP(I) BECOMES NEW
C ROW I.
C NA IS DIMENSION OF A,AP. NM IS DIMENSION OF M,MP.
C NP IS DIMENSION OF IP.
C
      REAL A,AP
      INTEGER M,MP,IP,NR,NC,I,I1,NIR,I2,
     *K,LA,LM,N1,J,J1,I3,N2,NM,NA,NP,LP
      DIMENSICN A(NA),M(NM),AP(NA),MP(NM),IP(NP)
C
C LP IS UNIT NUMBER OF LINE PRINTER.
      LP      = 6
C CLEAR MP.
      DO 1 I = 1,NM
    1  MP(I) = 0
C CHECK THAT CUTPUT DOES NOT OVER-WRITE INPUT.
      IF (M(1).EQ.0) GO TO 10
C UNPACK CONTROL INFORMATION.
      NR      =  IND(M,1,NM)
      NC      =  IND(M,2,NM)
C RECORD NUMBERS OF ROWS ,COLUMNS AND ELEMENTS IN MP.
      DO 2 I = 1,2
    2  MP(I) = M(I)
C LA,LM COUNT ELEMENTS IN AP,MP.
      LA      = 1
      LM      = 5+NR
C PERMUTE ROWS.
      DO  9 I= 1,NR
       N1     = 0
```

```
C J IS OLD NUMBER OF NEW ROW I.
      J      = IP(I)
      K      = IND(M,4+J,NM)
      CALL IPK(K,MP,4+I,NM)
      IF (J.GT.NR.OR.J.LE.0) GO TO 3
      GO TO 5
C J OUT OF RANGE--GIVE ERROR MESSAGE.
    3 WRITE (LP,4) I
    4 FORMAT(38H IN PERROW PERM CONTAINS INDEX OUT OF
     * 17HRANGE IN POSITION,I3)
      CALL EXIT
C PICK OUT START AND END OF ROW J.
    5 IF (J.EQ.1) GO TO 7
      J1     = J-1
      DO 6 I3= 1,J1
    6  N1    =N1+IND(M,4+I3,NM)
C NIRJ IS NUMBER IN ROW J OF A.
    7 NIRJ   = IND(M,4+J,NM)
      IF (NIRJ.EQ.0) GO TO 9
      N2     = N1+NIRJ
      N1     = N1+1
C TRANSFER ROW J OF A,M TO ROW I OF AP,MP.
      DO 8 I3= N1,N2
       AP(LA)=  A(I3)
       K     = IND(M,4+NR+I3,NM)
       CALL IPK(K,MP,LM,NM)
       LA    = LA+1
    8  LM    = LM+1
C END OF LOOP ON I(ROW NUMBER).
    9 CONTINUE
      RETURN
C ERROR MESSAGE.
   10 WRITE (LP,11)
   11 FORMAT(35H IN PERROW OUTPUT OVER-WRITES INPUT,
     * 34H OR INPUT HAS NO ROWS AND COLUMNS.)
      CALL EXIT
      END
C ***************************************************************
      SUBROUTINE RDSPMX(A,M,NA,NM)
C     ******************************
C
C READS A SPARSE MATRIX FROM CARDS INTO ARRAY A,STORING
C COLUMN INDICES AND CONTROL DATA IN M.
C NA IS DIMENSION OF A, NM IS DIMENSION OF M.
C
      REAL A
      INTEGER NR,NC,NE,JE,JR,JF,K,MIN,M,
     * I,NIR,J,J1,J2,L1,L2,NA,NM,LIMF,IN,LP
      DIMENSICN A(NA),M(NM),AIN(4),MIN(4)
C IN IS UNIT NUMBER OF CARD READER.
      IN     = 5
C LP IS UNIT NUMBER OF LINE PRINTER.
      LP     = 6
C NR,NC,NE ARE NUMBERS OF ROWS,COLS,AND ELEMENTS IN A.
      READ (IN,1) NR,NC,NE
    1 FORMAT(3I5)
C JE,JR COUNT NUMBER OF ELEMENTS,ROWS.
      JE     = 1
      JR     = 1
      DO 2 I = 1,NM
    2  M(I)  = 0
      IERR   = 0
      LIMF   = 0
C JF IS NUMBER OF CONTROL DATA.
      JF     = 4+NR
C K COUNTS ELEMENTS WITHIN ROW.
      K      = 0
C AIN,MIN ARE ELEMENTS AND INDICES AS READ FROM CARD.
    3 READ (IN,4) (AIN(I),MIN(I),I=1,4)
    4 FORMAT(4(E15.8,I5))
      DO 10 I = 1,4
C CHECK FOR RCW-SENTINEL.
       IF (MIN(I).GE.90000) GO TO 9
C CHECK VALIDITY OF COLUMN-INDEX.
       IF (MIN(I).LE.NC) GO TO 6
       WRITE (LP,5) MIN(I),JE
    5  FORMAT(15H COLUMN INDEX (,I5,20H)GREATER THAN NUMBER,
     *  24H OF COL IN ELEMENT NUM. ,I5)
       IERR  = 1
C STORE ELEMENT.
    6  IF (JE.LE.NA.AND.(JE+JF).LE.2*NM) GO TO 8
       IF (LIMF.EQ.1) GO TO 10
       LIMF  = 1
       WRITE (LP,7)
    7  FORMAT(21HOARRAY FULL IN RDSPMX)
       GO TO 10
    8  A(JE) = AIN(I)
       CALL IPK(MIN(I),M,JE+JF,NM)
       JE    = JE+1
       K     = K+1
       GO TO 10
C CHECK FOR END-OF-MATRIX SENTINEL.
    9  IF (MIN(I).EQ.99999) GO TO 11
C RECORD NUMBER OF ELEMENTS IN ROW JR OF A.
       CALL IPK(K,M,4+JR,NM)
       K         = 0
       JR        = JR+1
   10 CONTINUE
C READ NEW CARD.
      GO TO 3
C AT END OF MATRIX CHECK NUMBER OF ROWS IS AS STATED.
   11 JR     = JR-1
      IF (JR.EQ.NR) GO TO 13
      WRITE (LP,12) JR,NR
   12 FORMAT(17H NUMBER OF ROWS (,I5,17H) DOES NOT EQUAL ,
     * 15HSTATED NUMBER (,I5,1H))
      IERR   = 1
C CHECK NUMBER OF ELEMENTS.
   13 JE     = JE-1
      IF (JE.EQ.NE) GO TO 15
      WRITE (LP,14) JE,NE
   14 FORMAT(21H NUMBER OF ELEMENTS (,I5,11H) DOES NOT ,
     * 21HEQUAL STATED NUMBER (,I5,1H))
      IERR   = 1
C CHECK ASCENDING ORDER OF INDICES.
   15 J      = JF
      DO 19 I= 1,JR
C NIR IS NUMBER IN ROW I OF A.
       NIR   = IND(M,4+I,NM)
       J2    = NIR+J
       J1    = J+2
       IF (NIR.LE.1) GO TO 18
       DO 17 K = J1,J2
        L1     = IND(M,K-1,NM)
        L2     = IND(M,K,NM)
        IF (L1.LT.L2) GO TO 17
        K1     = K-J1+2
        WRITE (LP,16) K1,I
   16   FORMAT( 9H ELEMENT ,I5,8H IN ROW ,I5,11H HAS WRONG
     *   12HCCLUMN INDEX)
        IERR   = 1
   17  CONTINUE
   18  J     = J2
   19 CONTINUE
C STORE CONTRCL DATA.
      CALL IPK(NR,M,1,NM)
      CALL IPK(NC,M,2,NM)
      M(2)   = NE
      IF (IERR.GE.1) CALL EXIT
C
      RETURN
      END
C     **********************************
      SUBROUTINE RVSPMX(A,M,IR,V,N,NA,NM)
C     *************************************
C EXTRACTS ROW IR OF SPARSE MATRIX IN A,STORING RESULT IN
C VECTOR V IN EXTENDED FORM, I.E. INCLUDING ZERO ELEMENTS.
C M CONTAINS COLUMN INDICES OF A.
C N,NA,NM ARE DIMENSIONS OF V,A,M.
C
      REAL A,V
      INTEGER M,IR,N,NRA,I,NIRS,IR1,JM,K,J,NIRA,NA,NM,LP
      DIMENSICN A(NA),M(NM),V(N)
C LP IS UNIT NUMBER OF LINE PRINTER.
      LP     = 6
C NRA IS NUMBER OF ROWS IN A.
      NRA    = IND(M,1,NM)
      IF (IR.GT.NRA) GO TO 2
C N IS NUMBER OF COLS IN A (EQUALS NUMBER OF ELEMENTS IN V).
      N      = IND(M,2,NM)
C CLEAR V.
      DO 1 I = 1,N
    1  V(I)  = 0.0
C NIRS WILL BE NUMBER OF ELEMENTS IN ROWS PRIOR TO IR.
      NIRS   = 0
      IF (IR-1) 2,6,4
    2 WRITE (LP,3)
    3 FORMAT(34H IN RVSPMX ROW NUMBER OUT OF RANGE)
      CALL EXIT
    4 IR1    = IR-1
      DO 5 I = 1,IR1
    5  NIRS  = NIRS+IND(M,4+I,NM)
    6 JM     = 4+NRA+NIRS
C NIRA IS NUMBER IN ROW IR.
      NIRA   = IND(M,4+IR,NM)
      IF (NIRA.LE.0) GO TO 8
C TRANSFER ELEMENTS OF ROW IR.
      DO 7 I = 1,NIRA
       K     = IND(M,JM+I,NM)
       J     = NIRS+I
    7  V(K)  = A(J)
    8 RETURN
      END
C     ************************************
      SUBROUTINE SMSPMX(A,M,AN,MN,S,NA,NM)
C     **************************************
C MULTIPLY SPARSE MATRIX IN A BY SCALAR S, STORING RESULT IN
C AN.
C M,MN CONTAIN COLUMN INDICES FOR A,AN.
C NA IS DIMENSION OF A,AN. NM IS DIMENSION OF M,MN.
C
      REAL A,AN,S
      INTEGER M,MN,NEA,I,NRA,N,NA,NM,LP
      DIMENSICN A(NA),M(NM),AN(NA),MN(NM)
C
C LP IS UNIT NUMBER OF LINE PRINTER.
      LP     = 6
C CHECK THAT CUTPUT DOES NOT OVER-WRITE INPUT.
      MN(1)  = 0
      IF (M(1).EQ.0) GO TO 5
      IF (S.EC.0.0) GO TO 3
C NEA IS NUMBER OF ELEMENTS IN A.
      NEA    = M(2)
C MULTIPLY A BY S.
      DO 1 I = 1,NEA
    1  AN(I) = A(I)*S
C NRA IS NUMBER OF ROWS IN A.
      NRA    = IND(M,1,NM)
C N IS NUMBER OF WORDS IN M.
      N      = (5+NRA+NEA)/2
```

```
C MOVE M.
      DO 2 I = 1,N
    2  MN(I) = M(I)
      RETURN
    3 MN(1)  = M(1)
      NRA    = IND(M,1,NM)
      K      =(5+NRA)/2
      DO 4 I = 2,K
    4  MN(I) = 0
      RETURN
C ERROR MESSAGE.
    5 WRITE (LP,6)
    6 FORMAT(35H IN SMSPMX OUTPUT OVER-WRITES INPUT,
     * 34H OR INPUT HAS NO ROWS AND COLUMNS.)
      CALL EXIT
      END
C ************************************************************
      SUBROUTINE TRSPMX(A,M,AT,MT,NA,NM,IP,NP)
C     *****************************************
C
C TRANSPOSE A SPARSE MATRIX IN A, STORING THE RESULT IN AT.
C M,MT CONTAIN COLUMN INDICES OF A,AT.
C NA IS DIMENSION OF A,AT. NM IS DIMENSION OF M,MT.
C IP(I) WILL BE NUMBEK OF ELEMENTS IN COLUMN  I OF A, ALSO
C IP(I) WILL BE POINTER TO FIRST ELEMENT IN ROW I OF AT.
C NP IS DIMENSION OF IP.
C
      INTEGER I,IC,IFC,IFR,IT,J1,IP
      REAL A,AT
      DIMENSICN A(NA),M(NM),AT(NA),MT(NM),IP(NP)
C LP IS UNIT NUMBER OF LINE PRINTER.
      LP     = 6
C CLEAR MT.
      DO 1 I = 1,NM
    1  MT(I) = 0
C CHECK THAT AT DOES NOT OVER-WRITE A.
      IF (M(1).EQ.0) GO TO 8
C UNPACK CONTROL INFORMATION. NR,NC,NE ARE NUMBERS OF ROWS,
C  COLUMNS ANC ELEMENTS IN A.
      NR     = IND(M,1,NM)
      NC     = IND(M,2,NM)
      NE     = M(2)
C CHECK FOR PCSSIBLE OVERFLOW OF MT.
      L      = 4+NC+NE
      IF (L.GT.2*NM) UO TO 10
C PACK NUMBER OF ROWS(NC), COLUMNS(NR) AND ELEMENTS OF AT.
      CALL IPK(NC,MT,1,NM)
      CALL IPK(NR,MT,2,NM)
      MT(2)  = M(2)
C IFR,IFC ARE NUMBER OF CONTROL DATA IN A,AT.
      IFR    = 4+NR
      IFC    = 4+NC
C CLEAR IP.
      DO 2 I = 1,NC
    2  IP(I) = 0
C COUNT NUMBER OF ELEMENTS IN EACH COLUMN OF A(ROW OF AT).
      DO 3 I = 1,NE
       K     = IND(M,IFR+I,NM)
    3  IP(K) = IP(K)+1
C PACK NUMBERS OF ELEMENTS IN ROWS OF AT.
      DO 4 J = 1,NC
       K     = IP(J)
    4  CALL IPK(K,MT,4+J,NM)
C SET UP POINTER TO FIRST ELEMENT IN ROW I OF AT.
       NICA1 = IP(1)
      IP(1)  = 1
      DO 5 I = 2,NC
       NICA  = IP(I)
       IP(I) = IP(I-1)+NICA1
    5  NICA1 = NICA
C PROCESS ROWS OF A. J1 IS POSITION OF FIRST ELEMENT OF
C CURRENT ROW OF A.
      J1     = 1
C I COUNTS ROWS OF A.
      DO 7 I = 1,NR
C NIRA IS NUMBER OF ELEMENTS IN CURRENT ROW OF A.
       NIRA  = IND(M,4+I,NM)
       IF (NIRA.EQ.0) GO TO 7
       J2    = J1+NIRA-1
       DO 6 J = J1,J2
C K IS COLUMN NUMBER OF J*TH ELEMENT IN A, I.E.ROW NUMBER
C  IN AT.
        K      = IND(M,IFR+J,NM)
C IT IS POSITION OF CURRENT ELEMENT IN AT.
        IT     = IP(K)
        AT(IT) = A(J)
        CALL IPK(I,MT,IT+IFC,NM)
    6   IP(K)  = IP(K)+1
    7  J1    = J2+1
      RETURN
C ERROR MESSAGES.
    8 WRITE (LP,9)
    9 FORMAT(27H IN TKSPMX AT OVER-WRITES A,
     * 34H OR INPUT HAS NC ROWS AND COLUMNS.)
      CALL EXIT
   10 WRITE (LP,11)
   11 FORMAT(27H IN TKSPMX MT WILL OVERFLOW)
      CALL EXIT
      END
C ************************************************************
      SUBROUTINE WRSPMX(A,M,TIT,NA,NM)
C     ********************************
C WRITE SPARSE MATRIX A.
C M CONTAINS COLUMN INDICES OF A.
C TIT CONTAINS DESCRIPTION OF A.
C NA,NM ARE DIMENSIONS OF A,M.
C
      REAL A,AOUT,TIT
      INTEGER M,P,I,L,NRA,JF,NIRA,J,J2,K2,K,KJ,MOUT,NA,NM,LP
      DIMENSICN A(NA),M(NM),TIT(10),AOUT(5),MOUT(5)
C
C LP IS UNIT NUMBER OF LINE PRINTER.
      LP     = 6
C P IS PAGE CCUNTER.
      P      = 1
C HEADING AND DESCRIPTION.
      WRITE (LP,1) (TIT(I),I=1,10),P
    1 FORMAT(23H1PRINTOUT OF SPARSE MAT,
     * 4HRIX.,13X,10A4,10X,5HPAGE ,I5//)
C L IS LINE CCUNTER.
      L      = 2
C NRA IS NUMBER OF ROWS IN A.
      NRA    = IND(M,1,NM)
      JF     = 4+NRA
C NIRS IS NUMBER IN RUWS ALREADY WRITTEN.
      NIRS   = 0
C I IS ROW COLNTER.
      DO 8 I = 1,NRA
C NIRA IS NUMBER IN ROW OF A.
       NIRA  = IND(M,4+I,NM)
       IF (NIRA.EQ.0) GO TO 8
C J COUNTS ELEMENTS WRITTEN.
       J     = 0
       IF (L.LT.51) GU TO 3
C AT END OF PAGE WRITE NEW HEADING ON NEXT PAGE,UPDATING
C PAGE NUMBER.
    2  P     = P+1
       WRITE (LP,1) (TIT(K),K=1,10),P
       L     = 2
    3  WRITE (LP,4) I
    4  FORMAT(12HOROW NUMBER ,I5//
     * 1X,5(4X,7HELEMENT,5X,3HCOL,5X))
       L     = L+4
C EXTRACT NEXT LINE OF OUTPUT.
    5  J2    = MINO(NIRA,J+5)
       K2    = J2-J
       DO 6 K  = 1,K2
        KJ     = K+J+NIRS
        MOUT(K)= IND(M,KJ+JF,NM)
    6   AOUT(K)= A(KJ)
       WRITE (LP,7) (AOUT(K),MOUT(K),K=1,K2)
    7  FORMAT(1X,5(E15.7,I5,4X))
       L     = L+1
       J     = J+5
       IF (J.GE.NIRA) GO TO 8
       IF (L-55) 5,2,2
    8 NIRS   = NIRS+NIRA
C LAST ROW WRITTEN.
      RETURN
      END
C ************************************************************
```

Remark on Algorithm 408 [F4]

A Sparse Matrix Package (Part I) [J.M. McNamee, *Comm. ACM 14* (Apr. 1971), 265–273]

Arthur H.J. Sale [Recd. 6 Aug. 1971]
Basser Computing Department, University of Sydney, Sydney, Australia

Key Words and Phrases: matrix, sparse matrix, matrix manipulation, Fortran standards
CR Categories: 514

There are a number of minor flaws in the presentation of Algorithm 408. The first concerns the liberal use of a subroutine *EXIT* not described in the Algorithm, nor to be found in the Fortran standard as an intrinsic procedure. Probably the use of this particular routine is self-evident (especially to IBM users), but it is difficult to justify using it when the *STOP* statement is available. The safest way to correct this flaw is to write a short program that scans the algorithm text replacing occurrences of *CALL EXIT* by *STOP*; by my count there are 25 of these. The alternative, of supplying a subroutine named *EXIT* has a trap: a subprogram must contain a *RETURN* (see [1, Sec. 8.4.1.1(5) of the standard]), so the routine must be (a) in nonstandard Fortran or machine code, or (b) something like:

```
SUBROUTINE EXIT
J=0
IF (J.EQ.0) STOP
RETURN
END
```

The other flaw occurs in the very last line of the algorithm: an *END* statement delimits a program (see [1, Sec. 3.2.2]), so that the comment following it must belong to another program segment (which does not have an *END* and is in error). The cure is simple: remove the comment.

There is also a minor criticism one might make of the efficiency of the subprograms *IND* and *IPK*, which are frequently called. In practice the advantage of using available Fortran versions will often outweigh the gain in speed possible by lapsing into assembly language, and therefore the following versions are offered as probably compiling to a more efficient code. They utilize the intrinsic function *MOD* (often compiled in-line) and remove needless computations and assignments.

```
  FUNCTION IND(M, I, NM)
  DIMENSION M(NM)
  J = (I+1) / 2
  IF (MOD(I, 2))1, 1, 2
1 IND = MOD(M(J), 10000)
  RETURN
2 IND = M(J) / 10000
  RETURN
  END

  SUBROUTINE IPK(K, M, I, NM)
  DIMENSION M(NM)
  J = (I+1) / 2
  IF (MOD(I, 2)) 1, 1, 2
1 M(J) = M(J) + K
  RETURN
2 M(J) = K*10000 + M(J)
  RETURN
  END
```

References

1. Fortran vs Basic Fortran. *Comm. ACM 7* (Oct. 1964), 590–625.

Remark on Algorithm 408 [F4]

A Sparse Matrix Package (Part I)
[John Michael McNamee, *Comm. ACM 14* (Apr. 1971), 265–273]

E.E. Lawrence [Recd. 1 February 1972, 12 March 1973]
Central Application Laboratory, Mullard Limited, New Road, Mitcham, Surrey CR4 4XY, England

The subroutines constituting Algorithm 408 were, with the exception of *MVSPMX* and *WRSPMX*, tested on an IBM 360/65 using CALL/360-0S. The author's alteration (iii) was introduced, i.e. declaration of the *M*-array to be half length. Other changes were introduced in order: (a) to make the algorithm more conversational in a time shared environment; and (b) to improve the speed of the sorting procedure in *PERCOL*.

The following deficiencies in the algorithm were noted

1. The dimensional parameters of *ACSPMX*, *ADSPMX*, and *MUSPMX* are incomplete. As an illustration of this consider the two matrices

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 0 & 0 & 4 \\ 0 & 0 & 3 & 0 \\ 0 & 2 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

each of which has four nonzero elements.

Then the sum matrix has eight such elements, and in general, for two matrices with n_1 and n_2 nonzero elements, the number of nonzero elements, n_3, in the sum matrix is in the range $0 \leq n_3 \leq n_1 + n_2$.

However in *ADSPMX* the condition used is $n_1 = n_2 = n_3$.

Similar arguments apply to *ACSPMX* and *MUSPMX*.

To correct this requires extensions to the parameter lists and dimension statements, and also it changes the conditional statements within the subroutines concerned.

This shows up with the CALL/360-0S system since the compiler performs subscript checking. It would not be evident on most compilers including the IBM Fortran IV G compiler. It is, however, bad practice to rely on default effects of compilers.

2. There are three, probably copying, errors in *MUSPMX* (page 270).

(i) Line 33 should be:
IF(NCA.EQ.NCB) GO TO 3

(ii) Line 55 should be:
DO 14 J = 1, NRB

(iii) Line 102 should be:
CALL IPK(NRB,MC,2,NM)

ACM Transactions on Mathematical Software, Vol. 3, No. 3, September 1977, Page 308

REMARK ON ALGORITHM 408

A Sparse Matrix Package (Part I) [F4]
[J.M. McNamee, *Comm. ACM 14*, 4 (1971), 265–273]

Paolo Sipala [Recd 10 October 1976]
Istituto di Elettrotecnica, University of Trieste, Trieste, Italy

The subroutines RDSPMX, ADSPMX, MUSPMX, TRSPMX, MVSPMX, and WRSPMX of ACM Algorithm 408 were tested after conversion to Basic Fortran, and the following corrections appear to be needed:

(1) In ADSPMX, the line after statement number 9 should be changed to

```
IF (T.EQ.0.0) GO TO 911
```

and before statement number 11 the following line should be inserted:

```
911   JB = JB + 1
```

(2) In TRSPMX, after statement number 5 the following line should be inserted:

```
J2 = 0
```

The error in ADSPMX showed up when adding two matrices containing elements with opposite values in corresponding positions, which should cancel; the error in TRSPMX was noted when transposing a matrix having a null first line.

ACM Transactions on Mathematical Software, Vol. 4, No. 3, September 1978, Pages 295.

REMARK ON ALGORITHM 408

A Sparse Matrix Package (Part 1) [F4]
[J.M. McNamee, *Comm. ACM 14,* 4 (April 1971), 265-273]

Fred Gustavson [Recd 25 January 1978]
T.J. Watson Research Center, IBM, Yorktown Heights, NY 10598

The subroutine TRSPMX of ACM Algorithm 408 was compared with Algorithm HALFP [1] and the following corrections appear to be needed:

(1) Before statement DO 5 . . . , insert the line

```
IF (NC.LE.1) GO TO 100
```

(2) After label 5 insert the line

```
100 J2 = 0
```

The need for correction (1) is required when the matrix is a column vector (NC = 1). The need for correction (2) was noted in [2] as TRSPMX fails when transposing a matrix with an empty first row.

REFERENCES

1. GUSTAVSON, F.G. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Software 4,* 3 (Sept. 1978), 250-269.
2. SIPALA, P., Remark on Algorithm 408. *ACM Trans. Math. Software 3,* 3 (Sept. 1977), 303.

and the following corrections appear to be needed:

(1) In ADSPMX, the line after statement number 9 should be changed to

```
IF (T.EQ.0.0) GO TO 911
```

and before statement number 11 the following line should be inserted:

```
911     JB = JB + 1
```

(2) In TRSPMX, after statement number 5 the following line should be inserted:

```
J2 = 0
```

The error in ADSPMX showed up when adding two matrices containing elements with opposite values in corresponding positions, which should cancel; the error in TRSPMX was noted when transposing a matrix having a null first line.

ACM Transactions on Mathematical Software, Vol. 4, No. 3, September 1978, Pages 295.

REMARK ON ALGORITHM 408

A Sparse Matrix Package (Part 1) [F4]
[J.M. McNamee, *Comm. ACM 14,* 4 (April 1971), 265–273]

Fred Gustavson [Recd 25 January 1978]
T.J. Watson Research Center, IBM, Yorktown Heights, NY 10598

The subroutine TRSPMX of ACM Algorithm 408 was compared with Algorithm HALFP [1] and the following corrections appear to be needed:

(1) Before statement DO 5 . . . , insert the line

```
IF (NC.LE.1) GO TO 100
```

(2) After label 5 insert the line

```
100 J2 = 0
```

The need for correction (1) is required when the matrix is a column vector (NC = 1). The need for correction (2) was noted in [2] as TRSPMX fails when transposing a matrix with an empty first row.

REFERENCES

1. GUSTAVSON, F.G. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Software 4,* 3 (Sept. 1978), 250–269.
2. SIPALA, P., Remark on Algorithm 408. *ACM Trans. Math. Software 3,* 3 (Sept. 1977), 303.

ACM Transactions on Mathematical Software, Vol. 6, No. 3, September 1980, Pages 456–457.

REMARK ON ALGORITHM 408

A Sparse Matrix Package (Part 1) [F4]
[J.M. McNamee, *Commun. ACM 14,* 4 (April 1971), 265–273]

U. Harms, H. Kollakowski, and G. Möller [Received 15 May 1978 and 15 August 1978; accepted 12 December 1979]
Regionales Rechenzentrum für Niedersachsen, Technische Universität Hannover, Wunstorfer Straße, D-3000 Hannover, West Germany

When implementing Algorithm 408 on a CDC Cyber 76-12 and a Cyber 73-16, the errors noted by Lawrence [2] are corrected. In ARSPMX the dimensional parameters were incomplete and have been completed. Thus it is possible to add, for example, two sparse matrices having different numbers of nonzero elements.

There is another severe error in ADSPMX, as pointed out by Sipala [3]: when adding two elements whose sum is zero, ADSPMX gives an incorrect result. For example, when

$$A = \begin{bmatrix} 1 & 0 & 2 \\ 2 & 3 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \qquad B = \begin{bmatrix} -1 & 0 & 2 \\ -2 & 3 & 0 \\ 0 & 1 & 0 \end{bmatrix},$$

then the result of $A + B$ by ADSPMX is

$$C = \begin{bmatrix} -1 & 0 & 4 \\ -2 & 6 & 0 \\ 0 & 2 & 0 \end{bmatrix},$$

not the correct sum.

The necessary changes in ADSPMX are

(1) line 73: IF (T.EQ.0.0) GO TO 30
(2) before line 86: 30 CONTINUE
JB = JB + 1

Then the zero sum is ignored. For better definition, all elements of C are set to zero. If all elements of C are zero, then a message is printed later in the code.

There are some minor changes in some other subroutines; for example, in ARSPMX, MCSPMX, and MRSPMX the variable NEA is set but never used. The same thing happens to the variable NC in PERROW. For this reason NEA and NC are eliminated in the subroutines presented here.

In MCSPMX the statement M(2) = L − 1 (1 line before statement 6) should be changed to MN(2) = L − 1, and in MRSPMX the statement M(2) = L − 1 (2 lines before statement 8) should be changed to MN(2) = L − 1.

Moreover, some errors in TRSPMX, pointed out by Sipala [3] and Gustafson [1], have been corrected.

REFERENCES

1. GUSTAFSON, F. Remark on Algorithm 408. *ACM Trans. Math. Softw. 4*, 3 (Sept. 1978), 295.
2. LAWRENCE, E.E. Remark on Algorithm 408, A sparse matrix package (part I). *Commun. ACM 16*, 9 (Sept. 1973), 578.
3. SIPALA, P. Remark on Algorithm 408, A sparse matrix package (part I). *ACM Trans. Math. Softw. 3*, 3 (Sept. 1977), 303.

ALGORITHM

[Code for Algorithm 408 with all the corrections given here is available from the ACM Algorithms Distribution Service (see inside back cover for order form) or may be found in microfiche form in "Collected Algorithms from ACM."]

NAME(n): indicates a Fortran module with n records
NAME^{T}(n): indicates "NAME" is included for testing purposes
Contents: IND(18), IPK(14), ANTIP(39), RDSPMX(121), WRSPMX(62), ANDPMX(152), SUSPMX(148), MVSPMX(24), SMSPMX(52), CVSPMX(30), ERSPMX(29), ECSPMX(28), INSPMX(19), OTSPMX(21), PERCOL(101), PERROW(84), RVSPMX(50), MUSPMX(134), TRSPMX(95), MCSPMX(89), MRSPMX(83), TEST^{T}(536), CHECK^{T}(37), MAIN^{T}(56), RANDO^{T}(13), RANDU^{T}(8), ACSPMX(142), ARSPMX(163)

ALGORITHM 409
Discrete Chebychev Curve Fit [E2]

H. Schmitt [Recd. 23 June 1970 and 12 Oct 1970]
Rechenzentrum der Technischen Hochschule
Darmstadt, West Germany

Key Words and Phrases: approximation, polynomial approximation, exchange algorithm, Chebychev approximation, Remez algorithm

CR Categories: 5.11, 5.13

```
procedure approx (m, n, k, x, y, epsh) transients: (maxit, ref)
  results: (hmax, h, a) exits: (exparameter, exmaxit, exsign);
  value m, n, k, epsh;  integer m, n, k, maxit;  real epsh, hmax;
  array x, y, h, a;  integer array ref;
  label exparameter, exmaxit, exsign;
```

comment This procedure computes the best approximation polynomial in the sense of Chebychev of required degree *m* to a set of *n* distinct points given by their abscissas and ordinates (array *x*, *y* [1:*n*]). The abscissas must be arranged in increasing order $x[1] < x[2] < \cdots < x[n]$. The desired polynomial is even, odd, or mixed for $k = 2$, $k = 1$, or $k = 0$, respectively. It is expected that $x[1] \geq 0$ in case of $k = 2$ and $x[1] > 0$ in case of $k = 1$. Leveling according to the exchange method described by Stiefel [1] is done up to a tolerance of *abs*(*epsh*). The sign of *epsh* decides whether *ref* is expected to supply entry data (cf. parameter *ref*).

maxit enters an upper limit for the number of exchange steps allowed and returns the number of steps actually performed. The parameter *ref* is used to carry entry data only if $epsh < 0$. It is an integer array containing the subscripts of the points to be used as initial reference. The lower array bound is 1, the upper bound (say *p*) is $m + 2$ in the case of mixed ($k = 0$) polynomials, *entier* $((m+3)/2)$ in the case of odd ($k = 1$), and *entier* $((m+4)/2)$ in the case of even ($k = 2$) polynomials. It is expected that $1 \leq ref[1] < ref[2] < \cdots < ref[p] \leq n$. Unless an initial reference is not explicitly given by means of the array *ref* and indicated by a value $epsh < 0$, the points lying next to the so-called Chebychev abscissas (with regard to the interval [*x*[1], *x*[*n*]]) are determined to start off the algorithm. As output, this parameter returns the reference belonging to the approximation polynomial.

The output parameters are *hmax* to return the maximum deviation, an array *h*[1:*n*] to return the approximation errors at all given points, and an array *a*[0:*m*] to carry the polynomial coefficients. The array *h* containing the approximation errors is introduced as a formal parameter to allow a drawing of the error function to be made outside the procedure. This provides a means to look at the quality of the computed approximation and is recommended to the user. A totally leveled approximation polynomial should have an error function with well characterized extrema of equal height.

Three emergency exits are provided for extraordinary events. *exparameter* is an exit to be used when entry data are entered incorrectly. *exmaxit* is used when the best fit is not found within the maximum number of exchange steps allowed. In this case, the parameter *ref* denotes a new reference which may be used as entry data for a further call of *approx*. The exit *exsign* is used when the approximation errors at the points of reference do not alternate in sign. In this case, accuracy of the computer is insufficient to generate an approximation polynomial of the required degree.

Acknowledgment. The author wishes to express his appreciation to Prof. Dr. W. Barth for many valuable discussions on the subject of Chebychev approximation.

Reference

1. Stiefel, E. L. Numerical methods of Chebychev approximation. In *On Numerical Approximation*, R. Langer, (Ed.), U. Wisconsin Press, 1958, pp. 217–232;

```
begin
  integer i, j, p, q1, q2, r;  Boolean k0, k1;
  k0 := k = 0;  k1 := k = 1;
  q1 := if k1 then 1 else 0;
  q2 := if k0 then 1 else 2;
  for i := 0 step 1 until m do a[i] := 0;
  if ¬ k0 then m := entier ((m−q1) × 0.5 + 0.1);
  p := m + 2;
  comment Check for properly given parameters;
  if n < p ∨ m < 0 ∨ ¬ k0 ∧ (¬ k1 ∨ x[1] ≤ 0)
    ∧ (k ≠ 2 ∨ x[1] < 0) then go to exparameter;
  for i := 2 step 1 until n do
    if x[i] ≦ x[i−1] then go to exparameter;
  begin
    procedure exchange (n, p, h, epsh, z, equal);
      value n, p, epsh;
      real epsh;  integer n, p;  label equal;
      array h;  integer array z;
    comment This procedure performs the exchange technique.
      The number of points and the number of reference points
      are entered by n and p. The approximation errors at different
      points are compared relative to epsh. The subscripts of the
      points of reference are carried by z[1] ··· z[p] of the integer
      array z[0:p+1], a parameter which serves to enter the
      former and return the new reference. z[0] and z[p+1] are
      for internal use only and are expected to have the values 0
      and n + 1. If both the old and new references are equal to
      each other, a jump to the label equal occurs. No global
      quantities are contained within this procedure;
    begin
      integer i, j, l, index, indl, indr, sig, ze;
      real hz1, hzp, max, maxl, maxr;
      l := 0;  sig := −sign (h[z[1]]);
      if sig = 0 then sig := 1;
      for i := 1 step 1 until p do
      begin
        max := 0;  sig := −sig;  ze := z[i+1] − 1;
        for j := z[i−1] + 1 step 1 until ze do
        if (h[j]-max) × sig > 0 then
        begin max := h[j];  index := j end;
        if abs (max-h[z[i]]) > abs(max) × epsh then
        begin z[i] := index;  l := 1 end
        end;
        maxl := maxr := 0;
        for j := z[p] + 1 step 1 until n do
        if (maxr −h[j]) × sig > 0 then
        begin maxr := h[j];   indr := j end;
        hz1 := h[z[1]];   sig := sign(hz1);
        for j := 1 step 1 until z[1] − 1 do
        if (maxl−h[j]) × sig > 0 then
        begin maxl := h[j];   indl := j end;
        maxl := abs(maxl);   maxr := abs(maxr);
        hz1 := abs(hz1);   hzp := abs(h[z[p]]);
        if l = 0 then
        begin
          if maxl − hzp ≦ maxl × epsh ∧
          maxr − hz1 ≦ maxr × epsh then go to equal
        end;
        if maxl = 0 ∧ maxr = 0 then go to end;
        if maxl > maxr then
        begin
          if maxl > hzp then to go shl
          else if maxr ≧ hz1 then to go shr
        end
        else
        begin
          if maxr > hz1 then go to shr
          else if maxl ≧ hzp then go to shl
        end;
        go to end;
shr:
        index := z[1];
        for i := 1 step 1 until p − 1 do z[i] := z[i+1];
        z[p] := indr;
        if maxl > 0 then
        for i := 1 step 1 until p − 1 do
        begin
          if abs (h[indl]) ≧ abs (h[z[i]]) then
          begin j := z[i];   z[i] := indl;   indl := index;
            index := j end
          else go to end
        end;
        go to end;
shl:
        index := z[p];
        for i := p step − 1 until 2 do z[i] := z[i−1];
        z[1] := indl;
        if maxr > 0 then
        for i := p step −1 until 2 do
        begin
          if abs (h[indr]) ≧ abs(h[z[i]]) then
          begin j := z[i];   z[i] := indr;   indr := index;
            index := j end
          else go to end
        end;
end:
      end procedure exchange;
      real arg, max, pi, q, s, t, dt, x1, xa, xe;   Boolean b1, b2;
      array xx[1:n], aa, daa[0:m], c, d [1:p];
      integer array z[0:p+1];
      comment Set up of initial reference;
      z[0] := 0;   z[p+1] := n + 1;
      if epsh < 0 then
      begin
        j := 0;
        for i := 1 step 1 until p do
        begin
          r := z[i] := ref[i];
          if j < r then j := r else go to exparameter
        end;
        if j > n then go to exparameter;
        epsh := abs (epsh);   go to m1
      end;
      pi := 3.14159265;   x1 := x[1];   xe := x[n];
      if k0 then
      begin xa := xe + x1;   xe := xe − x1;
        arg := pi/(m+1) end
      else
      begin xa := 0;   xe := xe + xe;
        arg := pi/(2×(m+1)+q1) end;
      for j := p step −1 until 1 do
      begin
        x1 := xa + xe × cos (arg × (p−j));   r := z[j+1];
        for i := r − 1 step −1 until 2 do
        if x[i] + x[i−1] ≦ x1 then go to m0;
        i := 1;
m0:
        z[j] := if r > i then i else r − 1
      end;
      if z[1] ≧ 1 then go to m1;
      for j := 1, j + 1 while z[j] < j do z[j] := j;
m1:
      for i := 0 step 1 until m do aa[i] := 0;
      for i := 1 step 1 until n do
      begin h[i] := y[i];   q := x[i];
        xx[i] := if k0 then q else q × q
      end;
      b1 := b2 := false;   r := −1;   t := 0;
iterat:
      r := r + 1;   s := 1.0;
      comment Computation of the divided difference schemes;
      if k1 then
      begin
        for i := 1 step 1 until p do
        begin
          s := −s;   j := z[i];   q := x[j];
          c[i] := (h[j] + s × t)/q;   d[i] := s/q
        end
      end
      else
      for i := 1 step 1 until p do
      begin s := −s;   c[i] := h[z[i]] + s × t;   d[i] := s end;
      for i := 2 step 1 until p do
        for j := p step −1 until i do
      begin
        q := xx[z[j]] − xx[z[1+j−i]];
        c[j] := (c[j] − c[j−1])/q;
        d[j] := (d[j] − d[j−1])/q
      end;
      dt := −c[p]/d[p];   t := t + dt;
      comment Computation of the polynomial coefficients;
      for i := m step −1 until 0 do
      begin
        daa[i] := c[i+1] + dt × d[i+1];   q := xx[z[i+1]];
        for j := i step 1 until m − 1 do
          daa[j] := daa[j] − q × daa[j+1]
      end;
      for i := 0 step 1 until m do aa[i] := aa[i] + daa[i];
      comment Evaluation of the polynomial to get the approxima-
        tion errors;
      max := 0;
      for i := 1 step 1 until n do
      begin
        s := aa[m];   q := xx[i];
        for j := m − 1 step −1 until 0 do s := s × q + aa[j];
        if k1 then s := s × x[i];
        q := h[i] := y[i] − s;
        if abs (q) > max then max := abs(q)
```

```
    end;
    comment Test for alternating signs;
    j := -sign (h[z[1]]);
    for i := 2 step 1 until p do
      if sign (h[z[i]]) = j then j := -j else
      begin b1 := true;   go to m2 end;
    comment Search for another reference;
    exchange (n, p, h, epsh, z, m2);
    if r < maxit then go to iterat else b2 := true;
    comment Results to output parameters;
m2:
    for i := 0 step 1 until m do a[q1+i×q2] := aa[i];
    for i := 1 step 1 until p do ref[i] := z[i];
    hmax := max;   maxit := r;
    if b1 then go to exsign;
    if b2 then go to exmaxit
  end
end procedure approx
```

ALGORITHM 410
Partial Sorting [M1]

J. M. Chambers [Recd. 15 July 1970]
Bell Telephone Laboratories, Murray Hill, NJ 07974

Key Words and Phrases: sorting, partial sorting, order statistics
CR Categories: 5.31

Description

We introduce the notion of partial sorting as follows. Given an array A of N elements the result of sorting the array (in place) is to arrange the elements of A so that
$A(1) \leq A(2) \leq \cdots \leq A(N)$.
An equivalent statement is that, for $J = 1, 2, \cdots, N$, $A(J)$ is a value such that for $1 \leq I < J < K \leq N$
$A(I) \leq A(J) \leq A(K)$ (1)
This property is also equivalent to the statement that $A(J)$ is the Jth order statistic [4] of A, for all J.

Partial sorting is a procedure which rearranges A so that (1) holds for some selected values of J, but not necessarily for all J. The advantage of using partial sorting, where possible, is that the cost is substantially less than for sorting, when the number of order statistics required is small compared to N.

Such will frequently be the case, for example, in statistical applications, when the sample is to be summarized using some of the order statistics. For large N only a portion of the sample would be needed, even for displays such as the empirical distribution function.

Specifically, in the algorithm *PSORT* below, the user supplies the array A of size N and a set of indices *IND* of size *NI*. On return, A will have been rearranged so that relation (1) holds, i.e. $A(J)$ has the value it would have if A were sorted, for J = *IND*(1), *IND*(2), ..., *IND*(*NI*).

For example, suppose A is the vector (10., 8., 3., 5., 7., 2.) and *IND* is the vector (2, 5). Then after a partial sort of A with *IND*, $A(2) = 3.$ and $A(5) = 8.$.

The method used is based on Hoare's method [1, 2] as developed by Singleton [3]. Hoare's method consists of choosing an element $A(m)$ and splitting the array into three portions which are respectively smaller than, equal to, and larger than this element. The method is then applied recursively to the first and third portions, until the data is completely sorted. Successive versions leading to [2] alter the method in four important respects. (i) instead of choosing $A(m)$ arbitrarily, the median of the first, last and middle element are chosen; (ii) the recursion is simulated, rather than explicit; (iii) short sequences (less than 10 in [3]) are sorted by a "sinking" sort; (iv) a different treatment of "tied" observations is introduced.

Hoare's method is very well suited to handle the partial sorting problem. The algorithm is modified simply by passing over the portion of A in which none of the indices in *IND* are found. Once we have established a segment of A which is known not to contain any of the desired order statistics, there is no need to sort it further. The special case of $NI = 1$ was treated in procedure *FIND* of [1].

For a fixed number of indices, the cost of applying *PSORT* is very nearly proportional to N, as opposed to the full sort, with cost of the order of $N\log(N)$. Because of the simplicity of the modified algorithm, the cost of *PSORT* will almost always be significantly less than the cost of the full sort, providing *NI* is substantially less than N. Notice, however, that a full sort will be carried out unless some adjacent elements of *IND* differ by more than 10.

The following restrictions are to be noted: it is assumed that *IND* is initially sorted into ascending order; A is of type *REAL*; if N is the dimension of the A array then the arrays *INDU*, *INDL*, *IU*, *IL* must have dimension K where $N < 2^{K+1}$, (see [3]);

Examples. Table I gives some examples of the performance of *PSORT* on various size arrays with various initial orderings. The examples were constructed as follows. Samples of N were simulated with a standard normal marginal distribution, and a correlation ρ with an ordered normal sample. (Specifically we generated a_i, b_i for $i = 1, \cdots, N$ as independent standard normal variates, then formed $y_i = \rho a_i + (1 - \rho^2)^{\frac{1}{2}} b_i$ and sorted the y_i, carrying along the a_i. The resulting a_i are the desired input to *PSORT*.)

Computations were carried out in two ways. By replacing the comparisons of elements in A by special functions, the number of comparisons required was counted, and is shown in the columns of Table I headed C. This gives a machine independent result, but does not include the costs of transposition, logic, etc. Therefore, we also give timings for the original algorithm, on a GE 635 computer, in the columns headed T. The unit of time is one millisecond.

The results of Table I suggest, as one would expect, that the most expensive case, for given value of *NI*, is for the desired order statistics to be evenly spaced; i.e. $jN/(NI+1)$ for $j = 1, \cdots, NI$. For this worst case, the cost does grow proportionately to N (a little less than that, in the table).

A comparison with the full sort, using Singleton's algorithm, is included for sample size 500.

Table I. Examples of *PSORT*. C = number of comparisons, T = time in 10^{-3} sec.

					Correlation with ordered data									
					−1.0		−0.5		0.0		+0.5		+1.0	
N	*NI*	*IND*			*C*	*T*	*C*	*T*	*C*	*T*	*C*	*T*	*C*	*T*
100	2	33	67		303	11.0	384	14.2	392	14.2	386	13.6	291	9.0
100	3	25	50	75	323	11.8	468	17.5	429	15.9	470	16.7	329	10.2
500	2	33	67		1122	36.0	1356	43.7	1169	36.7	1362	41.3	1121	27.8
500	3	25	50	75	1182	37.9	1414	46.3	1307	41.5	1406	43.1	1181	29.9
500	3	125	250	375	1628	49.3	2213	70.1	2184	71.3	2205	67.2	1748	43.9
500		Call to *SORT*				151.3		151.2		150.2		150.4		150.9
1000	3	250	500	750	3258	96.9	4870	145.1	4438	137.27	4725	140.4	3503	85.6

References

1. Hoare, C. A. R. Algorithms 63, Partition; 64, Quicksort; and 65, Find. *Comm ACM 4* (July 1961), 321–322.
2. Hoare, C. A. R. Quicksort. *Comput. J. 5* (1962), 10–15.
3. Singleton, R. S. Algorithm 347 Sort. *Comm. ACM 12* (1969), 185–186.
4. Wilks, S. S. *Mathematical Statistics.* Wiley, New York, 1962, p. 234.

Algorithm

```
      SUBROUTINE PSORT(A,N~~,IND~~,NI)
C PARAMETERS TO PSORT HAVE THE FOLLOWING MEANING
C A     ARRAY TO BE SORTED
C N     NUMBER OF ELEMENTS IN A
C IND   ARRAY OF INDICES IN ASCENDING ORDER
C NI    NUMBER OF ELEMENTS IN IND
      DIMENSICN A(N),INC(NI)
      DIMENSICN INDU(16),INDL(16)
      DIMENSICN IU(16),IL(16)
      INTEGER P
      JL=1
      JU=NI
      INDL(1)=1
      INDU(1)=NI
C ARRAYS INDL, INDU KEEP ACCOUNT OF THE PORTION OF IND RELATED TO THE
C CURRENT SEGMENT OF DATA BEING ORDEREC.
      I=1
      J=N
      M=1
5     IF(I.GE.J) GO TU 70
C FIRST ORDER A(I),A(J),A((I+J)/2), ANC USE MEDIAN TO SPLIT THE DATA
10    K=I
      IJ=(I+J)/2
      T=A(IJ)
      IF(A(I).LE.T) GU TO 20
      A(IJ)=A(I)
      A(I)=T
      T=A(IJ)
20    L=J
      IF(A(J).GE.T) GO TO 40
      A(IJ)=A(J)
      A(J)=T
      T=A(IJ)
      IF(A(I).LE.T) GU TO 40
      A(IJ)=A(I)
      A(I)=T
      T=A(IJ)
      GO TO 4C
30    A(L)=A(K)
      A(K)=TT
40    L=L-1
      IF(A(L).GT.T) GU TO 40
      TT=A(L)
C SPLIT THE DATA INTC A(I TO L).LT.T, A(K TO J).GT.T
50    K=K+1
      IF(A(K).LT.T) GO TO 50
      IF(K.LE.L) GO TC 30
      INDL(M)=JL
      INDU(M)=JU
      P=M
      M=M+1
C SPLIT THE LARGER OF THE SEGMENTS
      IF(L-I.LE.J-K) GC TC 60
      IL(P)=I
      IU(P)=L
      I=K
C SKIP ALL SEGMENTS NCT CORRESPONDING TO AN ENTRY IN IND
55    IF(JL.GT.JU) GO TO 70
      IF(IND(JL).GE.I) GO TO 58
      JL=JL+1
      GO TO 55
58    INDU(P)=JL-1
      GO TO 8C
60    IL(P)=K
      IU(P)=J
      J=L
65    IF(JL.GT.JU) GC TO 70
      IF(IND(JU).LE.J) GO TO 68
      JU=JU-1
      GO TO 65
68    INDL(P)=JU+1
      GO TO 8C
7C    M=M-1
      IF(M.EQ.0) RETURN
      I=IL(M)
      J=IU(M)
      JL=INDL(M)
      JU=INDU(M)
      IF(JL.GT.JU) GC TO 70
8C    IF(J-I.CT.10) GO TO 10
      IF(I.EQ.1) GO TO 5
      I=I-1
9C    I=I+1
      IF(I.EQ.J) GO TC 70
      T=A(I+1)
      IF(A(I).LE.T) GU TO 90
      K=I
10C   A(K+1)=A(K)
      K=K-1
      IF(T.LT.A(K)) GU TO 100
      A(K+1)=T
      GO TO 9C
      END
```

Algorithm 411

Three Procedures for the Stable Marriage Problem [H]

D.G. McVitie* and L.B. Wilson (Recd. 12 Aug. 1968 and 15 July 1969)
Computing Laboratory, University of Newcastle
upon Tyne, Newcastle upon Tyne, NE1 7RU,
England

Key Words and Phrases: assignment problems, assignment procedures, combinatorics, discrete mathematics, operations research, stable marriage problem, university entrance

CR Categories: 5.30

Part 1

```
procedure GS (malechoice, femalechoice, marriage, count, n);
  value n;  integer count, n;
  integer array malechoice, femalechoice, marriage;
comment This procedure finds the male optimal stable marriage
  solution using the Gale and Shapley algorithm. The result is
  left in the integer array marriage. Thus marriage [i] is the man
  whom the ith woman marries. n is the size of the problem,
  count is the number of proposals made before the stable marriage
  is found. malechoice and femalechoice are the choice matrices
  for the men and women respectively, i.e. femalechoice[i, j]
  is the jth choice of the ith woman. The femalechoice array is
  changed to the integer array fc, where fc[i, j] is the choice num-
  ber (first, second, third, . . .) of the jth man to woman i. This
  new arrangement is adopted for convenience when the women
  compare proposals. All the women keep a dummy man 0 in sus-
  pense initially. This dummy man is given a choice number n + 1
  so that he will be given up as soon as any other offer is made;
begin
  integer i, m, j;  Boolean array refuse [0:n];
  integer array fc [1:n, 0:n], proposal, malecounter [1:n];
  for i := 1 step 1 until n do
  begin
    for j := 1 step 1 until n do
      fc [i, femalechoice [i, j]] := j;
    comment The femalechoice array is rearranged for conven-
      ience in the marriage part of the procedure;
      refuse [i] := true;  marriage [i] := 0;
      malecounter [i] := 1;  fc [i, 0] := n + 1
  end;
  count := 0;
PROPOSE:
  m := 0;
  comment Now the rejected men propose to the next woman in
    their choice lists. Initially all the men propose to their first
    choices;
  for i := 1 step 1 until n do
    if refuse [i] then
    begin
      proposal [i] := malechoice [i, malecounter [i]];
      malecounter [i] := malecounter [i] + 1;
      m := m + 1; refuse [i] := false
    end
    else proposal [i] := -1;
  if m = 0 then go to FINISH;
  comment The procedure terminates if at any stage no proposals
    are made by the men;
  count := count + m;
  comment In the next part of the procedure all the women who
    have had a proposal decide whether to reject it or the one
    they are keeping in suspense;
  for i := 1 step 1 until n do
    if proposal [i] > 0 then
    begin
      j := proposal [i];
      if fc [j, i] > fc [j, marriage [j]] then refuse [i] := true
      else
      begin refuse [marriage [j]] := true; marriage [j] := i end
    end;
  go to PROPOSE;
FINISH:
end of procedure GS
```

Part 2

```
procedure MW(malechoice, femalechoice, marriage, count, n);
  value n;  integer count, n;
  integer array malechoice, femalechoice, marriage;
comment The heading is the same as for the GS procedure and
  the formal parameters have the same meaning. Also the female-
  choice array has been rearranged in the array fc as before, and
  the women given initially a dummy man 0 with choice number
  n + 1;
begin
  integer i, j;
  integer array fc [1:n, 0:n], malecounter [1:n];
  procedure PROPOSAL (i);  value i;  integer i;
  comment This procedure makes the next proposal for man i
    and calls the procedure REFUSAL to see what effect this
    proposal will have. The procedure does nothing if man i is
    the dummy man 0;
  if i ≠ 0 then
  begin
    integer j;  count := count + 1;
    j := malecounter[i];  malecounter[i] := j + 1;
    REFUSAL(i, malechoice[i, j])
  end;
  procedure REFUSAL(i, j);  value i, j;  integer i, j;
  comment This procedure decides whether woman j should keep
    the man she is holding in suspense in marriage[j] or man i
    who has just proposed to her. Whichever she rejects goes
    back to the procedure PROPOSAL to make his next proposal;
  if fc[j, marriage[j]]>fc[j, i] then
  begin
    integer k;
    k := marriage[j];  marriage[j] := i;
    PROPOSAL(k)
  end
```

*Now at Software Science, Ltd., Wilmslow, Cheshire, England.

```
   else PROPOSAL(i);
   for i := 1 step 1 until n do
   begin
      for j := 1 step 1 until n do
         fc[i, femalechoice[i,j]] := j;
      marriage[i] := 0;   malecounter[i] := 1; fc[i, 0] := n + 1
   end;
   count := 0;
   for i := 1 step 1 until n do PROPOSAL(i);
   comment  This for statement operates the algorithm and after
      the ith cycle a set of stable marriages exists for the men 1 to i
      and i of the women;
end of procedure MW
```

Part 3

```
procedure  ALL STABLE MARRIAGES (malechoice, femalechoice,
   n, STABLE MARRIAGE);
   value n;
   integer array malechoice, femalechoice;
   integer n;  procedure STABLE MARRIAGE;
comment  malechoice and femalechoice are the same arrays as were
   used in GS and MW, n is size of problem. STABLE MARRIAGE
   (marriage, n, count) is a procedure (with three parameters) writ-
   ten by the user which is entered when a new stable marriage is
   formed after count proposals. The marriage is stored such that
   marriage[i] contains the number of the man married to woman i.
   The locally declared Boolean array unchanged is used to make
   sure Rule (2) is not violated; i.e. during a breakmarriage opera-
   tion started on man i only men ≥ i may propose. The locally
   declared Boolean success is set true if breakmarriage to man i
   leads to a new stable marriage, otherwise it is set false;
begin
   integer array marriage, malecounter [0: n], fc [1: n, 0: n];
   Boolean array unchanged [0: n];
   integer i, j, k;   Boolean success;
   procedure breakmarriage(malecounter,marriage,i,n,count);
      value malecounter, marriage, i, n, count;
      integer i, n, count; integer array malecounter, marriage;
   comment  This procedure breaks the marriage of man i;
   begin
      integer j;
      marriage [malechoice [i, malecounter [i]−1]] := −i;
      proposal (i,malecounter,marriage,count);
      if ¬ success then go to EXIT;
      STABLE MARRIAGE (marriage,n,count);
      for j := i step 1 until n − 1 do
         breakmarriage (malecounter,marriage,j,n,count);
      comment  The lower limit i in the above for statement is the
         application of Rule(1) which after a successful break-
         marriage operation on man i restricts further breakmarriages
         to men ≥ i;
      for j := i + 1 step 1 until n − 1 do
         unchanged [j] := true;
EXIT:
      unchanged [i] := false;
   end of breakmarriage;
   procedure proposal (i, malec, marriage, c);
      value i;
      integer i,c; integer array malec, marriage;
   comment  In this procedure man i proposes to the next woman
      in his choice list, and calls the procedure refusal for this
      woman. If i is negative on entry then a successful break-
      marriage operation has been completed and a new stable
      marriage found. If the Boolean success is made false during
      a breakmarriage operation then it means that this break-
      marriage has failed;
   if i < 0 then success := true
   else if i = 0 ∨ malec [i] = n + 1 ∨ ¬ unchanged [i]
      then success := false
   else
   begin
      c := c + 1; j := malec [i];   malec [i] := j + 1;
      refusal (i,malechoice[i,j],malec,marriage,c)
   end of proposal;
   procedure refusal (i,j,malec,marriage,c);
      value i,j;
      integer i,j,c;   integer array malec, marriage;
   comment  This procedure decides whether woman j prefers man
      i or the man in marriage [j]. Whichever she rejects goes back
      to the procedure proposal to make his next choice;
   if fc [j, abs (marriage [j])]>fc[j,i] then
   begin
      k := marriage [j];   marriage [j] := i;
      proposal  (k,malec,marriage,c)
   end
   else proposal (i,malec,marriage,c);
   for i := 1 step 1 until n do
   begin
      for j := 1 step 1 until n do
         fc[i,femalechoice [i,j]] := j;
      marriage [i] := 0;   malecounter [i] := 1;
      fc[i,0] := n + 1;   unchanged [i] := true;
   end;
   count := 0;
   for i := 1 step 1 until n do
      proposal (i,malecounter,marriage,count);
   comment  Male optimal stable solution found;
   STABLE  MARRIAGE (marriage,n,count);
   for i := 1 step 1 until n − 1 do
      breakmarriage (malecounter, marriage, i,n,count);
end of procedure ALL STABLE MARRIAGES
```

Algorithm 412

Graph Plotter [J6]

Josef Čermák (Recd. 19 Mar. 1970 and 12 Nov. 1970)
Department of Physics, University of Chemical Technology, Pardubice, ČSSR.

Key Words and Phrases: plot, graph, lineprinter plot
CR Categories: 4.41

```
procedure graphplotter (N, x, y, m, n, xerror, yerror, g, L, S, EM,
  C0, C1, C2, C3, C4, label);
  value N, m, n, xerror, yerror, g, L, S;
  array x, y;  integer N, g, m, n, L, S;  real xerror, yerror;
  string EM, C0, C1, C2, C3, C4;  label label;
comment This procedure is functionally identical with Algorithm
  278. It needs, however, a significantly smaller array than Al-
  gorithm 278 for storage of the graph before it is printed. The
  procedure is intended to be used to give an approximate graph-
  ical display of a multivalued function y[i, j] of x[i], on a line
  printer. Output channel N is used for all output. The graph is
  plotted for those points such that 1 ≤ i ≤ m and 1 ≤ j ≤ n
  where 2 ≤ n ≤ 4. If n = 1, then y must be a one-dimensional
  array y[i] and the graph is plotted for x[i] and y[i] for 1 ≤ i ≤
  m. The format of the output is arranged so that a margin of g
  spaces appears on the left-hand edge of the graph. L and S
  specify the number of lines down the page and the number of
  spaces across the page which the graph is to occupy, respec-
  tively. The graph is printed so that lines 1 and L correspond tc
  the minimum and maximum values of x, and character posi-
  tions 1 and S correspond to the minimum and maximum values
  of y. That is to say, y is plotted across the page and x is plotted
  down the page. After the entire graph has been plotted, the
  minimum and maximum values for x and y are printed out in
  order xmin, xmax, ymin, ymax. The argument EM represents
  the character which is printed on the perimeter of the display.
  The argument C0 represents the character printed at empty
  positions. The arguments, C1, C2, C3, C4, represent the charac-
  ters printed for y[i, 1], y[i, 2], y[i, 3], and y[i, 4], respectively.
  At those points at which more than one character would ap-
  pear, the order of preference is C1, C2, C3, C4. Control is
  passed from graph-plotter to the point whose label appears as
  the parameter label if the range of x[i] is less than xerror, or
  if the range of y[i, j] is less than yerror, for all j. If the values of
  x[i] occur at equal intervals, choosing L = m will make one
  line of printout equivalent to one interval of x. The graph may
  look somewhat out of true proportion since this algorithm as-
  sumes that spacing along both axes is the same, but most line
  printers do not have the same spacing between adjacent lines
  as between adjacent characters on a line;
begin
  real p, q, xmax, xmin, ymax, ymin;
  integer i, j;
  integer array plot L, ind [1:L], plot S [1:S];
  xmax := xmin := x[1];
  for i := 2 step 1 until m do
  begin
    if x[i] > xmax then xmax := x[i] else
    if x[i] < xmin then xmin := x[i]
  end of hunt for maximum and minimum values of x;
  if n = 1 then go to N1A;
  ymax := ymin := y[1, 1]:
  for i := 1 step 1 until m do
    for j := step 1 until n do
    begin
      if y[i, j] > ymax then ymax := y[i, j] else
      if y[i, j] < ymin then ymin := y[i, j]
    end of hunt for maximum and minimum values of y;
escape:
  if abs(xmax-xmin) < xerror ∨ abs(ymax-ymin) < yerror
    then go to label;
  p := (L−1)/(xmax−xmin);  q := (S−1)/(ymax−ymin);
  for i := 1 step 1 until L do plot L[i] := 1;
  for i := m step−1 until 1 do
  begin
    integer r;
    r := 1 + entier ((x[i]−xmin)×p+0.5);
    plot L[r] := 0;  ind[r] := i
  end;
  NEWLINE (N, 1);  SPACE (N, g);
  comment NEWLINE and SPACE must be declared globally to
    graphplotter, NEWLINE (N, p) outputs p carriage returns
    and p line feeds on channel N, SPACE (N, p) outputs p
    blank character positions on channel N;
  for j := 1 step 1 until S do outstring (N, EM);
  for i := 1 step 1 until L do
  begin
    plot S[1] := plot S [S] := 1;
    for j := 2 step 1 until S − 1 do plot S[j] := 2;
    if plot L[i] = 0 then
    begin
      if n = 1 then
      plot S [1+entier (0.5+q×(y[ind[i]]−ymin))] := 3
      else
      for j := n step − 1 until 1 do
      plot S [1+entier (0.5+q×(y[ind [i], j]−ymin))] := j + 2
    end;
    NEWLINE (N, 1);  SPACE (N, g);
    for j := 1 step 1 until S do
    begin
      switch SW := SW1, SW2, SW3, SW4, SW5, SW6;
      go to SW [plot s[j]];
SW1:
      outstring (N, EM);  go to fin;
SW2:
      outstring (N, C0);  go to fin;
SW3:
      outstring (N, C1);  go to fin;
SW4:
      outstring (N, C2);  go to fin;
SW5:
      outstring [N, C3);  go to fin;
SW6:
      outstring (N, C4);
fin:
    end;
  end;
  NEWLINE (N, 2);  SPACE (N, g);
  for j := 1 step 1 until S do outstring (N, EM);
  NEWLINE (N, 2);  SPACE (N, g);  outreal (N, xmin);
  outreal (N, xmax);
  outreal (N, ymin);  outreal (N, ymax);
  go to end;
N1A:
  ymax := ymin := y[1];
  for i := 2 step 1 until m do
  begin
    if y[i] > ymax then ymax := y[i] else
    if y[i] < ymin then ymin := y[i]
  end of hunt for maximum and minimum values of y when
    n = 1;
  go to escape;
end:
end of graphplotter
```

Remark on Algorithm 412 [J6]

Graph Plotter [Joseph Cermak, *Comm. ACM 14* (July 1971), 492–493]

Richard P. Watkins [Recd. 31 Jan. 1972], Mathematics Department, Royal Melbourne Institute of Technology, Melbourne, Australia 3000

This algorithm is not functionally identical to Algorithm 278 as claimed. If the $x[i]$ values are not uniformly spaced or if $m > L$, it is possible for two or more of them to correspond to the same printer line. In this case, the array *ind* will contain only the largest of the values of i and only one set of $y[i, j]$ values, corresponding to that value of i, will be plotted.

The array *ind* is redundant. The following changes enable *plotL* to take over the functions of *ind* (where all line numbers refer to lines relative to the label *escape*):

a. Line 4. Replace

```
for i := 1 step 1 until L do plotL[i] := 1
```

by

```
for i := 1 step 1 until L do plotL[i] := 0
```

b. Line 9. Replace

```
plotL[r] := 0; ind[r] := i
```

by

```
plotL[r] := i
```

c. Line 21. Replace

```
if plotL[i] = 0 then
```

by

```
if plotL[i] > 0 then
```

d. Line 24. Replace

```
plotS [1 + entier(0.5 + q × (y[ind[i]] −ymin))] := 3
```

by

```
plotS [1 + entier(0.5 + q × (y[plotL[i]] −ymin))] := 3
```

e. Line 27. Replace

```
plotS [1 + entier(0.5 + q × (y[ind[i],j] − ymin))] := j + 2
```

by

```
plotS[1 + entier(0.5 + q × (y[plotL[i],j] − ymin))] := j + 2
```

(The referee has noted that there is a typographical error on the fifth line before the line labeled *escape*. Replace

```
for j := step 1 until n do
```

by

```
for j := 1 step 1 until n do
```

He has also noted that the array declaration for *ind* should be deleted if the above changes are made.—L.D.F.)

Algorithm 413

ENTCAF and ENTCRE: Evaluation of Normalized Taylor Coefficients of an Analytic Function [C5]

J.N. Lyness,* Argonne National Laboratory, Argonne, IL 60439, and G. Sande,† Department of Statistics, The University of Chicago, Chicago, IL 60637 (Recd. 17 June 1968, 12 Feb 1970, and 20 July 1970)

Editor's note: *The algorithm described here is available on magnetic tape. The text plus the listing of the algorithm will be printed in the* Collected Algorithms from CACM. *The charge for the taped algorithm is $16.00 (U.S. and Canada) or $18.00 (elsewhere). If the user sends us a small tape (wt. less than 1 lb.) we will copy the algorithm on it and return it to him at a charge of $10.00 (U.S. only). All orders are to be prepaid with checks payable to "ACM Algorithms." The algorithm is recorded as one file of BCD 80-character card images at 556 B.P.I., even parity, on seven-track tape. If requested, the algorithm is supplied at a density of 800 B.P.I. The cards for the algorithm are sequenced starting at 10 and incremented by 10. The sequence number is right-justified in column 80. Although every attempt is made to insure that the algorithm conforms to the printed description, no guarantee is made, nor is there a guarantee that the algorithm is correct.—L.D.F.*

Key Words and Phrases: Taylor coefficients, Taylor series, Cauchy integral, numerical integration, numerical differentiation, interpolation, complex variable, complex arithmetic, fast Fourier transform

CR Categories: 5.12, 5.13, 5.16

Description

Introduction. Two subroutines, *ENTCAF* and *ENTCRE*, coded in ANSI FORTRAN are described here. *ENTCAF* may be used to calculate approximations $r^s a_s^{(m)}$ to a set of normalized Taylor coefficients

$$r^s a_s = r^s f^{(s)}(\zeta)/s! \quad s = 0,1,2, \ldots . \tag{1.1}$$

The values of r and ζ, a complex number, are provided by the user together with a function subprogram that represents $f(z)$ as a complex-valued function of a complex variable. The user also provides a value of ϵ_{req}, the required absolute accuracy. The routine returns an accuracy estimate ϵ_{est} together with approximations $r^s a_s^{(m)}$ and a number m. These are supposed to satisfy

$$\begin{aligned} |\, r^s a_s^{(m)} - r^s a_s \,| &< \epsilon_{est} \quad s = 0,1,2, \ldots, m-1, \\ |\, r^s a_s \,| &< \epsilon_{est} \quad s = m, m+1, \ldots . \end{aligned} \tag{1.2}$$

A result status indicator *NCODE* is output. If $\epsilon_{est} > \epsilon_{req}$ this gives a brief indication of why the required accuracy was not achieved.

ENTCRE carries out the same task as *ENTCAF* in the case that ζ is real and also that $f(z)$ is real when z is real. In this special and common case, *ENTCRE* is about twice as economic as *ENTCAF*.

Outline of method. The Taylor coefficients a_s occur in the Taylor series

$$f(z) = \sum_{s=0}^{\infty} a_s (z - \zeta)^s, \quad |\, z - \zeta \,| < R_c, \tag{2.1}$$

where R_c is the radius of convergence of the Taylor series. Cauchy's theorem provides a set of integral representations. One of these is

$$r^s a_s = \frac{r^s}{2\pi i} \int_{C_r} \frac{f(z)}{(z - \zeta)^{s+1}}\, dz, \quad r < R_c, \tag{2.2}$$

where C_r is the circle $|\, z - \zeta \,| = r$. The approximation $r^s a_s^{(m)}$ is obtained by replacing the integral in (2.2) by an approximation based on an m-point trapezoidal rule approximation. Specifically,

$$r^s a_s \simeq r^s a_s^{(m)} = m^{-1} \sum_{j=0}^{m-1} \exp(-2\pi ijs/m) f(\zeta + r \exp(2\pi ij/m)), \tag{2.3}$$
$$s = 0,1, \ldots, m-1.$$

* Work performed under the auspices of the U.S. Atomic Energy Commission. †This research was supported in part by National Science Foundation research grant number NSF-GP-16071 from the Division of Mathematical and Physical Sciences.

The calculation is in two parts. The first part (stages 1, 2, and 3) is iterative in nature. Using (2.3) the approximations $a_0^{(m)}$ with $m = 1,2,4,8, \cdots$ are calculated. The function values are retained. The convergence criterion is based on the circumstance that the true value

$$a_0 = f(\zeta) \tag{2.4}$$

of one of the approximations $a_0^{(m)}$ may be determined by a single function evaluation. A rather involved convergence criterion based on the orderly approach of the sequence $a_0^{(m)}$, $m = 1,2,4, \ldots$, to its limiting value a_0 is used. This is described in some detail by Lyness [8].

When the convergence of $a_0^{(m)}$ to a_0 has been achieved the routine carries out the second part (stage 4). This consists of evaluating $r^s a_s^{(m)}$ from (2.3) for $s = 0,1, \cdots, m - 1$ using the function values calculated and retained during the first part. A fast Fourier transform technique is used for this calculation. This is particularly appropriate since m is a power of two. The derivation and implementation of this technique is described in Gentleman and Sande [5, pp. 566–7]. The specialized version used in *ENTCRE* is described in Sande [9].

Restrictions: theoretical. There are two restrictions of a theoretical nature.

1. The value of r must be less than the radius of convergence, R_c, of the Taylor series. So long as this condition is satisfied, it can be shown (see [5] and [8]) that

$$\begin{aligned} &| r^s a_s | < K\rho^s, \\ &| r^s a_s^{(m)} - r^s a_s | < K\rho^{m+s}/(1 - \rho^m), \end{aligned} \tag{3.1}$$

where ρ is any number greater than r/R_c and K depends on ρ. Thus the approximations approach their limiting values and there are only a finite number of normalized Taylor coefficients whose magnitude exceeds ϵ_{req}. If this restriction is violated, that is, a value of $r \geq R_c$ is chosen, then in general the sequence $r^s a_s^{(m)}$ converges, but not to $r^s a_s$. Instead it converges to the integral on the right in (2.2), but (2.2) is not generally vailid if $r \geq R_c$. Thus the routine itself fails to converge since $a_0^{(m)}$ does not approach $f(\zeta)$ in the limit of increasing m.

2. The function $f(z)$ must not be an odd function of $(z - \zeta)$. While the convergence criterion based on (2.4) has much to recommend it, it does have one serious drawback. If it happens (as it does in the case $f(z) = sin(z)$; $\zeta = 0$) that

$$f(z - \zeta) = -f(\zeta - z), \tag{3.2}$$

then every approximation $a_0^{(m)}$ is zero, as is the true value a_0. The routine then finds that it converges immediately. In this case the problem should be reformulated. One defines $g(z) = f(z)/(z - \zeta)$ or $g(z) = (z - \zeta)f(z)$. The Taylor coefficients A_s of $g(z) = \zeta$ are then calculated using *ENTCAF*. A_s is the same as a_{s+1} or a_{s-1} as the case may be.

Restrictions: practical. There are two principal practical restrictions. These arise because (1) the computer uses finite length floating-point arithmetic; (2) execution cannot be allowed to continue indefinitely; at some stage it has to terminate whether or not the calculation is complete.

An output status parameter *NCODE* indicates to the user whether the results have been significantly affected by either of these restrictions.

1. *Roundoff error.* The routine requires as an input parameter the machine accuracy parameter ϵ_M. The approximations $r^s a_s^{(m)}$ given by (2.3) are of such a form that an estimate of the roundoff error level is

$$\epsilon_{r.o}^{(m)} = \epsilon_M \max_{j=0, \ldots, m-1} | f(\zeta + r \exp(2\pi ij/m)) |. \tag{3.3}$$

If, at any stage it appears that

$$\epsilon_{req} < 10\, \epsilon_{r.o}^{(m)}, \tag{3.4}$$

the routine internally replaces ϵ_{req} by 10 $\epsilon_{r.o}^{(m)}$ and either terminates (input *NCODE* negative) or continues with the calculation (input *NCODE* nonnegative).

2. *Physical upper limit.* This is defined by an input parameter *NMAX*. Iterations in the first part to calculate $a_0^{(m)}$, $m = 1,2,4,8, \ldots$, with $m <$ *NMAX* are possible. If convergence has not been achieved by this stage, the calculation is completed.

The output status parameter *NCODE* is +1 if all went well. In general *NCODE* = 0 if the calculation was terminated; is positive if it converged and negative if it did not converge; has magnitude 1 if roundoff error was not observed; and has magnitude 2 if roundoff error was observed.

If *NCODE* $\neq$ 0, the returned value ϵ_{est} corresponds to the estimated accuracy of *TCOF(J)* whether or not convergence or roundoff error occurred. If *NCODE* = 0, the quantity 10 $\epsilon_{r.o}^{(m)}$ is returned in place of ϵ_{est}.

Comments. The algorithms described here deliver approximations to a set of normalized Taylor coefficients $r^s a_s$. It is natural to ask why this choice of output was made, rather than perhaps a set of Taylor coefficients a_s or a set of derivatives $f^{(s)}(\zeta)$. The most immediate reason is that the algorithm naturally provides a set or normalized Taylor coefficients to a uniform absolute accuracy. The user specifies r and ϵ_{req} only. If, for example, one is interested in a set of derivatives, the specification of the accuracy requirements becomes very much more complicated. However, if one looks ahead to the use to which the Taylor coefficients are to be put, one finds in many cases that uniform accuracy in normalized Taylor coefficients corresponds to the sort of accuracy requirement which is most convenient.

As an illustration we consider a very trivial problem. We wish to represent $f''(x)$ as a polynomial in the interval $(-l,l)$ to an accuracy E. Clearly

$$f''(x) = \sum_{s=2}^{\infty} s(s - 1)a_s x^{s-2} = \frac{1}{r^2} \sum_{s=2}^{\infty} s(s - 1)a_s r^s \left(\frac{x}{r}\right)^{s-2}. \tag{4.1}$$

A very crude approach might be to take $r = l$ and $\epsilon = r^2E/6$. In this case the error in the sth term is less than $s(s - 1)E(x/l)^{s-2}/6$. One cannot be assured that for $x \simeq l$ these errors may not cooperate in such a way as to lose the required accuracy. However, if r is chosen to be greater than l and $\epsilon = r^2(1 - l/r)^3E/2$ then it follows at once that if the allowed error in a_s, r^s is less thah ϵ, the error in $f''(x)$ is less than E. These two approaches represent extremes. Neither take into account that the sequence $a_s r^s$ itself approaches zero and for high values of s it is unnecessary to bound the error in omitting such a term by ϵ. A more complicated formula based on (3.1) is derived by Lyness and Delves [5], eq. (2.9). But the underlying feature of any of these approaches to approximating (4.1) is that a uniform absolute accuracy for $a_s r^s$, $s = 0,1,2, \ldots$, is very convenient for this problem. If the algorithm instead calculated $f^{(s)}(0)$ to a specified relative accuracy, the determination of the accuracy to use in this problem would be very much more involved.

Possible modifications. The general approach to a numerical calculation by means of the numerical evaluation of contour integrals is at present an open field for investigation. The algorithms described here may be used in several problems known to the authors. These are: (a) determination of zeros of analytic function [7, 1, and 5]; (b) numerical differentiation [7, 6]; (c) numerical quadrature [8].

In particular applications, modifications of *ENTCAF* or *ENTCRE* can lead to more efficient calculations. Possible modifications include: (a) Provision for calculation of only some of the Taylor coefficients, for example, s even or $s \leq 12$; (b) Provision for a "subsequent return option" which allows the same calculation to be taken up at a later stage if it is found subsequently that higher accuracy is required; (c) Provision for an "early exit." Used in conjunction with (b) this would enable the program to consider intermediate results to determine whether to continue with the current values of r and ϵ, before a high investment of computer time has been made.

In fact, *ENTCRE* is a special modification of *ENTCAF* designed for a particular application, ζ real, $f(x)$ real. The output status parameter *NCODE* is of particular use in these applications since it allows appropriate remedial action to be taken under program control.

Algorithms which include modifications (b) and (c) above have been used by the first author. However, these involve complicated logic and are strongly connected with the particular application. The algorithms listed here may be modified by the user in particular applications for any large scale use. However, in pilot runs or small scale calculations they are adequate as they stand.

Comparisons and examples. In [6] and [8], several numerical examples are given, and comparisons with other methods are made. So far as the determination of zeros of an analytic function is concerned, the method described in [6] has some advantages in a global situation, but should not be used locally. For numerical quadrature, the method described [8] is definitely superior to standard methods if there is a nearby pole or singularity of a special type. In these cases a proper evaluation depends on the details of the problem under consideration.

It is in problems involving numerical differentiation that the method on which these algorithms are based show up to great advantage. This is simply because, once the use of complex function values is allowed, the numerical instability associated with numerical differentiation may be avoided.

In [6], a different but related method for numerical differentiation is described. The remarks about the roundoff error given there apply to these routines also. There as an example, the calculation of $f^{(5)}(0)$ was considered for

$$f(x) = e^x/(\sin^3(x) + \cos^3(x)).$$

The actual value of this derivative is an integer, namely

$$f^{(5)}(0) = -164.$$

In order to provide some sort of comparison, a special algorithm for numerical differentiation based on polynomial interpolation was written using only function values at real abscissas. A set of several dozen numerical experiments were carried out on a machine for which $\epsilon_M = 3 \times 10^{-11}$. The closest result was in error by 10^{-2}; the worst result had the wrong sign.

ENTCRE was then used for the same problem in an attempt to obtain seven-digit accuracy, i.e. an absolute accuracy of $E = 10^{-4}$. A sequence of values of r was used, with in each case $\epsilon_{req} = r^5 \times 10^{-4}/5!$ and input parameter $NCODE = -1$ to secure immediate termination if roundoff error prevented a sufficiently accurate result from being attained. With $r = 0.1$ and $r = 0.2$, execution terminated using in each case one complex and three real function values. With $r = 0.4$, the result

$$f^{(5)}(0) = -164.00000013$$

was obtained at a cost of 15 complex and three real function values ($m = 32$); the accuracy estimate given by the algorithm was

$$E_{est} = \epsilon_{est}\, 5!/r^s = 6 \times 10^{-6}.$$

Incidentally, an absolute accuracy of less than 10^{-4} was estimated and a better accuracy obtained for $r = 0.3$, 0.4, 0.5, 0.6, 0.7 with $m = 32$, 32, 64, 64, 128, respectively. For $r = 0.8$ and $r = 0.9$ the routine failed to converge with $m = 128$ giving absurd results and estimates. These latter values of r are greater than the radius of convergence $R_c = \pi/4$.

The role played by the output status parameter *NCODE* is illustrated in this example. With $r = 0.1$ and $r = 0.2$, the value of *NCODE* indicated immediately that the results were not to be taken seriously because of roundoff error. With $r = 0.8$ and $r = 0.9$, the value of *NCODE* indicated that the results were not to be taken seriously because of lack of convergence. Thus the calculation could have been carried out completely under program control, with a driver program finding for itself an appropriate value of r. An efficient program for this application would require modifications (a), (b), and (c) of the previous section.

The testing of the algorithm included the calculation of high-order derivatives. In general, it frequently happens that even when analytic closed expressions are known for such derivatives, these expressions are difficult to evaluate because of excessive subtraction error. Cases in point include the functions e^x/x and $sin(x)/x$. Programs were written to evaluate the first 80 derivatives of these functions at $x = 5$, 10, 20, 40, and 80. It turned out that meaningful results could be obtained. For example, for $f(x) = e^x/x$, using $r = 32$ and $\epsilon_{req} = 10^{-10}$, *ENTCRE* gives

$$f^{(25)}(40) = 3.6560469 \times 10^{16}$$

with an estimated relative accuracy of 2.5×10^{-9}. These results were compared with those obtained using an algorithm due to Gautschi and Klein [2, 3]. In all cases examined corresponding results agreed to within the calculated error estimate.

References

1. Delves, L.M., and Lyness, J.N. A numerical method for locating the zeros of an analytic function. *Math. Comput. 21* (1967), 543–560.
2. Gautschi, W., and Klein, B.J. Recursive computation of certain derivatives—A study of error propagation, *Comm. ACM 13* (Jan. 1970), 7–9.
3. Gautschi, W., and Klein, B.J. R282 Derivatives of e^x/x, $\cos(x)$ and $\sin(x)/x$. *Comm. ACM 13* (Jan. 1970), 53–54.
4. Gentleman, W.M., and Sande, G. Fast Fourier transforms—for fun and profit. Proc. AFIPS 1966 FJCC, Vol. 29, Spartan Books, New York, pp. 563–578.
5. Lyness, J.N., and Delves, L.M. On numerical contour integration round a closed contour. *Math. Comput. 21* (1967), 561–577.
6. Lyness, J.N., and Moler, C.B., Numerical differentiation of analytic functions. *SIAM J. Numer. Anal. 4* (1967), 202–210.
7. Lyness, J.N. Numerical algorithms based on the theory of complex variables. Proc. ACM 22nd Nat. Conf. 1967, pp. 125–134.
8. Lyness, J.N. Quadrature methods based on complex function values. *Math. Comput. 23* (1969), 601–620.
9. Sande, G., Fast Fourier transform—A globaly complex algorithm with locally real implementation. Proc. 4th Ann. Princeton Symp. on Information Sciences and Systems, 1970, pp. 136–142.

Algorithm

```
      SUBROUTINE ENTCRE ( CFUN, ZETA, RCIRC, EPREQ, EPMACH, NMAX, NCODE,
     . EPEST, NTCOF, TCOF, WORK, NTAB, SINTAB )
C
C ** EVALUATION OF NORMALIZED TAYLOR COEFFICIENTS **
C **          OF A REAL ANALYTIC FUNCTION          **
C
C      ** GENERAL PURPOSE  **
C THIS ROUTINE EVALUATES A SET OF NORMALIZED TAYLOR COEFFICIENTS
C TCOF(J+1) = (RCIRC**J) * (J-TH DERIVATIVE OF CFUN(Z) AT Z=ZETA)
C DIVIDED BY FACTORIAL(J) ... J = 0,1,2,3...NMAX-1.
C TO A UNIFORM ABSOLUTE ACCURACY **EPEST** USING FUNCTION
C VALUES OF CFUN(Z) AT POINTS IN THE COMPLEX PLANE LYING ON
C THE CIRCLE OF RADIUS **RCIRC** WITH CENTER AT Z = ZETA.
C THIS ROUTINE IS A SPECIAL VERSION OF ENTCAF FOR USE WHEN
C ZETA IS REAL AND ALSO CFUN(Z) IS REAL WHEN Z IS REAL.
C
C      ** THEORETICAL RESTRICTIONS **
C RCIRC MUST BE SMALLER THAN THE RADIUS OF CONVERGENCE OF
C THE TAYLOR SERIES.  THE PROBLEM HAS TO BE REFORMULATED
C SHOULD CFUN(Z) HAPPEN TO BE AN ODD FUNCTION
C OF  (Z - ZETA)  , THAT IS IF THE RELATION
C ** -CFUN(-(Z-ZETA))=CFUN(Z-ZETA) **  IS AN IDENTITY.
C
C      ** REQUIREMENTS FOR CALLING PROGRAM **
C CALLING PROGRAM MUST CONTAIN CONTROL STATEMENTS DESCRIBED
C NOTES (3) AND (4) BELOW.  IT MUST ALSO ASSIGN VALUES TO
C INPUT PARAMETERS.  THE ROUTINE REQUIRES TWO SUBPROGRAMS.
C HFCOF  (LISTED AFTER ENTCRE) AND CFUN (SEE NOTE (4) BELOW).
C
C      **INPUT PARAMETERS**
C (1) CFUN    NAME OF COMPLEX FUNCTION SUBPROGRAM.
C (2) ZETA    REAL POINT ABOUT WHICH TAYLOR EXPANSION IS REQUIRED
C (3) RCIRC   RADIUS (REAL)
C (4) EPREQ   THE ABSOLUTE ACCURACY (REAL) TO WHICH THE
C             NORMALIZED TAYLOR COEFFICENTS, TCOF(J), ARE REQUIRED
C (5) EPMACH  THE MACHINE ACCURACY PARAMETER (REAL)
C             (OR AN UPPER BOUND ON THE RELATIVE ACCURACY OF
C             QUANTITES LIKELY TO BE ENCOUNTERED).
C (6) NMAX    PHYSICAL UPPER LIMIT ON THE SIZE AND LENGTH
C             OF THE CALCULATION.  THE MAXIMUM NUMBER OF
C             COEFFICIENTS CALCULATED WILL BE THAT POWER OF TWO
C             LESS THAN OR EQUAL TO NMAX.  NMAX IS ASSUMED TO
C             BE AT LEAST 8.  (SEE NOTE(3) BELOW).
C (7) NCODE   .GE.0  THE ROUTINE WILL DO AS WELL AS IT CAN.
C             .LT.0  THE ROUTINE WILL ABORT AT AN EARLY STAGE
C             IF THE REQUIRED ACCURACY CANNOT BE ATTAINED
C             BECAUSE OF ROUND OFF ERROR.
```

```
C (12) NTAB   IN NORMAL RUNNING, NTAB SHOULD BE SET TO ZERO
C             BEFORE THE FIRST CALL TO ENTCRE, BUT LEFT ALONE
C             AFTER THAT.  (FOR MORE SOPHISTICATED USE, SEE
C             OUTPUT PARAMETERS (12) AND (13) AND NOTE(2) BELOW)
C
C     ** OUTPUT PARAMETERS **
C  (1),(2),(3),(4),(5),(6) IDENTICAL WITH INPUT VALUES.
C (7)   NCODE  RESULT STATUS INDICATOR.
C              TAKES ONE OF FIVE VALUES AS FOLLOWS,
C              = +1. CONVERGED NORMALLY.
C              = -1. DID NOT CONVERGE. NO ROUND OFF ERROR
C              TROUBLE.
C              = +2. CONVERGED, BUT WITH A HIGHER TOLERANCE
C              SET BY THE ROUND OFF LEVEL. (EPEST.GT.EPREQ)
C              = -2. DID NOT CONVERGE IN SPITE OF HIGHER
C              TOLERANCE SET BY ROUND OFF LEVEL.
C              =  0. RUN WAS ABORTED BECAUSE EPREQ IS
C              UNATTAINABLE DUE TO ROUND OFF LEVEL AND INPUT
C              NCODE IS NEGATIVE.
C (8)   EPEST  ESTIMATE OF ACTUAL UNIFORM ABSOLUTE ACCURACY
C              IN ALL TCOF.  EXCEPT, IF NCODE.EQ.0  ESTIMATE
C              OF ROUND OFF LEVEL.
C (9)   NTCOF  NUMBER OF NONTRIVIAL VALUES OF TCOF ACTUALLY
C              CALCULATED.  THEY ARE BASED ON NTCOF/2+2 CALLS
C              OF CFUN (THREE CALLS WERE FOR PURELY
C              REAL ARGUMENT).
C (10) TCOF    REAL DIMENSION (DIM).  APPROXIMATIONS TO THE
C              NORMALIZED TAYLOR COEFFICIENTS, EXCEPT WHEN
C              OUTPUT NCODE = 0. (SEE NOTE(3) BELOW)
C (11) WORK    INTERNAL WORKING AREA OF REAL DIMENSION (DIM)
C              (SEE NOTE(3) BELOW.) CONTENTS IS IDENTICAL WITH
C              THAT OF TCOF.
C (12) NTAB    NUMBER OF VALUES OF SINTAB AVAILABLE
C              (SEE NOTE (2) BELOW).
C (13) SINTAB REAL DIMENSION (DIM/4). (SEE NOTES (2) AND (3)
C              BELOW.) SINTAB(J+1) = SIN(PI*J/2*NTAB)
C              J = 0,1,2,...NTAB-1.
C              (A QUARTER CYCLE) OTHER LOCATIONS ARE EMPTY.
C
C     ** NOTES ON INPUT/OUTPUT PARAMETERS **
C  NOTE(1)**  NCODE IS USED BOTH AS INPUT AND OUTPUT PARAMETER.
C NORMALLY IT RETAINS THE VALUE +1 AND NEED NOT BE RESET
C BETWEEN NORMAL RUNS.
C  NOTE(2)**  THE APPEARANCE OF NTAB AND SINTAB IN THE
C CALLING SEQUENCE ALLOWS THE USER TO MAKE USE OF - OR TO
C PRECOMPUTE - THESE NUMBERS IN ANOTHER PART OF THE PROGRAM
C SHOULD HE SO DESIRE.  NTAB MUST BE A POWER OF TWO OR 0.
C  NOTE(3)**  THE APPEARANCE OF NMAX,TCOF,WORK AND SINTAB IN
C THE CALLING SEQUENCE ALLOWS THE SCOPE OF THE SUBPROGRAM AND
C THE AMOUNT OF STORAGE TO BE ASSIGNED BY THE CALLING
C PROGRAM, WHICH SHOULD CONTAIN A CONTROL STATEMENT TO THE
C FOLLOWING EFFECT -
C  REAL TCOF(DIM), WORK(DIM), SINTAB(DIM/4)
C WHERE DIM IS NORMALLY A POWER OF TWO.  NMAX IS NORMALLY
C EQUAL TO DIM,  BUT MAY BE LESS THAN DIM.
C  NOTE(4)**  CFUN(Z)  IS A USER PROVIDED COMPLEX VALUED
C FUNCTION SUBPROGRAM WITH A COMPLEX VALUED ARGUMENT.  THE
C CALLING PROGRAM MUST CONTAIN CONTROL STATEMENTS AS FOLLOWS -
C  EXTERNAL CFUN      COMPLEX CFUN
C
C     ** BOOKKEEPING PARAMETERS FOR STAGE ONE **
C  NCONV    1   CONVERGENCE ACHIEVED.
C          -1   NO CONVERGENCE ACHIEVED.
C  NROUND   1   NO ROUND OFF TROUBLE OBSERVED.
C           2   ROUND OFF TROUBLE OBSERVED.
C  NABORT   0   UPDATE TOLERANCE AND CONTINUE ON APPEARANCE OF
C               OF ROUND OFF TROUBLE.
C           1   TERMINATE WHEN ROUND OFF TROUBLE OBSERVED.
C  EXACT    THE EXACT VALUE OF TCOF(1) WHICH IS CFUN(ZETA).
C  SAFETY   THIS IS A SAFETY FACTOR BY WHICH THE ROUTINE AVOIDS
C           THE ROUND OFF LEVEL.  IT IS SET TO 10.0 AND
C           APPEARS ONLY IN THE COMBINATION (SAFETY*EPMACH).
C           TO ALTER THIS FACTOR, OR TO REMOVE THE ROUND OFF
C           ERROR GUARD COMPLETELY, THE USER NEED ONLY ADJUST
C           THE INPUT PARAMETER EPMACH APPROPRIATELY.
C
C     ** QUANTITIES CALCULATED IN STAGE THREE(A) **
C  THIS IS THE FIRST PART OF ITERATION NUMBER NTCOF. PRESENTLY
C AVAILABLE ARE -
C SINTAB(J+1) = SIN(PI*J/2*NTAB) , J = 0,1,2,...NTAB-1.
C WE REQUIRE THE SEQUENCE SIN(PI*J/2*(NTCOF/4)),
C J = 1,3,5,...(NTCOF/4-1).
C IF (NTCOF.LE.4*NTAB) THESE NUMBERS ARE ALREADY AVAILABLE IN
C THE SINTAB TABLE SPACED AT AN INTERVAL 2*NSPACE = 8*NTAB/NTCOF.
C OTHERWISE, NTCOF = 8*NTAB AND THE SINTAB TABLE IS UPDATED.
C THIS INVOLVES REARRANGING THE NTAB VALUES AVAILABLE,
C CALCULATING AND STORING NTAB NEW VALUES AND UPDATING
C NTAB TO 2*NTAB.
C
C     ** QUANTITIES CALCULATED IN STAGE THREE(B) **
C  ITERATIONS ARE NUMBERED 8,16,32,...AT THE END OF ITERATION
C NTCOF. THE NTCOF/2 + 1 COMPLEX FUNCTION VALUES AT
C ABSCISSAS REGULARLY SPACED ON UPPER HALF OF CIRCLE ARE
C STORED IN THE TCOF VECTOR AS FOLLOWS.
C  TCOF(J+1)   =  REAL PART OF CFUN(Z(J)) J=0,1,2,...NTCOF/2.
C  TCOF(NTCOF-J+1) = IMAGINARY PART OF CFUN(Z(J))
C                     J=1,2,...(NTCOF/2-1).
C WHERE
C  Z(J) = ZETA + RCIRC*CEXP(2*PI*EYE*J/NTCOF)
C THIS INVOLVES A REARRANGEMENT OF THE NTCOF/4 + 1 FUNCTION
C VALUES AVAILABLE AT THE START OF THE ITERATION AND THE
C CALCULATION OF A FURTHER NTCOF/4 FUNCTION VALUES.  IN
C ADDITION FMAX AND APPROX ARE CALCULATED.  THESE ARE
C  FMAX    MAXIMUM MODULUS OF THE FUNCTION VALUES SO FAR
C          ENCOUNTERED.
C  APPROX AN APPROXIMATION TO TCOF(1)
C          BASED ON THESE FUNCTION VALUES.
C
C     ** QUANTITIES CALCULATED AT STAGE THREE(C) **
C  ERROR1  CURRENT VALUE OF THE ERROR = ABS(APPROX-EXACT).
C  ERROR2,ERROR3,ERROR4  VALUES OF ERROR AT END OF THREE
C          PREVIOUS ITERATIONS.
C  EPMACH  MACHINE ACCURACY PARAMETER. (INPUT PARAMETER)
C  EPREQ   REQUIRED ACCURACY. (INPUT PARAMETER)
C  EPRO    HIGHEST ACCURACY REASONABLY ATTAINABLE IN VIEW OF
C          THE SIZE OF THE FUNCTION VALUES SO FAR ENCOUNTERED.
C          (=10.0*EPMACH*FMAX)
C  EPCOF   CURRENTLY REQUIRED ACCURACY (=AMAX1(EPREQ,EPRO)).
C  EPEST   ESTIMATE OF CURRENT ACCURACY. (THE MAXIMUM OF EPRO
C          AND A FUNCTION OF ERRORS 1,2,3 AND 4) (OUTPUT PARAMETER)
C
C ** CONVERGENCE AND TERMINATION CHECKS IN STAGE THREE(C) **
C (1)  USES FMAX TO RAISE EPCOF ABOVE ROUND OFF LEVEL.  IF
C  THIS IS NECESSARY AND THE INPUT VALUE OF NCODE IS NEGATIVE,
C  IT TERMINATES SETTING NCODE = 0.
C (2)  USES APPROX TO EVALUATE CONVERGENCE OF TCOF(1) TOWARDS
C  EXACT.  IT MAY ASSIGN CONVERGENCE AND GO TO STAGE FOUR(A)
C  SETTING NCODE = +1 OR +2.
C (3)  USES NMAX TO CHECK PHYSICAL LIMIT.  IF THIS HAS BEEN
C  REACHED, IT GOES TO STAGE FOUR(A) SETTING NCODE = -1 OR -2.
C (4)  OTHERWISE CONTINUES NEXT ITERATION BY GOING TO STAGE THREE
C
C  ** CALCULATION OF FIRST NTCOF TAYLOR COEFFICIENTS IN
C  **  STAGE FOUR(A)
C A VERSION OF THE FAST FOURIER TRANSFORM USING A WORK ARRAY
C IS USED.  THE ARRAY **WORK** IS USED ONLY DURING THIS STAGE.
C THE WORK ARRAY ALLOWS THE PERMUTING OF INDICES ASSOCIATED
C WITH IN-PLACE FFTS TO BE SUPPRESSED.  THE FFT CALCULATES
C THE NECCESSARY SUMMATIONS EXCEPT FOR DIVIDING BY NTCOF.
C
C ** SETTING OF REMAINING TAYLOR COEFFICIENTS IN STAGE FOUR(B) **
C  THE CONVERGENCE CRITERION ALLOWS US TO INFER THAT THE
C NORMALIZED TAYLOR COEFFICIENTS OF ORDER GREATER THAN NTCOF
C ARE ZERO TO ACCURACY  EPEST. THEY ARE EVALUATED AS BEING
C EXACTLY ZERO.
      COMPLEX CFUN
      REAL ZETA,RCIRC,EPREQ,EPMACH,EPEST
      INTEGER NMAX,NCODE,NTCOF,NTAB
      REAL TCOF (1), WORK (1), SINTAB (1)
      INTEGER NABORT,NCONV,NDISP,NDOLIM,NPREV,NROUND,NSPACE
      INTEGER J,JCONJ,JCOS,JFROM,JRCONJ,JREFL,JSIN,JTO
      REAL APPROX,COSDIF,EPCOF,EPMIN,EPRO,EP32,EP42
      REAL ERROR1,ERROR2,ERROR3,ERROR4,EXACT,FMAX,FVALIM
      REAL FVALRE,RCOS,RSIN,SAFETY,SCALE,SUPPER,TWOPI
      COMPLEX FVAL,ZVAL
      COMPLEX CMPLX
C ***    STAGE ONE    ***
C ---------------------
C INITIALISE BOOKKEEPING PARAMETERS AND EXACT VALUE OF TCOF(1).
      NROUND = 1
      NABORT = 0
      IF (NCODE.LT.0) NABORT = 1
      EPCOF = EPREQ
      SAFETY = 10.0
      ZVAL = CMPLX(ZETA,0.0)
      FVAL = CFUN(ZVAL)
      FVALRE = REAL(FVAL)
      EXACT = FVALRE
C ***    STAGE TWO    ***
C ---------------------
C FIRST THREE ITERATIONS ( THOSE WITH NTCOF = 1,2,4 ).
      ZVAL = CMPLX(ZETA+RCIRC,0.0)
      FVAL = CFUN(ZVAL)
      FVALRE = REAL(FVAL)
      APPROX = FVALRE
      FMAX = ABS(FVALRE)
      TCOF(1) = FVALRE
      ERROR3 = ABS(APPROX-EXACT)
      ZVAL = CMPLX(ZETA-RCIRC,0.0)
      FVAL = CFUN(ZVAL)
      FVALRE = REAL(FVAL)
      APPROX = 0.5*(APPROX+FVALRE)
      FMAX = AMAX1(FMAX,ABS(FVALRE))
      TCOF(3) = FVALRE
      ERROR2 = ABS(APPROX-EXACT)
      ZVAL = CMPLX(ZETA,RCIRC)
      FVAL = CFUN(ZVAL)
      FVALRE = REAL(FVAL)
      FVALIM = AIMAG(FVAL)
      APPROX = 0.5*(APPROX+FVALRE)
      FMAX = AMAX1(FMAX,CABS(FVAL))
      TCOF(2) = FVALRE
      TCOF(4) = FVALIM
      ERROR1 = ABS(APPROX-EXACT)
      NTCOF = 4
      EPRO = FMAX*SAFETY*EPMACH
      IF (EPRO.LT.EPCOF) GO TO 300
        EPCOF = EPRO
        NROUND = 2
        IF (NABORT.EQ.0) GO TO 300
        NCODE = 0
        EPEST = EPRO
        GO TO 470
C ***    STAGE THREE    ***
C -----------------------
C COMMENCE ITERATION NUMBER NTCOF.
  300 CONTINUE
      NPREV = NTCOF
      NTCOF = 2*NTCOF
C ***    STAGE THREE(A)    ***
C --------------------------
C UPDATE SINTAB TABLE IF NECESSARY.
      IF (4*NTAB.GE.NTCOF) GO TO 340
      IF (NTAB.GE.2) GO TO 310
        SINTAB(1) = 0.0
        SINTAB(2) = SQRT(0.5)
        NTAB = 2
        GO TO 340
  310 CONTINUE
      NDOLIM = NTAB-1
      DO 320 J = 1,NDOLIM
        JFROM = NTAB-J
        JTO = 2*JFROM
        SINTAB(JTO+1) = SINTAB(JFROM+1)
  320 CONTINUE
      NTAB = 2*NTAB
      TWOPI = 8.0*ATAN(1.0)
      COSDIF = COS(TWOPI/FLOAT(4*NTAB))
      NDOLIM = NTAB-3
      DO 330 J = 1,NDOLIM,2
        SINTAB(J+1) = (0.5*SINTAB(J)+0.5*SINTAB(J+2))/COSDIF
  330 CONTINUE
      SINTAB(NTAB) = COSDIF
  340 CONTINUE
C ***    STAGE THREE(B)    ***
C --------------------------
C UPDATE LIST OF FUNCTION VALUES IN TCOF,
C CALCULATE FMAX AND APPROX.
      NDOLIM = NPREV-1
      DO 350 J = 1,NDOLIM
        JFROM = NPREV-J
        JTO = 2*JFROM
        TCOF(JTO+1) = TCOF(JFROM+1)
  350 CONTINUE
      SUPPER = 0.0
      NDOLIM = (NPREV/2)-1
      NSPACE = (4*NTAB)/NTCOF
      DO 360 J = 1,NDOLIM,2
        JSIN = J*NSPACE
        JCOS = NTAB-JSIN
        RSIN = RCIRC*SINTAB(JSIN+1)
        RCOS = RCIRC*SINTAB(JCOS+1)
        JCONJ = NTCOF-J
        ZVAL = CMPLX(ZETA+RCOS,RSIN)
        FVAL = CFUN(ZVAL)
        FVALRE = REAL(FVAL)
        FVALIM = AIMAG(FVAL)
        SUPPER = SUPPER+FVALRE
        FMAX = AMAX1(FMAX,CABS(FVAL))
        TCOF(J+1) = FVALRE
        TCOF(JCONJ+1) = FVALIM
        JREFL = NPREV-J
        JRCONJ = NTCOF-JREFL
        ZVAL = CMPLX(ZETA-RCOS,RSIN)
        FVAL = CFUN(ZVAL)
        FVALRE = REAL(FVAL)
        FVALIM = AIMAG(FVAL)
        SUPPER = SUPPER+FVALRE
        FMAX = AMAX1(FMAX,CABS(FVAL))
        TCOF(JREFL+1) = FVALRE
        TCOF(JRCONJ+1) = FVALIM
  360 CONTINUE
      APPROX = 0.5*APPROX+SUPPER/FLOAT(NPREV)
C ***   STAGE THREE(C)    ***
C -------------------------
C CONVERGENCE AND TERMINATION CHECK.
      ERROR4 = ERROR3
      ERROR3 = ERROR2
      ERROR2 = ERROR1
      ERROR1 = ABS(APPROX-EXACT)
      EPRO = FMAX*SAFETY*EPMACH
      IF (EPRO.LT.EPCOF) GO TO 370
        EPCOF = EPRO
        NROUND = 2
        IF (NABORT.EQ.0) GO TO 370
        NCODE = 0
        EPEST = EPRO
        GO TO 470
  370 CONTINUE
      ERROR4 = AMAX1(ERROR4,EPRO)
      ERROR3 = AMAX1(ERROR3,EPRO)
      EP42 = ERROR2*((ERROR2/ERROR4)**(4.0/3.0))
      EP32 = ERROR2*((ERROR2/ERROR3)**2)
      EPMIN = AMIN1(ERROR2,EP32,EP42)
      EPEST = AMAX1(ERROR1,EPMIN,EPRO)
      IF (EPEST.GT.EPCOF) GO TO 380
        NCONV = 1
        GO TO 400
  380 CONTINUE
      IF (2*NTCOF.LE.NMAX) GO TO 300
        NCONV = -1
C ***   STAGE FOUR(A)    ***
C -------------------------
C CALCULATION OF FIRST NTCOF TAYLOR COEFFICIENTS USING F.F.T.
  400 CONTINUE
      NCODE = NCONV*NROUND
      NDISP = NTCOF
  410 CONTINUE
      NDISP = NDISP/2
      CALL HFCOF (NTCOF,NDISP,TCOF,WORK,NTAB,SINTAB)
      IF (NDISP.GT.1) GO TO 430
      DO 420 J = 1,NTCOF
        TCOF(J) = WORK(J)
  420 CONTINUE
      GO TO 440
  430 CONTINUE
      NDISP = NDISP/2
      CALL HFCOF (NTCOF,NDISP,WORK,TCOF,NTAB,SINTAB)
      IF (NDISP.GT.1) GO TO 410
  440 CONTINUE
      SCALE = 1.0/FLOAT(NTCOF)
      DO 450  J = 1,NTCOF
        TCOF(J) = TCOF(J)*SCALE
        WORK(J) = TCOF(J)
  450 CONTINUE
C ***   STAGE FOUR(B)    ***
C -------------------------
C SETTING OF REMAINING TAYLOR COEFFICIENTS.
      IF (NTCOF.GE.NMAX) GO TO 470
      NOOLIM = NTCOF+1
      DO 460  J = NUOLIM,NMAX
        TCOF(J) = 0.0
        WORK(J) = 0.0
  460 CONTINUE
  470 CONTINUE
      RETURN
C END OF ENTCRE
      END
      SUBROUTINE HFCOF ( NTCOF, NDISP, TCOF, WORK, NTAB, SINTAB )
C
C ** HERMITIAN FOURIER COEFFICIENTS **
C
C      **  GENERAL PURPOSE  **
C THIS ROUTINE DOES ONE PASS OF A FAST FOURIER TRANSFORM.
C THE INDEXING IS ARRANGED SO THAT THE COEFFICIENTS ARE IN
C ORDER AT THE END OF THE LAST PASS.  THIS INDEXING REQUIRES
C THE USE OF SEPARATE ARRAYS FOR INPUT AND OUTPUT OF THE
C PARTIAL RESULTS.  THIS ROUTINE IS CALLED ONCE FOR EACH PASS.
C
C      **  INPUT PARAMETERS  **
C  (1)  NTCOF     NUMBER OF COEFFICIENTS TO BE PROCESSED.
C  (2)  NDISP     MAXIMUM VALUE OF DISPLACEMENT INDEX.
C  (3)  TCOF      (REAL) INPUT ARRAY.
C  (5)  NTAB      NUMBER OF ENTRIES IN SINTAB.
C  (6)  SINTAB    (REAL) TABLE OF VALUES OF SINE.
C                 SINTAB(J+1)=SIN(PI*J/2*NTAB), J=0,1,2...NTAB-1
C
```

```
C
C      **  OUTPUT PARAMETERS  **
C  (4)  WORK      (REAL) OUTPUT ARRAY.
C
C      **  INDEXING OF ARRAYS  **
C THE TWO POINT FOURIER TRANSFORM IS APPLIED TO THE POINTS
C OF TCOF WITH INDICES
C      JDISP*NPREV+JREPL  AND  JDISP*NPREV+JREPL+NHALF
C THE RESULTS ARE MODIFIED BY THE APPROPRIATE TWIDDLE FACTOR
C AND STORED IN WORK WITH INDICES
C      JDISP*NNEXT+JREPL  AND  JDISP*NNEXT+JREPL+NPREV
C  WHERE
C      NDISP      PRODUCT OF REMAINING FACTORS.
C      NPREV      PRODUCT OF PREVIOUS FACTORS.
C      NNEXT      PRODUCT OF PREVIOUS AND CURRENT FACTORS.
C      NHALF      PRODUCT OF PREVIOUS AND REMAINING FACTORS.
C      JREPL      REPLICATION INDEX = 1,2,...NPREV.
C      JDISP      HERMITIAN SYMMETRY IN THIS INDEX RESULTS IN
C                 THREE CASES.
C                 1)  INITIAL POINT - JDISP=0. INPUT POINTS
C                 ARE PURELY REAL AND OUTPUT POINTS ARE
C                 PURELY REAL.
C                 2)  MIDDLE POINT - JDISP=NDISP/2 - NOT
C                 ALWAYS PRESENT.  INPUT POINTS ARE COMPLEX AND
C                 OUTPUT POINTS ARE PURELY REAL.
C                 3)  INTERMEDIATE POINTS - JDISP=1,2,..(NDISP/2-1)
C                 - NOT ALWAYS PRESENT.  INPUT POINTS ARE
C                 COMPLEX AND OUTPUT POINTS ARE COMPLEX.
C
C ON INPUT, THE HERMITIAN SYMMETRY IS IN A BLOCK OF LENGTH
C 2*NDISP, I.E. THE POINT CONJUGATE TO JDISP IS 2*NDISP-JDISP.
C ON OUTPUT, THE HERMITIAN SYMMETRY IS IN A BLOCK OF LENGTH
C NDISP, I.E. THE POINT CONJUGATE TO JDISP IS NDISP-JDISP.
C A HERMITIAN SYMMETRIC BLOCK HAS REAL PARTS AT THE FRONT
C IMAGINARY PARTS (WHEN THEY EXIST) AT THE CONJUGATE
C POSITIONS AT THE BACK.
C
C THE TWIDDLE FACTOR CEXP(-PI*EYE*J/NDISP), J=1,2...(NDISP/2-1)
C IS OBTAINED AS SEPARATE REAL AND IMAGINARY PARTS FROM
C THE SINTAB TABLE.  THE IMAGINARY PART SIN(PI*J/NDISP) IS
C FOUND AT A SPACING OF NSPACE=2*NTAB/NDISP IN SINTAB.
C THE REAL PART IS FOUND AT A CONJUGATE POSITION IN THE TABLE.
C
      INTEGER NTCOF,NDISP,NTAB
      REAL TCOF (1), WORK (1), SINTAB (1)
      REAL CS,IS,IU,I0,I1,RS,RU,R0,R1,SN
      INTEGER JCONJ,JCOS,JDISP,JREPL,JSIN,JT,JTC,JW,JWC,KT0,KT1
      INTEGER KT2,KT3,KW0,KW1,KW2,KW3,NHALF,NMIDL,NNEXT,NPREV,NSPACE
      NHALF = NTCOF/2
      NPREV = NTCOF/(2*NDISP)
      NNEXT = NTCOF/NDISP
      NMIDL = (NDISP-1)/2
      NSPACE = (2*NTAB)/NDISP
C INITIAL POINTS OF BLOCKS.
      DO 100 JREPL = 1,NPREV
        KT0 = JREPL
        KT1 = KT0+NHALF
        KW0 = JREPL
        KW1 = KW0+NPREV
        R0 = TCOF(KT0)
        R1 = TCOF(KT1)
        WORK(KW0) = R0+R1
        WORK(KW1) = R0-R1
  100 CONTINUE
C INTERMEDIATE POINTS OF BLOCKS.
      IF (NMIDL.LT.1) GO TO 400
      DO 300 JDISP = 1,NMIDL
        JCONJ = NDISP-JDISP
        JSIN = JDISP*NSPACE
        JCOS = NTAB-JSIN
        SN = SINTAB(JSIN+1)
        CS = SINTAB(JCOS+1)
        JT = JDISP*NPREV
        JTC = JCONJ*NPREV
        JW = JDISP*NNEXT
        JWC = JCONJ*NNEXT
        DO 200  JREPL = 1,NPREV
          KT0 = JT+JREPL
          KT1 = KT0+NHALF
          KT2 = JTC+JREPL
          KT3 = KT2+NHALF
          KW0 = JW+JREPL
          KW1 = KW0+NPREV
          KW2 = JWC+JREPL
          KW3 = KW2+NPREV
          R0 = TCOF(KT0)
          I0 = TCOF(KT3)
          R1 = TCOF(KT2)
          I1 = -TCOF(KT1)
          RS = R0+R1
          IS = I0+I1
          RU = R0-R1
          IU = I0-I1
          WORK(KW0) = RS
          WORK(KW2) = IS
          WORK(KW1) = RU*CS+IU*SN
          WORK(KW3) = IU*CS-RU*SN
  200   CONTINUE
  300 CONTINUE
  400 CONTINUE
C MIDDLE POINTS OF BLOCKS.
      IF (NDISP.LE.1) GO TO 600
      JT = (NDISP/2)*NPREV
      JW = (NDISP/2)*NNEXT
      DO 500 JREPL = 1,NPREV
        KT0 = JT+JREPL
        KT1 = KT0+NHALF
        KW0 = JW+JREPL
        KW1 = KW0+NPREV
        R0 = TCOF(KT0)
        I0 = TCOF(KT1)
        WORK(KW0) = 2.0*R0
        WORK(KW1) = 2.0*I0
  500 CONTINUE
  600 CONTINUE
      RETURN
C END OF HFCOF
      END
      SUBROUTINE ENTCAF ( CFUN, ZETA, RCIRC, EPREQ, EPMACH, NMAX, NCODE,
     . EPEST, NTCOF, TCOF, WORK, NTAB, EXPTAB )
C
```

```
C
C ** EVALUATION OF NORMALIZED TAYLOR COEFFICIENTS **
C **              OF AN ANALYTIC FUNCTION             **
C
C      **  GENERAL PURPOSE  **
C THIS ROUTINE EVALUATES A SET OF NORMALIZED TAYLOR COEFFICIENTS
C TCOF(J+1) = (RCIRC**J) * (J-TH DERIVATIVE OF CFUN(Z) AT Z=ZETA)
C DIVIDED BY FACTORIAL(J) ... J = 0,1,2,3...NMAX-1.
C TO A UNIFORM ABSOLUTE ACCURACY **EPEST** USING FUNCTION
C VALUES OF CFUN(Z) AT POINTS IN THE COMPLEX PLANE LYING ON
C THE CIRCLE OF RADIUS **RCIRC** WITH CENTER AT Z = ZETA.
C
C      ** THEORETICAL RESTRICTIONS **
C RCIRC MUST BE SMALLER THAN THE RADIUS OF CONVERGENCE OF
C THE TAYLOR SERIES.  THE PROBLEM HAS TO BE REFORMULATED
C SHOULD CFUN(Z) HAPPEN TO BE AN ODD FUNCTION OF (Z - ZETA),
C THAT IS IF THE RELATION **-CFUN(-(Z-ZETA))=CFUN(Z-ZETA)**
C IS AN IDENTITY.
C
C      ** REQUIREMENTS FOR CALLING PROGRAM **
C CALLING PROGRAM MUST CONTAIN CONTROL STATEMENTS DESCRIBED
C IN NOTES (3) AND (4) BELOW.  IT MUST ALSO ASSIGN VALUES TO
C INPUT PARAMETERS.  THE ROUTINE REQUIRES TWO SUBPROGRAMS.
C CFCOF (LISTED AFTER ENTCAF) AND CFUN (SEE NOTE(4) BELOW).
C
C      **INPUT PARAMETERS**
C  (1) CFUN   NAME OF COMPLEX FUNCTION SUBPROGRAM.
C  (2) ZETA   COMPLEX POINT ABOUT WHICH TAYLOR EXPANSION
C             IS REQUIRED.
C  (3) RCIRC  RADIUS (REAL)
C  (4) EPREQ  THE ABSOLUTE ACCURACY (REAL) TO WHICH THE
C             NORMALIZED TAYLOR COEFFICIENTS, TCOF(J), ARE REQUIRED
C  (5) EPMACH THE MACHINE ACCURACY PARAMETER (REAL) (OR AN
C             UPPER BOUND ON THE RELATIVE ACCURACY OF
C             QUANTITIES LIKELY TO BE ENCOUNTERED).
C  (6) NMAX   PHYSICAL UPPER LIMIT ON THE SIZE AND LENGTH OF
C             THE CALCULATION.  THE MAXIMUM NUMBER OF
C             COEFFICIENTS CALCULATED WILL BE THAT POWER OF
C             TWO LESS THAN OR EQUAL TO NMAX.  NMAX IS
C             ASSUMED TO BE AT LEAST 4. (SEE NOTE(3) BELOW.)
C  (7) NCODE  .GE.0 THE ROUTINE WILL DO AS WELL AS IT CAN.
C             .LT.0 THE ROUTINE WILL ABORT AT AN EARLY
C             STAGE IF THE REQUIRED ACCURACY CANNOT BE
C             ATTAINED BECAUSE OF ROUND OFF ERROR.
C (12) NTAB   IN NORMAL RUNNING, NTAB SHOULD BE SET TO ZERO
C             BEFORE THE FIRST CALL TO ENTCAF, BUT LEFT ALONE
C             AFTER THAT. (FOR MORE SOPHISTICATED USE, SEE
C             OUTPUT PARAMETERS (12) AND (13) AND NOTE(2)
C             BELOW.)
C
C      ** OUTPUT PARAMETERS **
C  (1),(2),(3),(4),(5),(6) IDENTICAL WITH INPUT VALUES.
C  (7) NCODE  RESULT STATUS INDICATOR.  TAKES ONE OF FIVE
C             VALUES AS FOLLOWS.
C             =+1. CONVERGED NORMALLY.
C             =-1. DID NOT CONVERGE. NO ROUND OFF ERROR TROUBLE
C             =+2. CONVERGED, BUT WITH A HIGHER TOLERANCE SET
C             BY THE ROUND OFF LEVEL. (EPEST.GT.EPREQ)
C             =-2. DID NOT CONVERGE IN SPITE OF HIGHER
C             TOLERANCE SET BY ROUND OFF LEVEL.
C             = 0. RUN WAS ABORTED BECAUSE EPREQ IS
C             UNATTAINABLE DUE TO ROUND OFF LEVEL AND INPUT
C             NCODE IS NEGATIVE.
C  (8) EPEST  ESTIMATE OF ACTUAL UNIFORM ABSOLUTE ACCURACY
C             IN ALL TCOF, EXCEPT IF NCODE.EQ.0 ESTIMATE OF
C             ROUND OFF LEVEL.
C  (9) NTCOF  NUMBER OF NONTRIVIAL VALUES OF TCOF ACTUALLY
C             CALCULATED.  THEY ARE BASED ON NTCOF+1 CALLS
C             OF CFUN.
C (10) TCOF   COMPLEX DIMENSION (DIM).  APPROXIMATIONS TO
C             THE NORMALIZED TAYLOR COEFFICIENTS, EXCEPT WHEN
C             OUTPUT NCODE = 0. (SEE NOTE(3) BELOW.)
C (11) WORK   INTERNAL WORKING AREA OF COMPLEX DIMENSION (DIM).
C             (SEE NOTE(3) BELOW). CONTENTS IS IDENTICAL
C             WITH THAT OF TCOF.
C (12) EXPTAB COMPLEX DIMENSION (DIM/2). (SEE NOTES (2) AND
C             (3) BELOW.)  EXPTAB(J+1) = CEXP(PI*EYE*J/NTAB)
C             J = 0,1,2,...,NTAB-1.   (A HALF CYCLE)
C             OTHER LOCATIONS ARE EMPTY.
C
C      ** NOTES ON INPUT/OUTPUT PARAMETERS **
C NOTE(1)** NCODE IS USED BOTH AS INPUT AND OUTPUT PARAMETER.
C  NORMALLY IT RETAINS THE VALUE +1 AND NEED NOT BE RESET
C  BETWEEN NORMAL RUNS.
C NOTE(2)** THE APPEARANCE OF NTAB AND EXPTAB IN THE CALLING
C  SEQUENCE ALLOWS THE USER TO MAKE USE OF - OR TO PRECOMPUTE -
C  THESE NUMBERS IN ANOTHER PART OF THE PROGRAM SHOULD HE
C  SO DESIRE.  NTAB MUST BE A POWER OF TWO OR 0.
C NOTE(3)** THE APPEARANCE OF NMAX, TCOF, WORK, AND EXPTAB
C  IN THE CALLING SEQUENCE ALLOWS THE SCOPE OF THE SUBPROGRAM
C  AND THE AMOUNT OF STORAGE TO BE ASSIGNED BY THE CALLING
C  PROGRAM, WHICH SHOULD CONTAIN A CONTROL STATEMENT TO THE
C  FOLLOWING EFFECT
C  COMPLEX TCOF(DIM), WORK(DIM), EXPTAB(DIM/2)
C  WHERE DIM IS NORMALLY A POWER OF TWO.  NMAX IS NORMALLY
C  EQUAL TO DIM, BUT MAY BE LESS THAN DIM.
C NOTE(4)** CFUN(Z) IS A USER PROVIDED COMPLEX VALUED
C  FUNCTION SUBPROGRAM WITH A COMPLEX VALUED ARGUMENT.  THE
C  CALLING PROGRAM MUST CONTAIN CONTROL STATEMENTS AS FOLLOWS
C  EXTERNAL CFUN
C  COMPLEX CFUN
C
C      ** BOOKKEEPING PARAMETERS FOR STAGE ONE **
C  NCONV    1  CONVERGENCE ACHIEVED.
C          -1  NO CONVERGENCE ACHIEVED.
C  NROUND   1  NO ROUND OFF TROUBLE OBSERVED.
C           2  ROUND OFF TROUBLE OBSERVED.
C  NABORT   0  UPDATE TOLERANCE AND CONTINUE ON APPEARANCE OF
C              ROUND OFF TROUBLE.
C           1  TERMINATE WHEN ROUND OFF TROUBLE OBSERVED.
C  EXACT   THE EXACT VALUE OF TCOF(1) WHICH IS CFUN(ZETA).
C  SAFETY  THIS IS A SAFETY FACTOR BY WHICH THE ROUTINE AVOIDS
C      THE ROUND OFF LEVEL.  IT IS SET TO 10.0 AND APPEARS
C      ONLY IN THE COMBINATION (SAFETY*EPMACH).  TO ALTER THIS
C      FACTOR, OR TO REMOVE THE ROUND OFF ERROR GUARD
C      COMPLETELY, THE USER NEED ONLY ADJUST THE INPUT
C      PARAMETER EPMACH APPROPRIATELY.
C
C
C      ** QUANTITIES CALCULATED IN STAGE THREE(A) **
C  THIS IS THE FIRST PART OF ITERATION NUMBER NTCOF. PRESENTLY
C  AVAILABLE ARE  EXPTAB(J+1) = CEXP(PI*EYE*J/NTAB),
C  J = 0,1,2,...NTAB-1.
C  WE REQUIRE THE SEQUENCE  CEXP(PI*EYE*J/NTCOF/2)),
C  J= 1,3,5,...(NTCOF/2-1).
C  IF (NTCOF.LE.2*NTAB) THESE NUMBERS ARE ALREADY AVAILABLE
C  IN THE EXPTAB TABLE SPACED AT AN INTERVAL  2*NSPACE = 4*NTAB/NTCOF.
C  OTHERWISE, NTCOF = 4*NTAB AND THE EXPTAB TABLE IS UPDATED.
C  THIS INVOLVES REARRANGING THE NTAB VALUES AVAILABLE,
C  CALCULATING AND STORING NTAB NEW VALUES AND UPDATING
C  NTAB TO 2*NTAB.
C
C      ** QUANTITIES CALCULATED IN STAGE THREE(B) **
C  ITERATIONS ARE NUMBERED 4,8,16,... AT THE END OF
C  ITERATION NUMBER NTCOF, THE NTCOF COMPLEX FUNCTION
C  VALUES AT ABCISSAS REGULARLY SPACED ON CIRCLE ARE STORED
C  IN THE TCOF VECTOR AS FOLLOWS
C   TCOF(J+1) = CFUN(Z(J))  J=0,1,2,...,NTCOF-1
C  WHERE
C   Z(J) = ZETA + RCIRC*CEXP(2*PI*EYE*J/NTCOF)
C  THIS INVLOVES A REARRANGEMENT OF THE NTCOF/2 FUNCTION
C  VALUES AVAILABLE AT THE START OF THE ITERATION AND THE
C  CALCULATION OF A FURTHER NTCOF/2 FUNCTION VALUES.  IN
C  ADDITION FMAX AND APPROX ARE CALCULATED.  THESE ARE
C   FMAX    MAXIMUM MODULUS OF THE FUNCTION VALUES SO FAR
C           ENCOUNTERED.
C   APPROX AN APPROXIMATION TO TCOF(1) BASED ON THESE
C           FUNCTION VALUES.
C
C      ** QUANTITIES CALCULATED AT STAGE THREE(C) **
C ERROR1 CURRENT VALUE OF THE ERROR = CABS(APPROX-EXACT).
C ERROR2, ERROR3, ERROR4 VALUES OF ERROR AT END OF THREE
C        PREVIOUS ITERATIONS.
C EPMACH MACHINE ACCURACY PARAMETER. (INPUT PARAMETER)
C EPREQ  REQUIRED ACCURACY. (INPUT PARAMETER)
C EPRO   HIGHEST  ACCURACY REASONABLY ATTAINABLE IN VIEW OF
C        THE SIZE OF THE FUNCTION VALUES SO FAR ENCOUNTERED.
C        (=10.0*EPMACH*FMAX)
C EPCOF  CURRENTLY REQUIRED ACCURACY (=AMAX1(EPREQ,EPRO)).
C EPEST  ESTIMATE OF CURRENT ACCURACY. (THE MAXIMUM OF EPRO AND
C        A FUNCTION OF ERRORS 1,2,3 AND 4. (OUTPUT PARAMETER)
C
C      ** CONVERGENCE AND TERMINATION CHECKS IN STAGE THREE(C) **
C (1) USES FMAX TO RAISE EPCOF ABOVE ROUND OFF LEVEL.
C  IF THIS NECESSARY AND THE INPUT VALUE OF NCODE IS NEGATIVE,
C  IT TERMINATES SETTING NCODE=0.
C (2) USES APPROX TO EVALUATE CONVERGENCE OF TCOF(1) TOWARDS
C  EXACT.  IT MAY ASSIGN CONVERGENCE AND GO TO STAGE FOUR(A)
C  SETTING NCODE=+1 OR +2. (CONVERGENCE IS NOT CHECKED FOR
C  FOUR OR FEWER POINTS).
C (3) USES NMAX TO CHECK PHYSICAL LIMIT.  IF THIS HAS BEEN
C  REACHED, IT GOES TO STAGE FOUR(A) SETTING NCODE=-1 OR -2.
C (4) OTHERWISE CONTINUES NEXT ITERATION BY GOING TO STAGE
C  THREE.
C
C **CALCULATION OF FIRST NTCOF TAYLOR COEFFICIENTS IN STAGE FOUR(A)
C  A VERSION OF THE FAST FOURIER TRANSFORM USING A WORK ARRAY
C IS USED.  THE ARRAY **WORK** IS USED ONLY DURING THIS STAGE.
C THE WORK ARRAY ALLOWS THE PERMUTING OF INDICES ASSOCIATED
C WITH IN-PLACE FFTS TO BE SUPPRESSED.  THE FFT CALCULATES
C THE NECCESSARY SUMMATIONS EXCEPT FOR DIVIDING BY NTCOF.
C
C **SETTING OF REMAINING TAYLOR COEFFICIENTS IN STAGE FOUR(B)
C  THE CONVERGENCE CRITERION ALLOWS US TO INFER THAT THE
C NORMALIZED TAYLOR COEFFICIENTS OF ORDER GREATER THAN NTCOF
C ARE ZERO TO ACCURACY EPEST.
C THEY ARE EVALUATED AS BEING EXACTLY ZERO.
      COMPLEX CFUN
      COMPLEX ZETA
      REAL RCIRC,EPREQ,EPMACH,EPEST
      INTEGER NMAX,NCODE,NTCOF,NTAB
      COMPLEX TCOF (1), WORK (1), EXPTAB (1)
      INTEGER NABORT,NCONV,NDISP,NDOLIM,NPREV,NROUND,NSPACE
      REAL COSDIF,EPCOF,EPMIN,EPRO,EP32,EP42,ERROR1,ERROR2
      REAL ERROR3,ERROR4,FMAX,SAFETY,SCALE,TWOPI
      COMPLEX APPROX,EXACT,FVAL,REXP,SUM,ZVAL
      INTEGER J,JCONJ,JFROM,JTAB,JTO
      COMPLEX CMPLX,CONJG
C ***    STAGE ONE    ***
C --------------------
C INITIALISE BOOKKEEPING PARAMETERS AND EXACT VALUE OF TCOF(1).
      NROUND = 1
      NABORT = 0
      IF (NCODE.LT.0) NABORT = 1
      EPCOF = EPREQ
      SAFETY = 10.0
      ZVAL = ZETA
      FVAL = CFUN(ZVAL)
      EXACT = FVAL
C ***    STAGE TWO    ***
C --------------------
C FIRST TWO ITERATIONS ( THOSE WITH NTCOF = 1,2 ).
      ERROR3 = 0.0
      ZVAL = ZETA+CMPLX(RCIRC,0.0)
      FVAL = CFUN(ZVAL)
      APPROX = FVAL
      FMAX = CABS(FVAL)
      TCOF(1) = FVAL
      ERROR2 = CABS(APPROX-EXACT)
      ZVAL = ZETA-CMPLX(RCIRC,0.0)
      FVAL = CFUN(ZVAL)
      APPROX = 0.5*(APPROX+FVAL)
      FMAX = AMAX1(FMAX,CABS(FVAL))
      TCOF(2) = FVAL
      ERROR1 = CABS(APPROX-EXACT)
      NTCOF = 2
C ***    STAGE THREE    ***
C ----------------------
C COMMENCE ITERATION NUMBER NTCOF.
  300 CONTINUE
      NPREV = NTCOF
      NTCOF = 2*NTCOF
C ***    STAGE THREE(A)    ***
C -------------------------
C UPDATE EXPTAB TABLE IF NECESSARY.
      IF (2*NTAB.GE.NTCOF) GO TO 340
      IF (NTAB.GE.2) GO TO 310
        EXPTAB(1) = (1.0,0.0)
        EXPTAB(2) = (0.0,1.0)
        NTAB = 2
        GO TO 340
  310 CONTINUE
      NDOLIM = NTAB-1
      DO 320 J = 1,NDOLIM
        JFROM = NTAB-J
        JTO = 2*JFROM
        EXPTAB(JTO+1) = EXPTAB(JFROM+1)
  320 CONTINUE
      NTAB = 2*NTAB
      TWOPI = 8.0*ATAN(1.0)
      COSDIF = COS(TWOPI/FLOAT(2*NTAB))
      NDOLIM = NTAB-3
      DO 330 J = 1,NDOLIM,2
        EXPTAB(J+1) = (0.5*EXPTAB(J)+0.5*EXPTAB(J+2))/COSDIF
  330 CONTINUE
      EXPTAB(NTAB) = (0.5*EXPTAB(NTAB-1)-(0.5,0.0))/COSDIF
  340 CONTINUE
C ***   STAGE THREE(B)    ***
C -------------------------
C UPDATE LIST OF FUNCTION VALUES IN TCOF.
C CALCULATE FMAX AND APPROX.
      NDOLIM = NPREV-1
      DO 350 J = 1,NDOLIM
        JFROM = NPREV-J
        JTO = 2*JFROM
        TCOF(JTO+1) = TCOF(JFROM+1)
  350 CONTINUE
      SUM = (0.0,0.0)
      NSPACE = (2*NTAB)/NTCOF
      DO 360 J = 1,NDOLIM,2
        JTAB = J*NSPACE
        REXP = RCIRC*EXPTAB(JTAB+1)
        ZVAL = ZETA+REXP
        FVAL = CFUN(ZVAL)
        SUM = SUM+FVAL
        FMAX = AMAX1(FMAX,CABS(FVAL))
        TCOF(J+1) = FVAL
        JCONJ = NTCOF-J
        ZVAL = ZETA+CONJG(REXP)
        FVAL = CFUN(ZVAL)
        SUM = SUM+FVAL
        FMAX = AMAX1(FMAX,CABS(FVAL))
        TCOF(JCONJ+1) = FVAL
  360 CONTINUE
      APPROX = 0.5*APPROX+SUM/FLOAT(NTCOF)
C ***   STAGE THREE(C)    ***
C -------------------------
C CONVERGENCE AND TERMINATION CHECK.
      ERROR4 = ERROR3
      ERROR3 = ERROR2
      ERROR2 = ERROR1
      ERROR1 = CABS(APPROX-EXACT)
      EPRO = FMAX*SAFETY*EPMACH
      IF (EPRO.LT.EPCOF) GO TO 370
        EPCOF = EPRO
        NROUND = 2
        IF (NABORT.EQ.0) GO TO 370
        NCODE = 0
        EPEST = EPRO
        GO TO 470
  370 CONTINUE
      IF (NTCOF.LE.4) GO TO 380
      ERROR4 = AMAX1(ERROR4,EPRO)
      ERROR3 = AMAX1(ERROR3,EPRO)
      EP42 = ERROR2*((ERROR2/ERROR4)**(4.0/3.0))
      EP32 = ERROR2*((ERROR2/ERROR3)**2)
      EPMIN = AMIN1(ERROR2,EP32,EP42)
      EPEST = AMAX1(ERROR1,EPMIN,EPRO)
      IF (EPEST.GT.EPCOF) GO TO 380
        NCONV = 1
        GO TO 400
  380 CONTINUE
      IF (2*NTCOF.LE.NMAX) GO TO 300
        NCONV = -1
C ***   STAGE FOUR(A)    ***
C -------------------------
C CALCULATION OF FIRST NTCOF TAYLOR COEFFICIENTS USING F.F.T.
  400 CONTINUE
      NCODE = NCONV*NROUND
      NDISP = NTCOF
  410 CONTINUE
      NDISP = NDISP/2
      CALL CFCOF (NTCOF,NDISP,TCOF,WORK,NTAB,EXPTAB)
      IF (NDISP.GT.1) GO TO 430
      DO 420 J = 1,NTCOF
        TCOF(J) = WORK(J)
  420 CONTINUE
      GO TO 440
  430 CONTINUE
      NDISP = NDISP/2
      CALL CFCOF (NTCOF,NDISP,WORK,TCOF,NTAB,EXPTAB)
      IF (NDISP.GT.1) GO TO 410
  440 CONTINUE
      SCALE = 1.0/FLOAT(NTCOF)
      DO 450 J = 1,NTCOF
        TCOF(J) = TCOF(J)*SCALE
        WORK(J) = TCOF(J)
  450 CONTINUE
C ***   STAGE FOUR(B)    ***
C -------------------------
C SETTING OF REMAINING TAYLOR COEFFICIENTS.
      IF (NTCOF.GE.NMAX) GO TO 470
      NDOLIM = NTCOF+1
      DO 460 J = NDOLIM,NMAX
        TCOF(J) = (0.0,0.0)
        WORK(J) = (0.0,0.0)
  460 CONTINUE
  470 CONTINUE
      RETURN
C END OF ENTCAF
      END
      SUBROUTINE CFCOF ( NTCOF, NDISP, TCOF, WORK, NTAB, EXPTAB )
C
C ** COMPLEX FOURIER COEFFICIENTS **
C      **  GENERAL PURPOSE  **
C THIS ROUTINE DOES ONE PASS OF A FAST FOURIER TRANSFORM.
C THE INDEXING IS ARRANGED SO THAT THE COEFFICIENTS ARE IN
C ORDER AT THE END OF THE LAST PASS.  THIS INDEXING REQUIRES
C THE USE OF SEPARATE ARRAYS FOR INPUT AND OUTPUT OF THE
C PARTIAL RESULTS.  THIS ROUTINE IS CALLED ONCE FOR
C EACH PASS.
C
C      **  INPUT PARAMETERS  **
C  (1)  NTCOF    NUMBER OF COEFFICIENTS TO BE PROCESSED.
C  (2)  NDISP    MAXIMUM VALUE OF DISPLACEMENT INDEX.
C  (3)  TCOF     (COMPLEX) INPUT ARRAY.
C  (5)  NTAB     NUMBER OF ENTRIES IN EXPTAB.
C  (6)  EXPTAB   (COMPLEX) TABLE OF VALUES OF COMPLEX EXPONENTIAL.
C                EXPTAB(J+1) = CEXP(PI*EYE*J/NTAB),
C                J = 0,1,2,...,NTAB-1.
C
C      **  OUTPUT PARAMETERS  **
C  (4)  WORK     (COMPLEX) OUTPUT ARRAY.
C
C      **  INDEXING OF ARRAYS  **
C THE TWO POINT FOURIER TRANSFORM IS APPLIED TO THE POINTS
C OF TCOF WITH INDICES
C  (JDISP-1)*NPREV+JREPL  AND  (JDISP-1)*NPREV+JREPL+NHALF.
C THE RESULTS ARE MODIFIED BY THE APPROPRIATE TWIDDLE FACTOR
C AND STORED IN WORK WITH INDICES
C  (JDISP-1)*NNEXT+JREPL  AND  (JDISP-1)*NNEXT+JREPL+NPREV
C WHERE
C      NDISP      PRODUCT OF REMAINING FACTORS.
C      NPREV      PRODUCT OF PREVIOUS FACTORS.
C      NNEXT      PRODUCT OF PREVIOUS AND CURRENT FACTORS.
C      NHALF      PRODUCT OF PREVIOUS AND REMAINING FACTORS.
C      JDISP      DISPLACEMENT INDEX = 1,2...,NDISP.
C      JREPL      REPLICATION INDEX = 1,2...,NPREV.
C
C THE TWIDDLE FACTOR CEXP(-PI*EYE*J/NDISP), J=0,1,99NDISP-1
C IS OBTAINED BY TAKING THE CONJUGATE OF ELEMENTS SPACED
C EVERY NSPACE=NTAB/NDISP OF EXPTAB.
      INTEGER NTCOF,NDISP,NTAB
      COMPLEX TCOF (1), WORK (1), EXPTAB (1)
      COMPLEX CONJG
      COMPLEX ROT,Z0,Z1
      INTEGER J,JDISP,JREPL,JTAB,JT,JW
      INTEGER KT0,KT1,KW0,KW1,NHALF,NNEXT,NPREV,NSPACE
      NHALF = NTCOF/2
      NPREV = NTCOF/(2*NDISP)
      NNEXT = NTCOF/NDISP
      NSPACE = NTAB/NDISP
      DO 200 JDISP = 1,NDISP
        J = JDISP-1
        JTAB = J*NSPACE
        ROT = CONJG(EXPTAB(JTAB+1))
        JT = J*NPREV
        JW = J*NNEXT
        DO 100 JREPL = 1,NPREV
          KT0 = JT+JREPL
          KT1 = KT0+NHALF
          KW0 = JW+JREPL
          KW1 = KW0+NPREV
          Z0 = TCOF(KT0)
          Z1 = TCOF(KT1)
          WORK(KW0) = Z0+Z1
          WORK(KW1) = (Z0-Z1)*ROT
  100   CONTINUE
  200 CONTINUE
      RETURN
C END OF CFCOF
      END
```

Algorithm 414

Chebyshev Approximation of Continuous Functions by a Chebyshev System of Functions [E2]

G.H. Golub and L.B. Smith* (Recd. Oct. 11, 1967, Jan. 27, 1969, and Apr. 11, 1970) Dept. of Computer Science, Stanford University, Stanford CA 94305

Key Words and Phrases: approximation, Chebyshev approximation, Remez algorithm
CR Categories: 5.13

The second algorithm of Remez can be used to compute the minimax approximation to a function, $f(x)$, by a linear combination of functions, $\{Q_i(x)\}_0^n$, which form a Chebyshev system. The only restriction on the function to be approximated is that it be continuous on a finite interval $[a,b]$. An Algol 60 procedure is given, which will accomplish the approximation. This implementation of the second algorithm of Remez is quite general in that the continuity of $f(x)$ is all that is required whereas previous implementations have required differentiability, that the end points of the interval be "critical points," and that the number of "critical points" be exactly $n + 2$. Discussion of the method used and of its numerical properties is given as well as some computational examples of the use of the algorithm. The use of orthogonal polynomials (which change at each iteration) as the Chebyshev system is also discussed.

Description

1. *Introduction.* Given a Chebyshev system, $\varphi_0(x)$, $\varphi_1(x)$, ..., $\varphi_n(x)$, we define the Chebyshev or minimax approximation to a continuous function $f(x)$ over an interval $[a, b]$ to be the function

$$P_n(x) = c_0\varphi_0(x) + \cdots + c_n\varphi_n(x), \tag{1.1}$$

such that ϵ is minimized, where

$$\epsilon = \mathop{max}_{a\leq x\leq b} |f(x) - P_n(x)|. \tag{1.2}$$

If $\varphi_i(x) = x^i$, we have the minimax polynomial approximation of degree n to $f(x)$. If $\varphi_i(x) = T_i(x)$, where $T_i(x)$ denotes the Chebyshev polynomial of the first kind of order i, we have the minimax approximation as a sum of Chebyshev polynomials. For the definition of a Chebyshev system, see Achieser [3, p. 73].

The algorithm presented here computes the coefficients c_i, $i = 0, 1, \ldots, n$, in (1.1) for any given Chebyshev system $\varphi_i(x)$, $i = 0, 1, \ldots, n$. The algorithm is based on the second algorithm of Remez [1], and also makes use of the exchange method described by Stiefel [2].

The characterization of the error curve, given by

$$\epsilon(x) = \sum_{i=0}^{n} c_i\varphi_i(x) - f(x), \tag{1.3}$$

is the basis for the second algorithm of Remez. It is shown, for example, by Rice [11, p. 56] that $p_n{}^*(x) = \sum_{i=0}^{n} c_i\varphi_i(x)$ is the Chebyshev approximation to $f(x)$ on $[a, b]$ if and only if there exists a set of points $a \leq x_0 < x_1 < x_2 < \cdots < x_{n+1} \leq b$ such that

(a) $\epsilon(x_{i+1}) = -\epsilon(x_i)$,
(b) $|\epsilon(x_i)| = \epsilon^*$, and
(c) $\mathop{max}_{a\leq x\leq b} |\epsilon(x)| = \epsilon^*$.

Thus, when the computed error curve attains this "equal ripple' character with at least $n + 1$ sign changes in $[a,b]$ we know we have the desired minimax approximation.

The second algorithm of Remez, based on the characterization, can be outlined in three steps.

(i) Choose an initial set of points, the reference set, $a \leq x_0 < x_1 < \cdots < x_{n+1} \leq b$.
(ii) Compute the discrete Chebyshev approximation to $f(x)$ on the reference set.
(iii) Adjust the points of the reference set to be the extrema of the error curve (1.3).

Steps (ii) and (iii) are repeated until convergence is obtained.

Proof of the existence of the minimax polynomial (given by (1.1) and (1.2) with $\{\varphi_i\}_0^n$, a Chebyshev system) is given by Achieser [3, p. 74].

Proof that the second algorithm of Remez converges for any starting values for the critical points is given by Novodvorskii and Pinsker [4]. If $f(x)$ is differentiable, Veidinger [12] proves that the convergence is quadratic. That is

$$\epsilon^* - \epsilon^{(k)} = O(\epsilon^* - \epsilon^{(k-1)})^2, \text{ as } k \rightarrow \infty,$$

where ϵ^* is the maximum error for the Chebyshev approximation and $\epsilon^{(k)}$ is the maximum error at the kth iteration. A survey article concerned with minimax approximations is given by Fraser [8].

2. *Applicability.* The algorithm presented herein has wide applicability in that it can be used to approximate any continuous function given on an arbitrary closed interval. In addition, the

* Now at Computing Center, University of Colorado, Boulder, CO 80302

approximating function is not restricted to polynomials or Chebyshev polynomials, but is allowed to be any linear Chebyshev system, to be supplied by the user. Three simplifying assumptions often made in an implementation of the second algorithm of Remez are:

(a) Differentiability of $f(x)$, the function to be approximated. (see [6], for example)
(b) The end points of the interval are critical points (see [8, p. 299]).
(c) The existence of exactly $n + 2$ points of extreme value on the error curve (see [8, p. 299]).

None of these three assumptions is made for this algorithm.

3a. Formal parameter list: input to the procedure

n integer degree of the Chebyshev system of functions to be used in the fit $\{\varphi_0(x), \varphi_1(x), \cdots, \varphi_n(x)\}$.

a lower end point of the interval of approximation, of type real.

b upper end point of the interval of approximation, of type real.

kstart integer controlling the number of points ($kstart \times (n+2)$) used in the initial approximation. See (i) in Section 5.

kmax integer allowing control of the number of times *k* is increased above *kstart*.

loops integer allowing control over the number of iterations taken by Remez's second algorithm if convergence is not yet attained.

f a real procedure to compute the function $f(x)$ to be approximated; procedure heading required:

```
real procedure f(x);
value x;
real x;
```

the argument is the untransformed variable *x*. $f(x)$ must be continuous in the interval $[a, b]$.

chebyshev a procedure to evaluate the Chebyshev system of functions being used at some point, *x*, in the interval $[a, b]$; procedure heading required:

```
procedure chebyshev(n, x, t);
value n, x;
integer n;
real x;
real array t;
```

n is the degree of the system, *x* is the point in $[a, b]$, and *t* is an array that will contain the values $t[i] = \varphi_i(x)$, $i = 0, 1, \ldots, n$.

eps a real procedure to compute the error curve given by (5.1); procedure heading required:

```
real procedure eps(x, c, n);
value x, n;
real x;
integer n;
real array c;
```

x is a point in $[a, b]$, *n* is the degree of the system, and *c* is an array containing the coefficients of the approximation, $c[i] = c_i$ in (5.1).

exchange a procedure, [10] for example, to locate the $n + 2$ subset of $m + 1$ given points which determine the minimax polynomial on those $m + 1$ points; procedure heading required:

```
procedure exchange (a,d,c,m,n,refset,emax,singular,r);
value m,n; integer m,n; real emax;
real array a,d,c,r;
integer array refset;
label singular;
```

a is a real $m + 1$ by $n + 1$ array, *d* is a $m + 1$ component vector, *c* is a $n + 2$ component vector, $m + 1$ is the integer number of points $(x_0, \ldots, x_m)$, *n* is the degree of the system, *refset* is a $n + 2$ component integer vector, *emax* is a real number and *singular* is a label. *r* is a vector containing the $m + 1$ values of the residual at the $m + 1$ points under consideration. On entry the components of *a* and *d* are

$a[i,j] = \varphi_j(x_i)$ and

$d[i] = f(x_i), \qquad i = 0(1)m, \qquad j = 0(1)n.$

Upon exit from *exchange*, the array *c* contains the coefficients of the minimax function found, *refset* contains the subscripts identifying the points used to compute the minimax function, i.e. the reference set, and *emax* contains the value of the maximum deviation of the minimax function from $f(x)$ on the points x_i, $i = 0(1)m$.

3b. Formal parameter list: output from the procedure

c the array of coefficients c_i of eq. (5.1).

emax the maximum modulus of the error curve (5.1) for the final approximation function, of type real.

trouble a label to which control is transferred if *remez* does not converge properly.

why an integer whose value on exit will be set to one of the following:

why = −1 if number of added points is greater than *n*. (See step (ii) in Section 5.)
why = 1 if trouble occurs in procedure *quadraticmax*.
why = 2 if trouble occurs in procedure *exchange*.
why = 3 if no convergence after iterating *loops* times.
why = 4 converged according to the maximum and minimum residual comparison.
why = 5 converged according to *why* = 4 and the critical point test.
why = 6 converged according to *why* = 4 and the coefficient test.
why = 7 converged according to *why* = 4 and both the critical point and the coefficient tests.
why = 8 converged according to critical point test only.
why = 9 converged according to coefficient test only.
why = 10 converged according to critical point and coefficient tests.

4. *Organization and notational details.* The algorithm calls for three procedures, in addition to the function $f(x)$ to be approximated, as indicated by the formal parameter list.

exchange Based on Stiefel's Exchange algorithm, which finds the $n + 2$ subset of $m + 1$ given points which determine the minimax polynomial. Use [10], for example.

eps To be supplied by user: *eps* computes the error curve

$$\epsilon(x) = \sum_{i=0}^{n} c_i \varphi_i(x) - f[x] \tag{4.1}$$

where the c_i, $i = 0, \ldots, n$, are parameters and the $\varphi_i(x)$, $i = 0, 1, \ldots, n$, are the Chebyshev system of functions being used to fit the function $f(x)$. For best results $\epsilon(x)$ should be computed in double precision and then rounded to single precision accuracy. If $f(x)$ cannot be calculated easily or efficiently in double precision at least the sum, $\sum_{i=0}^{n} c_i \varphi_i(x)$, should be accumulated in double precision and rounded to single.

chebyshev To be supplied by user: *chebyshev* evaluates the Chebyshev system $\varphi_i(x)$, $i = 0, 1, \ldots, n$ for a given argument *x*. *chebyshev* is called by *eps*.

The functions $\epsilon(x)$ and $\varphi_i(x)$ (computed by *eps* and *chebyshev*) can often be computed by simple recursive procedures. For example, if the Chebyshev system used is the set of Chebyshev polynomials, there is a well-known recurrence relation ($\varphi_{i+1}(x) = 2x\varphi_i(x) - \varphi_{i-1}$) that can be used to efficiently evaluate the required functions.

An outline of the organization of the algorithm is given in the following steps:

(i) Let $m = k \times (n+2)$, take $m + 1$ points in the interval $[a,b]$ and use *exchange* to determine the "best" polynomial (i.e. the

$$c_i \ni \max_{0 \leq j \leq n} \left| \sum_{i=0}^{n} c_i \varphi_i(x_j) - f(x_j) \right| = \text{minimum})$$

on those points. Exchange will pick $n + 2$ of the original points as critical points. The $m + 1$ points are chosen equally spaced or as the zeros of $T_{m-1}(x) - T_{m-3}(x)$ with $k \geq 1$.

(ii) Use the $n + 2$ points chosen by *exchange* in step (i) and ν other local extrema (subject to the conditions discussed under Example 2, Section 6) as input to the procedure *quadraticmax* ($\nu \geq 0$).

(iii) Procedure *quadraticmax* adjusts the $n + \nu + 2$ critical points to be the abscissas of the extrema of the error curve given by (4.1). Section 5b gives a discussion of how the adjustments are computed. After adjustment the new points are tested for alternation of sign, and if the property has been lost, we increase k and go back to step (i).

(iv) The adjusted critical points are then input to *exchange* which finds the new coefficients c_i, $i = 0, 1, \cdots, n$ for the "best" polynomial on the adjusted $n + \nu + 2$ points.

(v) Now convergence tests can be applied to the coefficients c_i, found in step (iv), to the critical points x_i, $i = 0, 1, \cdots, n$ and to the extreme values of (4.1). If not converged, go back to step (iii) since the previous critical points will not be the exact extreme points after the approximating polynomial is changed in step (iv).

5a. *Discussion of numerical properties and methods: accuracy and convergence.* The accuracy of the approximations generated by this procedure is limited by the precision of the arithmetic used and the accuracy of the subsidiary procedures *f*, *exchange*, *eps*, and *chebyshev*. The use of double precision in *eps*, for example, can improve the results of *remez* since it will then have a "smoother" error curve to work on. This use of double precision in *eps* is strongly recommended by the authors. The maximum absolute error of the approximation is output from *remez* and depends, of course, on n, the degree of approximation.

The procedure is deemed to have converged when the coefficients of the approximating function or the critical points have satisfied certain relative criteria between successive iterations. We use the notation $c_i^{(n)}$ to represent the ith coefficient at the nth iteration and similarly, $x_i^{(n)}$ represents the ith critical point at the nth iteration.

When

$$\max_i | c_i^{(n)} - c_i^{(n-1)} | \leq epsc | c_i^{(n)} | \tag{5.1}$$

or

$$\max_i | x_i^{(n)} - x_i^{(n-1)} | \leq epsx | x_i^{(n)} | \tag{5.2}$$

we consider the procedure to have converged. If $| c_i^{(n)} |$ or $| x_i^{(n)} |$ is very small the relative test is not appropriate. In that case we test $| c_i^{(n)} - c_i^{(n-1)} |$ and $| x_i^{(n)} - x_i^{(n-1)} |$ against allowed absolute errors, *absepsc* and *absepsx*. Typical values for the constants (for an 11-decimal place machine) could be

$$\begin{aligned} epsc &= 10^{-8} \\ epsx &= 10^{-4} \\ absepsc &= 10^{-8} \\ absepsx &= 10^{-4} \end{aligned} \tag{5.3}$$

A third convergence criterion is the comparison of the maximum and minimum magnitudes of the error curve at the critical points. Let

$$maxr = \max_i | \epsilon(x_i^{(n)}) |$$

and

$$minr = \min_i | \epsilon(x_i^{(n)}) |$$

where $\{x_i^{(n)}\}$ are the critical points chosen at the nth iteration, and then make the following test. If $maxr \leq rcompare \times minr$ then claim convergence. A typical value for the constant *rcompare* could be 1.0000005.

When the maximum absolute error approaches $10^{-s}(f_m)$, where s is the number of places available in the machine, and f_m is $\max_{a \leq x \leq b} |f(x)|$, we are approaching the limit of obtainable accuracy. We are working with

$$\epsilon(x) = P_n(x) - f(x) \tag{5.4}$$

so when $\epsilon(x)$ is nearly equal to $10^{-s}f(x)$, we are losing about s places in the subtraction in (5.4). This is where judicious use of double precision can be made to increase accuracy if necessary. $P_n(x)$ can be computed in double precision and a single precision difference formed, or for even further accuracy $f(x)$, if possible, could be computed in double precision and the double precision difference taken.

A comparison of the discrete approximation on a finite number of points in an interval, and the continuous approximation which this algorithm finds, is studied by Rivlin and Cheney in [9]. Rice [11, pp. 66–70] discusses the question of convergence (and rate of convergence) of the discrete approximation to the continuous approximation. This relates to the question of how large to choose k in step (i), Section 4. We have found that for well-behaved functions like e^x on $[-1,1]$ a value for k of about 3 gives good starting values. On the other hand a function like $1/(x-\lambda)$ on $[-1,1]$ with $\lambda > 1$ and λ near 1 requires k to be about 15 to obtain good starting values. The choice of k should be large enough so that the initial approximation chosen by the procedure *exchange* is close enough to the final approximation to insure that the "alternation of sign" property is never lost during the iterations. There is no known method of choosing such a k a priori. This is why the algorithm tests for "alternation of signs" at each iteration and increases k if the property is lost.

5b. *Discussion of numerical properties and methods: Locating the extrema of* $\epsilon(x)$. Most of the programming effort is involved in locating the extrema of the error function $\epsilon(x)$. The programming is similar to that done by C.L. Lawson in a *Fortran* program to compute the best minimax approximation [7]. $\epsilon(x)$ is given by

$$\epsilon(x) = \sum_{i=0}^{n} c_i \varphi_i(x) - f(x).$$

The procedure *exchange* then is used to compute the coefficients of the minimax function. That is, given $n + \nu + 2$ points, $\nu \geq 0$, *exchange* computes the coefficients of the function $\sum_{i=0}^{n} c_i \varphi_i(x)$ such that on the discrete set of points $\epsilon(x_j)$, $j = 0, 1, \cdots, n + \nu + 1$ has at least $n + 2$ extreme values (at the given points) equal in magnitude and of alternating signs. The satisfaction of this condition when the points are indeed the extrema of the continuous $\epsilon(x)$ guarantees that $\sum_{i=0}^{n} c_i \varphi_i(x)$ is the unique minimax approximating function that we seek.

5b.1 *Discussion of numerical properties and methods: Parabolic approximation to locate extremum.* Given the initial guesses x_i, $i = 0, 1, \cdots, n + \nu + 1$ (at each iteration) for the abcissas of the extrema of the error curve, we must locate these critical points more precisely. We consider two cases. First the interior points, and secondly the least and greatest of the initial guesses which may be equal to the respective end points of the interval on which the function is to be approximated.

For interior points we do the following. Take

$$\begin{aligned} u &= x_i \\ v &= x_i + \alpha(x_{i+1} - x_i) \\ w &= x_i + \alpha(x_{i-1} - x_i) \end{aligned} \tag{5.5}$$

where α is a parameter $0 < \alpha < 1$ (e.g. $\alpha = 0.1$). We then determine the parabola through the three points $\epsilon(u)$, $\epsilon(v)$, and $\epsilon(w)$. The abscissa, x^*, corresponding to the vertex of this parabola is then taken as the next guess for the ith "critical point." The point x^* is given by

$$x^* = \frac{1}{2} \frac{[(u^2 - v^2)\epsilon(w) + (v^2 - w^2)\epsilon(u) + (w^2 - u^2)\epsilon(v)]}{[(u - v)\epsilon(w) + (v - w)\epsilon(u) + (w - u)\epsilon(v)]}. \tag{5.6}$$

For computational purposes x^* is not computed directly by (5.6) since for u, v, and w very close, the denominator will be quite small. Therefore, the denominator of (5.6) is computed

$$d = [(u-v)\epsilon(w)+(v-w)\epsilon(u)+(w-u)\epsilon(v)], \tag{5.7}$$

and then by dividing out (5.6), we express x^* as

$$x^* = \begin{cases} \frac{1}{2}(u+v) & \text{if } d = 0 \\ \frac{1}{2}(u+v) + \frac{1}{2}\frac{(v-u)(u-w)\,[\epsilon(v)-\epsilon(w)]}{d} & \text{if } d \neq 0. \end{cases} \tag{5.8}$$

Once x^* is computed, it is then tested to insure acceptability since for u, v, and w very close, machine roundoff may introduce spurious results. Also, the value of α or the nature of the function $f(x)$ and therefore of $\epsilon(x)$ may introduce an unacceptable value for x^* in which case u, v, or w, whichever has highest ordinate value, is used for x^*. If x^* is acceptable it can replace u, v, or w, whichever has the lowest (in abolute value) ordinate value on the error curve $\epsilon(x)$, and a second x^* is computed. This iteration will converge to the abcissa of the extremum near x_i if roundoff is ignored and u, v, and w are sufficiently close to that point. (Compare convergence to Muller's method for solving algebraic equations [5].) However, this iteration need not be carried out excessively (2-4 iterations should be sufficient) since during each iteration of the overall process we recompute the approximating function and thereby obtain a new error curve whose extrema will not necessarily have the same abscissas.

For the end points (5.5) cannot apply since x_{i+1} and x_{i-1} do not exist at the right and left ends respectively. Therefore we take, at the left end for example,

$$\begin{aligned} u &= x_i \\ v &= x_i + \alpha(x_{i+1}-x_i) \\ w &= \begin{cases} x_i + \beta(x_{i+1}-x_i) & \text{if } x_i = a \\ x_i + \alpha(a-x_i) & \text{if } a < x_i\,, \end{cases} \end{aligned} \tag{5.9}$$

with the requirement that $\alpha \neq \beta$. The right end is handled similarly. Again the parabola through the three points $\epsilon(u)$, $\epsilon(v)$, and $\epsilon(w)$ is used to determine x^*. The tests for acceptability and iterations are performed as they were for the interior points.

5b.2 *Discussion of numerical properties and methods: Crude search to locate extremum.* In case approximation by parabola does not yield an acceptable value for the abscissa of an extremum, the following rather crude method works effectively. We simply divide the interval under consideration into l equal intervals (e.g. $l=10$) and examine the ordinate of the error curve at the end points of the intervals. The points to the left and right of the point with maximum ordinate (in absolute value) then define a new interval upon which the process is repeated. This subdivision continues until the subintervals become smaller than some specified value (e.g. 10^{-5}). The method causes the function to be evaluated more often than the parabolic approximation, but works successfully at a point where the error curve has a sharp cusp-like extremum.

The choice of $l = 10$ in this crude search procedure is arbitrary. In fact, for an initial interval of length I, a smaller value, say $l = 4$, would reduce the subinterval size to $10^{-5}\cdot I$ with a minimum of 21 function evaluations, whereas using $l = 10$ would require at least 51 function evaluations. However, small values of l increase the chances of missing the true extremum.

To decide whether to use this crude search or not we employ a relative test. Let the parabolic choice be x^* and the three points used to compute x^* be u, v, and w. Then one would expect (hope) that $|\epsilon(x^*)| \geq |\epsilon(u)|$, $|\epsilon(v)|$, and $|\epsilon(w)|$, in which case x^* has the desired properties. However, if $\epsilon_m = max_{x=u,v,w}|\epsilon(x)|$, and $|\epsilon(x^*)| < \epsilon_m$, then we must doubt the acceptability of x^* and perhaps use the crude method to determine x^*. We found a successful way to make this decision was to use the crude method if $||\epsilon(x^*)| - \epsilon_m| > C\cdot\epsilon_m$, where C is an arbitrary constant (e.g. 10^{-4}).

Fig. 1

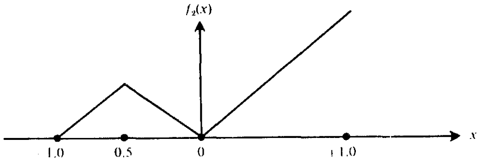


Table I. Coefficients c_i of "best" polynomial $P_4(x) = \sum_{i=0}^{4} c_i T_i(x)$ (to 6D)

i	Start	Iteration 1	Iteration 2	Iteration 3
0	1.266 063	1.266 066	1.266 066	1.266 066
1	1.130 321	1.130 318	1.130 318	1.130 318
2	0.271 495	0.271 495	0.271 495	0.271 495
3	0.044 337	0.044 336	0.044 336	0.044 336
4	0.005 523	0.005 519	0.005 519	0.005 519

Table II. Critical points, x_j, of best polynomial (to 6D)

j	Start	Iteration 1	Iteration 2	Iteration 3
0	−1.000 000	−1.000 000	−1.000 000	−1.000 000
1	−0.771 429	−0.797 573	−0.797 682	−0.797 682
2	−0.257 143	−0.278 189	−0.279 152	−0.279 152
3	0.314 286	0.339 805	0.339 061	0.339 061
4	0.828 571	0.820 978	0.820 536	0.820 536
5	1.000 000	1.000 000	1.000 000	1.000 000

Table III. Comparison of starting values x_j for $f(x) = e^x$, $n = 4$ (to 3D)

j	$T_5(x) - T_3(x) = 0$ or $\lvert T_5(x)\rvert = 1$	*exchange* on $6(N+2)$ points equally spaced	*exchange* on 201 points equally spaced	TRUE (computed)
0	−1.000	−1.000	−1.000	−1.000
1	−0.809	−0.771	−0.800	−0.798
2	−0.309	−0.257	−0.280	−0.279
3	0.309	0.314	0.340	0.339
4	0.809	0.829	0.820	0.821
5	1.000	1.000	1.000	1.000
$D_{\max}$	0.030	0.027	0.002	—

Table IV. Critical points chosen at each iteration.

Iteration	The n+2 points used (see Figure 3)									
1st	1	2	3	4	7	8	9	10	11	12
2nd	1	2	3	6	7	8	9	10	11	12
3rd	1	2	3	6	7	8	9	10	11	12

6. *Examples*. The procedure was tested on the Burroughs B5500 at the Stanford Computation Center using Burroughs Extended Algol.

We have chosen two examples to illustrate the use of the algorithm. The first is the function

$$f_1(x) = e^x \text{ on } [-1,1] \tag{6.1}$$

and the second is

$$\begin{aligned} f_2(x) &= 1 + x, & -1.0 \le x < -0.5 \\ &= -x, & -0.5 \le x < 0.0 \\ &= x, & 0.0 \le x \le 1.0. \end{aligned} \tag{6.2}$$

The first example, $f_1(x)$, is an infinitely differentiable function so that the error curve (4.1) is also differentiable, whereas $f_2(x)$ (see Figure 1) is continuous, but its derivative, $f_2'(x)$, has discontinuities at $x = -0.5$ and at $x = 0.0$, which cause the error curve to have a discontinuous derivative. We examine $f_2(x)$ as it provides an interesting example of approximating a function which is only continuous. In both cases we used Chebyshev polynomials as the Chebyshev system of functions.

Example 1. $[f_1(x) = e^x]$. Tables I and II show how the critical points and the coefficients of the approximating polynomial converge as we approximate $f_1(x) = e^x$ by a 4th-degree sum of Chebyshev polynomials. Figures differing from the final result are underlined at each step.

Table I shows that the coefficients of the "best" polynomial have converged to 6D after only one iteration; however, the critical points don't converge until the second iteration as shown by Table II. In other words, the polynomial does not change coefficients very much with a small change in the critical points. The starting points shown in Table II are chosen by *exchange* from $6 \times (n+2) = 36$ (for $n=4$) equally spaced points in the interval $[-1,1]$.

Various methods for choosing the starting values for the critical points have been proposed. These include the zeros of $T_{n+1}(x) - T_{n-1}(x)$, which are also the extrema of $T_{n+1}(x)$, and what we propose here is to let *exchange* choose $n + 2$ points from some original set of $k(n+2)$ points where $k \geq 1$. The original $k(n+2)$ points may be equally spaced, or they may be the zeros of $T_{k(n+2)+1}(x) - T_{k(n+2)-1}(x)$.

Table III compares various starting values for this example, $f_1(x) = e^x (n=4)$. D_{max} represents the maximum deviation from the "TRUE" values.

Example 2. $[f_2(x)]$. Approximation of $f_2(x)$ by an 8th degree sum of Chebyshev polynomials $(n=8)$ poses the problem of having an error curve with more than $N + 2$ local extrema. This problem also arises when approximating an even or odd function (see [6]). We resolve the problem by including all the local extrema of the error function, $\epsilon(x)$, which have the alternation of sign property, in the search for $n + 2$ critical points. That is, if the abcissas of the extrema are ordered algebraically, the signs of the corresponding ordinates must alternate. We obtain starting guesses for local extrema by having *exchange* pick $n + 2$ starting points from some original set of points, together with the corresponding first approximating polynomial, and then examining the resultant residuals. If the table of residuals indicates an extremum not already chosen by *exchange*, which has the correct alternating sign, then the corresponding abcissa is included as a critical point for later iterations. k must be chosen greater than 1 in order for this method to work.

Figure 2 shows the error curve, $\epsilon(x)$, for the first and third iterations of approximating $f_2(x)$ by an 8th-degree linear combination of Chebyshev polynomials.

Table IV indicates how the choice of critical points can change from one iteration to the next. If we had not included the additional extrema at points 5 and 6 at the first iteration, we would have arrived at the approximation whose error curve is illustrated by Figure 3. That is $n + 2$ extrema of the error curve have equal magnitude and alternating signs, but another extremum exists with larger modulus.

Fig. 2

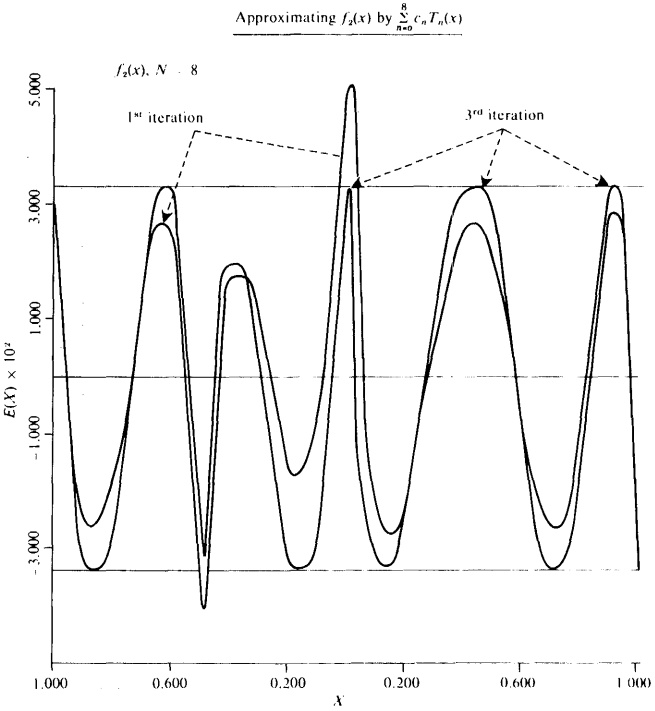


Fig. 3

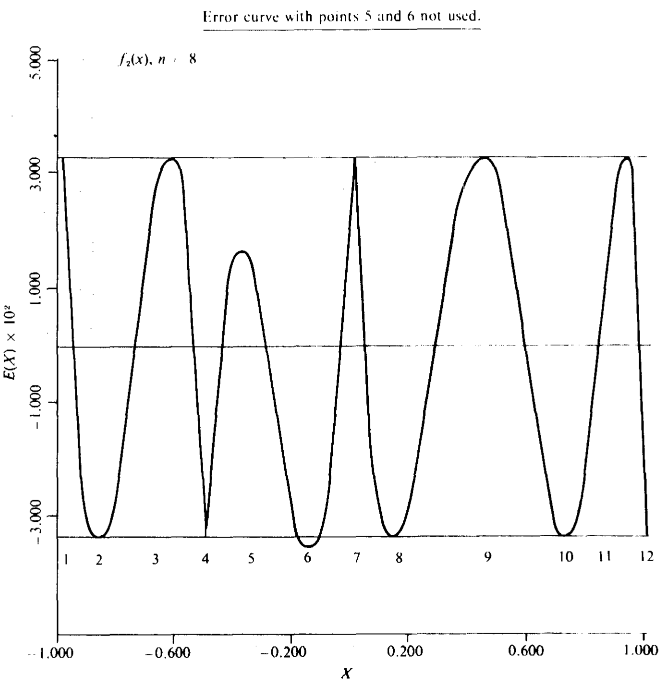


Table V. Comparison of starting values x_j for $f(x) = f_2(x)$, $n = 8$ (to 4D)

j	$T_9(x) - T_7(x) = 0$	*exchange* on 33 points equally spaced	*exchange* on 201 points equally spaced	TRUE (computed)
0	−1.0000	−1.0000	−1.00	−1.0000
1	−0.9397	−0.8750	−0.86	−0.8565
2	−0.7660	−0.6250	−0.62	−0.6248
3	−0.5000	−0.1250	−0.14	−0.1424
4	−0.1736	0.0	0.0	0.0
5	0.1736	0.1250	0.15	0.1456
6	0.5000	0.4375	0.44	0.4413
7	0.7660	0.7500	0.73	0.7290
8	0.9397	0.9375	0.93	0.9289
9	1.0000	1.0000	1.000	1.0000
$D_{\max}$	0.3750	0.0210	0.0048	—

As an interesting comparison to Table III we give a similar table for $f(x) = f_2(x)$. D_{max} represents the maximum deviation from the "TRUE" values in Table V.

7. *Use of orthogonal polynomials.* Consider the polynomials $p_0(x), p_1(x), \cdots, p_n(x)$ orthogonal on the set of points $x_0 < x_1 < \cdots < x_m$. Such polynomials are described by Forsythe [13], and they form a Chebyshev system. This is easily seen since any licear combination,

$$P(x) = \sum_{i=0}^{n} c_i p_i(x), \tag{7.1}$$

is a polynomial of degree n which has exactly n zeros. Hence on any interval, $P(x)$ has no more than n zeros. This satisfies the definition of a Chebyshev system.

It is known, see Forsythe [13], that orthogonal polynomials have advantages over standard polynomials in least squares data-fitting. In the Remez algorithm, if a new set of polynomials, orthogonal on the critical points, is computed each time the critical points are adjusted, convergence is assured. This can be proved by nothing that at each iteration the best orthogonal polynomial fit is equivalent to the best fit that would be obtained if the Chebyshev system were held constant as standard polynomials. Perhaps this use of orthogonal polynomials will have computational advantages over, say, standard polynomials on the interval [0,1].

The use of orthogonal polynomials for the Chebyshev system has been implemented and tried successfully on a Burroughs B5500 but as yet we have no illustrations of any dramatic advantages over any other Chebyshev system.

References

1. Remez, E.Y. General computational methods of Chebyshev approximation. In *The Problems with Linear Real Parameters*, AEC-tr-4491, Books 1 and 2, English translation by US AEC.
2. Stiefel, E.L. Numerical methods of Chebyshev approximation. In *On Numerical Approximation*, R.E. Langer (Ed.) U. of Wisconsin Press, Madison, 1959.
3. Achieser, N.I. *Theory of Approximation.* (Trans. by C.J. Hyman), Frederick Ungar Publ. Co., New York, 1956.
4. Novodvorskii, E.N., and Pinsker, I.S. On a process of equalization of maxima. *Uspehi Mat. Nauk. 6* (1951), 174–181. (Trans. by A. Shenitzer, available from New York U. Library.)
5. Muller, D.E. A method for solving algebraic equations using an automatic computer. *Math Tables Aids Comp. 10* (1956), 208–215.
6. Murnaghan, E.D., and Wrench, J.W. Rep. No. 1175, David Taylor Model Basin, Md., 1960.
7. Lawson, C.L. Private communication.
8. Fraser, W. A survey of methods of computing minimax and near minimax polynomial approximations for functions of a single independent variable. *J. ACM 12* (July 1965), 295–314.
9. Rivlin, T.J., and Cheney, E.W. A comparison of uniform approximations on an interval and a finite subset thereof. *SIAM J. on Numer. Anal. 3* (June 1966).
10. Bartels, R.H., and Golub, G.H. Computational considerations regarding the calculation of Chebyshev solutions for overdetermined linear equation systems by the exchange method. Tech. Rep. No. CS67, Comput. Sci. Dep., Stanford U. (June 1967). Also Algorithm 328 *Comm. ACM 11* (June 1968), 401–406, 428–430.
11. Rice, J.R. *The Approximation of Functions*, Vol. 1, Reading Mass. Addison-Wesley, 1964.
12. Veidinger, L. On the numerical determination of the best approximations in the Chebyshev sense. *Numer. Math. 2* (1960), 95–105.
13. Forsythe, G.E. Generation and use of orthogonal polynomials for data-fitting with a digital computer. *J. SIAM 5* (June 1957), 74–88.

Algorithm

```
procedure remez (n, a, b, kstart, kmax, loops, f, chebyshev, eps,
    exchange, c, emax, trouble, why);
  value n, a, b, kstart, kmax, loops;
  real array c;  real a, b, emax;  label trouble;
  integer n, kstart, kmax, loops, why;
  real procedure f, eps; procedure chebyshev, exchange;
comment Procedure remez finds the best fit (in the minimax sense) to
  a function f using a linear combination of functions which form a
  Chebyshev system. The exchange algorithm of E.L. Stiefel is used
  to obtain starting values for the critical points and the Remez
  algorithm is then used to find the best fit;
begin
  procedure quadraticmax(n, x, niter, alfa, beta, ok, a, b, c, nadded,
    eps);
    value n, niter, alfa, beta, nadded;  array x, c;
    integer n, niter, nadded;  real alfa, beta, a, b;
    Boolean ok;  real procedure eps;
  comment Procédure quadraticmax is called to adjust the values of
    the critical points in each iteration of the Remez algorithm. The
    points are adjusted by fitting a parabola to the error curve in a
    neighborhood, or if that proves unsatisfactory a brute force de-
    termination of the extrema is used;
  begin
    integer i, count1, count2, nhalf, signepsxstar, signu, signv, signw,
      jmax, ncrude, j, nn;
    real u, v, w, denom, epsu, epsv, epsw, xstar, epsxstar, xxx, misse,
      missx, dx, emax, etmp;
    integer array signepsx [0 : n + 1];  array epsx [0 : n + 1];
    nn := n − nadded;
    comment On arbitrary parameters...
      ncrude The number of divisions used in the brute force
        search for extrema.
      nhalf The parameter (alpha) which determines the size of
        interval to be examined for an extremum is reduced by
        half if a bad value for xstar is computed, however this
        reduction may occur only nhalf times.
      misse If the value of the error curve at a new critical point
        differs from the previous value by a relative difference of
        more than misse then the brute force method is brought in.
      missx The brute force method keeps searching until it is
        within missx of an extremum;
    comment Set values of the constants;
    ncrude := 10;  nhalf := 4;  misse := $1.0_{10} -2$;  missx :=
      $1.0_{10} -5$;
    comment Compare missx with absepsx. They should be equal;
    for i := 0 step 1 until n + 1 do
    begin
      epsx[i] := eps(x[i], c, nn);
      signepsx[i] := sign(epsx[i]);
    end;
    for i := step 1 until n + 1 do
    begin
      comment If the starting values for the critical points do not
        alternate the sign of eps(x), then we go to the label trouble;
      if signepsx[i] × signepsx[i−1] ≠ −1 then go to trouble;
    end;
    comment First find all the interior extrema. Then we will find
      the end extrema, which may occur at the ends of the interval;
    for i := 1 step 1 until n do
    begin
      count1 := 0;  count2 := 0;
L1:
      u := x[i];
      v := u + alfa × (x[i+1] − u);  w := u + alfa ×
        (x[i−1] − u);
      epsu := epsx[i];  signu := signepsx[i];
      epsv := eps(v, c, nn);  signv := sign(epsv);
      epsw := eps(w, c, nn);  signw := sign(epsw);
      if ¬ signu = signv ∨ ¬ signv = signw then go to L3;
      comment If the sign of eps(x) at the three points is not the
        same, we go to L3 where alfa is reduced to make the points
        closer together;
      epsu := abs(epsu);  epsv := abs(epsv);  epsw := abs(epsw);
L2:
      denom := 2.0 × ((epsv − epsu) × (w − u) + (epsw −
        epsu) × (u − v));
      if denom = 0.0 then xstar := 0.5 × (v + w) else xstar :=
        0.5 × (v + w) + (v − u) × (u − w) × (epsv − epsw)/
        denom;
      count1 := count1 + 1;
      comment Test xstar to be sure it is what we want. Is it be-
        tween x[i−1] and x[i+1]? Is eps(xstar) ≥ eps(u, v, w)? If
        xstar is too bad, go to L3 and reduce alfa unless alfa has
        been reduced nhalf times. Otherwise if ok, go to savexstar;
      if xstar = u ∨ xstar = v ∨ xstar = w then
      begin
        epsxstar := eps(xstar, c, nn); signepsxstar := sign
          (epsxstar);
        epsxstar := abs(epsxstar);  go to savexstar
      end;
      if xstar ≤ x[i−1] ∨ xstar ≥ x[i+1] then go to L3;
      epsxstar := eps(xstar, c, nn);
      signepsxstar := sign(epsxstar);
      epsxstar := abs(epsxstar);
      if signepsxstar ≠ signu ∨ epsxstar < epsu ∨ epsxstar <
        epsv ∨ epsxstar < epsw then
      begin
        if epsu ≥ epsv ∧ epsu ≥ epsw then
        begin
          if abs(epsxstar − epsu) > misse × epsu then go to
            LBL2;
          xstar := u;  epsxstar := epsu;  signepsxstar := signu·
          go to savexstar;
        end;
        if epsv ≥ epsu ∧ epsv ≥ epsw then
        begin
          if abs(epsxstar − epsv) > misse × epsv then go to
            LBL2;
          xstar := v;  epsxstar := epsv;  signepsxstar := signv;
          go to savexstar·
        end;
        if abs(epsxstar − epsw) > misse × epsw then go to
          LBL2;
        xstar := w;  epsxstar := epsw;  signepsxstar := signw;
        go to savexstar;
LBL2:
        jmax := 0;
LBL1:
        dx := (v−w)/ncrude;  emax := 0.0;  xxx := w − dx;
        for j := 0 step 1 until ncrude do
        begin
          xxx := xxx + dx;  jmax := jmax + 1;
          etmp := eps(xxx, c, nn);
          if abs(etmp) > emax then
          begin
            emax := epsxstar := abs(etmp);
            signepsxstar := sign(etmp);
            u := xstar := xxx;
            v := u + dx;  w = u − dx;
          end
        end;
        if dx > missx then go to LBL1;
        comment Make sure v and w are within bounds;
        if v ≥ x[i+1] then go to L3;
        if w ≤ x[i−1] then go to L3;
        go to savexstar
      end;
      if count1 > niter then go to savexstar;
      if epsu ≤ epsw then
      begin
        if epsv < epsu then
L4:
        begin
          comment v is minimum;
          if xstar > u then
          begin
            v := xstar;  epsv := epsxstar;  go to L2;
          end;
          if xstar > w then
          begin
            epsv := epsu;  v := u;
            epsu := epsxstar;  u := xstar;
            go to L2;
          end
          else
          begin
            v := u;  epsv := epsu;
            u := w;  epsu := epsw;
            w := xstar;  epsw := epsxstar;
            go to L2;
          end;
        end
        else
        begin
          comment u is minimum;
          if xstar ≥ v then
          begin
            u := v;  epsu := epsv;
            v := xstar; epsv := epsxstar;
            go to L2;
          end;
          if xstar ≥ w then
          begin
            u := xstar;  epsu := epsxstar;
            go to L2;
          end
          else
          begin
            u := w;  epsu := epsw;
            w := xstar; epsw := epsxstar;
            go to L2;
          end;
        end;
      end
      else
      begin
        if epsv < epsw then
        begin
          comment v is minimum;  go to L4;
        end
        else
        begin
          comment w is minimum;  if xstar ≥ v then
          begin
            w := u;  epsw := epsu;
            u := v;  epsu := epsv;
            v := xstar;  epsv := epsxstar;
            go to L2;
          end;
          if xstar ≥ u then
          begin
            w := u;  epsw := epsu;
            u := xstar; epsu := epsxstar;
          go to L2;
          end
          else
          begin
            w := xstar;  epsw := epsxstar;
            go to L2;
          end;
        end;
      end;
L3:
      count2 := count2 + 1;
      if count2 > nhalf then go to trouble;
      alfa := 0.5 × alfa;
      comment The factor 0.5 used in reducing alpha is arbitrarily
        chosen;
      go to L1;
      comment Replace x[i] by xstar after checking alternation of
        signs;
    savexstar:
      if i > 1 ∧ signepsxstar × signepsx[i−1] ≠ −1 then go to
        trouble;
      signepsx[i] := signepsxstar;
      x[i] := xstar;
    end;
    comment This is the end of the loop on i which finds all interior
      extrema. Now we proceed to locate the extrema at or near
      the two endpoints (left end, then right end);
    comment We assume beta > alfa;
    for i := 0, n + 1 do
    begin
      count1 := 0;  count2 := 0;
L8:
      u := x[i];  if i = 0 then
      begin
        if a < u then w := u + alfa × (a − u) else w := u +
        beta × (x[1] − u);
        v := u + alfa × (x[1] − u);
      end
      else
      begin
        if b > u then w := u + alfa × (b − u) else w := u +
        beta × (x[n] − u);
        v := u + alfa × (x[n] − u);
      end;
      epsu := epsx[i];  signu := signepsx[i];
      epsv := eps(v, c, nn);  signv := sign(epsv);
      epsw := eps(w, c, nn);  signw := sign(epsw);
      if signv ≠ signu ∨ signv ≠ signw then go to L7;
      epsu := abs(epsu);  epsv := abs(epsv);  epsw := abs(epsw);
L5:
      denom := 2.0 × (epsu × (v−w) + epsv × (w−u) + epsw ×
        (u−v));
      if denom = 0.0 then xstar := 0.5 × (w+v) else xstar :=
        0.5 × (v+w) + (v−u) × (u−w) × (epsv − epsw)/
        denom;
      if i = 0 ∧ (xstar < a ∨ xstar ≥ x[1]) then
      begin
        xstar := a;  epsxstar := eps(a, c, nn);
        signepsxstar := sign(epsxstar); epsxstar := abs (epsxstar);
      end
      else
      if i = n + 1 ∧ (xstar > b ∧ xstar ≤ x[n]) then
      begin
        xstar := b;  epsxstar := eps(b, c, nn);
        signepsxstar := sign(epsxstar); epsxstar := abs (epsxstar);
      end
      else
      begin
        epsxstar := eps(xstar, c, nn);
        signepsxstar := sign(epsxstar);
        epsxstar := abs(epsxstar);
      end;
      count1 := count1 + 1;
      if i = 0 ∧ xstar ≥ x[1] then go to L7;
      if i = n + 1 ∧ xstar ≤ x[n] then go to L7;
      if xstar = u ∨ xstar = v ∨ xstar = w then go to L6;
      if signepsxstar ≠ signu ∨ epsxstar < epsu ∨ epsxstar <
        epsv ∨ epsxstar < epsw then
      begin
        if epsu ≥ epsv ∧ epsu ≥ epsw then
        begin
          xstar := u;  epsxstar := epsu;
          signepsxstar := signu;  go to L6;
        end;
        if epsv ≥ epsu ∧ epsv ≥ epsw then
        begin
          xstar := v;  epsxstar := epsv;
          signepsxstar := signv;  go to L6;
        end;
        xstar := w;  epsxstar := epsw;
        signepsxstar := signw; go to L6;
      end;
      if count1 > niter then go to L6;
      if epsu < epsw then
      begin
        if epsv < epsu then
        begin
          comment v is minimum;
          v := xstar;  epsv := epsxstar;
          go to L5;
        end
        else
        begin
          comment u is minimum;
          u := xstar;  epsu := epsxstar;
          go to L5;
        end;
      end
      else
      begin
        if epsv < epsw then
        begin
          comment v is minimum;
          v := xstar;  epsv := epsxstar;
          go to L5;
        end
        else
        begin
          comment w is minimum; w := xstar; epsw := epsxstar;
          go to L5;
        end
      end;
L7:
      count2 := count2 + 1;
      if count2 > nhalf then go to trouble;
      alfa := 0.5 × alfa;  beta := 0.5 × beta;
      go to L8;
      comment Replace x[i] by xstar after checking its sign;
```

```
L6:
        if i = 0 ∧ signepsxstar × signepsx[1] ≠ − 1 then go to
            trouble;
        if i ≠ 0 ∧ signepsxstar × signepsx[n] ≠ − 1 then go to
            trouble;
        signepsx[i] := signepsxstar;  x[i] := xstar;
      end;
      go to done;
trouble:
      ok := false;  go to L9;
done:
      ok := true;
L9:
   end  quadraticmax;
   comment Procedure start computes the arrays which are then in-
      put to exchange to find the best approximation on the points at
      hand;
   procedure start (m, n, a, d, xi, chebyshev, f);
      value m, n;  integer m, n;
      array a, d, xi;
      procedure chebyshev;  real procedure f;
   begin
      integer i, j;  real array t[0:n];
      for i := 0 step 1 until m do
      begin
        chebyshev (n, xi[i], t);
        for j := 0 step 1 until n do a[i,j] := t[j];
        d[i] := f(xi[i]);
      end
   end start;
   comment Now the procedure remez;
   real epsc, alfa, beta, epsx, absepsc, absepsx, rcompare, dx, maxr,
      minr, tempr, minsep;
   integer m, i, itemp, j, niter, nloop, k, nadded, isub, maxri, ilast,
      signnow, jj;
   integer signnew;  integer array refset[0 : n + 1 + n];
   comment Assume number of points added ≤ n;
   integer array ptsadd[0 : n];
   array clast[0 : n + 1], xq, xqlast[0 : n + 1 + n];
   Boolean firsttime, ok, convx, convc, addit;
   why := 0; k := kstart;
   comment Come here if k gets changed;
newk:
   m := n + 1 +(k − 1) × (n + 2);
   begin
      array r, xi, d[0 : m], aa[0 : m, 0 : n + 1];
      firsttime := true;  convx := false; convc := false;
      nloop := 0;
      comment This makes the initial points spaced according to the
         extrema of the Chebyshev polynomial of degree m − 1;
      for i := 0 step 1 until m do
      xi[i] := (a+b)/2.0 − (b−a) × cos((3.14159265359 × i)/m)/
         2.0;
      comment 3.14159... is π;
      dx := (b−a)/m;
      comment To use equally spaced points a statement such as the
         following could be used. for i := 0 step 1 until m do xi[i] :=
         a + i × dx;
      start(m, n, aa, d, xi, chebyshev, f),
   comment The following constants are used in testing for conver-
         gence
         epsc used in relative test on coefficients
         absepsc used in absolute test on coefficients
         epsx used in relative test on critical points
         absepsx used in absolute test on critical points
         rcompare used to compare relative magnitudes of max and
         min values of residual on the critical points;
      epsc := 1.0₁₀ − 7; absepsc := 1.0₁₀ − 7;  epsx := 1.0₁₀ − 5;
         absepsx := 1.0₁₀ − 5;
      rcompare := 1.0000005;
```

```
      comment epsx and absepsx should be the same as missx in pro-
         cedure quadraticmax. epsc and absepsc should be adjusted
         according to knowledge of the expected magnitudes of the
         coefficients (if known). It is best to depend on the critical
         points and/of the max and min of the residuals for conver-
         gence criteria;
      comment Now call on exchange to find the first approximation
         to the best approximating function;
      exchange (aa, d, c, m, n, refset, emax, singular, r);
      comment The subscripts of the points chosen are in array ref-
         set[0:n+1], the coefficients of the best approximating func-
         tion on the m points are in c[0:n], the residuals in r;
      comment The reference set, the coefficients at this step, and/or
         the residuals may be written at this point;
      for i := 0 step 1 until n do clast[i] := c[i];
      comment Now we are going to look for any extrema not given
         by the points chosen by exchange;
      comment Make sure critical points are algebraically ordered;
      for i := 0 step 1 until n do for j := i + 1 step 1 until n + 1 do
      begin
        if refset[j] < refset[i] then
        begin
          itemp := refset[j];  refset[j] := refset[i];
          refset[i] := itemp;
        end
      end;
      nadded := 0;  maxr := 0;  maxri := 0;  ilast := 0;
      signnow := sign(r[0]);
      for i := 0 step 1 until m + 1 do
      begin
        if i = m + 1 then go to LBL;
        if sign(r[i]) ≠ 0 ∧ sign(r[i]) = signnow then
        begin
          if abs(r[i]) > maxr then
          begin maxri := i;  maxr := abs(r[i]);  end
        end
        else
LBL:
        begin
          if i < m + 1 then signnow := sign(r[i]);
          addit := true;
          for j := 0 step 1 until n + 1 do
          begin
            for jj := ilast step 1 until i − 1 do
            begin
              if jj = refset[j] then addit := false;
            end
          end;
          if addit then
          begin
            nadded := nadded + 1;  if nadded > n then
            begin
              comment We assume nadded is always ≤ n. If nadded
                 is > n, why is set to −1 and we go to the label
                 trouble. This can be modified by changing this test
                 and changing the declarations for ptsadd, refset, xq,
                 and xqlast above;
              why := −1;
              go to trouble
            end;
            ptsadd[nadded] := maxri;
            refset [n + 1 + nadded] := maxri;
          end;
          if i < m + 1 then
          begin
            ilast := i;  maxr := abs(r[i]);  maxri := i;
          end
        end
      end;
      comment We now have n + 2 + nadded points to send to
         quadraticmax for adjustment;
```

```
    m := n + nadded;
    comment Make sure critical points are algebraically ordered;
    for i := 0 step 1 until m do for j := i + 1 step 1 until m + 1
      do
    begin
      if refset[j] < refset[i] then
      begin
        itemp := refset[j];  refset[j] := refset[i];
        refset[i] := itemp;
      end
    end;
    for i := 0 step 1 until m + 1 do xq[i] := xi[refset [i]];
    niter := 2;
    comment This is the number of times to iterate in quadraticmax;
    alfa := 0.15;  beta := 0.2;
    comment alfa and beta are used to determine the points used
      in quadraticmax to fit a parabola. They are arbitrary subject
      to: 0 < alfa < beta < 1. Also beta should be fairly small
      to keep the points on one side of zero;
    comment This is the beginning of the loop that calls on
      quadraticmax, exchange, etc.;
loop:
    nloop := nloop + 1;
    quadraticmax(m, xq, niter, alfa, beta, ok, a, b, c, nadded, eps);
    if ¬ ok then
    begin
      k := k + 1;  if k > kmax then
      begin why := 1;  go to trouble;  end;
      go to newk;
    end;
    if ¬ firsttime then
    begin
      comment Compare the largest and smallest of the residuals
        at the critical points (after adjustment);
      comment Set minr to a large number;
      maxr := 0.0;  minr := 1.0₁₀50;
      for i := 0 step 1 until n + 1 do
      begin
        addit := true;
        for j := 1 step 1 until nadded do if refset[i] = ptsadd[j]
          then addit := false;
        if addit then
        begin
          tempr := abs(eps (xq [refset [i]], c, n));
          if tempr > maxr then maxr := tempr else if tempr <
            minr then minr := tempr;
        end
      end;
      if maxr ≤ rcompare × minr then why := 4;
    end;
    comment Compare xq to xqlast;
    if ¬ firsttime then
    begin
      convx := true;
      for i := 0 step 1 until m + 1 do
      begin
        if abs(xq [i] − xqlast[i]) > absepsx then
        begin
          if abs (xq [i] − xqlast[i]) ≥ epsx × abs(xq [i]) ∧
            xq[i] ≠ 0.0 then convx := false;
          if xq[i] = 0.0 ∧ abs(xq [i] − xqlast[i]) > absepsx
            then convx := false;
        end;
        xqlast[i] := xq[i];
      end
    end
    else
```

```
    begin
      firsttime := false;
      for i := 0 step 1 until m + 1 do xqlast[i] := xq[i];
      for i := 0 step 1 until n do clast[i] := c[i];
    end;
    comment Get ready to call exchange again;
    start(m + 1, n, aa, d, xq, chebyshev, f);
    exchange(aa, d, c, m + 1, n, refset, emax, singular, r);
    comment Now compare the new coefficients to the last set of
      coefficients;
    if ¬ firsttime then
    begin
      convc := true;
      for i := 0 step 1 until n do
      begin
        if abs(c[i] − clast[i]) ≥ epsc × abs(c[i]) ∧ c[i] ≠ 0.0
          then convc := false;
        if  c[i] = 0.0  ∧  abs(c[i] -- clast[i]) > absepsc  then
          convc := false;  clast[i] := c[i];
      end
    end;
    comment Set the parameter why to the proper value according
      to the following:
        why = 4 if maxr ≤ rcompare × minr.
        why = 5 if "4" and convx = true.
        why = 6 if "4" and convc = true.
        why = 7 if "4" and convx = convc = true.
        why = 8 if convx = true.
        why = 9 if convc = true.
        why = 10 if convx = convc = true. Any value of why ≥
          4 indicates convergence;
    if why = 4 ∧ convx then why := 5;
    if why = 4 ∧ convc then why := 6;
    if why = 5 ∧ convc then why := 7;
    if why = 0 ∧ convx then why := 8;
    if why = 0 ∧ convc then why := 9;
    if why = 8 ∧ convc then why := 10;
    if why ≥ 4 then go to converged;
    if nloop ≥ loops then
    begin why := 3;  go to trouble end;
    comment We go to label trouble in calling program if no con-
      vergence after a number of iterations equal to loops;
    go to loop;
singular:
    why := 2;  go to trouble;
    comment We come to singular if exchange gets into trouble;
converged:
  end;
  comment End of block using m in array declarations;
  comment There are four exits to the label trouble . . .
    (why=1) if k gets > kmax
    (why=2) if exchange gets into trouble
    (why=3) if no convergence after iterating loops number of
      times
    (why=−1) if number of added points is greater than n;
end remez
```

Algorithm 415

Algorithm for the Assignment Problem (Rectangular Matrices) [H]

F. Bourgeois, and J.C. Lassalle [Recd. 21 Sept. 1970 and 20 May 1971] CERN, Geneva, Switzerland

Key Words and Phrases: operations research, optimization theory, assignment problem, rectangular matrices
CR Categories: 5.39, 5.40

Description

This algorithm is a companion to [3] where the theoretical background is described.

References

1. Silver, R. An Algorithm for the assignment problem. *Comm. ACM 3* (Nov. 1960), 605–606.
2. Munkres, J. Algorithms for the assignment and transportation problems. *J. SIAM 5* (Mar. 1957), 32–38.
3. Bourgeois, F. and Lassalle, J. C. An extension of the Munkres algorithm for the assignment problem to rectangular matrices. *Comm. ACM 15* (Dec. 1971), 802–804.

Algorithm

```
procedure assignment (a, n, m, x, total);
  value a, n, m; integer n, m;
  real total; array a; integer array x;
comment: a[i, j] is an n × m matrix, x[1], x[2], ..., x[n] are assigned
  integer values which minimize total := sum(i := 1(1)n) of the
  elements a[i, x[i]]. If m > n the x[i] are distinct and are a subset
  of the integers 1, 2, ..., m. If m = n the x[i] are a permutation
  of the integers 1, 2, ..., n. If m < n the set of x[i] consists of
  some permutation of the integers 1, 2, ..., m interspersed with
  n − m zeros. The permutation and the positions of the zeros are
  chosen in such a way as to minimize the above sum with the
  convention that a[i, o] is to be taken equal to zero. imin =
  min(n, m) and imax = max(n, m) must be such that: imin > 0,
  imax > 1.
  This procedure is based on that of Silver [1] which uses the
  assignment algorithm of Munkres [2]. Silver's procedure has
  been extended to handle the case n ≠ m;
begin
  switch switch := NEXT, L1, NEXT 1, MARK;
  real min;
  integer array c[1:n], cb[1:m], lambda[1:m], mu[1:n], r[1:n],
    y[1:m];
  integer cbl, cl, cl0, i, j, k, l, rl, rs, sw, imin, imax, flag;
  total := 0; imin := m; imax := n;
  if n > m then go to JA;
  imin := n; imax := m;
  for i := 1 step 1 until n do
  begin
    min := a[i, 1];
    for j := 2 step 1 until m do if a[i, j] < min then min := a[i, j];
    for j := 1 step 1 until m do a[i, j] := a[i, j] − min;
    total := total + min;
  end;
  if m > n then go to JB;
JA:
  for j := 1 step 1 until m do
  begin
  min := a[1, j];
    for i := 2 step 1 until n do if a[i, j] < min then min := a[i, j];
    for i := 1 step 1 until n do a[i, j] := a[i, j] − min;
    total := total + min;
  end;
JB:
  for i := 1 step 1 until n do x[i] := 0;
  for j := 1 step 1 until m do y[j] := 0;
  for i := 1 step 1 until n do
  begin
    for j := 1 step 1 until m do
    begin
      if a[i, j] ≠ 0 ∨ x[i] ≠ 0 ∨ y[j] ≠ 0 then go to J1;
      x[i] := j; y[j] := i;
J1:
    end;
  end;
  comment Start labeling;
START:
  flag := n; rl := cl := 0; rs := 1;
  for i := 1 step 1 until n do
  begin
    mu[i] := 0;
    if x[i] ≠ 0 then go to I1;
    rl := rl + 1; r[rl] := i; mu[i] := −1;
    flag := flag − 1;
I1:
  end;
  if flag = imin then go to FINI;
  for j := 1 step 1 until m do lambda[j] := 0;
  comment Label and scan;
LABEL:
  i := r[rs]; rs := rs + 1;
  for j := 1 step 1 until m do
  begin
    if a[i, j] ≠ 0 ∨ lambda[j] ≠ 0 then go to J2;
    lambda [j] := i; cl := cl + 1; c[cl] := j;
    if y[j] = 0 then go to MARK;
    rl := rl + 1; r[rl] := y[j]; mu[y[j]] := i;
J2:
  end;
  if rs ≦ rl then go to LABEL;
  comment Renormalize;
  sw := 1; cl0 := cl; cbl := 0;
  for j := 1 step 1 until m do
  begin
    if lambda[j] ≠ 0 then go to J3;
    cbl := cbl + 1; cb[cbl] := j;
```

```
J3:
    end;
    min := a[r[1], cb[1]];
    for k := 1 step 1 until rl do
    begin
      for l := 1 step 1 until cbl do
      if a[r[k], cb[l]] < min then min := a[r[k], cb[l]];
    end;
    total := total + min × (rl+cbl−imax);
    for i := 1 step 1 until n do
    begin
      if mu[i] ≠ 0 then go to I2;
      if cl0 < 1 then go to I3;
      for l := 1 step 1 until cl0 do a[i, c[l]] := a[i, c[l]] + min;
      go to I3;
I2:
      for l := 1 step 1 until cbl do
      begin
        a[i, cb[l]] := a[i, cb[l]] − min;
        go to switch[sw];
NEXT:
        if a[i, cb[l]] ≠ 0 ∨ lambda[cb[l]] ≠ 0 then go to L1;
        lambda[cb[l]] := i;
        if y[cb[l]] = 0 then
        begin
          j := cb[l];   sw := 2;   go to L1;
        end;
        cl := cl + 1;   c[cl] := cb[l];   rl := rl + 1;
        r[rl] := y[cb[l]];
L1:
      end;
I3:
    end;
    go to switch[sw + 2];
NEXT1:
    if cl0 = cl then go to LABEL;
    for i := cl0 + 1 step 1 until cl do mu[y[c[i]]] := c[i];
    go to LABEL;
    comment Mark new column and permute;
MARK:
    y[j] := i := lambda[j];
    if x[i] = 0 then begin x[i] := j;   go to START;
  end;
  k := j;   j := x[i];   x[i] := k;   go to MARK;
FINI:
end
```

Algorithm 416

Rapid Computation of Coefficients of Interpolation Formulas [E1]

Sven-Åke Gustafson* [Recd. 21 Aug. 1969]
Computer Science Department, Stanford University,
Stanford, CA 94305

Key Words and Phrases: divided differences, Newton's interpolation formula
CR Category: 5.13

Description

This algorithm is a companion to [1] where the theoretical background is described

References

1. Gustafson, Sven-Åke. Rapid computation of interpolation formulae and mechanical quadrature rules. *Comm. ACM 14* (Dec. 1971), 797–801.

Algorithm

```
procedure INTP (dx, f, c, ord, n);
  value n;   real array dx, f, c;
  integer array ord;   integer n;
begin
comment INTP determines the coefficients of the polynomial of de-
  gree less than n which reproduces given function values and
  divided differences. The parameters of INTP are:
```

idenlifier	*type*	*comment*
n	integer	
ord	integer array	Array bounds [1:*n*]
dx, f, c	real array	Array bounds [1:*n*]

n is the number of coefficients of the interpolating polynomial. *ord* gives the character of the input data: if *ord*[*i*] = 1 then *x*[*i*] should be an argument and *f*[*i*] the corresponding function value. But if *ord*[*i*] > 1 then *f*[*i*] should contain a divided difference with a number of arguments equal to *ord*[*i*]. In this case *dx*[*i*] should contain the difference between the argument of highest index of *f*[*i*] and that of *f*[*i*−1].

Upon execution of *INTP* the coefficients of the desired polynomial are stored in *c* in such a manner that the coefficient in front of the power t^{i-1} is contained in *c*[*i*]. Other parameters are not changed. Caution: The given data must be such that it is possible to construct Newton's interpolation formula with divided differences from them. We must also have *ord*[1] = 1. Observe that if derivatives of *f* are given the corresponding divided differences with confluent arguments must be evaluated and given as input data.

Examples of use of *INTP*:

Example 1. Determine the polynomial of degree less than *n* which interpolates a function *f* at *n* distinct points x_i, $i = 1, 2, \dots, n$. Input data: $dx[i] = x_i$, $f[i] = f_i$, $ord[i] = 1$, $i = 1, 2, \dots, n$.

Example 2. Let x_1, x_2, x_3, x_4 be four given points. We know f_1, $f_{1,2}$, $f_{2,3}$, and f_4. Determine the polynomial of degree 3 which reproduces these quantities. Input data: $n = 4$,

$dx[1] = x_1$	$ord[1] = 1$	$f[1] = f_1$
$dx[2] = x_2 - x_1$	$ord[2] = 2$	$f[2] = f_{1,2}$
$dx[3] = x_3 - x_2$	$ord[3] = 2$	$f[3] = f_{2,3}$
$dx[4] = x_4$	$ord[4] = 1$	$f[4] = f_4$

Example 3. The same problem when we are given $f(-1)$, $f'(-1)$, $f''(-1)$, and $f(1)$. Input data: $n = 4$,

$dx[1] = -1$	$ord[1] = 1$	$f[1] = f(-1)$
$dx[2] = 0$	$ord[2] = 2$	$f[2] = f'(-1)$
$dx[3] = 0$	$ord[3] = 3$	$f[3] = 0.5 \cdot f''(-1)$
$dx[4] = 1$	$ord[4] = 1$	$f[4] = f(1)$

For further details see [1];

```
integer i, j, k; real ai, h, d, xx;
  real array arg [1 : n];
  comment Initiate phase DI;
  for i := 1 step 1 until n do
    arg[i] := if ord[i] = 1 then dx[i] else dx[i] + arg[i-1];
  comment Phase DI;
  for i := 2 step 1 until n do
  begin
    j := ord[i];
    if j = 1 then go to divde;
    d := f[i];
    for k := i step -1 until i - j + 2 do f[k] := f[k-1];
    f[i-j+1] := d;
    h := dx[i];   ai := arg[i];
    for k := i - j + 2 step 1 until i - 1 do
      f[k] := f[k] + f[k-1] × (ai-arg[k-1]);
    f[i] := f[i] + f[i-1] × h;
    arg[i] := ai;
divde:
    for k := i - j step -1 until 1 do
      f[k] := (f[k+1]-f[k])/(arg[i]-arg[k]);
  end i-loop;
  comment phase DII;
  c[1] := f[1];   if n = 1 then go to ready;
  for i := 2 step 1 until n do
  begin
    xx := arg[i];   c[i] := c[i-1];
    for k := i - 1 step -1 until 2 do
    c[k] := -xx × c[k] + c[k-1];
    c[1] := f[i] - xx × c[1]
  end second i-loop;
ready:
end INTP
```

* Present Address: Inst. F. Informations Behandling (Numeisk analys), KTH, 10044 Stockholm, Sweden.

Algorithm 417

Rapid Computation of Weights of Interpolatory Quadrature Rules [D1]

Sven-Åke Gustafson* [Recd. 21 Aug. 1969]
Computer Science Department, Stanford University, Stanford, CA 94305

Key Words and Phrases: divided differences
CR Categories: 5.16

Description

This algorithm is a companion to [1] where the theoretical background is described

Reference

1. Gustafson, Sven-Åke. Rapid computation of interpolation formulae and mechanical quadrature rules. *Comm. ACM 14* (Dec. 1971), 797–801.

Algorithm

```
procedure INTG(y, dx, m, ord, n);
  value n;  real array y, dx, m;
  integer array ord;  integer n;
begin
comment INTG determines weights in quadrature rules of the form
```

$$\int_a^b f(t)\,d\alpha(t) = \sum_{i=1}^{n} m_i f_i^{ord(i)} \qquad (1)$$

Here $f_i^{ord(i)}$ can be a function value or derivative or divided dfference of order 1. The weights m_i are determined such as to render the rule exact when the integrand f is a polynomial of degree less than n. The parameters of *INTG* are:

identifier	*type*	*comment*
n	integer	
ord	integer array	Array bounds [1:*n*]
y, dx, m	real array	Array bounds [1:*n*]

n is the number of abscissae in formula (1). *ord* gives the character of the quantities $f_i^{ord(i)}$: if $ord[i] = 1$ then $f_i^{ord(i)}$ is the function value f_i, if $ord[i] = 2$, then $f_i^{ord(i)}$ is a divided difference with two arguments. (The procedure does not handle cases where $ord[i] > 2$.)

If $ord[i] = 1$, then $dx[i]$ should contain the argument corresponding to $f_i^{ord(i)}$, else $dx[i]$ should contain the difference between the arguments of highest index in $f_i^{ord(i)}$ and that of $f_{i-1}^{ord(i-1)}$.

y should contain the moments, that is in $y[r]$ must be stored the number

$$\int_a^b t^{r-1}\,d\alpha(t)$$

Upon execution of *INTG* the weight m_i is stored in $m[i]$. Other parameters are not changed. Example of use of *INTG*: Determine the coefficients m_1, m_2, m_3, and m_4 in the rule

$$\int_{-1}^{+1} f(x)\,dx = m_1 f(-1) + m_2 f'(-1) + m_3 f(1) + m_4 f'(1)$$

Input data: $n = 4$

$dx[1] = -1$	$ord[1] = 1$	$y[1] = 2$
$dx[2] = 0$	$ord[2] = 2$	$y[2] = 0$
$dx[3] = 1$	$ord[3] = 1$	$y[3] = 2/3$
$dx[4] = 0$	$ord[4] = 2$	$y[4] = 0$

Restriction: We can only have $ord[i] = 1$ or $ord[i] = 2$. Furthermore the given data must be such that it is possible to construct Newton's interpolation formula with divided differences from the set $x_i f_i^{ord(i)}$ $i = 1, 2, \ldots, n$. We must also have $ord[1] = 1$. For further details, see [1];

```
integer i, j, k;  real t;
real array x[1:n];
comment Initiate phase PI;
for i := 1 step 1 until n do
begin
  m[i] := y[i];
  x[i] := if ord[i] = 1 then dx[i] else dx[i] + x[i−1]
end;
comment Phase PI;
for j := 2 step 1 until n do
begin
  t := x[j−1];
  for i := n step −1 until j do m[i] := m[i] − t × m[i−1]
end;
comment Phase PII;
for k := 1 step 1 until n − 1 do
begin
  comment transform from descending diagonal k to descending
  diagonal k + 1;
  if k = n − 1 ∧ ord[n] = 2 then go to ready;
  t := x[k];  m[n] := m[n]/(x[n]−t);
  for i := n − 1 step −1 until k + 2 do m[i] := (m[i] − m[i+1])/
  (x[i]−t);
  if ord[k+1] = 2 then
  begin
    m[k+1] := m[k+1] − m[k+2];  go to on;
  end;
  if k + 1 < n then
  m[k+1] := (m[k+1] − m[k+2])/(if ord[k+1] = 2 then
  dx[k+1] else x[k+1]−t);
  if ord[k] = 1 ∧ ord[k+1] = 1 then
  begin
    m[k] := m[k] − m[k+1];  go to on
  end;
  for i := k − 1 step −1 until 1 do
  if ord[i] = 1 then
  begin
    j := i;  go to next
  end;
next:
  t := m[k+1];
  for i := k step −1 until j + 1 do m[i] := m[i] − t × dx[i];
  m[j] := m[j] − t;
on:
  end;
ready:
end INTG;
```

* Present address: Inst. F. Informations Behandling (numerisk analys), KTH, 10044 Stockholm, Sweden.

Algorithm 418

Calculation of Fourier Integrals [D1]

Bo Einarsson [Recd. 25 Aug. 1970, 30 Oct. 1970, and 25 Jan. 1971]
Research Institute of National Defense, Box 98,
S-147 00 Tumba, Sweden

Key Words and Phrases: quadrature, Filon quadrature, integration, Filon integration, Fourier coefficients, Fourier integrals, Fourier series, spline, spline approximation, spline quadrature, extrapolation, Richardson extrapolation

CR Categories: 5.16

Description

The most commonly used formula for calculating Fourier integrals is Filon's formula, which is based on the approximation of the function by a quadratic in each double interval. In order to obtain a better approximation the cubic spline fit is used in [1]. The obtained formulas do not need the explicit calculation of the spline fit, but in addition to the function values at all intermediate points, the values of the first and second derivatives at the boundary points are required. However, these values are often obtained from symmetry conditions. If the derivatives at the end-points are unknown, they may be calculated from a cubic spline fit, for example by using some exterior points or by using two extra interior conditions for the spline fit. It can also be noted that in certain periodic cases the terms containing the derivatives will cancel, and their values will be superfluous. The use of Algorithm 353 [2] is recommended if the frequency ω/π is a positive integer and the interval is [0,1]. Test computations reported in [1] indicate that the spline formula is more accurate than Filon's formula. Both are of the fourth order. The expansion of the error term in powers of the step length contains only even powers, and therefore the use of Richardson extrapolation is very efficient.

The algorithm presented here is similar to Algorithm 353 by Chase and Fosdick [2], but in the present routine, Richardson extrapolation is included in order to obtain faster convergence.

The routine *FSPL2* evaluates the integrals

$$C = \int_2^{12} e^{-x} \cos(\omega x)\, dx \text{ and } S = \int_2^{12} e^{-x} \sin(\omega x)\, dx.$$

using the algorithm described in [1]. *FSPL2* contains a feature which selects an initial integration step size such that at least two quadrature nodes are within each full period of the trigonometric function, cf. [3]. This step size is reduced by halving until the specified accuracy is obtained or the maximum number of interval halvings of the original interval is reached. Two evaluations are always performed. If the interval $[a, b]$ is long, it is advised to take special precautions.

The use of Richardson extrapolation, which is performed in the subroutine *ENDT2*, decreases the number of function evaluations by a factor 4 in several of the test examples. It is possible to introduce the fast Fourier transform in order to obtain faster computation of the inner loop of the algorithm. Another extension is to calculate the central part of the integral with the spline algorithm and the tails with the method in [4], which gives accurate results even when the function $f(x)$ is slowly decreasing if the frequency ω is large.

Finally we give some test examples for both single and double precision computation of

$$C = \int_a^b f(x) \cos(\omega x)\, dx \text{ and } S = \int_a^b f(x) \sin(\omega x)\, dx$$

								Error in computed	
	MAX	*LC*	*LS*	*W*	*EPS*	*C*	*S*	*C*	*S*
SP Input	10	1	1	0.05	10^{-6}				
Output		5	5			0.133645	0.020190	-30.10^{-8}	-15.10^{-8}
SP Input	10	1	1	50.0	10^{-6}				
Output		9	9			0.001417	0.002306	-13.10^{-8}	-14.10^{-8}
DP Input	15	1	1	0.05	10^{-14}				
Output		7	7			0.133645	0.020190	-68.10^{-17}	-7.10^{-17}
DP Input	15	1	1	50.0	10^{-14}				
Output		11	11			0.001417	0.002306	3.10^{-17}	8.10^{-17}

References

1. Einarsson, Bo. Numerical calculation of Fourier integrals with cubic splines. *BIT 8* (1968), 279–286.
2. Chase, Stephen M., and Fosdick, Lloyd D. Algorithm 353, Filon quadrature. *Comm. ACM 12* (Aug. 1969), 457–458.
3. Einarsson, Bo. Remark on algorithm 353, Filon quadrature. *Comm. ACM 13* (Apr. 1970), 263.
4. Gustafson, Sven-Åke, and Dahlquist, Germund. On the computation of slowly convergent Fourier integrals. Presented at Nov. 1970 meeting in Oberwolfach and to appear in *Methoden und Verfahren der Mathematischen Physik.*
5. Einarsson, Bo. On the calculation of Fourier integrals. Preprints of the IFIP Congress 71, Booklet TA-1, North-Holland Pub. Co., Amsterdam, 1971, pp. 99-103. To appear in Information Processing 71, same publication.

Algorithm

```
      SUBROUTINE FSPL2
     *           (F,A,B,FPA,FPB,FBA,FBB,W,EPS,MAX,C,S,LC,LS)
C
C THIS ROUTINE COMPUTES THE FOURIER INTEGRALS
C C=INTEGRAL F(X) COS WX DX FROM X=A TO X=B
C S=INTEGRAL F(X) SIN WX DX FROM X=A TO X=B
C
C WITH THE SPLINE PROCEDURE IN B. EINARSSON, NUMERICAL
C CALCULATION OF FOURIER INTEGRALS WITH CUBIC SPLINES,
C BIT, VOL. 8, PP. 279-286, 1968.
C
C REPEATED RICHARDSON EXTRAPOLATION IS USED.
C
C THIS SUBROUTINE HAS ADAPTED SEVERAL IDEAS FROM
C ALGORITHM 353, FILON QUADRATURE  BY  CHASE AND FOSDICK,
C COMM. ACM, VOL. 12, PP. 457-458, 1969.
C
C F(X)=THE FUNCTION TO BE INTEGRATED, SUPPLIED BY THE USER
C AND DECLARED 'EXTERNAL' IN THE CALLING PROGRAM.
C
      DATA PI / 3.141592653589793 /
C
C A=LOWER QUADRATURE LIMIT  AND  B=UPPER QUADRATURE LIMIT
C IF A.GE.B THE COMPUTATION IS BYPASSED AND THE SIGNS OF
C LC, LS, AND EPS ARE CHANGED.
C
C FPA AND FPB ARE THE VALUES OF THE DERIVATIVE OF F(X).
C FBA AND FBB ARE THE CORRESPONDING VALUES OF THE SECOND
C DERIVATIVE AT THE POINTS A AND B.
C W=THE ANGULAR FREQUENCY
C
C EPS = REQUIRED ACCURACY, DEFINED BY
C          |ERROR| ≤ EPS*(1.+|C|)
C AND
C          |ERROR| ≤ EPS*(1.+|S|)
C IF CONVERGENCE IS NOT OBTAINED, THE VALUE
C OF EPS IS RETURNED WITH NEGATIVE SIGN.
C
C MAX=THE MAXIMUM NUMBER OF PARTITIONS  (INTERVAL HALVINGS)
C IN THIS ROUTINE THE INTERNAL VARIABLE MXN DEFINED BELOW
C IS USED INSTEAD OF MAX.
C LC POSITIVE ON ENTRY INDICATES THAT C IS WANTED.
C LS POSITIVE ON ENTRY INDICATES THAT S IS WANTED.
C ON EXIT LC AND LS GIVE THE NUMBER OF PARTITIONS USED
C FOR THE COMPUTATION OF C AND S.
C
C THIS ROUTINE CALLS THE SUBROUTINE ENDT2.
C
      DIMENSION PVTC(7),PVTS(7)
      IF(EPS.LT.0.) GOTO 5
      IF(A.LT.B) GOTO 10
      EPS=-EPS
    5 LC=-LC
      LS=-LS
      RETURN
   10 N=1
      W1=ABS(W)
      TEMP=2.0*(B-A)*W1/PI
      IF(TEMP.GT.2.0) N=ALOG(TEMP)/0.693
C 0.693=ALOG(2.) ROUNDED DOWNWARDS.
      MXN=MAX0(MAX,N+1)
      FA=F(A)
      FB=F(B)
      COSA=COS(W1*A)
      SINA=SIN(W1*A)
      COSB=COS(W1*B)
      SINB=SIN(W1*B)
      H=(B-A)/FLOAT(2**N)
      NSTOP=2**N-1
      NST=1
C TMAX IS THE SWITCH-OVER POINT FOR TETA.
C ANALYSIS SHOWS THAT WITH A 56 BIT FLOATING POINT MANTISSA
      TMAX=0.2
C IS SUITABLE, WHILE WITH A 24 BIT MANTISSA WE PREFER
      TMAX=1.
C TMAXB IS THE SWITCH-OVER POINT IN BETA, WHERE THE
C CANCELLATION IS STRONGEST.
      TMAXB=5.*TMAX
C LLC AND LLS ARE USED BY THE ROUTINE IN COMPUTED-GO-TO
C STATEMENTS. AS SOON AS LLS AND LLC HAVE BEEN DEFINED,
C WE CAN USE LS AND LC AS RETURN PARAMETERS (SEE ABOVE).
      IF(LS)11,11,12
   11 LLS=2
      GOTO 13
   12 LLS=1
   13 IF(LC)14,14,15
   14 LLC=2
      GOTO 17
   15 LLC=1
   17 CONTINUE
      SUMCOS=0.5*(FA*COSA+FB*COSB)
      SUMSIN=0.5*(FA*SINA+FB*SINB)
C ALL OF THE ABOVE IS EXECUTED ONLY ONCE PER CALL.
C NOW THE ITERATION BEGINS.
C THE CONSTANT 'M' IS USED IN THE RICHARDSON EXTRAPOLATION.
C M-1 IS THE NUMBER OF TIMES THE ORIGINAL STEP LENGTH 'H'
C HAS BEEN DIVIDED BY TWO.
      M=1
   20 CONTINUE
      H2=H*H
      TETA=W1*H
      DO 65 I=1,NSTOP,NST
        X=A+H*FLOAT(I)
        WX=W1*X
        GOTO (50,55),LLS
   50   SUMSIN=SUMSIN+F(X)*SIN(WX)
   55   GOTO (60,65),LLC
   60   SUMCOS=SUMCOS+F(X)*COS(WX)
   65   CONTINUE
      T2=TETA*TETA
      TEMP=1.0-SIN(0.5*TETA)**2/1.5
      IF (TETA-TMAX) 70,70,75
C 70 IS THE POWER SERIES FOR SMALL TETA, 75 IS THE CLOSED
C FORM USED WITH LARGER VALUES OF TETA.
C THE COEFFICIENTS OF THE DIFFERENT POWER SERIES BELOW ARE
C GIVEN IN EXACT FORM, COMPARE WITH THE REFERENCE ABOVE.
   70 ALFA=TETA*(1.0-T2*(2.0/15.0-T2*(19.0/1680.0-
     -T2*(13.0/25200.0-T2*(293.0/19958400.0-
     -T2*181.0/619164000.0)))))/12.0
      DELTA=-1.0/12.0+T2*(1.0/90.0-T2*(5.0/12096.0-
     -T2*(1.0/129600.0-T2/11404800.0)))
      EPSIL=1.0-T2*(1.0/6.0-T2*(0.0125-T2*(17.0/30240.0-
     -T2*(31.0/1814400.0-T2/2661120.0))))
      T3=T2
   72 BETA=TETA*H2*(1.0-T2/21.0*(1.0-T2*(1.0/48.0-
     -T2*(1.0/3960.0-T3/494208.0)))))/180.0
      GOTO 80
C CLOSED FORM OF THE COEFFICIENTS.
   75 TEMP1=(0.5*TETA)**2
      TEMP2=SIN(0.5*TETA)**2/TEMP1
      TEMP3=SIN(TETA)/TETA
      ALFA=(TEMP-TEMP2*TEMP3)/TETA
      DELTA=(TEMP-TEMP2)/T2
      EPSIL=TEMP2*TEMP2
      IF (TETA-TMAXB) 76,76,78
   76·T3=T2*(1.-T2*(1./175.-T2*(1./40800.-T2/12209400.)))
      GOTO 72
   78 BETA=(TEMP-TEMP3)/(TETA*W1*W1)
C HAVE CALCULATED THE COEFFICIENTS, NOW READY FOR THE
C INTEGRATION FORMULAS.
   80 GOTO (81,85),LLS
   81 TS=H*((BETA*FBB-ALFA*FB)*COSB+(ALFA*FA-BETA*FBA)*COSA+
     +DELTA*H*(FPB*SINB-FPA*SINA)+EPSIL*SUMSIN)/TEMP
      CALL ENDT2(PVTS,TS,EPS,S,LLS,M)
      LS=N
   85 GOTO (86,90),LLC
   86 TC=H*((ALFA*FB-BETA*FBB)*SINB+(BETA*FBA-ALFA*FA)*SINA+
     +DELTA*H*(FPB*COSB-FPA*COSA)+EPSIL*SUMCOS)/TEMP
      CALL ENDT2(PVTC,TC,EPS,C,LLC,M)
      LC=N
   90 CONTINUE
C NOW TEST TO SEE IF DONE.
      IF(LLC+LLS-3) 92,92,100
   92 N=N+1
C THIS IS THE BEGINNING OF THE ITERATION.
      IF(N-MXN) 95,95,99
   95 H=0.5*H
      NST=2
      NSTOP=2**N
      M=M+1
      GOTO 20
   99 EPS=-EPS
  100 CONTINUE
      IF(LS.GT.0.AND.W.LT.0.0) S=-S
      RETURN
      END
```

```
      SUBROUTINE ENDT2(PREVOT,QUANT,EPS,VALUE,L,M)
C
C ENDT2 IS A SUBROUTINE THAT PERFORMS RICHARDSON EXTRA-
C POLATION OF THE VALUES 'QUANT' WHICH ARE INTRODUCED INTO
C THE ROUTINE EACH TIME IT IS CALLED, EACH TIME WITH
C INCREASING VALUE OF 'M', STARTING WITH M = 1. THE CURRENT
C VALUES ARE STORED IN THE ARRAY 'PREVOT', WHERE 'PREVOT(1)'
C AT EXIT IS EQUAL TO 'QUANT'. THE BEST VALUE FOR THE MOMENT
C IS GIVEN IN 'VALUE'. ENDT2 REQUIRES THE PRESENT VALUE TO
C AGREE WITH THE PREVIOUS VALUE TO WITHIN EPS2, WHERE
C EPS2 = EPS*(1.0 + ABS(PRESENT VALUE)).
C EPS IS SUPPLIED BY THE USER.
C THE ERROR EXPANSION IS OF THE FORM
C ERROR = C4*H**4 + C6*H**6 + C8*H**8 + ... + CN*H**N + ...
C
      DIMENSION PREVOT(7),RICH(7)
      DATA  RICH(1) /    0.0/,  RICH(2) /   15.0/,
     *      RICH(3) /   63.0/,  RICH(4) /  255.0/,
     *      RICH(5) / 1023.0/,  RICH(6) / 4095.0/,
     *      RICH(7) /16383.0/
C RICH(1) = 0  IS NOT USED
C RICH(K) = 2**(2*K) - 1,  K=2,3,4,5,6,7
      TEMP2=PREVOT(I)
      PREVOT(1)=QUANT
      TEMP1=QUANT
      IF(M.EQ.1) GOTO 30
   20 REPS=EPS*(1.0+ABS(QUANT))
      DO 23 K=2,M
        DIFF=TEMP1-TEMP2
        IF(ABS(DIFF)-REPS) 25,25,22
   22   IF(K.EQ.8) GOTO 30
        TEMP1=TEMP1+DIFF/RICH(K)
        TEMP2=PREVOT(K)
        PREVOT(K)=TEMP1
   23   CONTINUE
      GO TO 30
   25 L*2
   30 VALUE=TEMP1
      RETURN
      END
```

Remark on Algorithm 418 [D1]

Calculation of Fourier Integrals [Bo Einarsson, *Comm. ACM 15* (Jan. 1972), 47–48]

Bo Einarsson [Recd. 31 Jan. 1972]
Research Institute of National Defense, Box 98,
S-147 00 Tumba, Sweden

Key Words and Phrases: quadrature, Filon quadrature, integration, Filon integration, Fourier coefficients, Fourier integrals, Fourier series, spline, spline approximation, spline quadrature, extrapolation, Richardson extrapolation
CR Categories: 5.16

Algorithm 418 looks confusing since the first 12 lines of the Fortran listing have been lost at the printing. Another error is that the two formula lines in the description are interchanged; the routine of course evaluates the general Fourier cosine and sine integrals. Finally, in the last line of the references, for "publication," read "publisher." The beginning of the algorithm is

```
      SUBROUTINE FSPL2
     *           (F,A,B,FPA,FPB,FBA,FBB,W,EPS,MAX,C,S,LC,LS)
C
C THIS ROUTINE COMPUTES THE FOURIER INTEGRALS
C`C=INTEGRAL F(X) COS WX DX FROM X=A TO X=B
C S=INTEGRAL F(X) SIN WX DX FROM X=A TO X=B
C
C WITH THE SPLINE PROCEDURE IN B. EINARSSON, NUMERICAL
C CALCULATION OF FOURIER INTEGRALS WITH CUBIC SPLINES,
C BIT, VOL. 8, PP. 279-286, 1968.
C
C REPEATED RICHARDSON EXTRAPOLATION IS USED.
```

Remark on Algorithm 418 [D1]

Calculation of Fourier Integrals [Bo Einarsson, *Comm. ACM 15* (Jan. 1972), 47–48]

Robert Piessens [Recd. 1 June 1973]
Applied Mathematics and Programming Division, University of Leuven, B-3030 Heverlee, Belgium

The algorithm has been tested in double precision on an IBM 370/155 with success. However, in the case that the Fourier cosine integral C and the Fourier sine integral S of the function $F(x)$ are wanted simultaneously (LC and LS positive on entry), the efficiency can be improved, since each value of $F(x)$ is then computed twice. This causes a considerable waste of computing time, which can easily be avoided by the following alterations:
(i) insert statement
FX = F(X)
5 lines after statement 20.
(ii) replace statement 50 by
50 SUMSIN = SUMSIN + FX*SIN(WX)
and statement 60 by
60 SUMCOS = SUMCOS + FX*COS(WX)

Algorithm 419

Zeros of a Complex Polynomial [C2]

M.A. Jenkins
Queen's University, Kingston, Ontario, Canada
and
J.F. Traub* [Recd. 10 Aug. 1970]
Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213

Key Words and Phrases: roots, roots of a polynomial, zeros of a polynomial
CR Categories: 5.15

Description

The subroutine *CPOLY* is a Fortran program to find all the zeros of a complex polynomial by the three-stage complex algorithm described in Jenkins and Traub [4]. (An algorithm for real polynomials is given in [5].) The algorithm is similar in spirit to the two-stage algorithms studied by Traub [1, 2]. The program finds the zeros one at a time in roughly increasing order of modulus and deflates the polynomial to one of lower degree. The program is extremely fast and the timing is quite insensitive to the distribution of zeros. Extensive testing of an Algol version of the program, reported in Jenkins [3], has shown the program to be very reliable.

The program is written in a portable subset of ANSI Fortran. It has been successfully used on the IBM 360/65, the GE 635 and the CDC 6600. The program is a translation of the Algol 60 procedure *cpolyzerofinder* appearing in [3].

MCON, the final subroutine of the program, sets four variables which describe the precision and range of the floating point arithmetic being used. Instructions for setting *MCON* variables are given in the *MCON* comments. The algorithm will accept polynomials of maximal degree 49.

The authors would like to thank K. Paciorek and M.T. Dolan for their assistance in preparing the Fortran version of the program and P. Businger and C. Lawson for suggesting improvements to the program.

* This work was done while J.F. Traub was at Bell Telephone Laboratories.

References

1. Traub, J.F. A class of globally convergent iteration functions for the solution of polynomial equations. *Math. Comp. 20* (1966), 113–138.
2. Traub, J.F. The calculation of zeros of polynomials and analytic functions. In *Mathematical Aspects of Computer Science, Proceedings Symposium Applied Mathematics, Vol. 19,* Amer. Math. Soc., Providence, R.I., 1967, pp. 138–152.
3. Jenkins, M.A. Three-stage variable-shift iterations for the solution of polynomial equations with a posteriori error bounds for the zeros. Diss., Rep. CS 138, Comput. Sci. Dep., Stanford U., Stanford, Cal., 1969.
4. Jenkins, M.A., and Traub, J.F. A three-stage variable-shift iteration for polynomial zeros and its relation to generalized Rayleigh iteration. *Numer. Math. 14* (1970), 252–263.
5. Jenkins, M.A., and Traub, J.F. A three-stage algorithm for real polynomials using quadratic iteration. *SIAM J. Numer. Anal.* 7 (1970), 545–566.

Algorithm

```
      SUBROUTINE CPOLY(OPR,OPI,DEGREE,ZEROR,ZEROI,FAIL)
C FINDS THE ZEROS OF A COMPLEX POLYNOMIAL.
C OPR, OPI  -  DOUBLE PRECISION VECTORS OF REAL AND
C IMAGINARY PARTS OF THE COEFFICIENTS IN
C ORDER OF DECREASING POWERS.
C DEGREE    -  INTEGER DEGREE OF POLYNOMIAL.
C ZEROR, ZEROI  -  OUTPUT DOUBLE PRECISION VECTORS OF
C REAL AND IMAGINARY PARTS OF THE ZEROS.
C FAIL      -  OUTPUT LOGICAL PARAMETER,  TRUE  ONLY IF
C LEADING COEFFICIENT IS ZERO OR IF CPOLY
C HAS FOUND FEWER THAN DEGREE ZEROS.
C THE PROGRAM HAS BEEN WRITTEN TO REDUCE THE CHANCE OF OVERFLOW
C OCCURRING. IF IT DOES OCCUR, THERE IS STILL A POSSIBILITY THAT
C THE ZEROFINDER WILL WORK PROVIDED THE OVERFLOWED QUANTITY IS
C REPLACED BY A LARGE NUMBER.
C COMMON AREA
      COMMON/GLOBAL/PR,PI,HR,HI,QPR,QPI,QHR,QHI,SHR,SHI,
     *    SR,SI,TR,TI,PVR,PVI,ARE,MRE,ETA,INFIN,NN
      DOUBLE PRECISION SR,SI,TR,TI,PVR,PVI,ARE,MRE,ETA,INFIN,
     *    PR(50),PI(50),HR(50),HI(50),QPR(50),QPI(50),QHR(50),
     *    QHI(50),SHR(50),SHI(50)
C TO CHANGE THE SIZE OF POLYNOMIALS WHICH CAN BE SOLVED, REPLACE
C THE DIMENSION OF THE ARRAYS IN THE COMMON AREA.
      DOUBLE PRECISION XX,YY,COSR,SINR,SMALNO,BASE,XXX,ZR,ZI,BND,
     *    OPR(1),OPI(1),ZEROR(1),ZEROI(1),
     *    CMOD,SCALE,CAUCHY,DSQRT
      LOGICAL FAIL,CONV
      INTEGER DEGREE,CNT1,CNT2
C INITIALIZATION OF CONSTANTS
      CALL MCON(ETA,INFIN,SMALNO,BASE)
      ARE = ETA
      MRE = 2.0D0*DSQRT(2.0D0)*ETA
      XX = .70710678
      YY = -XX
      COSR = -.060756474
      SINR = .99756405
      FAIL = .FALSE.
      NN = DEGREE+1
C ALGORITHM FAILS IF THE LEADING COEFFICIENT IS ZERO.
      IF (OPR(1) .NE. 0.0D0 .OR. OPI(1) .NE. 0.0D0) GO TO 10
          FAIL = .TRUE.
          RETURN
C REMOVE THE ZEROS AT THE ORIGIN IF ANY.
   10 IF (OPR(NN) .NE. 0.0D0 .OR. OPI(NN) .NE. 0.0D0) GO TO 20
          IDNN2 = DEGREE-NN+2
          ZEROR(IDNN2) = 0.0D0
          ZEROI(IDNN2) = 0.0D0
          NN = NN-1
          GO TO 10
C MAKE A COPY OF THE COEFFICIENTS.
   20 DO 30 I = 1,NN
          PR(I) = OPR(I)
          PI(I) = OPI(I)
          SHR(I) = CMOD(PR(I),PI(I))
   30 CONTINUE
C SCALE THE POLYNOMIAL.
      BND = SCALE (NN,SHR,ETA,INFIN,SMALNO,BASE)
      IF (BND .EQ. 1.0D0) GO TO 40
      DO 35 I = 1,NN
           PR(I) = BND*PR(I)
           PI(I) = BND*PI(I)
   35 CONTINUE
C START THE ALGORITHM FOR ONE ZERO.
   40 IF (NN.GT. 2) GO TO 50
C CALCULATE THE FINAL ZERO AND RETURN.
           CALL CDIVID(-PR(2),-PI(2),PR(1),PI(1),ZEROR(DEGREE),
     *     ZEROI(DEGREE))
           RETURN
C CALCULATE BND, A LOWER BOUND ON THE MODULUS OF THE ZEROS.
   50 DO 60 I = 1,NN
           SHR(I) = CMOD(PR(I),PI(I))
   60 CONTINUE
      BND = CAUCHY(NN,SHR,SHI)
C OUTER LOOP TO CONTROL 2 MAJOR PASSES WITH DIFFERENT SEQUENCES
C OF SHIFTS.
      DO 100 CNT1 = 1,2
C FIRST STAGE CALCULATION, NO SHIFT.
           CALL NOSHFT(5)
C INNER LOOP TO SELECT A SHIFT.
           DO 90 CNT2 = 1,9
C SHIFT IS CHOSEN WITH MODULUS BND AND AMPLITUDE ROTATED BY
C 94 DEGREES FROM THE PREVIOUS SHIFT.
                XXX = COSR*XX-SINR*YY
                YY = SINR*XX+COSR*YY
                XX = XXX
                SR = BND*XX
                SI = BND*YY
C SECOND STAGE CALCULATION, FIXED SHIFT.
                CALL FXSHFT(10*CNT2,ZR,ZI,CONV)
                IF (.NOT. CONV) GO TO 80
C THE SECOND STAGE JUMPS DIRECTLY TO THE THIRD STAGE ITERATION.
C IF SUCCESSFUL THE ZERO IS STORED AND THE POLYNOMIAL DEFLATED.
                     IDNN2 = DEGREE-NN+2
                     ZEROR(IDNN2) = ZR
                     ZEROI(IDNN2) = ZI
                     NN = NN-1
                     DO 70 I = 1,NN
                          PR(I) = QPR(I)
                          PI(I) = QPI(I)
   70                CONTINUE
                     GO TO 40
   80           CONTINUE
C IF THE ITERATION IS UNSUCCESSFUL ANOTHER SHIFT IS CHOSEN.
   90      CONTINUE
C IF 9 SHIFTS FAIL, THE OUTER LOOP IS REPEATED WITH ANOTHER
C SEQUENCE OF SHIFTS.
  100 CONTINUE
C THE ZEROFINDER HAS FAILED ON TWO MAJOR PASSES.
C RETURN EMPTY HANDED.
      FAIL = .TRUE.
      RETURN
      END
      SUBROUTINE  NOSHFT(L1)
C COMPUTES  THE DERIVATIVE  POLYNOMIAL AS THE INITIAL H
C POLYNOMIAL AND COMPUTES L1 NO-SHIFT H POLYNOMIALS.
C COMMON AREA
      COMMON/GLOBAL/PR,PI,HR,HI,QPR,QPI,QHR,QHI,SHR,SHI,
     *    SR,SI,TR,TI,PVR,PVI,ARE,MRE,ETA,INFIN,NN
      DOUBLE PRECISION SR,SI,TR,TI,PVR,PVI,ARE,MRE,ETA,INFIN,
     *    PR(50),PI(50),HR(50),HI(50),QPR(50),QPI(50),QHR(50),
     *    QHI(50),SHR(50),SHI(50)
      DOUBLE PRECISION XNI,T1,T2,CMOD
      N = NN-1
      NM1 = N-1
      DO 10 I = 1,N
           XNI = NN-I
           HR(I) = XNI*PR(I)/FLOAT(N)
           HI(I) = XNI*PI(I)/FLOAT(N)
   10 CONTINUE
      DO 50 JJ = 1,L1
           IF (CMOD(HR(N),HI(N)) .LE. ETA*10.0D0*CMOD(PR(N),PI(N)))
     *     GO TO 30
           CALL CDIVID(-PR(NN),-PI(NN),HR(N),HI(N),TR,TI)
           DO 20 I = 1,NM1
                J = NN-I
                T1 = HR(J-1)
                T2 = HI(J-1)
                HR(J) = TR*T1-TI*T2+PR(J)
                HI(J) = TR*T2+TI*T1+PI(J)
   20      CONTINUE
           HR(1) = PR(1)
           HI(1) = PI(1)
           GO TO 50
C IF THE CONSTANT TERM IS ESSENTIALLY ZERO, SHIFT H COEFFICIENTS.
   30      DO 40 I = 1,NM1
                J = NN-I
                HR(J) = HR(J-1)
                HI(J) = HI(J-1)
   40      CONTINUE
           HR(1) = 0.0D0
           HI(1) = 0.0D0
   50 CONTINUE
      RETURN
      END
      SUBROUTINE FXSHFT(L2,ZR,ZI,CONV)
C COMPUTES L2 FIXED-SHIFT H POLYNOMIALS AND TESTS FOR
C CONVERGENCE.
C INITIATES A VARIABLE-SHIFT ITERATION AND RETURNS WITH THE
C APPROXIMATE ZERO IF SUCCESSFUL.
C L2 - LIMIT OF FIXED SHIFT STEPS
C ZR,ZI - APPROXIMATE ZERO IF CONV IS .TRUE.
C CONV  - LOGICAL INDICATING CONVERGENCE OF STAGE 3 ITERATION
C COMMON AREA
      COMMON/GLOBAL/PR,PI,HR,HI,QPR,QPI,QHR,QHI,SHR,SHI,
     *    SR,SI,TR,TI,PVR,PVI,ARE,MRE,ETA,INFIN,NN
      DOUBLE PRECISION SR,SI,TR,TI,PVR,PVI,ARE,MRE,ETA,INFIN,
     *    PR(50),PI(50),HR(50),HI(50),QPR(50),QPI(50),QHR(50),
     *    QHI(50),SHR(50),SHI(50)
      DOUBLE PRECISION ZR,ZI,OTR,OTI,SVSR,SVSI,CMOD
           LOGICAL CONV,TEST,PASD,BOOL
      N = NN-1
C EVALUATE P AT S.
      CALL POLYEV(NN,SR,SI,PR,PI,QPR,QPI,PVR,PVI)
      TEST = .TRUE.
      PASD = .FALSE.
C CALCULATE FIRST T = -P(S)/H(S).
      CALL CALCT(BOOL)
C MAIN LOOP FOR ONE SECOND STAGE STEP.
      DO 50 J = 1,L2
           OTR = TR
           OTI = TI
C COMPUTE NEXT H POLYNOMIAL AND NEW T.
           CALL NEXTH(BOOL)
           CALL CALCT(BOOL)
           ZR = SR+TR
           ZI = SI+TI
C TEST FOR CONVERGENCE UNLESS STAGE 3 HAS FAILED ONCE OR THIS
C IS THE LAST H POLYNOMIAL.
           IF ( BOOL .OR. .NOT. TEST .OR. J .EQ. L2) GO TO 50
           IF (CMOD(TR-OTR,TI-OTI) .GE. .5D0*CMOD(ZR,ZI)) GO TO 40
                IF (.NOT. PASD) GO TO 30
C THE WEAK CONVERGENCE TEST HAS BEEN PASSED TWICE, START THE
C THIRD STAGE ITERATION, AFTER SAVING THE CURRENT H POLYNOMIAL
C AND SHIFT.
                     DO 10 I = 1,N
                          SHR(I) = HR(I)
                          SHI(I) = HI(I)
   10                CONTINUE
                     SVSR = SR
                     SVSI = SI
                     CALL VRSHFT(10,ZR,ZI,CONV)
                     IF (CONV) RETURN
C THE ITERATION FAILED TO CONVERGE. TURN OFF TESTING AND RESTORE
C H,S,PV AND T.
                     TEST = .FALSE.
                     DO 20 I = 1,N
                          HR(I) = SHR(I)
                          HI(I) = SHI(I)
   20                CONTINUE
                     SR = SVSR
                     SI = SVSI
                     CALL POLYEV(NN,SR,SI,PR,PI,QPR,QPI,PVR,PVI)
                     CALL CALCT(BOOL)
                     GO TO 50
   30           PASD = .TRUE.
                GO TO 50
   40      PASD = .FALSE.
   50 CONTINUE
C ATTEMPT AN ITERATION WITH FINAL H POLYNOMIAL FROM SECOND STAGE.
      CALL VRSHFT(10,ZR,ZI,CONV)
      RETURN
      END
      SUBROUTINE VRSHFT(L3,ZR,ZI,CONV)
C CARRIES OUT THE THIRD STAGE ITERATION.
C L3 - LIMIT OF STEPS IN STAGE 3.
C ZR,ZI   - ON ENTRY CONTAINS THE INITIAL ITERATE, IF THE
C ITERATION CONVERGES IT CONTAINS THE FINAL ITERATE
C ON EXIT.
C CONV    -  .TRUE. IF ITERATION CONVERGES
C COMMON AREA
      COMMON/GLOBAL/PR,PI,HR,HI,QPR,QPI,QHR,QHI,SHR,SHI,
     *    SR,SI,TR,TI,PVR,PVI,ARE,MRE,ETA,INFIN,NN
      DOUBLE PRECISION SR,SI,TR,TI,PVR,PVI,ARE,MRE,ETA,INFIN,
     *    PR(50),PI(50),HR(50),HI(50),QPR(50),QPI(50),QHR(50),
     *    QHI(50),SHR(50),SHI(50)
      DOUBLE PRECISION ZR,ZI,MP,MS,OMP,RELSTP,R1,R2,CMOD,DSQRT,ERREV,TP
      LOGICAL CONV,B,BOOL
      CONV = .FALSE.
      B = .FALSE.
      SR = ZR
      SI = ZI
C MAIN LOOP FOR STAGE THREE.
      DO 60 I = 1,L3
C EVALUATE P AT S AND TEST FOR CONVERGENCE.
           CALL POLYEV(NN,SR,SI,PR,PI,QPR,QPI,PVR,PVI)
           MP = CMOD(PVR,PVI)
           MS = CMOD(SR,SI)
           IF (MP .GT. 20.0D0*ERREV(NN,QPR,QPI,MS,MP,ARE,MRE))
     *        GO TO 10
C POLYNOMIAL VALUE IS SMALLER IN VALUE THAN A BOUND ON THE ERROR
C IN EVALUATING P, TERMINATE THE ITERATION.
                CONV = .TRUE.
                ZR = SR
                ZI = SI
                RETURN
   10      IF (I .EQ. 1) GO TO 40
                IF (B .OR. MP .LT.OMP .OR. RELSTP .GE. .05D0)
     *             GO TO 30
C ITERATION HAS STALLED. PROBABLY A CLUSTER OF ZEROS. DO 5 FIXED
C SHIFT STEPS INTO THE CLUSTER TO FORCE ONE ZERO TO DOMINATE.
                     TP = RELSTP
                     B = .TRUE.
                     IF (RELSTP .LT. ETA) TP = ETA
                     R1 = DSQRT(TP)
                     R2 = SR*(1.0D0+R1)-SI*R1
                     SI = SR*R1+SI*(1.0D0+R1)
                     SR = R2
                     CALL POLYEV(NN,SR,SI,PR,PI,QPR,QPI,PVR,PVI)
                     DO 20 J = 1,5
                          CALL CALCT(BOOL)
                          CALL NEXTH(BOOL)
   20                CONTINUE
      OMP = INFIN
                     GO TO 50
C EXIT IF POLYNOMIAL VALUE INCREASES SIGNIFICANTLY.
   30           IF (MP*.1D0 .GT. OMP) RETURN
   40      OMP = MP
C CALCULATE NEXT ITERATE.
   50      CALL CALCT(BOOL)
           CALL NEXTH(BOOL)
           CALL CALCT(BOOL)
           IF (BOOL) GO TO 60
           RELSTP = CMOD(TR,TI)/CMOD(SR,SI)
           SR = SR+TR
           SI = SI+TI
   60 CONTINUE
      RETURN
      END
      SUBROUTINE CALCT(BOOL)
C COMPUTES  T = -P(S)/H(S).
C BOOL   - LOGICAL, SET TRUE IF H(S) IS ESSENTIALLY ZERO.
C COMMON AREA
      COMMON/GLOBAL/PR,PI,HR,HI,QPR,QPI,QHR,QHI,SHR,SHI,
     *    SR,SI,TR,TI,PVR,PVI,ARE,MRE,ETA,INFIN,NN
      DOUBLE PRECISION SR,SI,TR,TI,PVR,PVI,ARE,MRE,ETA,INFIN,
     *    PR(50),PI(50),HR(50),HI(50),QPR(50),QPI(50),QHR(50;,
     *    QHI(50),SHR(50),SHI(50)
      DOUBLE PRECISION HVR,HVI,CMOD
      LOGICAL BOOL
      N = NN-1
C EVALUATE H(S).
      CALL POLYEV(N,SR,SI,HR,HI,QHR,QHI,HVR,HVI)
      BOOL = CMOD(HVR,HVI) .LE. ARE*10.0D0*CMOD(HR(N),HI(N))
      IF (BOOL) GO TO 10
          CALL CDIVID(-PVR,-PVI,HVR,HVI,TR,TI)
          RETURN
   10 TR = 0.0D0
      TI = 0.0D0
      RETURN
      END
      SUBROUTINE NEXTH(BOOL)
C CALCULATES THE NEXT SHIFTED H POLYNOMIAL.
C BOOL   -  LOGICAL, IF .TRUE. H(S) IS ESSENTIALLY ZERO
C COMMON AREA
      COMMON/GLOBAL/PR,PI,HR,HI,QPR,QPI,QHR,QHI,SHR,SHI,
     *    SR,SI,TR,TI,PVR,PVI,ARE,MRE,ETA,INFIN,NN
      DOUBLE PRECISION SR,SI,TR,TI,PVR,PVI,ARE,MRE,ETA,INFIN,
     *    PR(50),PI(50),HR(50),HI(50),QPR(50),QPI(50),QHR(50),
     *    QHI(50),SHR(50),SHI(50)
      DOUBLE PRECISION T1,T2
      LOGICAL BOOL
      N = NN-1
      NM1 = N-1
      IF (BOOL) GO TO 20
          DO 10 J = 2,N
              T1 = QHR(J-1)
              T2 = QHI(J-1)
              HR(J) = TR*T1-TI*T2+QPR(J)
              HI(J) = TR*T2+TI*T1+QPI(J)
   10     CONTINUE
          HR(1) = QPR(1)
          HI(1) = QPI(1)
          RETURN
C IF H(S) IS ZERO REPLACE H WITH QH.
   20 DO 30 J = 2,N
          HR(J) = QHR(J-1)
          HI(J) = QHI(J-1)
   30 CONTINUE
      HR(1) = 0.0D0
      HI(1) = 0.0D0
      RETURN
      END
      SUBROUTINE POLYEV(NN,SR,SI,PR,PI,QR,QI,PVR,PVI)
C EVALUATES A POLYNOMIAL  P  AT  S  BY THE HORNER RECURRENCE
C PLACING THE PARTIAL SUMS IN Q AND THE COMPUTED VALUE IN PV.
      DOUBLE PRECISION PR(NN),PI(NN),QR(NN),QI(NN),
     *    SR,SI,PVR,PVI,T
      QR(1) = PR(1)
      QI(1) = PI(1)
      PVR = QR(1)
      PVI = QI(1)
      DO 10 I = 2,NN
          T = PVR*SR-PVI*SI+PR(I)
          PVI = PVR*SI+PVI*SR+PI(I)
          PVR = T
          QR(I) = PVR
          QI(I) = PVI
   10 CONTINUE
      RETURN
      END
      DOUBLE PRECISION FUNCTION ERREV(NN,QR,QI,MS,MP,ARE,MRE)
C BOUNDS THE ERROR IN EVALUATING THE POLYNOMIAL BY THE HORNER
C RECURRENCE.
C QR,QI - THE PARTIAL SUMS
C MS    -MODULUS OF THE POINT
C MP    -MODULUS OF POLYNOMIAL VALUE
C ARE, MRE -ERROR BOUNDS ON COMPLEX ADDITION AND MULTIPLICATION
      DOUBLE PRECISION QR(NN),QI(NN),MS,MP,ARE,MRE,E,CMOD
      E = CMOC(QR(1),QI(1))*MRE/(ARE+MRE)
      DO 10 I = 1,NN
          E = E*MS+CMOD(QR(I),QI(I))
   10 CONTINUE
      ERREV = E*(ARE+MRE)-MP*MRE
      RETURN
      END
      DOUBLE PRECISION FUNCTION CAUCHY(NN,PT,Q)
C CAUCHY COMPUTES A LOWER BOUND ON THE MODULI OF THE ZEROS OF A
C POLYNOMIAL - PT IS THE MODULUS OF THE COEFFICIENTS.
      DOUBLE PRECISION Q(NN),PT(NN),X,XM,F,DX,DF,
     *   DABS,DEXP,DLOG
      PT(NN) = -PT(NN)
C COMPUTE UPPER ESTIMATE OF BOUND.
      N = NN-1
      X = DEXP( (DLOG(-PT(NN)) - DLOG(PT(1)))/FLOAT(N) )
      IF (PT(N).EQ.0.CD0) GO TO 20
C IF NEWTON STEP AT THE ORIGIN IS BETTER, USE IT.
          XM = -PT(NN)/PT(N)
          IF (XM.LT.X) X=XM
C CHOP THE INTERVAL (0,X) UNITL F<=0.
   20 XM = X*.1D0
      F = PT(1)
      DO 30 I = 2,NN
          F = F*XM+PT(I)
   30 CONTINUE
      IF (F.LE. 0.0D0) GO TO 40
          X = XM
          GO TO 20
   40 DX = X
C DO NEWTON ITERATION UNTIL X CONVERGES TO TWO DECIMAL PLACES.
   50 IF (DABS(DX/X) .LE. .005D0) GO TO 70
          Q(1) = PT(1)
          DO 60 I = 2,NN
              Q(I) = Q(I-1)*X+PT(I)
   60     CONTINUE
          F = Q(NN)
          DF = Q(1)
          DO 65 I = 2,N
              DF = DF*X+Q(I)
   65     CONTINUE
          DX = F/DF
          X = X-DX
          GO TO 50
   70 CAUCHY = X
      RETURN
      END
      DOUBLE PRECISION FUNCTION SCALE(NN,PT,ETA,INFIN,SMALNO,BASE)
C RETURNS A SCALE FACTOR TO MULTIPLY THE COEFFICIENTS OF THE
C POLYNOMIAL. THE SCALING IS DONE TO AVOID OVERFLOW AND TO AVOID
C UNDETECTED UNDERFLOW INTERFERING WITH THE CONVERGENCE
C CRITERION.  THE FACTOR IS A POWER OF THE BASE.
C PT - MODULUS OF COEFFICIENTS OF P
C ETA,INFIN,SMALNO,BASE - CONSTANTS DESCRIBING THE
C FLOATING POINT ARITHMETIC.
      DOUBLE PRECISION PT(NN),ETA,INFIN,SMALNO,BASE,HI,LO,
     *    MAX,MIN,X,SC,DSQURT,DLOG
C FIND LARGEST AND SMALLEST MODULI OF COEFFICIENTS.
      HI = DSQRT(INFIN)
      LO = SMALNO/ETA
      MAX = 0.0D0
      MIN = INFIN
      DO 10 I = 1,NN
          X = PT(I)
          IF (X .GT. MAX) MAX = X
          IF (X .NE. 0.0D0 .AND. X.LT.MIN) MIN = X
   10 CONTINUE
C SCALE ONLY IF THERE ARE VERY LARGE OR VERY SMALL COMPONENTS.
      SCALE = 1.0D0
      IF (MIN .GE. LO .AND. MAX .LE. HI) RETURN
      X = LO/MIN
      IF (X .GT. 1.0D0) GO TO 20
          SC = 1.0D0/(DSQRT(MAX)*DSQRT(MIN))
          GO TO 30
   20 SC = X
      IF (INFIN/SC .GT. MAX) SC = 1.0D0
   30 L = DLOG(SC)/DLOG(BASE) + .500
      SCALE = BASE**L
      RETURN
      END
      SUBROUTINE CDIVID(AR,AI,BR,BI,CR,CI)
C COMPLEX DIVISION C = A/B, AVOIDING OVERFLOW.
      DOUBLE PRECISION AR,AI,BR,BI,CR,CI,R,D,T,INFIN,DABS
      IF (BR .NE. 0.0D0  .OR. BI .NE. 0.0D0) GO TO 10
C DIVISION BY ZERO, C = INFINITY.
          CALL MCON (T,INFIN,T,T)
          CR = INFIN
          CI = INFIN
          RETURN
   10 IF (DABS(BR) .GE. DABS(BI)) GO TO 20
          R = BR/BI
          D = BI+R*BR
          CR = (AR*R+AI)/D
          CI = (AI*R-AR)/D
          RETURN
   20 R = BI/BR
      D = BR+R*BI
      CR = (AR+AI*R)/D
      CI = (AI-AR*R)/D
      RETURN
      END
      DOUBLE PRECISION FUNCTION CMOD(R,I)
C MODULUS OF A COMPLEX NUMBER AVOIDING OVERFLOW.
      DOUBLE PRECISION R,I,AR,AI,DABS,DSQURT
      AR = DABS(R)
      AI = DABS(I)
      IF (AR .GE. AI) GO TO 10
          CMOC = AI*DSQRT(1.0D0+(AR/AI)**2)
          RETURN
   10 IF (AR .LE. AI) GO TO 20
          CMOD = AR*DSQRT(1.0D0+(AI/AR)**2)
          RETURN
   20 CMOD = AR*DSQRT(2.0D0)
      RETURN
      END
      SUBROUTINE MCON(ETA,INFINY,SMALNO,BASE)
C MCON PROVIDES MACHINE CONSTANTS USED IN VARIOUS PARTS OF THE
C PROGRAM. THE USER MAY EITHER SET THEM DIRECTLY OR USE THE
C STATEMENTS BELOW TO COMPUTE THEM. THE MEANING OF THE FOUR
C CONSTANTS ARE -
C ETA       THE MAXIMUM RELATIVE REPRESENTATION ERROR
C WHICH CAN BE DESCRIBED AS THE SMALLEST POSITIVE
C FLOATING-POINT NUMBER SUCH THAT 1.0D0 + ETA IS
C GREATER THAN 1.0D0.
C INFINY    THE LARGEST FLOATING-POINT NUMBER
C SMALNO    THE SMALLEST POSITIVE FLOATING-POINT NUMBER
C BASE      THE BASE OF THE FLOATING-POINT NUMBER SYSTEM USED
C LET T BE THE NUMBER OF BASE-DIGITS IN EACH FLOATING-POINT
C NUMBER(DOUBLE PRECISION). THEN ETA IS EITHER .5*B**(1-T)
C OR B**(1-T) DEPENDING ON WHETHER ROUNDING OR TRUNCATION
C IS USED.
C LET M BE THE LARGEST EXPONENT AND N THE SMALLEST EXPONENT
C IN THE NUMBER SYSTEM. THEN INFINY IS (1-BASE**(-T))*BASE**M
C AND SMALNO IS BASE**N.
C THE VALUES FOR BASE,T,M,N BELOW CORRESPOND TO THE IBM/360.
      DOUBLE PRECISION ETA,INFINY,SMALNO,BASE
      INTEGER M,N,T
      BASE = 16.0D0
      T = 14
      M = 63
      N = -65
      ETA = BASE**(1-T)
      INFINY = BASE*(1.0D0-BASE**(-T))*BASE**(M-1)
      SMALNO = (BASE**(N+3))/BASE**3
      RETURN
      END
```

Remark on Algorithm 419 [C2]

Zeros of a Complex Polynomial [M.A. Jenkins and J.F. Traub, *Comm. ACM 15* (Feb. 1972), 97–99]

David H. Withers [Rec. 9 Oct. 1972 and 14 May 1973]
IBM, Essex Junction, VT 04352

The published algorithm has performed satisfactorily for all except one (degenerate) case. When removing zeros at the origin, the algorithm does not stop if all roots have been located. An error will occur if the polynomials, $X^N = 0$ or $a_N = 0$ are given to the algorithm. The difficulty may be avoided by inserting after statement 40 the statement

IF (*NN.EQ.* 1)*RETURN*

The referee pointed out the second type of degenerate case above and two typographical errors:

1. In the initialization of constants section *COSR* should be initialized by *COSR* = –.069756474.
2. In the *FUNCTIONS SCALE* and *CMOD*, the declaration of *DSQRT* as *DOUBLE PRECISION* was accidentally typed as *DSQURT*.

Algorithm 420

Hidden-Line Plotting Program [J6]

Hugh Williamson [Recd. 4 March 1970 and 4 Feb. 1971]
Tracor Computing Corporation, Austin, Texas

Key Words and Phrases: hidden-line plot, surface plot
CR Categories: 4.9

Description

HIDE produces a two-dimensional representation of a surface or figure by plotting segments of a succession of curves; each curve is plotted where it is not hidden by any of the curves previously plotted (that is, where it does not fall below any of them as they appear in the two-dimensional representation).

The calling sequence is described in some detail in the comment cards at the first of the subroutine.

The following are options:

(1) Translate the arrays before plotting to simulate stepping in the depth dimension.

(2) Draw any of the following: an $8\frac{1}{2}$ by 11 inch border, axes, and a title. (Whether this option is exercised or not, labeling may be added by the calling program.)

(3) Draw the unhidden part of the underside of a figure. In this case, the lines are assumed to be hidden where they fall above those previously plotted. This option together with the program's capability to plot the visible maximum can be used to represent the unhidden areas of both the top and the underside of a surface. This can be done by plotting all visible segments of each successive curve, beginning with the farthest in the foreground, as in the exemplary driver routine that produces the graph titled Test for plotting routine *HIDE*. Or all the segments to represent the top of the surface can be drawn first, and then all the segments to represent the underside. The method used in the driver routine listed is advantageous in that only one of the curves to be plotted must be stored at a time, but it is disadvantageous in that two sets of working arrays are required.

Explicit provisions are not made in *HIDE* for perspective plots or for rotations. If, however, the arrays to be plotted are properly transformed before *HIDE* is called, such effects can be achieved.

The arrays *XG*, *G*, *XH*, and *H* must be dimensioned in the calling program. *G* vs. *XG* is the visual maximum function; that is, after the first $n - 1$ curves have been plotted, *G* vs. *XG* is the function such that the nth curve falls below one or more of the first $n - 1$ curves (as they appear in the two-dimensional graph) at exactly the same points where it falls below *G* vs. *XG*. (Thus the intersections of the nth curve with *G* vs. *XG* are endpoints of intervals within which the nth curve is entirely hidden or entirely visible.)

The number of points used in arrays *G* and *XG* after n curves have been plotted is the sum of:

(1) the number of original data points of any of the first n curves that lie on the curve *G* vs. *XG*,

(2) the number of intersections of different curves that lie on *G* vs. *XG* (if the kth curve coincides with the maximum function of the first $k - 1$ curves over an interval, every data point of the kth curve with an abcissa within that interval is considered an intersection), and

(3) the number of points needed to simulate discontinuities in the maximum function; this number is no greater than four times the number of curves to be plotted for the graph.

An adequate dimension for *XG* and *G* and for the working arrays *XH* and *H* is an upper bound for the number of points that will be needed for the visual maximum function.

Developed on Tracor Computing Corporation's UNIVAC 1108 system, *HIDE* calls several basic systems plotting routines. In the listing, these calls are preceded by comment cards with asterisks across the lines. If *HIDE* were to be used on a different computer system, calls could be substituted to the corresponding routines of that system, or, if more flexibility were desired, to user-supplied routines.

Although partially explained in comment cards, the calling sequences of systems routines called by *HIDE* will be described more fully here. On TCC's system, these routines write pen codes on magnetic tapes, which are used to drive offline drum plotters.

PDATA(X,Y,N,J,L,XMIN,DX,YMIN,DY,HT)

This routine plots curves.

X is the abcissa array.

Y is the ordinate array.

N is the number of points $(X(I), Y(I))$ to be plotted.

$|J|$ is the number of data points from plotted symbol to plotted symbol. If $J = 0$, a line plot will be produced. If *J* is negative, only the symbols will be plotted. If *J* is positive, both the line and the symbols will be plotted.

L specifies the symbol to be plotted (the table correlating values of *L* with symbols would be of interest only to users of TCC's system).

XMIN is the x value at the plotting reference point, which is the origin for plotting pen movements (but not necessarily the data origin).

DX is the x increment per inch for the plot.

YMIN is the y value at the plotting reference point.

DY is the y increment per inch.

HT is the height in inches of the symbols.

MOVPEN(X,Y,I)

(X,Y) is the point in inches relative to the reference point to which the plotting pen is to be moved.

$|I| = 1$ if the pen is to be left as it was prior to this call (up or down).

$[I] = 2$ if the pen is to be placed down before movement.

$|I| = 3$ if the pen is to be picked up before movement.

If *I* is negative, (X,Y) will become the new reference point. Other options exist which are not used by *HIDE*.

PSYMB(X,Y,HT,T,TH,N)

This routine plots alphanumeric information.

(X,Y) is the position in inches relative to the reference point of the lower left-hand corner of the first symbol.

HT is the height of the symbols.

T is the starting location in core for the information to be plotted.

TH is the angle in degrees counter-clockwise relative to horizontal at which the symbols are to be plotted.

N is the number of symbols to be plotted.

PAXIS(*X,Y,T,N,S,TH,FMIN,DF*)

This routine draws and labels a linear axis.

(*X,Y*) is the point in inches relative to the plotting reference point of the beginning of the axis.

T is the starting location in core for a title for the axis.

| *N* | is the number of characters in the title. If *N* is negative, the labeling will be on the clockwise side of the axis; otherwise, on the counter-clockwise side.

S is the length in inches of the axis.

TH is the counter-clockwise angle in degrees relative to horizontal at which the axis is to be drawn.

FMIN is the data value at the start of the axis.

DF is the data increment per inch. *FMIN* and *DF* are necessary for labeling the axis.

PLTOFF, which is not called by *HIDE* but is called by driver routines, writes the remaining information in the buffer on the plot tape and writes an end-of-file mark.

If *HIDE* were to be used on a computer system with word length different from 36 bits, it is possible that *EPS*1, the relative abscissa increment used to simulate discontinuities in the visual maximum function, should be changed. *EPS*1, which is defined in a data statement near the beginning of the program, should be one or two orders of magnitude larger than the smallest recognizable relative difference in single precision floating point arithmetic.

The helpful suggestions made by the referees for improving the capabilities of *HIDE* are greatly appreciated by the author.

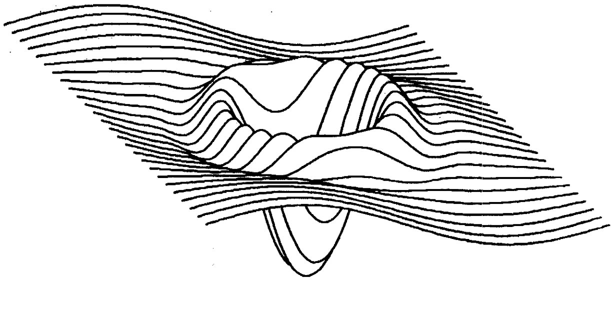
Algorithm

```
C
C THIS DRIVER RØUTINE FØR HIDE PRØDUCES THE GRAPH TITLED
C TEST FØR PLØTTING RØUTINE HIDE.
C
      DIMENSIØN X(150),Y(150),Y1(150),XG(500),G(500),XH(500)
     1          ,H(500),XG1(500),G1(500),TI(14)
      DATA NG,NG1,N,N1,NFNS,MAXDIM,XMIN,DELTAX,YMIN,DELTAY,
     1    XLNTH,YLNTH,XX/0,-3,150,-150,26,500,0.,1.05,-1.,
     2                   .67,-6.,-3.,3.141592654/
      EQUIVALENCE (XH(1),TI(1))
      READ 1,TI
    1 FØRMAT(13A6,A2)
      STEP = 3.141592654/74.5
      X(1) = 0.
      Y1(1) = 0.
      DØ 2 I = 2,N
        X(I) = X(I-1)+STEP
    2   Y1(I) = .2*SIN(X(I))
      Z = 0.
      STEP = 3.141592654/12.5
C
C THE CALLS TØ HIDE NECESSARY TØ PLØT THE TØP AND BØTTØM
C (UNDERSIDE) ØF A SURFACE ARE MADE IN THE FØLLØWING LØØP.
C
      DØ 3 I = 1,NFNS
        CZ = CØS(Z)
        DØ 4 J = 1,N
    4     Y(J) = Y1(J)*CZ-(EXP(-(X(J)-XX)**2-(Z-XX)**2)*
     1             CØS(1.75*((X(J)-XX)**2+(Z-XX)**2)))*1.5
C
C PLØT THE PART ØF THE ITH CURVE THAT LIES ØN THE UNHIDDEN
C PART ØF THE TØP ØF THE SURFACE.
C
        CALL HIDE(X,Y,XG,G,XH,H,NG,MAXDIM,N,NFNS,TI,XLNTH,
     1             YLNTH,XMIN,DELTAX,YMIN,DELTAY)
C
C PLØT THE PART ØF THE ITH CURVE THAT LIES ØN THE UNHIDDEN
C PART ØF THE UNDERSIDE ØF THE SURFACE.
C (NØTE.  IF PART ØF THE UNDERSIDE FALLS BELØW YMIN, BUT
C STAYS WITHIN THE DESIRED AREA ØN THE PLØT, HIDE WILL STILL
C PERFØRM THE PLØTTING CØRRECTLY.)
C
        CALL HIDE(X,Y,XG1,G1,XH,H,NG1,MAXDIM,N1,0,6HNØTTLE,
     1             XLNTH,YLNTH,XMIN,DELTAX,YMIN,DELTAY)
    3   Z = Z+STEP
C
C CALL SYSTEMS RØUTINES TØ MØVE THE PEN ØFF THE GRAPH TØ THE
C RIGHT AND TØ TERMINATE THE PLØT.
C
      CALL MØVPEN(10.,-2.,-3)
      CALL PLTØFF
      END
      SUBROUTINE HIDE(X,Y,XG,G,XH,H,NG,MAXDIM,N1,NFNS,TITLE,
     1                XLNTH,YLNTH,XMIN,DELTAX,YMIN,DELTAY)
C
C THIS SUBROUTINE PRODUCES A 2-DIMENSIONAL REPRESENTATION OF
C A 3-DIMENSICNAL FIGURE OR SURFACE.
C THE FIRST CALL TO HIDE IS FOR INITIALIZATION AND PLOTTING
C THE CURVE FARTHEST IN THE FOREGROUND.  ON EACH SUBSEQUENT
C CALL, A CURVE FARTHER IN THE BACKGROUND IS PLOTTED.
C X IS THE ABCISSA ARRAY FOR THE CURVE TO BE PLOTTED BY
C HIDE ON THIS CALL.  THE X VALUES MUST BE INCREASING.
C IF X(I) GE X(I+1) FOR SOME I, MAXDIM WILL BE SET TO ZERO,
C AND A RETURN WILL BE EXECUTED.
C Y IS THE ORCINATE ARRAY.
C G VS. XG IS THE CURRENT VISUAL MAXIMUM FUNCTION ON EACH
C RETURN FROM HIDE.
C XH AND H ARE WORKING ARRAYS.
C ON EACH RETURN FROM HIDE, NG IS THE NUMBER OF POINTS IN
C THE CURRENT MAXIMUM FUNCTION.
C ON THE FIRST CALL, NG IS A NONPOSITIVE INTEGER WHICH
C SPECIFIES CERTAIN OPTIONS.
C -1 DO NOT DRAW THE 8 1/2 BY 11 INCH BORDER.
C -? PLOT UNHIDDEN MINIMUM RATHER THAN MAXIMUM.  IN THIS
C    CASE, G VS. XG WILL BE THE NEGATIVE OF THE VISUAL
C    MINIMUM FUNCTION.
C -3 DO NOT PLOT BORDER, PLOT MINIMUM RATHER THAN MAXIMUM.
C 0 PLOT BORDER, PLOT MAXIMUM.
C IF THE BORDER IS DRAWN, ITS LEFT, BOTTOM CORNER WILL BE
C WHERE THE PLOTTING REFERENCE POINT WAS JUST BEFORE THE
C FIRST CALL TO HIDE, AND THE REFERENCE POINT WILL BE MOVED
C 1 INCH RIGHT AND 2 INCHES UP.  IF THE BORDER IS NOT DRAWN,
C THE REFERENCE POINT WILL NOT BE MOVED BY HIDE.
C MAXDIM IS THE DIMENSION IN THE CALLING PROGRAM OF THE
C ARRAYS XG, G, XH, AND H.  IF ONE OF THESE ARRAYS WOULD
C HAVE BEEN OVERFLOWED, MAXDIM IS SET EQUAL TO ITS NEGATIVE,
C AND A RETURN IS EXECUTED.
C N1 IS THE NUMBER OF POINTS (X(I),Y(I)) TO BE PLOTTED IN
C A GIVEN CALL TO HIDE.
C IF N1 IS LESS THAN 0, Y VS. X WILL NOT BE PLOTTED, BUT ON
C SUBSEQUENT CALLS, PLOTTING WILL BE DONE AS IF
C ((X(I),Y(I)),I=1,-N1) HAD BEEN PLOTTED (WHERE UNHIDDEN).
C N1 WILL BE RETURNED AS ITS ABSOLUTE VALUE.
C NFNS IS THE TOTAL NO. OF CURVES TO BE PLOTTED FOR THIS
C GRAPH IF TRANSLATING THE ARRAYS TO SIMULATE STEPPING IN
C THE DEPTH DIMENSION IS DESIRED.  IF NO TRANSLATION IS
C DESIRED, NFNS SHOULD BE NEGATIVE.  IF THE SAME TRANSLATION
C AS IN THE PREVIOUS CALL TO HIDE IS DESIRED, NFNS SHOULD BE
C ZERO.  THE NFNS=0 OPTION MAY BE SPECIFIED FOR INDIVIDUAL
C CURVES AFTER THE FIRST FOR A GIVEN GRAPH.  ALL
C TRANSLATIONS WHICH ARE PERFORMED WILL HAVE EQUAL STEP SIZE
C DETERMINED BY THE VALUES IN THE INITIAL CALL FOR XLNTH,
C YLNTH, AND NFNS.
C TITLE IS AN 80-CHARACTER TITLE.
C IF TITLE(1)=6HNOTTLE, THE TITLE WILL NOT BE PLOTTED.
C TITLE(1) ANC XH(1) OR H(1) MAY BE THE SAME LOCATION IF THE
C TITLE IS NOT NEEDED AFTER IT IS PLOTTED.
C XLNTH IS THE LENGTH IN INCHES OF THE HORIZONTAL AXIS.
C IF XLNTH IS LESS THAN 0, THE X-AXIS AND THE DEPTH AXIS
C WILL NOT BE DRAWN.  IN ANY CASE, UNLESS THIS OPTION IS
C SUPPRESSED THROUGH NFNS, THE ITH CURVE WILL BE TRANSLATED
C (I-1)*(9.-ABS(XLNTH))/(NFNS-1) INCHES TO THE LEFT.  THIS
C PLUS A SIMILAR VERTICAL TRANSLATION IS DONE TO SIMULATE
C STEPPING IN THE DEPTH DIMENSION.
C XMIN-(9.-ABS(XLNTH))*DELTAX WILL BE THE ABCISSA VALUE AT
C THE PLOTTING REFERENCE POINT (WHICH IS WHERE THE
C HORIZONTAL AND VERTICAL AXES WOULD INTERSECT IF DRAWN).
C YLNTH IS THE LENGTH OF THE VERTICAL AXIS IN INCHES.
C IF YLNTH IS LESS THAN 0, THE VERTICAL AND DEPTH AXES WILL
C NOT BE DRAWN.  BUT IN ANY CASE, UNLESS THIS OPTION IS
C SUPPRESSED THROUGH NFNS, THE ITH CURVE WILL BE TRANSLATED
C (I-1)*(6.-ABS(YLNTH))/(NFNS-1) INCHES UP TO SIMULATE
C STEPPING IN THE DEPTH DIMENSION.  YMIN-(6.-ABS(YLNTH))*
C DELTAY WILL BE THE ORDINATE VALUE AT THE PLOTTING
C REFERENCE PCINT.
C IF TRANSLATIONS ARE PERFORMED, X AND Y WILL BE RESTORED TO
C THEIR ORIGINAL VALUES BEFORE THE RETURN TO THE CALLING
C PROGRAM.
C NOTE THAT IF ABS(XLNTH)=9, AND ABS(YLNTH)=6, THERE WILL BE
C NO TRANSLATION, AND, IF BORDER AND AXES ARE NOT DRAWN, THE
C DIMENSIONS CF THE PLOT ARE UNSPECIFIED.
C IF THE AXES AND BORDER ARE DRAWN, THE TOP OF THE VERTICAL
C AXIS AND THE RIGHT END OF THE HORIZONTAL AXIS ARE FIXED
C RELATIVE TO THE BORDER, AND THE DEPTH AXIS JOINS THE LEFT
C END OF THE HORIZONTAL AXIS AND THE BOTTOM OF THE VERTICAL AXIS.
C XMIN IS A LCWER BOUND FOR X.
C DELTAX IS THE X DATA INCREMENT PER INCH FOR THE PLOT.
C XMIN AND DELTAX DETERMINE THE PLOTTING SCALE FOR X.
C (SEE ABOVE.)
C YMIN AND DELTAY, SIMILARLY, DETERMINE THE SCALE FOR Y.
C IF AN ERROR RETURN IS MADE FROM HIDE, ALL FURTHER CALLS
C WILL RESULT ONLY IN THE EXECUTION OF A RETURN UNLESS
C MAXDIM IS RESET TO A POSITIVE VALUE.
C
      DIMENSICN X(1),Y(1),XG(1),G(1),H(1),XH(1),TITLE(1)
      INTEGER TITLE
C
C THE ONLY PURPOSE OF THE FOLLOWING EQUIVALENCE STATEMENT IS
C TO SAVE STORAGE.
      EQUIVALENCE (K1,IWHICH),(K2,SLOPE),(FNSM1,Z1),
     1            (IGGP1,K1),(K1,N2)
C
C EPS1 IS THE RELATIVE ABCISSA INCREMENT USED TO SIMULATE
C DISCONTINUITIES IN THE MAXIMUM FUNCTION.
      DATA EPS1/.000001/
      DATA NOTTLE/6HNOTTLE/
C
C THE FOLLOWING STATEMENT FUNCTION COMPUTES THE ORDINATE ON
C THE LINE JOINING (XI,YI) AND (XIP1,YIP1) CORRESPONDING TO
C THE ABCISSA XX.
      F(XX,XI,YI,XIP1,YIP1) = YI+(XX-XI)*(YIP1-YI)/(XIP1-XI)
      IF(MAXDIM.LE.0) RETURN
      DO 71 I = 2,N1
        IF(X(I-1).LT.X(I)) GO TO 71
        MAXDIM = 0
        GO TC 75
   71 CONTINUE
      IFPLOT = 1
      IF(N1.GT.0) GO TO 76
      N1 = -N1
      IFPLOT = 0
   76 IF(NG.GT.0) GO TO 5000
      IF(N1+4.LE.MAXDIM) GO TO 74
      MAXDIM = -MAXDIM
   75 RETURN
C
C WE WANT SIGN = 1 IF WE ARE PLOTTING MAXIMUM, = -1 IF
C MINIMUM.
   74 SIGN = 1.
      IF(NG.LT.-1) SIGN = -1.
C
C THE KTH CURVE TO BE PLOTTED WILL (OPTIONALLY) BE
C TRANSLATED BY THE VECTOR (-DXIN,DYIN)*(K-1) TO SIMULATE
C STEPPING IN THE DEPTH DIMENSION.
      IF(NFNS.LE.0) GO TO 46
      FNSM1 = NFNS-1
      DXIN = (9.-ABS(XLNTH))*DELTAX/FNSM1
      DYIN = (6.-ABS(YLNTH))*DELTAY/FNSM1
C
C SYSTEMS ROUTINE MOVPEN MOVES THE PEN TO A POINT WHOSE
C COORDINATES ARE SPECIFIED IN INCHES BY THE FIRST TWO
C PARAMETERS.  THE PEN IS PICKED UP IF THE ABSOLUTE VALUE OF
C THE THIRD PARAMETER IS 3, IS PUT DOWN IF 2, AND IS LEFT AS
C AFTER LAST CALL IF 1.  IF THE THIRD PARAMETER IS NEGATIVE,
C A NEW REFERENCE POINT WILL BE ESTABLISHED.
   46 IF(NG.EQ.-1.OR.NG.EQ.-3) GO TO 41
C
C DRAW 8 1/2 BY 11 INCH BORDER.
C ***********************************************************
      CALL MOVPEN(11.,0.,2)
      CALL MOVPEN(11.,8.5,1)
      CALL MOVPEN(0.,8.5,1)
      CALL MOVPEN(0.,0.,1)
      CALL MOVPEN(1.,2.0,-3)
C
C CALL SYSTEMS ROUTINE TO PLOT THE 80-CHARACTER TITLE.
C THE FIRST TWO ARGUMENTS ARE THE COORDINATES IN INCHES
C RELATIVE TO THE REFERENCE POINT OF THE LOWER LEFT-HAND
C CORNER OF THE FIRST CHARACTER.  THE THIRD ARGUMENT
C DETERMINES THE HEIGHT IN INCHES OF THE CHARACTERS.  THE
C FIFTH ARGUMENT GIVES THE ANGLE RELATIVE TO HORIZONTAL OF
C THE PLOTTED CHARACTERS.
C ***********************************************************
   41 IF(TITLE(1).NE.NOTTLE) CALL PSYMB(-.28,-1.,.14,
     1                                  TITLE,0.,80)
      IF(XLNTH.LT.0.) GO TO 42
C
C CALL SYSTEMS ROUTINE TO DRAW THE HORIZONTAL AXIS.  THE
C LEFT END IS SPECIFIED IN INCHES RELATIVE TO THE REFERENCE
C POINT BY THE FIRST TWO ARGUMENTS.
C ***********************************************************
      CALL PAXIS(9.-XLNTH,0.,1H ,-1,XLNTH,0.,XMIN,DELTAX)
      IF(YLNTH.LT.0.) GO TO 43
C
C DRAW THE DEPTH AXIS.
C ***********************************************************
      CALL MOVPEN(9.-XLNTH,0.,3)
      CALL MOVPEN(0.,6.-YLNTH,2)
   42 IF(YLNTH.LT.0.) GO TO 43
C
C DRAW THE VERTICAL AXIS.  THE BOTTOM POINT IS SPECIFIED IN
C INCHES RELATIVE TO THE REFERENCE POINT BY THE FIRST TWO
C ARGUMENTS.
C ***********************************************************
      CALL PAXIS(0.,6.-YLNTH,1H ,1,YLNTH,90.,YMIN,DELTAY)
C
C CURVES SUCCESSIVELY FARTHER IN THE BACKGROUND WILL BE
C PLOTTED WHERE THEY ARE NOT HIDDEN BY G VS. XG.  G VS XG
C WILL BE UPDATED EACH TIME A NEW CURVE IS DRAWN AND WILL BE
C THE VISUAL MAXIMUM (OR MINIMUM) FUNCTION OF THE CURVES
C ALREADY PLOTTED.
   43 INDEXT=3
      DO 3 J = 1,N1
         XG(INDEXT) = X(J)
         G(INDEXT) = SIGN*Y(J)
    3    INDEXT = INDEXT+1
C
C THE FOLLOWING PRECAUTIONARY STEP IS USED IN PLACE OF A
C TEST IN SUBROUTINE LOOKUP TO SEE IF THE VALUE FOR WHICH WE
C WANT AN INDEX IS OUTSIDE THE TABLE.
C THE LAST XG VALUE WILL BE SET EQUAL TO THE LAST ABCISSA
C OF THE CURVE TO BE PLOTTED IN THE NEXT CALL TO HIDE.
      EPS = EPS1*(ABS(XMIN)+ABS(DELTAX))
      NG = N1+4
      XG(1) = -FNSM1*DXIN+XMIN-ABS(XMIN)-ABS(XG(3))-1.
      XG(2) = XG(3)-EPS
      XG(N1+3) = XG(N1+2)+EPS
      ZZ=YMIN
      IF(SIGN.LT.0.) ZZ = -YMIN-50.*DELTAY
      G(1) = ZZ
      G(2) = ZZ
      G(N1+3) = ZZ
      G(NG) = ZZ
C
C CALL SYSTEMS ROUTINE TO PRODUCE A LINE PLOT OF
C (X(I),Y(I),I=1,N1) - THIS IS THE CURVE FARTHEST IN THE
C FOREGROUND.
C XSTART IS THE X VALUE AT THE REFERENCE POINT.
      XSTART = XMIN-(9.-ABS(XLNTH))*DELTAX
C
      IF(IFPLCT.EQ.1) CALL PDATA(X,Y,N1,0,1,XSTART,DELTAX,
     1                                  YMIN,DELTAY,.07)
      DXKK = C.
      DYKK = C.
      RELINC = DELTAX/DELTAY
      XG(NG) = SIGN
      RETURN
C
C STATEMENT 5000 IS REACHED IF ANY EXCEPT THE CURVE FARTHEST
C IN THE FOREGROUND IS TO BE PLOTTED.
 5000 SIGN = XG(NG)
      XG(NG) = X(N1)
C
C TRANSLATE THE ARRAYS BEFORE PLOTTING TO SIMULATE STEPPING
C IN THE DEPTH DIMENSION.
      IF(NFNS) 52,48,49
   49 DXKK = CXKK+DXIN
      DYKK = CYKK+DYIN
   48 DO 4 J = 1,N1
         Y(J) = SIGN*(Y(J)+DYKK)
    4    X(J) = X(J)-DXKK
   52 CALL LOCKUP(X(1),XG(1),JJ)
      IF(JJ.GE.MAXDIM) GO TO 700
      DO 31 J = 1,JJ
         XH(J) = XG(J)
   31    H(J) = G(J)
      IG = JJ+1
      XH(IG) = X(1)
      H(IG) = F(X(1),XG(JJ),G(JJ),XG(IG),G(IG))
C
C WE WILL BE MAKING TABLE LOOKUPS FOR AN INCREASING SEQUENCE
C OF NUMBERS - THEREFORE, WE DO NOT HAVE TO SEARCH FROM THE
C FIRST OF THE (XG AND X) TABLES EACH TIME.  HENCE INDEXG
C AND INDEXT.
      INDEXG = JJ
      INDEXT = 1
      Z1 = X(1)
      F1 = H(IG)-Y(1)
      IT = 2
      JJ = IG
      IF(H(IG).GE.Y(1)) GO TO 32
      IF(JJ.GE.MAXDIM) GO TO 700
      JJ = IG+1
      H(JJ) = Y(1)
      XH(JJ) = Z1+EPS
   32 LAST = C
      X1 = Z1
C
C FIND THE FIRST ZERO, Z2, OF THE FUNCTION G-Y TO THE RIGHT
C OF Z1.
 1100 IF(XG(IG).LT.X(IT)) GO TO 1001
C
C DO NOT JUMP IF WE ARE TO LOOK FOR A ZERO BETWEEN X1 AND
C X(I).
      IWHICH = 0
      X2 = X(IT)
      F2 = F(X2,XG(IG-1),G(IG-1),XG(IG),G(IG))-Y(IT)
      IT = IT+1
      GO TO 1C02
C
C COME TO 1001 IF WE ARE TO LOOK FOR A ZERO BETWEEN X1 AND
C XG(IG).
 1001 X2 = XG(IG)
      IWHICH = 1
      F2 = G(IG)-F(X2,X(IT-1),Y(IT-1),X(IT),Y(IT))
      IG = IG+1
C
C THE FUNCTION (G-Y) HAS A ZERO Z2 SUCH THAT X1 LE Z2 LE X2
C IF AND ONLY IF (G-Y AT X1) * (G-Y AT X2) LE 0.
C (G-Y IS ASSUMED, FOR PLOTTING PURPOSES, TO BE LINEAR ON
C EACH INTERVAL (X1,X2).)
 1002 IF(F1*F2.GT.0.) GO TO 1005
      SLOPE = (F2-F1)/(X2-X1)
      IGG = IG-1-IWHICH
      ITT = IT-2+IWHICH
      IF(ABS(SLOPE*RELINC).GT.1.E-6) GO TO 1007
C
C IF G AND Y CIFFER IMPERCEPTIBLY (FOR PLOTTING PURPOSES)
C ON THE INTERVAL (X1,X2), SET Z2=X2.  THIS STEP PREVENTS
C DIVISION BY ZERO.
      Z2 = X2
      GO TO 1C06
C
C OTHERWISE, COMPUTE THE ZERO Z2.
 1007 Z2 = X1-F1/SLOPE
      GO TO 1C06
C
C IF NO ZERO WAS FOUND BETWEEN X1 AND X2, CONTINUE THE
C SEARCH FOR ZEROES.
 1005 X1 = X2
      F1 = F2
      IF(IT.LE.N1) GO TO 1100
C
C IF THE END CF THE X TABLE HAS BEEN REACHED, CONSIDER THE
C INTERVAL FRCM THE LAST ZERO FOUND TO THE END OF THE X
C TABLE (PLOT, UPDATE MAXIMUM FUNCTION AS INDICATED).
 1008 LAST = 1
      Z2 = X(N1)
      CALL LOCKUP(Z2,XG(INDEXG),IGG)
      IGG = INDEXG+IGG-1
      ITT = N1-1
C
C IT IS NECESSARY TO PLOT Y VS. X ON THE INTERVAL (Z1,Z2)
C ONLY IF Y IS UNHIDDEN AT EACH ZZ SUCH THAT Z1 LT ZZ LT Z2.
C WE CHOOSE ZZ NEAR THE LEFT END OF THE INTERVAL FOR
C EFFICIENCY IN THE TABLE LOOKUP.
C NOTE THAT IT IS MORE EFFICIENT TO CHOOSE THIS VALUE FOR ZZ
C THAN, SAY, .99*X(INDEXT)+.01*X(INDEXT+1), WHICH WOULD
C ELIMINATE ONE OF THE TWO TABLE LOOKUPS, BUT WOULD
C NECESSITATE A TEST TO DETERMINE IF ZZ WAS BETWEEN Z1 AND
C Z2.
 1006 ZZ = .99*Z1+.01*Z2
      CALL LOCKUP(ZZ,X(INDEXT),K1)
      CALL LOCKUP(ZZ,XG(INDEXG),K2)
      K1 = K1+INDEXT-1
      K2 = K2+INDEXG-1
      IF(F(ZZ,X(K1),Y(K1),X(K1+1),Y(K1+1)).GT.
     1   F(ZZ,XG(K2),G(K2),XG(K2+1),G(K2+1))) GO TO 7
C
C IF Y IS HIDCEN BETWEEN Z1 AND Z2, UPDATE THE MAXIMUM
C FUNCTION
```

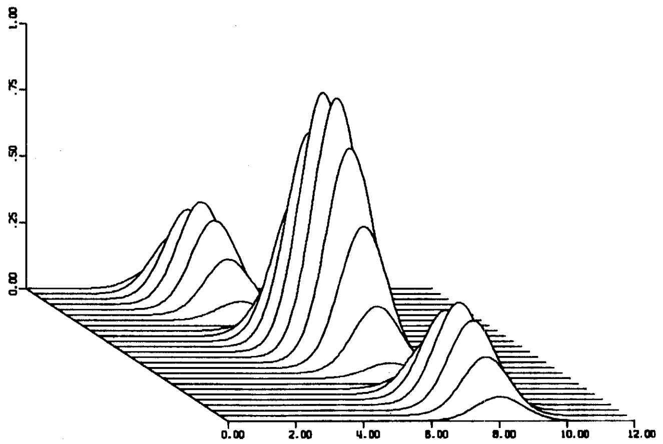
```
C FOR GENERALITY, THE MAXIMUM FUNCTION IS UPDATED EVEN IF
C THIS IS THE (NFNS)TH CURVE.
      IF(JJ+IGG-INDEXG.GE.MAXDIM) GO TO 700
      IF(INDEXG.EQ.IGG) GO TO 712
      J1 = INCEXG+1
      DO 12 I = J1,IGG
         JJ = JJ+1
         XH(JJ) = XG(I)
   12    H(JJ) = G(I)
  712 JJ = JJ+1
      XH(JJ) = Z2
      H(JJ) = F(Z2,XG(IGG),G(IGG),XG(IGG+1),G(IGG+1))
      INDEXG = IGG
      INDEXT = ITT
      GO TO 6C
C
C IF T IS NOT HIDDEN BETWEEN Z1 AND Z2, UPDATE THE MAXIMUM
C FUNCTION ANC PLOT.
    7 NGRAPH = ITT-INDEXT+2
      IF(JJ+NGRAPH-1.GT.MAXDIM) GO TO 700
      N2 = JJ
      IF(NGRAPH.EQ.2) GO TO 9
      J1 = INCEXT+1
      DO 11 I = J1,ITT
         JJ = JJ+1
         XH(JJ) = X(I)
   11    H(JJ) = Y(I)
    9 JJ = JJ+1
      XH(JJ) = Z2
      H(JJ) = F(Z2,X(ITT),Y(ITT),X(ITT+1),Y(ITT+1))
C
C CALL SYSTEMS ROUTINE TO PRODUCE LINE PLOT OF
C (XH(I),H(I),I=N2,N2+NGRAPH-1).
C ***************************************************************
      IF(IFPLCT.EQ.1) CALL PDATA(XH(N2),H(N2),NGRAPH,0,1,
     1                           XSTART,DELTAX,SIGN*YMIN,
     2                           SIGN*DELTAY,.07)
C
      INDEXT = ITT
      INDEXG = IGG
   60 IF(LAST.EQ.1) GO TO 61
      X1 = X2
      F1 = F2
      Z1 = Z2
C
C AFTER PLOTTING AND/UR UPDATING THE MAXIMUM FUNCTION ON THE
C INTERVAL (Z1,Z2), SEARCH FOR THE NEXT ZERO IF THE END OF
C THE ABCISSA TABLE XT HAS NOT BEEN REACHED.
      IF(IT.LE.N1) GO TO 1100
      GO TO 1C08
C
C AFTER Y VS. X HAS BEEN PLOTTED, FINISH UPDATING AND STORE
C THE NEW MAXIMUM FUNCTION.
C ALLOW FUR THE POSSIBILITY THAT THE PREVIOUS MAXIMUM
C FUNCTION EXTENDS TO THE RIGHT OF THE FUNCTION JUST
C PLOTTED.
   61 IF(XG(NG).LE.XG(NG-1)) NG = NG-1
      IF(XG(NG).LE.X(N1)) GO TO 33
      IF(JJ+3+NG-IGG.GT.MAXDIM) GO TO 700
      XH(JJ+1) = XH(JJ)+EPS
      JJ = JJ+1
      H(JJ) = F(X(N1),XG(IGG),G(IGG),XG(IGG+1),G(IGG+1))
      IGGP1 = IGG+1
      DO 34 J = IGGP1,NG
         JJ = JJ+1
         XH(JJ) = XG(J)
   34    H(JJ) = G(J)
   33 NG = JJ+2
      IF(NG.GT.MAXDIM) GO TO 700
      DO 13 I = 1,JJ
         G(I) = H(I)
   13    XG(I) = XH(I)
C
C THE FOLLOWING PRECAUTIONARY STEP IS USED IN PLACE OF A
C TEST IN SUBROUTINE LOOKUP TO SEE IF THE VALUE FOR WHICH WE
C WANT AN INDEX IS OUTSIDE THE TABLE.
C THE LAST XG VALUE WILL BE SET EQUAL TO THE LAST ABCISSA
C OF THE NEXT CURVE TO BE PLOTTED.
      XG(JJ+1) = XG(JJ)+EPS
      G(JJ+1) = YMIN+DYKK
      IF(SIGN.LT.0.) G(JJ+1) = -YMIN-50.*DELTAY+DYKK
      G(NG) = G(JJ+1)
C
C RESTORE ARRAYS X ANC Y BEFORE RETURNING.
   66 IF(NFNS.LT.0) GU TO 53
      DO 82 I = 1,N1
         X(I) = X(I)+CXKK
   82    Y(I) = SIGN*Y(I)-DYKK
   53 XG(NG) = SIGN
      RETURN
C
C IF STATEMENT 700 IS REACHED, DIMENSIONS WOULD HAVE BEEN
C EXCEEDED.  SEE COMMENTS ON CALLING SEQUENCE FOR HIDE.
  700 MAXDIM = -MAXDIM
      GO TO 66
      END
      SUBROUTINE LOOKUP(X,XTBL,J)
C
C THIS SUBROUTINE IS CALLED BY HIDE TO PERFORM A TABLE
C LOOKUP.  BECAUSE OF PRECAUTIONS TAKEN IN HIDE, A TEST TO
C SEE IF X IS OUTSIDE THE TABLE IS UNNECESSARY.
C
      DIMENSICN XTBL(1)
      J = 2
    4 IF(XTBL(J)-X) 1,2,3
    1 J = J+1
      GO TO 4
    2 RETURN
    3 J = J-1
      RETURN
      END
```

Driver

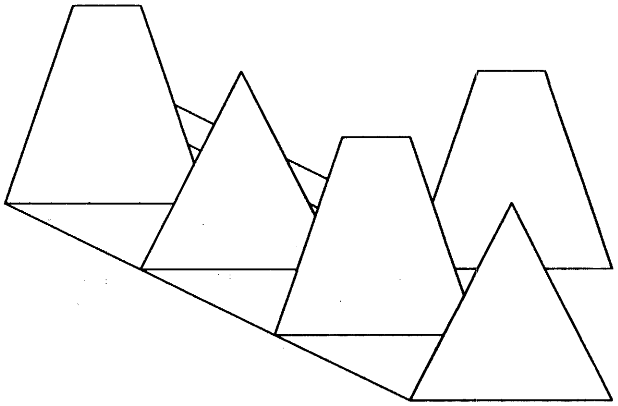
(a) Test for plotting routine *HIDE*.



(b) Test case for *HIDE*.



(c) Geometrical test case.



Remark on Algorithm 420 [J6]

Hidden-Line Plotting Program [H. Williamson, *Comm. ACM 15* (Feb. 1972), 100–103]

I.D.G. Macleod and A.M. Collins [Recd. 19 June 1972] Department of Engineering Physics, Research School of Physical Sciences, Australian National University, Canberra, A.C.T. 2600, Australia

The number of point pairs to be plotted in subroutine *HIDE* is indicated by the magnitude of parameter *N*1. If *N*1 is less than zero, the visual maximum function is updated but no plotting is carried out. In this case, however, the construction

```
      DO 71 I = 2, N1
      IF (X(I-1) .LT. X(I)) GO TO 71
      MAXDIM = 0
      GO TO 75
71    CONTINUE
```

is nonstandard and may lead to undesirable results. If the check for increasing *X* values is to be retained when *N*1 is negative, its absolute value should be used as the terminal value of the *DO* loop.

In sections 8.3.2 and 10.1.3, ANSI Fortran [1] indicates that where *X* is an array there should be no distinction between the use of *X* and the use of *X* [1] as parameters in a procedure reference. Nevertheless, some Fortran implementations (and languages such as Algol and PL/I) make such a distinction, in which case subroutine *LOOKUP* and the calls to it would have to be appropriately modified.

References

1. American National Standards Institute: Fortran. Publication X3.9-1966.

Fig. 1. Without verticals.

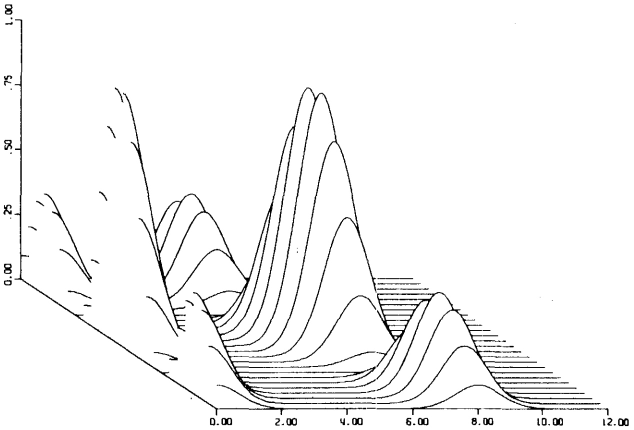
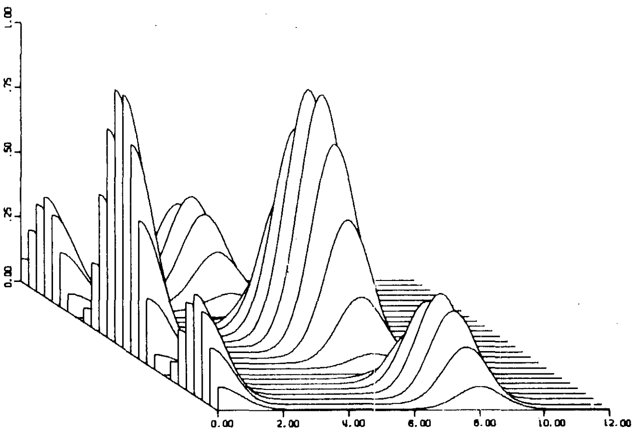


Fig. 2. With verticals to aid visualization.



Remark on Algorithm 420 [J6]

Hidden-Line Plotting Program
[Hugh Williamson, *Comm. ACM 15* (Feb. 1972), 100–103]

Hugh Williamson [Recd. 9 Oct. 1972]
National Con-Serv, Incorporated, Austin, Texas

The input quantities to subroutine *HIDE* referred to in the following paragraphs (e.g. *N*1, *NFNS*, "input curve to be plotted") are described in the block of comment statements at the beginning of *HIDE* as originally published.

If *N*1 < 0, *DO* loop 71 is not executed properly, since the upper limit, *N*1, is less than the lower limit, 2. This affects only checking for monotonicity in the input abscissa array; otherwise, if the inputs are correct, the performance of the program is not affected.

The error is corrected if the first 11 executable statements are replaced by the following (the first executable statement of the original program, which is not changed, is listed for clarity):

```
      IF(MAXDIM.LE.0) RETURN
      IFPLOT = 1
      IF(N1.GT.0) GO TO 76
      N1 = -N1
      IFPLOT = 0
76    DO 71 I = 2,N1
         IF(X(I-1).LT.X(I)) GO TO 71
         MAXDIM = 0
         GO TO 75
71    CONTINUE
         IF(NG.GT.0) GO TO 5000
```

On computers in which all variables are not automatically set to zero before execution, *FNSM*1 is not properly initialized if *NFNS* ≤ 0. To correct this, simply insert the statement

```
FNSM1 = 0.
```

before the statement

```
IF(NFNS.LE.0) GO TO 46
```

The latter is the sixth statement after Fortran statement number 74.

*FNSM*1 will still be improperly defined if *NFNS* = 1. If only one curve is to be plotted, however, translating to simulate stepping in the depth dimension will not be done, so set *NFNS* = −1 for only one curve to be plotted.

In some cases, the three-dimensional surface is easier to visualize if (nearly) vertical lines are drawn at the left edge of each curve; this effect is illustrated by Figures 1 and 2. The verticals are added by inserting (*XMIN*-ϵ, *YMIN*) as the first point in each input curve to be plotted, where ϵ is a small positive number ($10^{-4} \times DELTAX$ would be appropriate).

The author appreciates very much the comments received from readers of Communications regarding implementation of *HIDE* on different computers.

Remark on Algorithm 420 [J6]

Hidden-Line Plotting Program [Hugh Williamson, *Comm. ACM 15* (Feb. 1972), 100–103].

Blaine Gaither [Recd. 3 Apr. 1973]
New Mexico Institute of Mining and Technology (TERA), Socorro, NM 87801

The algorithm was compiled and run without corrections on an IBM 360 G44. It has been in use for a year now with no problems. However, there is danger of division by zero if *NFNS* equals 1. To eliminate this danger the statement:

```
IF(NFNS.EQ.1) NFNS = -1
```

should be inserted between the statements:

```
IF(NG.LT.-1) SIGN = -1
IF(NFNS.LE.0) GO TO 46
```

Depth axis may be added by the following changes. Where *ZMIN* and *ZMAX* are the values for the nearest and farthest curves respectively, replace the continuation card of *HIDE*'s subroutine statement with:

```
1       XLNTH,YLNTH,XMIN,DELTAX,YMIN,DELTAY,
        ZMIN,ZMAX)
```

In place of the statement labeled 42 insert:

```
  42 DELZ = ZMAX - ZMIN
     IF (DELZ) 9601, 9602, 9601
9601 XSC = XLNTH -·9.
     YSC = 6. - YLNTH
     IF (XSC) 9604, 9603, 9604
9603 ANGZ = 90.
     GO TO 9605
9604 ANGZ = ATAN(YSC/XSC)*57.29578
9605 ZLEN = SQRT(XSC*XSC+YSC*YSC)
     IF (ZLEN-1.) 9602, 9602, 9606
9606 CALL PAXIS (0.,YSC,1H,-1,ZLEN,ANGZ,ZMAX,
     -DELZ/ZLEN)
9602 IF (YLNTH.LT.0.) GO TO 43
```

If *ZMIN* equals *ZMAX* or if the length of the depth axis would be less than or equal to 1., these changes will have no effect. The max and min numbers on the depth axis may overlap with those of the horizontal and vertical axis.

Remark on Algorithm 420 [J6]

Hidden-Line Plotting Program [Hugh Williamson, *Comm. ACM 15* (Feb. 1972) 100–103.]

T.M.R. Ellis [Recd. 26 Mar. 1973 and 30 July 1973]
Computing Services, University of Sheffield, England

Algorithm 420 has been implemented on an ICL 1907 computer and used to plot the surface entitled "Test for Plotting Routine Hide" as well as a number of other surfaces. The system plotting routines for the ICL 1900 series computers more or less duplicate those used by Williamson, except in the case of *PDATA* for which no equivalent routine exists. There is however a system routine which draws a smooth curve through a set of points, and only slight modifications were required to reproduce the exact effect of *PDATA*.

The implementation was checked by the satisfactory reproduction of the "Test for Plotting Routine Hide," and subsequently it produced good representations of other surfaces. However, when attempting to plot a square-based pyramid, the program failed due to an error in *HIDE*.

When *HIDE* is searching for points at which the current line appears and disappears, it searches for the zeros of a function $(G-Y)$ where G is the current visual maximum (i.e. as already drawn) and Y is the current ordinate (as to be drawn). This search is carried out by comparing the values of the function $(G-Y)$ at adjacent points in the current line (Y) and/or the current visual maximum (G), as shown in Figure 1.

Fig. 1.

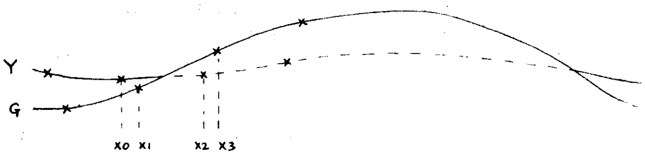


Fig. 2.

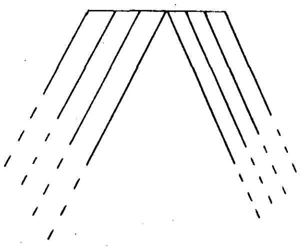


Fig. 3.

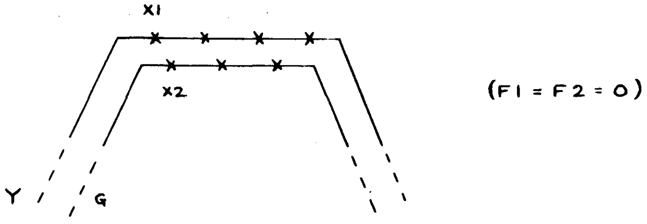
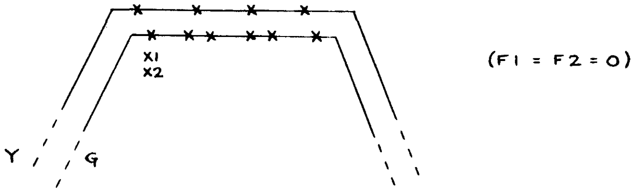


Fig. 4.



Due to the fact that each line drawn is shifted upward and to the left, in order to simulate perspective, data points on successive lines which in the actual surface would have the same abscissa will have different abscissa in the drawing. Thus $X0$ and $X1$ might represent the same value of the abscissa in the surface. At $X0$ and $X1$ in the above drawing the function $(G-Y)$ has a negative value, while at $X2$ and $X3$ it is positive. Clearly if $F1$ and $F2$ are the values of $(G-Y)$ at $X1$ and $X2$ there is a zero between $X1$ and $X2$ if and only if $F1$ and $F2$ have opposite signs. This is tested for by the statement:

```
1002 IF(F1*F2.GT.0.) GO TO 1005
```

If a zero is found to exist, its abscissa is calculated by linear interpolation, the slope of the line being determined by the next statement:

```
SLOPE = (F2-F1)/(X2-X1)
```

A check is subsequently made to avoid dividing by zero if *SLOPE* is too small.

In the case of the square based pyramid referred to above, the projection used was such that it was viewed down its rear face, and therefore all lines traversing the far face of the pyramid were both parallel to one another and passed through the same point on the graph (the peak of the pyramid). Thus for a part of their length all the lines after that which goes over the peak are drawn on top of each other, as shown in Figure 2. When plotting the second of these coincident lines the respective *G* and *Y* functions are therefore as shown in the exploded form in Figure 3.

This clearly means that for a number of consecutive abscissa values both *F*1 and *F*2 are zero. Due to the way in which *HIDE* keeps track of its path along the two functions *G* and *Y*, the effect of both *F*1 and *F*2 being zero is for the abscissa (*X*1) corresponding to the first of the two "zeros" to be entered in the visual maximum array for a second time. During the plotting of the next line therefore, the visual maximum function *G* vs. *XG* has two identical entries, and thus the stage comes when *X*1 corresponds to the first, and *X*2 to the second (see Figure 4).

If, as in this case, this (third) line would be coincident with the second (and the first) at this point, then $F1 = F2 = 0$ and the test at 1002 (above) will lead to the calculation of *SLOPE*, and hence failure due to the division by zero $(X2-X1)$.

The problem can, however, be very easily corrected by inserting the following statement immediately after the statement with label 1002:

```
IF(F1.EQ.FZ) GO TO 1005
```

Since this statement can only be reached if $F1*F2$ is less than or equal to zero, then clearly the jump will be made if and only if $F1 = F2 = 0$. In this case the second "zero" is ignored, and the program proceeds satisfactorily.

Remark on Algorithm 420 [J6]
Hidden-Line Plotting Program [Hugh Williamson, Comm. ACM 15 (Feb. 1972) 100–103] and Remark on Algorithm 420 [T.M.R. Ellis, *Comm. ACM 17* (June 1974), 324–325]

T.M.R. Ellis [Recd. 8 July 1974] Computing Services, University of Sheffield, England

There was an unfortunate printing error in my Remark on Algorithm 420 which made nonsense of the whole thing. The statement which should be inserted to correct the error discussed should, of course, be:

```
IF(F1.EQ.F2) GO TO 1005
```

and not: IF(F1.EQ.FZ) GO TO 1005 as printed.

Algorithm 421
Complex Gamma Function with Error Control [S14]

Hirondo Kuki* (Recd. 17 Aug. 1970 and 21 June 1971)
Computation Center, The University of Chicago,
Chicago, Illinois

Key words and phrases: complex gamma function, gamma function, complex loggamma function, loggamma function, round-off error control, inherent error control, run-time error estimates, error estimates, special functions

CR Categories: 4.9, 5.11, 5.12

Description

This Fortran program computes either the gamma function or the loggamma function of a complex variable in double precision. In addition, it provides an error estimate of the computed answer. The calling sequences are:
CALL CDLGAM (*Z*, *W*, *E*, 0) for the loggamma, and
CALL CDLGAM (*Z*, *W*, *E*, 1) for the gamma,
where *Z* is the double precision complex argument, *W* is the answer of the same type, and *E* is a single precision real variable. Before the call, the value of *E* is an estimate of the erroı in *Z*, and after the call, it is an estimate of the error in *W*.

For details of the characteristics of the program, an analysis of the algorithm, and the nature of the error estimate, see [1].

This program was tested on an IBM System 360 Model 65. A slightly modified version was used for this purpose to take advantage of the availability of such facilities as the *ENTRY* statement and functions of the type double precision complex. Compiled by OS/FORTRAN-H, opt. 2, it required 3188 bytes of storage. Performance statistics on samples of 500 arguments each, from seven disjoint regions within the square

$\{z = z_1 + iz_2 \,;\, |z_1|, |z_2| < 30\}$,
were as follows:

Region I	$0 \leq z_1$ and $\lvert z \rvert < 3$
Region II	$0 \leq z_1$ and $3 \leq \lvert z \rvert < 10$
Region III	$0 \leq z_1 < 10$, $-30 < z_2 < 30$, and $10 \leq \lvert z \rvert$
Region IV	$10 \leq z_1 < 30$, $-10 < z_2 < 10$
Region V	$10 \leq z_1 < 30$, $10 \leq \lvert z_2 \rvert < 30$
Region VI	$-30 < z_1 < 0$, $-1 < z_2 < 1$
Region VII	$-30 < z_1 < 0$, $1 \leq \lvert z_2 \rvert < 30$

	*time**			
Region	*log* Γ	Γ	*Max error***	*RMS error***
I	2100	2470	2.3×10^{-15}	8.7×10^{-16}
II	1800	2230	7.6×10^{-15}	2.8×10^{-15}
III	1700	1930	1.6×10^{-14}	7.8×10^{-15}
IV	920	1500	1.4×10^{-14}	7.1×10^{-15}
V	1000	1500	1.6×10^{-14}	7.9×10^{-15}
VI	2130	2330	2.6×10^{-14}	7.9×10^{-15}
VII	1900	2050	2.4×10^{-14}	9.5×10^{-15}

* Average time in μs.
** Generated absolute errors for computation of the loggamma function.

Essentially the same statistics were obtained as generated relative errors for computation of gamma function. Statistics on the effectiveness of the error estimate are found in [1].

References

1. Kuki, H. Complex gamma function with error control. *Comm. ACM 15* (Apr. 1972), 262–267.

Algorithm

```
      SUBROUTINE  CDLGAM(CARG,CANS,ERRØR,LFO)
C COMPLEX GAMMA AND LØGGAMMA FUNCTIØNS WITH ERRØR ESTIMATE
C
C CARG = A CØMPLEX ARGUMENT, GIVEN AS A VECTØR ØF 2 DØUBLE
C        PRECISIØN ELEMENTS CØNSISTING ØF THE REAL CØMPØNENT
C        FØLLØWED BY THE IMAGINARY CØMPØNENT
C CANS = THE CØMPLEX ANSWER, ØF THE SAME TYPE AS CARG
C ERRØR = A REAL VARIABLE.  IT STANDS FØR AN ESTIMATE ØF THE
C        ABSØLUTE ERRØR ØF THE ARGUMENT AS INPUT.  AS ØUTPUT
C        IT GIVES AN ESTIMATE ØF THE ABSØLUTE (FØR LØGGAMMA)
C        ØR THE RELATIVE (FØR GAMMA) ERRØR ØF THE ANSWER
C LFO  = FLAG.  SET IT TØ 0 FØR LØGGAMMA, AND 1 FØR GAMMA
      DØUBLE PRECISIØN CARG(2),CANS(2),CØEF(7),F0,F1,G0,G1,
     $    PI,DPI,HL2P,AL2P,DELTA,DE0,DE1,Z1,Z2,ZZ1,W1,W2,Y1,
     $    A,B,U,U1,U2,UU1,UU2,UUU1,UUU2,V1,V2,VV1,VV2,T1,T2,
     $    H,H1,H2,AL1,AL2,DN,EPS,ØMEGA
      DATA CØEF(1)/+0.64102564102564102 6D-2/
      DATA CØEF(2)/-0.19175269175269175 3D-2/
      DATA CØEF(3)/+0.84175084175084175 1D-3/
      DATA CØEF(4)/-0.59523809523809523 8D-3/
      DATA CØEF(5)/+0.79365079365079365 1D-3/
      DATA CØEF(6)/-0.27777777777777777 8D-2/
      DATA CØEF(7)/+0.83333333333333333 3D-1/
      DATA F0/840.07385296052619D0/,F1/20.0012308218942000D0/
      DATA G0/1680.1477059210524D0/,G1/180.0147704705204 2D0/
      DATA PI/3.1415926535897932 4D0/
      DATA DPI/6.2831853071795864 8D0/
      DATA HL2P/0.9189385332046727 42D0/
      DATA AL2P/1.8378770664093454 8D0/
C CØNSTANTS EPS AND ØMEGA ARE MACHINE DEPENDENT.
C EPS   IS THE BASIC RØUND-ØFF UNIT.  FØR S/360 MACHINES,
C IT IS CHØSEN TØ BE 16**-13.  FØR BINARY MACHINE ØF N-
C BIT ACCURACY SET IT TØ 2**(-N+1), AND INITIALIZE DEØ
C AS 5.0 RATHER THAN AS 2.0
C ØMEGA   IS THE LARGEST NUMBER REPRESENTABLE BY THE FLØAT
C PØINT REPRESENTATIØN ØF THE MACHINE.  FØR S/360
C      MACHINES, IT IS SLIGHTLY LESS THAN 16**63.
      DATA EPS/2.2D-16/
      DATA ØMEGA/7.23700538D75/
      Z1 = CARG(1)
      Z2 = CARG(2)
      DELTA = ABS(ERRØR)
      DE0 = 2.0D0
      DE1 = 0.0
```

* Deceased.

```
C FØRCE SIGN ØF IMAGINARY PART ØF ARG TØ NØN-NEGATIVE
      LF1 = 0
      IF (Z2 .GE. 0.0) GØ TØ 20
      LF1 = 1
      Z2 = -Z2
   20 LF2 = 0
      IF (Z1 .GE. 0.0) GØ TØ 100
C CASE WHEN REAL PART ØF ARG IS NEGATIVE
      LF2 = 1
      LF1 = LF1-1
      T1 = AL2P - PI*Z2
      T2 = PI*(0.5D0 - Z1)
      U  = -DPI*Z2
      IF (U .GE. -0.1054D0) GØ TØ 40
      A  = 0.0D0
C IF E**U .LT. 10**(-17), IGNØRE IT TØ SAVE TIME AND TØ AVØID
C IRRELEVANT UNDERFLØW
      IF (U .LE. -39.15D0) GØ TØ 30
      A  = DEXP(U)
   30 H1 = 1.0D0 - A
      GØ TØ 50
   40 U2 = U*U
      A  = -U*(F1*U2 + F0)
      H1 = (A + A)/((U2 + G1)*U2 + G0 + A)
      A  = 1.0D0 - H1
C DINT IS THE DØUBLE PRECISIØN VERSIØN ØF AINT, INTEGER EX-
C   TRACTIØN.  THIS FUNCTIØN IS NØT INCLUDED IN ANSI FØRTRAN
C   .  WHEN THIS FUNCTIØN IS NØT PRØVIDED BY THE SYSTEM,
C   EITHER SUPPLY IT AS AN EXTERNAL SUBRØUTINE (AND TYPE THE
C   NAME DINT AS DØUBLE PRECISIØN), ØR MØDIFY THE NEXT
C   STATEMENT AS THE EXAMPLE FØR S/360 INDICATES.  FØR S/360
C   REPLACE IT WITH
C
C     DØUBLE PRECISIØN SCALE
C     DATA SCALE/Z4F00000000000000/
C  50 B  = Z1 - ((Z1 - 0.5D0) + SCALE)
   50 B  = Z1 - DINT(Z1 - 0.5D0)
      H2 = A*DSIN(DPI*B)
      B  = DSIN(PI*B)
      H1 = H1 + (B+B)*B*A
      H  = DABS(H2) + H1 - DPI*A*DELTA
      IF (H .LE. 0.0) GØ TØ 500
      DE0 = DE0 + DABS(T1) + T2
      DE1 = PI + DPI*A/H
      Z1 = 1.0D0 - Z1
C CASE WHEN NEITHER REAL PART NØR IMAGINARY PART ØF ARG IS
C NEGATIVE.  DEFINE THRESHØLD CURVE TØ BE THE BRØKEN LINES
C CØNNECTING PØINTS 10F0*I, 10F4.142*I, 0.1F14.042*I,AND
C 0.1FØMEGA*I
  100 LF3 = 0
      Y1 = Z1 - 0.5D0
      W1 = 0.0
      W2 = 0.0
      K  = 0
      B  = DMAX1(0.1D0, DMIN1(10.0D0, 14.142D0-Z2)) - Z1
      IF (B .LE. 0.0) GØ TØ 200
C CASE WHEN REAL PART ØF ARG IS BETWEEN 0 AND THRESHØLD
      LF3 = 1
      ZZ1 = Z1
      N  = B + 1.0D0
      DN = N
      Z1 = Z1 + DN
      A  = Z1*Z1 + Z2*Z2
      V1 = Z1/A
      V2 = -Z2/A
C INITIALIZE U1+U2*I AS THE RIGHTMØST FACTØR 1-1/(Z+N)
      U1 = 1.0D0 - V1
      U2 = -V2
      K  = 6.0D0 - Z2*0.6D0 - ZZ1
      IF (K .LE. 0) GØ TØ 120
C FØRWARD ASSEMBLY ØF FACTØRS (Z+J-1)/(Z+N)
      N  = N - K
      UU1 = (ZZ1*Z1 + Z2*Z2) / A
      UU2 = DN*Z2/A
      VV1 = 0.0
      VV2 = 0.0
      DØ 110  J = 1,K
        B  = U1*(UU1+VV1) - U2*(UU2+VV2)
        U2 = U1*(UU2+VV2) + U2*(UU1+VV1)
        U1 = B
        VV1 = VV1 + V1
  110   VV2 = VV2 + V2
  120 IF (N .LE. 1) GØ TØ 140
C BACKWARD ASSEMBLY ØF FACTØRS 1-J/(Z+N)
      VV1 = V1
      VV2 = V2
      DØ 130  J = 2,N
        VV1 = VV1 + V1
        VV2 = VV2 + V2
        B  = U1*(1.0D0 - VV1) + U2*VV2
        U2 = -U1*VV2 + U2*(1.0D0 - VV1)
  130   U1 = B
  140 U  = U1*U1 + U2*U2
      IF (U .EQ. 0.0) GØ TØ 500
      IF (LF0 .EQ. 0) GØ TØ 150
      IF (K .LE. 0) GØ TØ 200
  150 AL1 = DLØG(U)*0.5D0
      IF (LF0 .NE. 0) GØ TØ 160
      W1 = AL1
      W2 = DATAN2(U2,U1)
      IF (W2 .LT. 0.0)  W2 = W2 + DPI
      IF (K .LE. 0) GØ TØ 200
  160 A  = ZZ1 + Z2 - DELTA
      IF (A .LE. 0.0) GØ TØ 500
      DE0 = DE0 - AL1
      DE1 = DE1 + 2.0D0 + 1.0D0/A
C CASE WHEN REAL PART ØF ARG IS GREATER THAN THRESHØLD
  200 A  = Z1*Z1 + Z2*Z2
      AL1 = DLØG(A)*0.5D0
      AL2 = DATAN2(Z2,Z1)
      V1 = Y1*AL1 - Z2*AL2
      V2 = Y1*AL2 + Z2*AL1
C EVALUATE ASYMPTØTIC TERMS.  IGNØRE THIS TERM, IF ABS VAL(ARG) .GT.
```

```
C 10**9, TØ SAVE TIME AND TØ AVØID IRRELEVANT UNDERFLØW
      VV1 = 0.0
      VV2 = 0.0
      IF (A .GT. 1.0D18) GØ TØ 220
      UU1 = Z1/A
      UU2 = -Z2/A
      UUU1 = UU1*UU1 - UU2*UU2
      UUU2 = UU1*UU2*2.0D0
      VV1 = CØEF(1)
      DØ 210  J = 2,7
        B  = VV1*UUU1 - VV2*UUU2
        VV2 = VV1*UUU2 + VV2*UUU1
  210   VV1 = B + CØEF(J)
      B  = VV1*UU1 - VV2*UU2
      VV2 = VV1*UU2 + VV2*UU1
      VV1 = B
  220 W1 = (((VV1 + HL2P) - W1) - Z1) + V1
      W2 = ((VV2 - W2) - Z2) + V2
      DE0 = DE0 + DABS(V1) + DABS(V2)
      IF (K .LE. 0)  DE1 = DE1 + AL1
C FINAL ASSEMBLY
      IF (LF2 .NE. 0) GØ TØ 310
      IF (LF0 .EQ. 0) GØ TØ 400
      A  = DEXP(W1)
      W1 = A*DCØS(W2)
      W2 = A*DSIN(W2)
      IF (LF3 .EQ. 0) GØ TØ 400
      B  = (W1*U1 + W2*U2) / U
      W2 = (W2*U1 - W1*U2) / U
      W1 = B
      GØ TØ 400
  310 H  = H1*H1 + H2*H2
      IF (H .EQ. 0.0) GØ TØ 500
      IF (LF0 .EQ. 0) GØ TØ 320
      IF (H .GT. 1.0D-2) GØ TØ 330
  320 A  = DLØG(H)*0.5D0
      IF (H .LE. 1.0D-2)  DE0 = DE0 - A
      IF (LF0 .NE. 0) GØ TØ 330
      W1 = (T1 - A) - W1
      W2 = (T2 - DATAN2(H2,H1)) - W2
      GØ TØ 400
  330 T1 = T1 - W1
      T2 = T2 - W2
      A  = DEXP(T1)
      T1 = A*DCØS(T2)
      T2 = A*DSIN(T2)
      W1 = (T1*H1 + T2*H2)/H
      W2 = (T2*H1 - T1*H2)/H
      IF (LF3 .EQ. 0) GØ TØ 400
      B  = W1*U1 - W2*U2
      W2 = W1*U2 + W2*U1
      W1 = B
  400 IF (LF1 .NE. 0)  W2 = -W2
C TRUNCATIØN ERRØR ØF STIRLINGS FØRMULA IS UP TØ 3*10**-17.
      DE1 = DE0*EPS + 3.0D-17 + DE1*DELTA
      GØ TØ 600
C CASE WHEN ARGUMENT IS TØØ CLØSE TØ A SINGULARITY
C
  500 W1 = ØMEGA
      W2 = ØMEGA
      DE1 = ØMEGA
C
  600 CANS(1) = W1
      CANS(2) = W2
      ERRØR = DE1
      RETURN
      END
```

Algorithm 422
Minimal Spanning Tree [H]

V. Kevin M. Whitney (Recd. 4 May 1970, 13 Oct. 1970, and 3 Aug. 1971)
Department of Electrical Engineering, University of Michigan, Ann Arbor, MI 48104

Key Words and Phrases: spanning tree, minimal spanning tree, maximal spanning tree
CR Category: 5.32

Description

This algorithm generates a spanning tree of minimal total edge length for an undirected graph specified by an array of inter-node edge lengths using a technique suggested by Dijkstra [1]. Execution time is proportional to the square of the number of nodes of the graph; a minimal spanning tree for a graph of 50 nodes is generated in 0.1 seconds on an IBM System 360/67. Previous algorithms [2, 3, 4, 5] require an amount of computation which depends on the graph topology and edge lengths and are best suited to graphs with few edges.

The nodes of the graph are assumed to be numbered from 1 to N. The length of an edge from node I to node J is given by array element $DM(I, J)$. If there is no edge from node I to node J, $DM(I, J)$ is given a value larger than the length of the longest edge of the graph, say 10^{10}. The diagonal elements of array DM are not used and may have any value. After execution of the algorithm, the edges of a minimal spanning tree are specified by pairs of nodes in array MST and the total edge length is given by CST.

The Dijkstra algorithm grows a minimal spanning tree by successively adjoining the nearest remaining node to a partially formed tree until all nodes of the graph are included in the tree. At each iterative step the nodes not yet included in the tree are stored in array NIT. The node of the partially completed tree nearest to node $NIT(I)$ is stored in $JI(I)$, and the length of edge from $NIT(I)$ to $JI(I)$ is stored in $UI(I)$. Hence the node not yet in the tree which is nearest to a node of the tree may be found by searching for the minimal element of array UI. That node, KP, is added to the tree and removed from array NIT. For each node remaining in array NIT, the distance from the nearest node of the tree (stored in array UI) is compared to the distance from KP, the new node of the tree, and arrays UI and JI are updated if the new distance is shorter. The nearest node selection and list updating are performed $N - 1$ times until all nodes are in the tree. A proof that this algorithm finds a minimal spanning tree and a discussion of the related shortest path tree algorithm will be found in either of references [1] or [6].

Most of the execution time for this algorithm is spent in the search and updating statements between statements labeled 200 and 500 which are executed $N - 1$ times. Since on the Kth execution a list of N–K items is searched and updated, the total execution time is proportional to N^2.

If the graph represented by the inter-node edge length array DM is not connected, the procedure will generate a minimal spanning forest containing the minimal spanning trees of the disjoint components joined together by edges of length 10^{10}. A disconnected graph is indicated by a value of 10^{10} for variable UK at step 500 during execution of the algorithm.

The algorithm may be simply modified to find a spanning tree of maximal total length by changing the loop between statements 300 and 400 to search for the most distant rather than for the nearest remaining node to be adjoined to the partially completed tree.

The data storage required for the algorithm may be reduced from $N(N + 5)$ locations to $5N$ locations by replacing array DM with an edge length function which calculates the required inter-node edge lengths as they are needed. Such a function will be called $N(N - 1)/2$ times and may extend considerably the size of problem which can be solved by this algorithm on a machine with limited core storage.

Acknowledgment is due E.L. Lawler for bringing reference [1] to my attention. This work was supported in part by Rome Air Development Center Contract F30602-69-C-0214 with the Systems Engineering Laboratory of the University of Michigan.

References

1. Dijkstra, E.W. A note on two problems in connection with graphs. *Numer. Math. 1*, 5 (Oct. 1959), 269–271.
2. Kruskal, J.B. Jr. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc. 7* (1956), 48–50.
3. Prim, R.C. Shortest connection networks and some generalizations. *Bell Syst. Tech. J. 36* (Nov. 1957), 1389–1401.
4. Obruca, A. Algorithm 1. MINTREE. *Comput. Bull.* (Sept. 1964), 67.
5. Loberman, H., and Weinberger, A. Formal procedures for connecting terminals with a minimum total wire length. *J. ACM 4*, 4 (Oct. 1957), 428–437.
6. Lawler, E.L. *An Introduction to Combinatorial Optimization Theory and Its Applications*, 2 vols. Holt, Rinehart & Winston, New York, 1971.

Algorithm

```
      SUBRØUTINE DMTØMS(DM,N,MST,IMST,CST)
C
C     THIS SUBRØUTINE FINDS A SET ØF EDGES ØF A LINEAR GRAPH
C     CØMPRISING A TREE WITH MINIMAL TØTAL EDGE LENGTH. THE
C     GRAPH IS SPECIFIED AS AN ARRAY OF INTER-NØDE EDGE LENGTHS.
C     THE EDGES ØF THE MINIMAL SPANNING TREE ØF THE GRAPH ARE
C     PLACED IN ARRAY MST.  EXECUTIØN TIME IS PRØPØRTIØNAL TØ
C     THE SQUARE ØF THE NUMBER ØF MØDES.
C
C     CALLING SEQUENCE VARIABLES ARE:
C
C     DM     ARRAY ØF INTER-NØDE EDGE LENGTHS.
C            DM(I,J) (1 .LE. I,J .LE. IN) IS THE LENGTH ØF
C            AN EDGE FRØM NØDE I TØ NØDE J.  IF THERE IS NØ
C            EDGE FRØM NØDE I TØ NØDE J, SET DM(I,J)=10.**10
C     N      NØDES ARE NUMBERED  1, 2, ..., N.
C
C     MST    ARRAY IN WHICH EDGE LIST ØF MST IS PLACED. MST(1,I)
C            IS THE ØRIGINAL NØDE AND MST(2,I) IS THE TERMINAL
C            NØDE ØF EDGE I FØR 1 .LE. I .LE. IMST.
C     IMST   NUMBER ØF EDGES IN ARRAY MST.
C     CST    SUM ØF EDGE LENGTHS ØF EDGES ØF TREE.
C
C     PRØGRAM VARIABLES    :
C
C     NIT    ARRAY ØF NODES NØT YET IN TREE.
C     NITP   NUMBER ØF NØDES IN ARRAY NIT.
C     JI(I)  NØDE ØF PARTIAL MST CLØSEST TØ NØDE NIT(I).
C     UI(I)  LENGTH ØF EDGE FRØM NIT(I) TØ JI(I).
C     KP     NEXT NØDE TØ BE ADDED TØ ARRAY MST.
```

```
C
      DIMENSIØN DM(50,50),MST(2,50)
      DIMENSIØN UI(50),JI(50),NIT(50)
C
C     INITIALIZE NØDE LABEL ARRAYS
C
      CST=0.
      NITP=N-1
      KP=N
      IMST=0
      DØ 100 I=1,NITP
         NIT(I)=I
         UI(I)=DM(I,KP)
  100    JI(I)=KP
C
C        UPDATE LABELS ØF NØDES NØT YET IN TREE.
C
  200 DØ 300 I=1,NITP
         NI = NIT(I)
         D=DM(NI,KP)
         IF(UI(I).LE.D)  GØ TØ 300
         UI(I)=D
         JI(I)=KP
  300    CØNTINUE
C
C     FIND NØDE ØUTSIDE TREE NEAREST TØ TREE.
C
      UK=UI(1)
      DØ 400 I=1,NITP
         IF(UI(I).GT.UK)  GØ TØ 400
         UK=UI(I)
         K=I
  400    CØNTINUE
C
C     PUT NØDES ØF APPRØPRIATE EDGE INTØ ARRAY MST.
C
      IMST=IMST+1
      MST(1,IMST)=NIT(K)
      MST(2,IMST)=JI(K)
      CST=CST+UK
      KP=NIT(K)
C
C     DELETE NEW TREE NØDE FRØM ARRAY IT.
C
      UI(K)=UI(NITP)
      NIT(K)=NIT(NITP)
      JI(K)=JI(NITP)
      NITP=NITP-1
  500 IF (NITP.NE.0) GØ TØ 200
C
C     WHEN ALL NØDES ARE IN TREE, QUIT.
C
      RETURN
      END
```

Remark on Algorithm 422 [H]

Minimal Spanning Tree [V.K.M. Whitney, *Comm. ACM 15* (Apr. 1972), 273–4]

B.W. Kernighan [Recd. 23 June 1972] Bell Telephone Laboratories, Incorporated, Murray Hill, New Jersey

An integer-arithmetic version of Algorithm 422 has been tested on the Honeywell 6070 using the Fortran A compiler, on several graphs. The algorithm produced correct results in all cases.

Algorithm 422 computes the minimal spanning tree by successively adding the nearest remaining node to a partially formed tree until all nodes of the graph are included in the tree. This procedure, which the author attributes to Dijkstra [1], was in fact independently developed by R.C. Prim [2], two years earlier.

References

1. Dijkstra, E.W. A note on two problems in connection with graphs. *Numerische Math. 1*, 5 (Oct. 1959), 269–271.
2. Prim, R.C. Shortest connection networks and some generalizations. *Bell Syst. Tech. J. 36* (Nov. 1957), 1389–1401.

Algorithm 423
Linear Equation Solver [F4]

Cleve B. Moler (Recd. 1 July 1970 and 1 Dec. 1970)
Department of Mathematics, The University of
Michigan, Ann Arbor, MI 48104
(This work was supported by the Office of Naval
Research under contract NR 044-377.)

Key Words and Phrases: matrix algorithms, linear equations, Fortran, paged memory, virtual memory, array processing

CR Categories: 4.22, 4.32, 5.14

Description

These routines are modifications of, and intended as replacements for, the corresponding routines in [1]. The modifications increase efficiency while retaining accuracy and ease of use. Consideration is made of the effect of Fortran array storage conventions and paged dynamic memory allocation schemes. When translated by a good Fortran compiler, the routines should be competitive with programs written directly in machine language. For more details, see [2].

Both routines must be used to solve a system of linear equations, *Ax = b*. *DECOMP* carries out that part of the computation which depends only on the matrix *A*. *SOLVE* uses these results to obtain the solution for any right hand side *b*.

References

1. Forsythe, G.E., and Moler, C.B. *Computer Solution of Linear Algebraic Systems.* Prentice-Hall, Englewood Cliffs, N.J., 1967.
2. Moler, Cleve B. Matrix computations with Fortran and paging. *Comm. ACM 15* (Apr. 1972), 268–270.

Algorithm

```
      SUBROUTINE DECOMP(N, NDIM, A, IP)
      REAL A(NDIM,NDIM),T
      INTEGER IP(NDIM)
C
C  MATRIX TRIANGULARIZATION BY GAUSSIAN ELIMINATION.
C  INPUT..
C     N = ORDER OF MATRIX.
C     NDIM = DECLARED DIMENSION OF ARRAY  A .
C     A = MATRIX TO BE TRIANGULARIZED.
C  OUTPUT..
C     A(I,J), I.LE.J = UPPER TRIANGULAR FACTOR, U .
C     A(I,J), I.GT.J = MULTIPLIERS = LOWER TRIANGULAR
C                      FACTOR, I-L .
C     IP(K), K.LT.N = INDEX OF K-TH PIVOT ROW.
C     IP(N) = (-1)**(NUMBER OF INTERCHANGES) OR 0 .
C  USE 'SOLVE' TO OBTAIN SOLUTION OF LINEAR SYSTEM.
C  DETERM(A) = IP(N)*A(1,1)*A(2,2)*...*A(N,N).
C  IF IP(N)=0, A IS SINGULAR, SOLVE WILL DIVIDE BY ZERO.
C  INTERCHANGES FINISHED IN U , ONLY PARTLY IN L .
C
      IP(N) = 1
      DO 6 K = 1,N
        IF(K.EQ.N) GO TO 5
        KP1 = K+1
        M = K
        DO 1 I = KP1,N
          IF(ABS(A(I,K)).GT.ABS(A(M,K))) M = I
1       CONTINUE
        IP(K) = M
        IF(M.NE.K) IP(N) = -IP(N)
        T = A(M,K)
        A(M,K) = A(K,K)
        A(K,K) = T
        IF(T.EQ.0.) GO TO 5
        DO 2 I = KP1,N
2         A(I,K) = -A(I,K)/T
        DO 4 J = KP1,N
          T = A(M,J)
          A(M,J) = A(K,J)
          A(K,J) = T
          IF(T.EQ.0.) GO TO 4
          DO 3 I = KP1,N
3           A(I,J) = A(I,J) + A(I,K)*T
4       CONTINUE
5       IF(A(K,K).EQ.0.) IP(N) = 0
6     CONTINUE
      RETURN
      END
```

```
      SUBROUTINE SOLVE(N, NDIM, A, B, IP)
      REAL A(NDIM,NDIM),B(NDIM),T
      INTEGER IP(NDIM)
C
C  SOLUTION OF LINEAR SYSTEM, A*X = B .
C  INPUT..
C    N = ORDER OF MATRIX.
C    NDIM = DECLARED DIMENSION OF ARRAY  A .
C    A = TRIANGULARIZED MATRIX OBTAINED FROM 'DECOMP'.
C    B = RIGHT HAND SIDE VECTOR.
C    IP = PIVOT VECTOR OBTAINED FROM 'DECOMP'.
C  DO NOT USE IF DECOMP HAS SET IP(N)=0.
C  OUTPUT..
C    B = SOLUTION VECTOR, X .
C
      IF(N.EQ.1) GO TO 9
      NM1 = N-1
      DO 7 K = 1,NM1
        KP1 = K+1
        M = IP(K)
        T = B(M)
        B(M) = B(K)
        B(K) = T
        DO 7 I = KP1,N
    7 B(I) = B(I) + A(I,K)*T
      DO 8 KB = 1,NM1
        KM1 = N-KB
        K = KM1+1
        B(K) = B(K)/A(K,K)
        T = -B(K)
        DO 8 I = 1,KM1
    8 B(I) = B(I) + A(I,K)*T
    9 B(1) = B(1)/A(1,1)
      RETURN
      END
```

Algorithm 424

Clenshaw-Curtis Quadrature [D-1]

W. Morven Gentleman (Recd. 5 Oct. 1970 and
13 Aug. 1971)
University of Waterloo, Waterloo, Ontario, Canada

Key Words and Phrases: quadrature, Chebyshev series, cosine transform, fast Fourier transform
CR Categories: 5.1

Description

Clenshaw-Curtis quadrature is one of the most effective automatic quadrature schemes available, particularly for integrands with some continuous derivatives. It can also be used for any piecewise continuous integrand, although it is not recommended for integrands with discontinuities.

The automatic scheme [1] consists of evaluating the $N + 1$ point Clenshaw-Curtis quadrature formula, together with some error estimate, for a sequence of N's until the estimated absolute error *ESTERR* is less than the product of the tolerated relative error *TOLERR* and the absolute value of the current estimate of the integral, or until the permitted number of function evaluations would be exceeded. The function subprogram *CCQUAD* uses the sequence $N = 6, 18, \ldots, 2*3**M$. The error estimate used is the absolute difference between the integral estimates for the current and preceding choices of N. Other error estimates exist [1] although they are not as reliable, and the cosine transform *CSXFRM*(1), ..., *CSXFRM*(*USED*) is returned so they can be computed if desired. The $N + 1$ point Clenshaw-Curtis quadrature formula shifts the interval (A, B) to the interval $(-1, 1)$, then integrates the polynomial which interpolates the integrand F at the Chebyshev points $cos(\pi s/N)$, $s = 0, 1, \ldots, N$. Because the cosine transform is an explicit representation of this polynomial, an approximation to the indefinite integral of the integrand in the interval can be obtained from the indefinite integral of this polynomial, which is another reason why the cosine transform is returned.

Earlier implementations of this quadrature scheme [e.g. 4] computed the cosine transform by a recursive method which was slow and suffered from rounding error, but *CCQUAD* uses a variant of the fast Fourier transform [2, 3] and is very fast and very resistant to rounding errors. Timings on several machines indicate that the total cost of the quadrature can be well described as the cost of computing two sines and two cosines for each integrand value used, plus, of course, the cost of computing the integrand values themselves. The variant of the *FFT* used obtains all sines and cosines as required, and does not build tables or march recurrance relations. Using a separate subroutine (*R3PASS*) to perform the passes of the *FFT* on interleaved subsequences of the original sequence is a device introduced by G. Sande to force compilers for many machines to generate optimal code for the *FFT*.

There is no requirement in the subprogram *CCQUAD* that *A* be less than *B*. There is also no requirement that *TOLERR* be positive: if it is not, the maximum permitted number of integrand values will be used. The stopping rule always depends on relative error: if this is meaningless because the true integral vanishes, the maximum permitted number of integrand values will be used. Because the number of nested *DO* loops used to generate integrand values in digit reversed order is fixed at eight, the maximum number of integrand values permitted is the smaller of *LIMIT* and $2*3**9 + 1 = 39367$. This should be ample but the restriction is easily changed.

Throughout the subprogram *CCQUAD* various statements appear with a *C* in column 1. If these comments are replaced by the statements themselves, intermediate results are written on unit number 6, enabling one to follow the decision process of the scheme. This can be very instructive in understanding the way the scheme works.

References

1. Gentleman, W.M. Implementing Clenshaw-Curtis quadrature, I Methodology and experience. *Comm. ACM 15* (May 1972), 337–342.
2. Gentleman, W.M. Implementing Clenshaw-Curtis quadrature, II Computing the cosine transformation *Comm. ACM 15* (May 1972) 343–346.
3. Gentleman, W.M., and Sande, G. Fast Fourier transforms—for fun and profit. Proc. AFIPS 1966 FJCC, Vol. 29, Spartan Books, New York, pp. 563–578.
4. Hopgood, F.R.A., and Litherland, C. Algorithm 279, Chebyshev quadrature. *Comm. ACM 9* (1966), 270.

Algorithm

```
      REAL FUNCTION CCQUAD (F,A,B,TOLERR,LIMIT,ESTERR,USED,
     *     CSXFRM)
C
C INPUT ARGUMENTS-
      REAL F,A,B,TOLERR
      INTEGER LIMIT
C OUTPUT ARGUMENTS-
      REAL ESTERR,CSXFRM(LIMIT)
      INTEGER USED
C
C    USING CLENSHAW-CURTIS QUADRATURE, THIS FUNCTION SUB-
C PROGRAM ATTEMPTS TO INTEGRATE THE FUNCTION F FROM A TO B
C TO AT LEAST THE REQUESTED RELATIVE ACCURACY TOLERR, WHILE
C USING NO MORE THAN LIMIT FUNCTION EVALUATIONS.  IF THIS
C CAN BE DONE, CCQUAD RETURNS THE VALUE OF THE INTEGRAL,
C ESTERR RETURNS AN ESTIMATE OF THE ABSOLUTE ERROR ACTUALLY
C COMMITTED, USED RETURNS THE NUMBER OF FUNCTION VALUES
C ACTUALLY USED, AND CSXFRM(1),...,CSXFRM(USED) CONTAINS
C N=USED-1 TIMES THE DISCRETE COSINE TRANSFORM, AS USUALLY
C DEFINED, OF THE INTEGRAND IN THE INTERVAL.  IF THE
C REQUESTED ACCURACY CANNOT BE ATTAINED WITH THE NUMBER OF
C FUNCTION EVALUATIONS PERMITTED, THE LAST (AND PRESUMABLY
C BEST) ANSWER OBTAINED IS RETURNED.
C
      REAL PI,RT3,CENTRE,WIDTH,SHIFT,FUND,ANGLE,C,S
      REAL OLDINT,NEWINT
      REAL T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12
```

```
C INSERT THE FOLLOWING STATEMENT TO TRACE PROGRAM FLOW
C      REAL SCLINT,SCLERR
       INTEGER N,N2,N3,N LESS 1,N LESS 3,MAX,M MAX,J,STEP
       INTEGER L(8),L1,L2,L3,L4,L5,L6,L7,L8
       INTEGER J1,J2,J3,J4,J5,J6,J7,J8,J REV
       EQUIVALENCE (L(1),L1),(L(2),L2),(L(3),L3),(L(4),L4),
     .    (L(5),L5),(L(6),L6),(L(7),L7),(L(8),L8),(J8,J REV)
       DATA PI,RT3/   3.14159265358 9E0,    1.732050807568E0 /
       DATA M MAX/   8 /
C
C
C                               INITIALIZATION
       CENTRE=(A+B)*.5E0
       WIDTH=(B-A)*.5E0
       MAX=MIN0(LIMIT,2*3**(M MAX+1))
       DO 10 J=1,M MAX
          L(J)=1
    10 CONTINUE
C
C
C                               COSINE TRANSFORM
C COMPUTE DOUBLE THE COSINE TRANSFORM WITH N=6
       N=6
C SAMPLE FUNCTION
       CSXFRM(1)=F(A)
       CSXFRM(7)=F(B)
       SHIFT=WIDTH*RT3*.5E0
       CSXFRM(2)=F(CENTRE-SHIFT)
       CSXFRM(6)=F(CENTRE+SHIFT)
       SHIFT=WIDTH*.5E0
       CSXFRM(3)=F(CENTRE-SHIFT)
       CSXFRM(5)=F(CENTRE+SHIFT)
       CSXFRM(4)=F(CENTRE)
C EVALUATE THE FACTORED N=6 COSINE TRANSFORM
       T1=CSXFRM(1)+CSXFRM(7)
       T2=CSXFRM(1)-CSXFRM(7)
       T3=2.E0*CSXFRM(4)
       T4=CSXFRM(2)+CSXFRM(6)
       T5=(CSXFRM(2)-CSXFRM(6))*RT3
       T6=CSXFRM(3)+CSXFRM(5)
       T7=CSXFRM(3)-CSXFRM(5)
       T8=T1+2.E0*T6
       T9=2.E0*T4+T3
       T10=T2+T7
       T11=T1-T6
       T12=T4-T3
       CSXFRM(1)=T8+T9
       CSXFRM(2)=T10+T5
       CSXFRM(3)=T11+T12
       CSXFRM(4)=T2-2.E0*T7
       CSXFRM(5)=T11-T12
       CSXFRM(6)=T10-T5
       CSXFRM(7)=T8-T9
       USED=7
C GO TO INTEGRAL COMPUTATION, BUT FIRST COMPUTE INTEGRAL FOR
C N=2
       GO TO 200
C
C
C COMPUTE REFINED APPROXIMATION
C SAMPLE FUNCTION AT INTERMEDIATE POINTS IN DIGIT REVERSED
C ORDER.  AS THE SEQUENCE IS GENERATED, COMPUTE THE FIRST
C (RADIX FOUR TRANSFORM) PASS OF THE FAST FOURIER TRANSFORM
   100 DO 110 J=2,M MAX
          L(J-1)=L(J)
   110 CONTINUE
       L(M MAX)=3*L(M MAX-1)
       J=USED
       FUND=PI/FLOAT(3*N)
       DO 120 J1=1,L1,1
          DO 120 J2=J1,L2,L1
             DO 120 J3=J2,L3,L2
                DO 120 J4=J3,L4,L3
                   DO 120 J5=J4,L5,L4
                      DO 120 J6=J5,L6,L5
                         DO 120 J7=J6,L7,L6
                            DO 120 J8=J7,L8,L7
                               ANGLE=FUND*FLOAT(3*J REV-2)
                               SHIFT=WIDTH*COS(ANGLE)
                               T1=F(CENTRE-SHIFT)
                               T3=F(CENTRE+SHIFT)
                               SHIFT=WIDTH*SIN(ANGLE)
                               T2=F(CENTRE+SHIFT)
                               T4=F(CENTRE-SHIFT)
                               T5=T1+T3
                               T6=T2+T4
                               CSXFRM(J+1)=T5+T6
                               CSXFRM(J+2)=T1-T3
                               CSXFRM(J+3)=T5-T6
                               CSXFRM(J+4)=T2-T4
                               J=J+4
   120 CONTINUE
C DO RADIX 3 PASSES OF FAST FOURIER TRANSFORM
       N2=2*N
       STEP=4
   150 J1=USED+STEP
       J2=USED+2*STEP
       CALL R3PASS (N2,STEP,N2-2*STEP,CSXFRM(USED+1),
     .    CSXFRM(J1+1),CSXFRM(J2+1))
       STEP=3*STEP
       IF (STEP .LT. N) GO TO 150
C
C COMBINE RESULTS
C FIRST DO J=0 AND J=N
       T1=CSXFRM(1)
       T2=CSXFRM(USED+1)
       CSXFRM(1)=T1+2.E0*T2
       CSXFRM(USED+1)=T1-T2
       T1=CSXFRM(N+1)
       T2=CSXFRM(N2+2)
       CSXFRM(N+1)=T1+T2
       CSXFRM(N2+2)=T1-2.E0*T2
C NOW DO REMAINING VALUES OF J
       N3=3*N
       N LESS 1=N-1
       DO 180 J=1,N LESS 1
          J1=N+J
          J2=N3-J
          ANGLE=FUND*FLOAT(J)
          C=COS(ANGLE)
          S=SIN(ANGLE)
          T1=C*CSXFRM(J1+2)-S*CSXFRM(J2+2)
          T2=(S*CSXFRM(J1+2)+C*CSXFRM(J2+2))*RT3
          CSXFRM(J1+2)=CSXFRM(J+1)-T1-T2
          CSXFRM(J2+2)=CSXFRM(J+1)-T1+T2
          CSXFRM(J+1)=CSXFRM(J+1)+2.E0*T1
   180 CONTINUE
C NOW UNSCRAMBLE
       T1=CSXFRM(N2+1)
       T2=CSXFRM(N2+2)
       DO 190 J=1,N LESS 1
          J1=USED+J
          J2=N2+J
          CSXFRM(J2)=CSXFRM(J1)
          CSXFRM(J1)=CSXFRM(J2+2)
   190 CONTINUE
       CSXFRM(N3)=T1
       CSXFRM(N3+1)=T2
       N=N3
       USED=N+1
C GO TO INTEGRAL COMPUTATION
       GO TO 210
C
C
C                               INTEGRAL EVALUATION
C INTEGRAL ESTIMATES ARE NOT SCALED BY WIDTH*N/2
C UNTIL FUNCTION CCQUAD RETURNS.
C
C WHEN N=6,EVALUATE INTEGRAL FOR N=2
   200 OLDINT=(T1+2.E0*T3)/3.E0
C
C
C EVALUATE NEW ESTIMATE OF INTEGRAL
   210 N LESS 3=N-3
       NEWINT=.5E0*CSXFRM(USED)/FLOAT(1-N**2)
       DO 220 J=1,N LESS 3,2
          J REV=N-J
          NEWINT=NEWINT+CSXFRM(J REV)/FLOAT(J REV*(2-J REV))
   220 CONTINUE
       NEWINT=NEWINT+.5E0*CSXFRM(1)
C
C
C                               TEST IF DONE
C TEST IF ESTIMATED ERROR ADEQUATE
       ESTERR=ABS(OLDINT*3.E0-NEWINT)
C INSERT THE FOLLOWING FOUR STATEMENTS TO TRACE PROGRAM FLOW
C      SCLINT=WIDTH*NEWINT/FLOAT(N/2)
C      SCLERR=WIDTH*(OLDINT*3.E0-NEWINT)/FLOAT(N/2)
C      WRITE (6,900) N,SCLINT,SCLERR
C 900 FORMAT (3H N=,I5,23H INTEGRAL ESTIMATED AS ,E15.8,
C    .   7H ERROR ,E15.8)
       IF (ABS(NEWINT)*TOLERR .GE. ESTERR) GO TO 400
C IF ESTIMATED ERROR TOO LARGE, REFINE SAMPLING IF PERMITTED
       OLDINT=NEWINT
       IF (3*N+1 .LE. MAX) GO TO 100
C IF REFINEMENT NOT PERMITTED, OR IF ESTIMATED ERROR
C SATISFACTORY, RESCALE ANSWERS AND RETURN
C INSERT THE FOLLOWING TWO STATEMENTS TO TRACE PROGRAM FLOW
C      WRITE (6,910)
C 910 FORMAT (25H REFINEMENT NOT PERMITTED)
   400 CCQUAD=WIDTH*NEWINT/FLOAT(N/2)
       ESTERR=WIDTH*ESTERR/FLOAT(N/2)
       RETURN
       END
C
C
C
C
       SUBROUTINE R3PASS (N2,M,LENGTH,X0,X1,X2)
C RADIX 3 PASS FOR FAST FOURIER TRANSFORM OF REAL SEQUENCE
C OF LENGTH N2
       INTEGER N2,M,LENGTH
       REAL X0(LENGTH),X1(LENGTH),X2(LENGTH)
C THE NOTATION OF REFERENCES 2 AND 3 IS USED IN THIS
C SUBROUTINE.
C M IS THE LENGTH OF THE TRANSFORM ALREADY ACCOMPLISHED,
C I.E. THE NUMBER OF DISTINCT VALUES OF THE FREQUENCY INDEX
C C HAT OF THESE TRANSFORMS, AND THE SPACING OF THE
C SEQUENCES TO BE TRANSFORMED.  EXPLICIT USE IS MADE OF THE
C FACT THAT M IS EVEN AND NOT LESS THAN FOUR.
       INTEGER HALF M,M3,K,K0,K1,J,J0,J1
       REAL TWOPI,HAFRT3,RSUM,RDIFF,RSUM2,ISUM,IDIFF,IDIFF2
       REAL FUND,ANGLE,C1,S1,C2,S2,R0,R1,R2,I0,I1,I2
       DATA TWOPI, HAFRT3/   6.283185307E0,   .866025403E0 /
       HALF M=(M-1)/2
       M3=M*3
       FUND=TWOPI/FLOAT(M3)
```

```
C DO ALL TRANSFORMS FOR C HAT=0, I.E. TWIDDLE FACTOR UNITY
      DO 10 K=1,N2,M3
          RSUM=(X1(K)+X2(K))
          RDIFF=(X1(K)-X2(K))*HAFRT3
          X1(K)=X0(K)-RSUM*.5E0
          X2(K)=RDIFF
          X0(K)=X0(K)+RSUM
   10 CONTINUE
C DO ALL TRANSFORMS FOR C HAT=CAP C/2, I.E. TWIDDLE FACTOR
C E(B/6)
      J=M/2+1
      DO 20 K=J,N2,M3
          RSUM=(X1(K)+X2(K))*HAFRT3
          RDIFF=(X1(K)-X2(K))
          X1(K)=X0(K)-RDIFF
          X2(K)=RSUM
          X0(K)=X0(K)+RDIFF*.5E0
   20 CONTINUE
C DO ALL TRANSFORMS FOR REMAINING VALUES OF C HAT.  OBSERVE
C THAT C HAT AND CAP C-C HAT MUST BE PAIRED
C CHOOSE A FREQUENCY INDEX
      DO 40 J=1,HALF M
          J0=J+1
          J1=M-J+1
C COMPUTE THE TWIDDLE FACTOR
          ANGLE=FUND*FLOAT(J)
          C1=COS(ANGLE)
          S1=SIN(ANGLE)
          C2=C1**2-S1**2
          S2=2.E0*S1*C1
C CHOOSE THE REPLICATION
          DO 30 K0=J0,N2,M3
              K1=K0-J0+J1
C OBTAIN TWIDDLED VALUES
              R0=X0(K0)
              I0=X0(K1)
              R1=C1*X1(K0)-S1*X1(K1)
              I1=S1*X1(K0)+C1*X1(K1)
              R2=C2*X2(K0)-S2*X2(K1)
              I2=S2*X2(K0)+C2*X2(K1)
C COMPUTE TRANSFORMS AND RETURN IN PLACE
              RSUM=R1+R2
              RDIFF=(R1-R2)*HAFRT3
              RSUM2=R0-.5E0*RSUM
              ISUM=I1+I2
              IDIFF=(I1-I2)*HAFRT3
              IDIFF2=I0-.5E0*ISUM
              X0(K0)=R0+RSUM
              X0(K1)=RSUM2+IDIFF
              X1(K0)=RSUM2-IDIFF
              X1(K1)=RDIFF+IDIFF2
              X2(K0)=RDIFF-IDIFF2
              X2(K1)=I0+ISUM
   30     CONTINUE
   40 CONTINUE
      RETURN
      END
```

Remark on Algorithm 424 [D 1]

Clenshaw-Curtis Quadrature [W.M.Gentleman, *Comm. ACM 15* (May 1972), 353–355.]

Albert J. Good [Recd. 19 December 1972] Systems, Science and Software, La Jolla, CA 92037

As published, this algorithm will not execute correctly under some compilers (e.g. Fortran V in the Univac 1108). One minor change is sufficient for proper operation: replace the variable *J REV* by the index *J8* inside the *DO* 120 loop.

The appearance of *J REV* and *J8* in an *EQUIVALENCE* statement is not meaningful since the memory location associated with a *DO* loop index does not always contain the current value of the index (this depends on the compiler).

ACM Transactions on Mathematical Software, Vol. 5, No. 2, June 1979, Pages 240.

REMARK ON ALGORITHM 424

Clenshaw-Curtis Quadrature [01]
[W.M. Gentleman, *Comm. ACM 15,* 5 (May 1972), 353-355]

K.O. Geddes [Recd 1 February 1978 and 17 April 1978]
Department of Computer Science, University of Waterloo, Waterloo, Ont., Canada N2L 3G1

This algorithm may be used to compute the Chebyshev series coefficients for a function F which is continuous on the interval $[-1,1]$, as noted in [1]. For this purpose, function *CCQUAD* would be called with $A = -1$, $B = 1$, and appropriate values of *TOLERR* and *LIMIT.* (For some applications, one would prefer instead to state the number of Chebyshev series coefficients desired.) The comments in function *CCQUAD* indicate that the array *CSXFRM* contains, on return, $N = USED - 1$ times the discrete cosine transform of F. Therefore, the values

$$CSXFRM(K)/N,\ 1 \le K \le NUMBER,$$

for some $NUMBER \le USED$, should be estimates for the first *NUMBER* Chebyshev series coefficients of F.

However, the published code produces an array *CSXFRM* with an incorrect sign on each value *CSXFRM*(K) for K even (i.e. the odd Chebyshev series coefficients will all have incorrect signs). This error does not affect the value of the definite integral computed by the algorithm because only the even terms in the Chebyshev series enter into the computation of the definite integral. The error does, however, affect the stated claim that "because the cosine transform is an explicit representation . . . , an approximation to the indefinite integral . . . can be obtained from the indefinite integral [of the truncated Chebyshev series]." The error can be corrected as follows.

Change the eighth and ninth executable statements

```
CSXFRM(1) = F(A)  to  CSXFRM(1) = F(B)
CSXFRM(7) = F(B)  to  CSXFRM(7) = F(A)
```

Change the statements one and four lines below this

```
SHIFT = WIDTH*RT3*.5EO   to  SHIFT = –WIDTH*RT3*.5EO
SHIFT = WIDTH*.5EO       to  SHIFT = –WIDTH*.5EO
```

Change the second and fifth statements following the eight nested "DO 120" statements

```
SHIFT = WIDTH*COS(ANGLE)     to  SHIFT = –WIDTH*COS(ANGLE)
SHIFT = WIDTH*SIN(ANGLE)     to  SHIFT = –WIDTH*SIN(ANGLE)
```

REFERENCES

1. GEDDES, K.O. Near-minimax polynomial approximation in an elliptical region. *SIAM J. Numer. Anal. 15* (1978), 1225–1233.

Algorithm 425

Generation of Random Correlated Normal Variables [G5]

Rex L. Hurst
Applied Statistics-Computer Science, Utah State University, Logan, UT 84321
and
Robert E. Knop* [Recd. 12 March 1970, 23 March 1971, and 9 Nov. 1971]
Department of Physics, Florida State University, Tallahassee, FL 32306

Key Words and Phrases: random number, normal density, normal distribution, Gaussian density, Gaussian distribution, simulation, Monte Carlo
CR Categories: 5.5

Description

We have programmed and made timing comparisons for two algorithms which sample the multivariate normal density

$$N(\mu, V) = | V^{-1} |/(2\pi)^{n/2} \cdot exp(- 1/2(Y - \mu)^T V^{-1}(Y - \mu)) \quad (1)$$

where V is an $n \times n$ covariance matrix, μ is an n component vector of means, and Y is an n component random vector [1].

The first algorithm proceeds by rotating coordinates to a system in which the covariance matrix is diagonal. In this system the multivariate normal density becomes equal to the product of its marginal densities, and each marginal density can be sampled independently of the others. After obtaining a sample vector in this rotated system, the coordinates are rotated back to the original system. In the following discussion this will be referred to as the matrix diagonalization algorithm [1].

The second algorithm proceeds by decomposing the multivariate normal density into the product of the marginal density of the first variate times the joint density of the remaining variates, conditional upon the value sampled for the first. This joint density is determined once the first variate has been sampled from its marginal density. The procedure is then applied to the second variate and iterated until values have been assigned to all components of the sample vector. In the following discussion this will be referred to as the conditional decomposition algorithm [1].

Both algorithms require that the covariance matrix be positive definite, and that it modify the argument *IENT* to indicate if this condition was not satisfied. Both algorithms perform extensive calculations on the covariance matrix the first time it is used. Subsequent sample vectors with the same covariance matrix bypass these calculations with considerable savings in execution time. Tests with eight variables produced the following execution times on an IBM 360/44:

	1 Matrix 500 Observations	1 Matrix 1000 Observations	200 Matrices 1 Observation
Matrix diagonalization	37 sec	72 sec	143 sec
Conditional decomposition	35 sec	68 sec	14 sec

We note that the conditional decomposition algorithm executes more rapidly in all cases.

Matrix Diagonalization. Suppose we define A to be the desired correlation structure; A can always be represented as $B\ I\ B^T$. We know the characteristic values λ_i of A are defined as the roots of the characteristic equation

$$| A - \lambda_i I | = 0. \quad (2)$$

The characteristic vector is a vector not identically zero satisfying, for characteristic value λ_i

$$(A - \lambda_i I)X_i = 0. \quad (3)$$

If A is symmetric, all roots different, and X_i are normalized, then

$$X_i^T X_j = \delta_{ij}$$

where δ_{ij} is the Kronecker delta. Let C be the matrix of characteristic vectors and D be a diagonal matrix of the characteristic roots:

$$C = [X_1, X_2 \cdots]$$

$$D = \begin{bmatrix} \lambda_1 & 0 & \cdots \\ 0 & \lambda_2 & \\ \vdots & & \end{bmatrix} \quad (5)$$

Then

$$C^T C = I \text{ and } C\ C^T = I. \quad (6)$$

The matrix C is thus orthogonal [2].

For an orthogonal matrix C and a symmetrix matrix A

$$C^T A\ C = D \text{ and } A = C\ D\ C^T, \quad (7)$$

therefore

$$A = C\ D^{\frac{1}{2}}\ I\ D^{\frac{1}{2}}\ C^T, \quad (8)$$

and we see that the matrix required to transform a set of independent normal variates to a new set with correlation matrix A is $B = C\ D^{\frac{1}{2}}$.

If A is distributed according to $N(0, A)$ (cf. (1)) and we define:

$$S = \begin{bmatrix} \sigma_1 & 0 & \cdots \\ 0 & \sigma_2 & \\ \vdots & & \end{bmatrix} \quad \text{and} \quad \mu = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \vdots \end{bmatrix}$$

then $(S Z + \mu)$ is distributed according to $N(\mu, \Sigma)$ where Σ is the variance-covariance matrix. To save computational time the matrix B may be defined

$$B = S\ C\ D^{\frac{1}{2}} \quad (10)$$

Subroutine *RANVR* receives a correlation matrix A, a vector of desired standard deviations *SD*, a positive definite test variable *IENT*, an argument for a random number generator *IARG*, variables for defining the order of $A(NV)$ and the order of the arrays used *NI*, and work arrays X, Y, and Z. Z is the return array. Upon return the diagonal of A contains the roots and the columns of X the vectors.

It requires a subroutine for computing characteristic values and vectors for real symmetric matrices [3-7], a subroutine for generat-

* This work was supported in part by the United States Atomic Energy Commission.

ing random normal deviates [8–12] which in turn requires a subroutine for generating random uniform numbers [13, 14]. We use a modification of Seraphin [14], which allows the generation of different sequences by modifying an entry argument.

```
Calling sequence      (BZ desired means)
     IENT = -1
     .
     .
     CALL RANVR(A, X, Y, Z, SD, NV, NI, IENT, IARG)
     IF (IENT·LE·0)     GO TO 5
     DO 4 I = 1, NV
4    Z(I) = Z(I) + BZ(I)
     .
     .
     .
5    Error handling if not positive definite.
```

Conditional Decomposition. To achieve the conditional decomposition of the multivariate normal density $N(0, V)$, we begin by partitioning the covariance matrix into the scalar v_{11}, the $1 \times (n-1)$ and $(n-1) \times 1$ vectors V_{12} and V_{21}, and the $(n-1) \times (n-1)$ matrix V_{22}:

$$V = \begin{pmatrix} v_{11} & V_{12} \\ V_{21} & V_{22} \end{pmatrix} \tag{11}$$

The inverse covariance matrix we represent as:

$$V^{-1} = \begin{pmatrix} r_{11} & R_{12} \\ R_{21} & R_{22} \end{pmatrix} \tag{12}$$

From $V V^{-1} = I$ we obtain the following relations:

$$\begin{aligned} 1/v_{11} &= r_{11} + R_{12}R_{22}^{-1}R_{21}, \\ R_{22} &= (V_{22} - V_{12}V_{21}/v_{11})^{-1}. \end{aligned} \tag{13}$$

The quadratic form of the multivariate normal density $N(0, V)$ can be written as:

$$Y^TV^{-1}Y = (y_1\, Y_2{}^T) \begin{pmatrix} r_{11} & R_{12} \\ R_{21} & R_{22} \end{pmatrix} \begin{pmatrix} y_1 \\ Y_2 \end{pmatrix} \tag{14}$$

Multiplying this out results in

$$Y^TV^{-1}Y = y_1r_{11}y_1 + (Y_2{}^TR_{22}Y_2 + Y_2{}^TR_{21}y_1 + y_1R_{12}Y_2). \tag{15}$$

Performing the matrix analog of completing the square on the term involving Y_2 allows this to be written as

$$\begin{aligned} Y^TV^{-1}Y = {} & y_1(r_{11} - R_{12}R_{22}^{-1}R_{21})y_1 \\ & + (Y_2 - (R_{22}^{-1}R_{21}y_1)^TR_{22}(Y_2 - R_{22}^{-1}R_{21}y_1). \end{aligned} \tag{16}$$

Substituting from (13) we obtain

$$\begin{aligned} Y^TV^{-1}Y = {} & y_1{}^2/v_{11} + (Y_2 - V_{21}y_1/v_{11})^T(V_{22} - V_{21}V_{12}/v_{11})^{-1} \\ & (Y_2 - V_{21}y_1/v_{11}). \end{aligned} \tag{17}$$

Thus the multivariate normal density $N(0, V)$ can be separated into the marginal density $N(0, v_{11})$ of the variate y, times the joint density

$$N(V_{21}y_1/v_{11}, V_{22} - V_{21}V_{12}/v_{11}) \tag{18}$$

of the vector Y_2 conditional upon y_1. This procedure is then repeated until every component of the random vector Y has been assigned a value.

Subroutine *RNVR* receives a covariance matrix *A*, a positive definite test variable *IENT*, an argument for a random number generator *IARG*, variables defining the order of *A*(*NV*) and the order of the arrays used *NI*, and work arrays *X*, *B*, *C*. *X* is the return array.

It requires a subroutine for generating random normal deviates which requires a subroutine for generating random uniform numbers.

```
Calling Sequence      (BZ desired means)
     IENT = -1
     .
     .
     CALL RNVR (Z, A, Y, C, NV, NI, IENT, IARG)
     IF (IENT.LE.0) GO TO 5
     DO 4 I = 1, NV
4    Z(I) = Z(I) + BZ(I)
     .
5    Error handling if not positive definite.
```

References

1. Anderson, T.W. *An Introduction to Multivariate Statistical Analysis.* Wiley, New York, 1958, p. 26.
2. Searle, S.R. *Matrix Algebra for the Biological Sciences.* Wiley, New York, 1966, p. 188.
3. Evans, Thomas G. Algorithm 85, Jacobi. *Comm. ACM 5* (Apr. 1962), 208.
4. Hillmore, J.S. Certification of Algorithm 85, Jacobi. *Comm. ACM 5* (Aug. 1962), 440.
5. Naur, P. Certification of Algorithm 85, Jacobi. *Comm. ACM 6* (Aug. 1963), 447–448.
6. Greenstadt, John. The determination of the characteristic roots of a matrix by the Jacobi method. In *Mathematical Methods for Digital Computers*, A. Ralston and H.S. Wilf (Eds.), Wiley, New York, 1967, pp. 84–91.
7. Stewart, G.W. Algorithm 384, Eigenvalues and eigenvectors of a real symmetric matrix. *Comm. ACM 13* (June 1970), 369–371.
8. Box, G., and Muller, M. A note on the generation of normal deviates. *Ann. Math. Stat. 28* (1958), 610.
9. Marsaglia, G. Expressing a random variable in terms of uniform random variables. *Ana. Math. Stat. 32* (1961), 894–898.
10. Knuth, Donald E. *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms.* Addison-Wesley, Reading, Mass., 1968.
11. Bell, James R. Algorithm 334, Random normal deviates. *Comm. ACM 11* (July 1968), 498.
12. Knop, R. Remark on Algorithm 334, Random normal deviates. *Comm. ACM 12* (May 1969), 281.
13. Strome, W. Murray. Algorithm 294, Uniform random. *Comm. ACM 10* (Jan. 1967), 40.
14. Seraphin, Dominic S. A fast random number generator for IBM 360. *Comm. ACM 12* (Dec. 1969), 695.

Algorithm

```
      SUBRØUTINE RNVR(X,A,B,C,NV,NI,IENT,IARG)
C     GENERATES A RANDØM NØRMAL VECTØR (M,S)
C     A        INPUT CØVARIANCE MATRIX, CØNDITIØNAL MØMENTS RETURN
C     Z,Y,C,   WØRK ARRAYS, RETURN VECTØR ØF RANDØM NØRMAL VARIABLES IN Z
C     NV,NI    ØRDER ØF CØVARIANCE MATRIX, ØRDER ØF ARRAY
C     IENT     -1= INITIAL ENTRY
C               O= RETURN IF NØT PØSITIVE DEFINITE
C               1= RETURN IF PØSITIVE DEFINITE
C     IARG     ARGUMENT FØR RANDØM NUMBER GENERATØR
      DIMENSIØN X(NI),A(NI,NI),B(NI),C(NI)
      IF(IENT) 1,9,6
C *** CØMPUTE CØNDITIØNAL MØMENTS
    1 NA=NV-1
      DØ 4 K=1,NA
        T=A(K,K)
        IF(T) 10,10,2
    2   NB=K+1
        C(K)=SQRT(T)
        DØ 3 I=NB,NV
    3     A(I,K)=A(K,I)/T
        DØ 4 I=NB,NV
          DØ 4 J=I,NV
    4       A(I,J)=A(I,J)-A(I,K)*A(K,J)
      IF(A(NV,NV)) 10,10,5
    5 IENT=1
      C(NV)=SQRT(A(NV,NV))
C *** CØMPUTE A RANDØM VECTØR
    6 DØ 8 I=1,NV
        B(I)=RNØR(IARG)*C(I)
        X(I)=B(I)
        IF(I.EQ.1) GØ TØ 8
        NB=I-1
        DØ 7 J=1,NB
    7     X(I)=X(I)+A(I,J)*B(J)
    8   CØNTINUE
    9 RETURN
   10 IENT=0
      RETURN
      END
      FUNCTIØN RNØR(IR)
C     GENERATES A RANDØM NØRMAL NUMBER (0,1)
C     IARG IS A LARGE ØDD INTEGER FØR A BEGINNING ARGUMENT
```

```
C      REQUIRES  FUNCTIØN RN WHICH GENERATES A UNIFØRM RANDØM NUMBER 0-1
       DATA I/0/
       IF(I.GT.0)GØ TØ 30
    10 X=2.0*RN(IR)-1.0
       Y=2.0*RN(IR)-1.0
       S=X*X+Y*Y
       IF(S.GE.(1.0))GØ TØ 10
       S=SQRT(-2.0*ALØG(S)/S)
       RNØR=X*S
       GØ2=Y*S
       I=1
       GØ TØ 40
    30 RNØR=GØ2
       I=0
    40 RETURN
       END
```

Remark on Algorithm 425 [G5]

Generation of Random Correlated Normal Variables [Rex L. Hurst and Robert E. Knop, *Comm. ACM 15* (May 1972), 355–357]

R.L. Page [Recd. 3 Oct. 1973]
Computer Science Program, Colorado State University,
Fort Collins, CO 80521

The work array parameters *B* and *C* of *SUBROUTINE RNVR*, which may prove cumbersome for some users, may be removed by making some minor changes. The removal of *C* is simple: simply change references to *C(I)* to *A(I, I)*. (The diagonal of *A* is presently unused once the conditional moments are computed.)

The vector *X* can be used in place of *B* provided its components are computed in reverse order. Thus, *DO* loop 8 (starting at statement 6) becomes two separate loops as shown below.

```
6  DO 7 I = 1, NV
7    X(I) = RNOR(IARG)*A(I, I)
   DO 8 I = 2, NV
     NB = NV-I+1
     DO 8 J = 1, NB
8      X(NB+1) = X(NB+1)+A(NB+1, J)*X(J)
```

The revised algorithm was tested on covariance matrices of orders two through six. Assuming the algorithm generates sample vectors from the zero mean normal distribution with the given covariance, the difference between the sample covariance and the given covariance, divided by the standard error of the covariance estimator, would give samples from a standard normal distribution. Our test did not contradict this assumption since 37 of 55 of these numbers, 67 percent, were in the range −1 to 1 (one would expect about 68 percent) and 54 of 55, 98 percent, were in the range −2 to 2 (one would expect about 95 percent).

Algorithm 426
Merge Sort Algorithm [M1]

C. Bron (Recd. 4 Feb. 1970 and 10 May 1971)
Technological University, Eindhoven, The Netherlands

Key Words and Phrases: sort, merge
CR Categories: 5.31

Description

Sorting by means of a two-way merge has a reputation of requiring a clerically complicated and cumbersome program. This ALGOL 60 procedure demonstrates that, using recursion, an elegant and efficient algorithm can be designed, the correctness of which is easily proved [2]. Sorting n objects gives rise to a maximum recursion depth of $[\log_2(n - 1) + 2]$. This procedure is particularly suitable for sorting when it is not desirable to move the n objects physically in store and the sorting criterion is not simple. In that case it is reasonable to take the number of compare operations as a measure for the speed of the algorithm. When n is an integral power of 2, this number will be comprised between $(n \times \log_2 n)/2$ when the objects are sorted to begin with and $(n \times \log_2 n - n + 1)$ as an upper limit. When n is not an integral power of 2, the above formulas are approximate.

It is assumed that each object can in some way be uniquely identified by one of the integers from 1 to n. This correspondence has to be supplied in the call by replacing *hi* and *lo* by two integer variables and the Jensen parameter *loafterhi* by a Boolean expression that yields the value **true** if the object identified by *lo* has to follow the object identified by *hi* in the ordered sequences, and **false** otherwise. Let e_i be the identifying integer of the ith object in the ordered sequence. Upon return from the procedure *sort* delivers the value of e_1 and the pointer array *pnt* will be filled in such a way that $pnt[e_i] = e_{i+1}$, $1 \leq i < n$, and $pnt[e_n] = 0$. Therefore the bounds of the actual array supplied for *pnt* will have to include the range $[1:n]$. Sorted subsequences that arise during the sorting process will have a similar chain structure.

The essence of the algorithm is to be found in the procedure *head*. It has the duty to form an ordered chain of desired length (*deslen*) from the objects identified by *count* + 1 through *count* + *deslen*. It does so by introducing a chain of length 1, consisting of object *count* + 1, and then repeatedly doubling the length of that chain by merging it with a chain of equal length the creation of which is left to a recursive call on *head*. If *deslen* is not an integral power of 2, a chain of length *deslen* can not be built by repeatedly doubling. In that case, before the last merge operation, a chain of length (desired length − present length) is created and merged with the present chain to produce the required result.

As an example of a call on the sorting procedure we supply *sort*(10 000, *chain*, *i*, *j*, $A[i] > A[j]$) although it should be stressed that the present version of the algorithm is not efficient when the sorting criterion is as simple as a comparison of two array elements. In such a case one does not only gain by replacing the calls on the formal parameter *loafterhi* by $A[lo] > A[hi]$ and declaring *lo* and *hi* as local variables of the procedure *sort*, but also one might resort to *in situ* sorting techniques like [1] that do not need the auxiliary array *pnt*. A comparison of this algorithm with *QUICKERSORT* [1] conducted under equivalent circumstances on the ALGOL system for the EL X8 showed no significant difference in speed when sorting arrays containing random numbers.

Acknowledgment. The author is grateful to Prof. E.W. Dijkstra for his contributions to this version of the algorithm, and to the referee for his careful analysis and valuable suggestions.

References

1. Scowen, R.S. Quickersort, *Comm. ACM 8* (Oct. 1965), 669–670.
2. Bron, C., Proof of a merge sort algorithm, May 1971 (unpublished).

Algorithm

```
integer procedure sort(n, pnt, lo, hi, loafterhi);
  value n; integer n, lo, hi; integer array pnt;
  Boolean loafterhi;
begin
  integer count, link;
  comment link is a working location for merging;
  integer procedure head(deslen);
    value deslen; integer deslen;
  comment The value of head will be the identifying integer of the
    object leading the sorted chain;
  begin
    integer beg, len, nextlen;
INTRODUCE NEW CHAIN OF LENGTH 1:
SUPPLY WITH END MARKER:
MAKE beg POINT TO ITS HEAD:
    beg := count := count + 1; pnt[beg] := 0; len := 1;
TEST: TO SEE WHETHER DESIRED LENGTH HAS BEEN
REACHED:
    if len < deslen then
    begin
      nextlen := if len < deslen − len then len
        else deslen − len;
INTRODUCE NEW CHAIN:
      hi := head(nextlen);
AND START MERGING:
FIND LEADING OBJECT OF MERGED CHAIN:
      lo := beg;
      if loafterhi then
      begin beg := hi; hi := lo; lo := beg end;
INITIALIZE CHAIN ON MECHANISM:
      link := lo;
CHAIN ON:
      lo := pnt[link];
TEST FOR END OF lo CHAIN:
      if lo ≠ 0 then
      begin
ADD LINK TO CHAIN:
        if loafterhi then
        begin
SWITCH LINK TO hi CHAIN:
          pnt[link] := link := hi; hi := lo
        end
        else
STEP DOWN IN lo CHAIN:
        link := lo;
        go to CHAIN ON
      end;
```

```
APPEND REMAINING TAIL:
        pnt[link] := hi;
        len := len + nextlen;
        go to TEST
      end;
      head := beg
   end head;
   count := 0; sort := head(n)
end sort;
```

Remark on Algorithm 426
Merge Sort Algorithm [M1]
[C. Bron, *Comm. ACM 15* (May 1972), 357–358]

C. Bron [Recd. 5 Nov. 1973]
Technological University of Twente, P.O. Box 217,
Enschede, The Netherlands

A remark in [3 p. 158] suggested to the author that Algorithm 426 needs only very minor modifications in order to handle the sorting of records that are chained to begin with. The algorithm then rearranges the chain and needs no additional array to store chaining information. Furthermore, the assumption that we should be able to associate each of the integers from 1 to *n* with each of the *n* records to be sorted is no longer necessary [2].

References

1. Bron, C. Algorithm 426, Merge Sort Algorithm. *Comm. ACM 15* (May 1972), 358.
2. Bron, C. An "In Situ" Merge Sort Algorithm. Tech. Note CB 64, Technological University of Twente, Enschede. The Netherlands.
3. Martin, W.A. Sorting. *Comp. Surv. 3* (1971), 147–174.

Algorithm 427
Fourier Cosine Integral [D1]

Peter Linz (8 June 1970, 3 Dec. 1970, and 11 Feb. 1971)
Department of Mathematics, University of California,
Davis, CA 95616

Key Words and Phrases: numerical integration, quadrature, adaptive quadrature, Filon quadrature, Fourier coefficients, Fourier integrals

CR Categories: 5.16

Description

The function *FRCOS* approximates

$$C(f, \omega) = \int_0^\infty f(t) \cos(\omega t)\, dt$$

by numerical evaluation of

$$C_T(f, \omega) = \int_0^T f(t) \cos(\omega t)\, dt.$$

The calling parameters for the function are:

1. *FC* is the name of the function subprogram, supplied by the user which computes $f(t)$. It is assumed that $f(t)$ is bounded in $[0, \infty)$ and is such that $\lim_{T\to\infty} C_T(f, \omega) = C(f, \omega)$.
2. *W* represents ω. It will normally be positive, although $\omega = 0$ will be handled correctly. In the latter case the algorithm reduces to an adaptive Simpson's rule. There is, however, some inefficiency in this because the cosine routine is used to compute *cos*(0.0) and some additional bookkeeping is done. The inefficiency may become significant if the time taken by the cosine routine is comparable to the time required to evaluate $f(t)$. The program will not work correctly for negative ω.
3. *T* should be chosen such that

$$C_T(f, \omega) = C(f, \omega)$$

within the required accuracy. The program actually evaluates $C_{TA}(f, \omega)$ where *TA* is chosen as follows:

(a) if $2^n 2\pi < \omega T \leq 2^{n+1} 2\pi$, for $n \geq -9$,
then $TA = 2^{n+1}2\pi/\omega$,
(b) if $\omega T \leq 2\pi/512$, then $TA = T$.

If an upper limit $2^n 2\pi$ is desired without adjustment, the *T* specified should be slightly smaller than this number (to avoid round-off error problems).

4. *ET* specifies the required (absolute) accuracy. The routine attempts to compute an answer which differs from $C_{TA}(f, \omega)$ by less than *ET*.
5. *HL* represents an upper limit on the stepsize; the integral over an interval is not considered to have converged unless the size of the interval is less than *HL*. Normally, *HL* can be chosen quite large, say $T/10$. However, when the integrand has a sharp peak, the choice of *HL* may be difficult. If it is chosen too large the peak may be missed altogether; if it is chosen small the computations become inefficient, since the limit is enforced everywhere. In such cases it might be preferable to use a variable *HL*, computed by means of a subprogram. *FRCOS* can be modified easily to do this.

The computations are done by means of an adaptive quadrature method described in detail in [1]. In summary, the approximate value of the integral over an interval $[a, b]$, denoted by $\hat{I}$, is computed as follows:

(1) If $b - a \leq \pi/256\omega$, Simpson's rule is used.
(2) If $\pi/256\omega < b - a < 2\pi/\omega$, Filon's method (referred to as *FILON* 2 below) is used. Here $\hat{I}$ is computed by

$$\hat{I} = h\{w_1 \cos(\omega a) + w_2 \sin(\omega a)\} f(a) + h w_3 \cos\left(\frac{\omega(a+b)}{2}\right) f\left(\frac{a+b}{2}\right) + h\{w_1 \cos(\omega b) - w_2 \sin(\omega b)\} f(b),$$

where

$$h = (b - a)/2,$$

$$w_1(\omega h) = \frac{1}{2h^2\omega^2}\left\{\cos(2\omega h) - \frac{4}{h\omega}\cos(\omega h)\sin(\omega h) + 3\right\},$$

$$w_2(\omega h) = \frac{1}{2h^2\omega^2}\left\{-\sin(2\omega h) + \frac{4}{h\omega}\sin^2(\omega h) - 2h\omega\right\},$$

$$w_3(\omega h) = \frac{4}{h^2\omega^2}\left\{\frac{1}{h\omega}\sin(\omega h) - \cos(\omega h)\right\}.$$

In the routine weights are needed only for $\omega h = \pi/2^p$, $p = 1, 2, \ldots, 9$. They have been precomputed to 14 significant digits and are stored in the arrays *W1C*, *W2C*, *W3C*, such that *W1C*(1) contains $w_1(\pi/2)$, *W1C*(2) contains $w_1(\pi/4)$, *W2C*(1) contains $w_2(\pi/2)$, etc. If higher accuracy is required the computation of the w's from the above formulas must be done with some care, since for small ωh large cancellation errors may occur. The use of multiple precision is recommended. Alternatively one may use the series expansions

$$w_1(\omega h) = \sum_{i=1}^{\infty} (-1)^i \frac{2^{2i-1}(2i-3)}{(2i+1)!}(\omega h)^{2i-2},$$

$$w_2(\omega h) = \sum_{i=1}^{\infty} (-1)^i \frac{2^{2i+3}i}{(2i+4)!}(\omega h)^{2i+1},$$

$$w_3(\omega h) = -\sum_{i=1}^{\infty} (-1)^i \frac{8i}{(2i+1)!}(\omega h)^{2i-2}.$$

(3) If $b - a = 2n\pi/\omega$, a special case of Filon's rule (called *FILON* 1) is used. Here

$$\hat{I} = \frac{4}{\omega^2(b-a)}\left\{f(a) - 2f\left(\frac{a+b}{2}\right) + f(b)\right\}.$$

The error is estimated by halving each interval and comparing the two estimates thus obtained. We denote by I the integral over $[a, b]$, by I_0 and I_1 the approximations with stepsize $(b - a)/2$ and $(b - a)/4$, respectively and write

$$I_0 = I + \epsilon_0,$$
$$I_1 = I + \epsilon_1.$$

If we know α such that

$$\epsilon_1 \simeq \alpha\epsilon_0,$$

then

$$\epsilon_1 \simeq \alpha(I_0 - I_1)/(1 - \alpha).$$

A given interval is split into parts until the estimated error is below a certain bound; once this is accomplished its contribution is added to the total integral and the next interval is considered. The error "allotted" to each interval depends on the size of the interval as well as on an estimate of the errors of all previously converged intervals.

The ratios α used in the error estimation are derived in [1]. The final expressions are:

(1) for Simpson's rule $\alpha = 1/16$,

(2) for *FILON* 1

$$\alpha = \frac{(b-a)^2/32 - 6/\omega^2}{(b-a)^2/8 - 6/\omega^2},$$

(3) for *FILON* 2

$$\alpha(\rho) = \frac{12\sin(\rho) - \rho^2\sin(\rho) - 6\rho - 6\rho\cos(\rho)}{12\sin(\rho) - 4\rho^2\sin(\rho) - 12\rho\cos(\rho)}$$

where $\rho = \omega(b-a)/2$.

For *FILON* 2 the α's are needed for $\rho = \pi/2^p$, $p = 1, 2, \ldots, 9$. They were precomputed and stored in the array *ER*, with *ER*(1) containing $\alpha(\pi/2)$, etc.

Computed values of *FC* are saved for later use, and it is possible that the space assigned for this purpose is exhausted before the computations are completed. In this case the routine returns with an error indication. (In the present implementation the value of *FRCOS* is set to 1.0×10^{30}, although this may be changed to suit the user.) Usually this occurs only if the routine is used improperly (e.g. *ET* has been specified so small that, due to round-off errors, the accuracy criterion cannot be met). While the assigned space appears to be adequate for most purposes, the user can easily change this by, say, doubling the sizes of the arrays *FS*, *PVAL*, and *AS*, and changing the overflow test.

The user should keep in mind that such an adaptive approach does not guarantee that the final answer has an error less than *ET*; accidental (false) convergence is always a possibility. While empirical evidence suggests that *FRCOS* is relatively immune to this, some examples of false convergence were encountered during the test of the algorithm. The user should always try to safeguard against this possibility, for example by making *ET* smaller than required, or by doing the computations twice with different values of *ET* and *HL*.

References

1. Linz, P. An adaptive quadrature algorithm for Fourier cosine integrals. (Unpublished manuscript available from author.)

Algorithm

```
      FUNCTION FRCOS(FC,W,T,ET,HL)
C   THIS ROUTINE COMPUTES THE FOURIER COSINE INTEGRAL FROM
C   ZERO TO INFINITY OF FC(T)*COS(W*T) BY AN ADAPTIVE
C   QUADRATURE METHOD USING A COMBINATION OF FILON AND
C   SIMPSON RULES
C       PARAMETERS
C   FC -MUST BE DECLARED EXTERNAL IN CALLING PROGRAM
C   W -VALUE MUST BE NON-NEGATIVE
C   T -UPPER LIMIT FOR QUADRATURE-SHOULD NORMALLY BE CHOSEN
C   SUCH THAT REST OF INTEGRAL IS NEGLIGIBLE. THE ACTUAL
C   LIMIT USED BY THE PROGRAM MAY BE SOMEWHAT LARGER THAN
C   THE GIVEN T(SEE INTRODUCTORY COMMENTS).
C   ET -REQUESTED ACCURACY(ABSOLUTE)
C   HL -LIMIT ON STEP SIZE-CONVERGENCE IN ANY SUBINTERVAL IS
C   NOT RECOGNIZED UNLESS SUBINTERVAL IS SMALLER THAN HL
      DIMENSION W1C(9),W2C(9),W3C(9),ER(9)
C   ARRAYS ER,W1C,W2C,W3C CONTAIN PRECOMPUTED CONSTANTS
C   NEEDED TO COMPUTE APPROXIMATE VALUES AND ERROR
C   ESTIMATES FOR FILON2(SEE COMMENTS).
      DATA ER(1),ER(2),ER(3),ER(4),ER(5),ER(6),ER(7),ER(8),
     $ ER(9)/    0.,.05061,.05969,.06181,.06233, .06246,
     $ .06249,.06249,.0625/
      DATA W1C(1),W1C(2),W1C(3),W1C(4),W1C(5),W1C(6),W1C(7),
     $       W1C(8),W1C(9)/
     $          4.0528473456934E-01,3.6761020369133E-01,
     $  3.4316760755741E-01,3.3587533234962E-01,
     $  3.3397411782348E-01,3.3349386085934E-01,
     $  3.3337348594489E-01,3.3334337278212E-01,
     $  3.3333584327653E-01/
      DATA W2C(1),W2C(2),W2C(3),W2C(4),W2C(5),W2C(6),W2C(7),
     $       W2C(8),W2C(9)/
     $          1.2059522143639E-01 , 1.9710810149097E-02,
     $  2.6328277852505E-03, 3.3459141708323E-04,
     $  4.1997086077777E-05,5.2550600306570E-06,
     $  6.5705211443498E-07,8.2136815416350E-08,
     $  1.0267267595664E-08/
      DATA W3C(1),W3C(2),W3C(3),W3C(4),W3C(5),W3C(6),W3C(7),
     $       W3C(8),W3C(9)/
     $  1.0320491018624,1.2528780015490,1.3128845799752,
     $  1.3281999871557,1.3320486700792,1.3330180847949,
     $ 1.3332530160151,1.3333132536798,1.3333283133997/
      DIMENSION FS(61),PVAL(30),AS(30)
C   ARRAYS FS,PVAL,AS ARE STORAGE FOR SAVED VALUES OF F
C   AND BOOK-KEEPING
      DATA PI2,PI256/6.2831853071796,.0122718463/
C   PI2=2*PI, PI256=PI/256
      DATA ALN2,ERC,ROC/.69314718,1.E+30,1.E-5/
C   ALN2=NATURAL LOG OF 2,ERC=ERROR VALUE RETURNED
C   BY FRCOS,ROC=CONSTANT USED TO ELIMINATE ROUNDOFF
C   PROBLEMS IN COMPUTING INTERVAL LIMITS
      EPS= ET
      VAL= 0.
      N= 1
      AS(1)=0.
      FS(1)= FC(0.)
      PVAL(1)=ERC
C   TEST IF UPPER LIMIT ADJUSTMENT IS NECESSARY
      WT=W*T
      IF(WT-PI256+ROC   ) 100,100,101
C   NOTE-CONSTANT ROC=1.E-5 USED THROUGHOUT PROGRAM TO
C   ELIMINATE EFFECT OF FLOATING POINT ROUNDOFF ERROR
C   SET UP FIRST INTERVAL FOR SIMPSON RULE
  100 FS(2)=FC(.5*T)*COS(.5*WT)
      FS(3) = FC(T)* COS(WT)
      B=T
      GO TO 105
C   ADJUST UPPER LIMIT
  101 NP= IFIX(ALOG(WT/PI256)/ALN2)+1
      TA=  2**NP* PI256/W
      B=TA
C   SET UP FIRST INTERVAL FOR FILON RULE
      FS(2)= FC(.5*TA)
      FS(3)= FC(TA)
C   TAKE LAST INTERVAL FROM LIST
  105 A=AS(N)
      HI=B-A
      WHI=W*HI
      N2=2*N
      F1= FS(N2-1)
      F2= FS(N2)
      F3= FS(N2+1)
      XQ= B-.75*HI
      XQ3 = B-.25*HI
C   TEST TO DETERMINE WHICH QUADRATURE RULE IS APPLICABLE
      IF( WHI   - PI256 -ROC   ) 110,110,111
  110 IF( WHI   - PI256 +ROC   ) 200,200,201
  111 IF( WHI   - PI2   -ROC   ) 220,220,230
C   ESTIMATE BY SIMPSON RULE
  200 FQ= FC(XQ)*COS(W*XQ)
      FQ3=FC(XQ3)*COS(W*XQ3)
      VNEW1= HI*(F1+4.*FQ+F2)/12.
      VNEW2= HI*(F2+4.*FQ3+F3)/12.
      VNEW= VNEW1+VNEW2
      ERR= (PVAL(N)-VNEW)/15.
      GO TO 300
C   SWITCH FROM FILON TO SIMPSON RULE
  201 F1 = F1* COS(W*A)
      F2 = F2* COS(W*(B-.5*HI))
      F3 = F3* COS(W*B)
      PVAL(N)= HI*(F1+4.*F2+F3)/6.
      GO TO 200
C   ESTIMATE BY FILON2
  220 H=.25*HI
      FQ= FC(XQ)
      FQ3= FC(XQ3)
      NH= IFIX(ALOG(PI2/WHI)/ALN2+ROC)+1
      W1= W1C(NH)
      W2=-W2C(NH)
      W3= W3C(NH)
      WA=W*A
      WA1=W*(B-.5*HI)
      WB=W*B
      CO1= COS(WA1)
      SI1= SIN(WA1)
      VNEW1 = H*((W1*COS(WA)+W2*SIN(WA))*F1 + W3*COS(W*XQ)*
     $           FQ+(W1*CO1-W2*SI1)*F2)
      VNEW2 = H*((W1*CO1 +W2*SI1)*F2 + W3*COS(W*XQ3)*FQ3
     $          +(W1*COS(WB) -W2*SIN(WB))*F3)
      VNEW=VNEW1+VNEW2
      ERT= ER(NH)
      ERR = ERT*(PVAL(N)-VNEW)/(1.-ERT)
C   SKIP CONVERGENCE TEST IF INTERVAL= ONE PERIOD
      IF(WHI- PI2+ ROC   ) 300,300,400
C   ESTIMATE BY FILON1
  230 FQ=FC(XQ)
      FQ3=FC(XQ3)
      W2=W*W
      CONST=8./(W2*HI)
      VNEW1= CONST*(F1-2.*FQ+F2)
      VNEW2= CONST*(F2-2.*FQ3+F3)
      VNEW= VNEW1+VNEW2
      W2=6./W2
      W3=HI*HI
      ERT=(W3/32.-W2)/(W3/8.-W2)
      ERR= ERT*(PVAL(N)-VNEW)/(1.-ERT)
C   CONVERGENCE TEST
C   SKIP CONVERGENCE TEST IF HI.GT.HL
  300 IF(HI- HL) 301,301,400
  301 IF(ABS(ERR)-EPS*HI/8) 500,500,400
C   CONVERGENCE NOT OBTAINED -SPLIT INTERVAL AND ADD TO LIST
```

```
C  TEST FOR POSSIBLE LIST OVERFLOW
  400 IF(N-30) 401,600,600
  401 FS(N2+3)= F3
      FS(N2+2)= FO3
      FS(N2+1)= F2
      FS(N2)= FO
      AS(N+1)=A+.5*HI
      PVAL(N)=VNEW1
      PVAL(N+1)=VNEW2
      N=N+1
      GO TO 105
C  CONVERGENCE OBTAINED -ADD FXTRAPOLATED PARTIAL SUM TO
C  TOTAL--ADJUST ERROR AND INTERVAL
  500 VAL= VAL +VNEW-ERR
      EPS = EPS-ABS(ERR)
      N=N-1
      B=A
      IF(N) 700,700,105
C  CONVERGENCE FAILURE -ROUTINE RETURNS ERC=1.E+30
C  OPTIONAL ERROR MESSAGE MAY BE INSERTED HERE
  600 FRCOS=ERC
      RETURN
C  COMPUTATIONS COMPLETED SUCCESSFULLY
  700 FRCOS= VAL
      RETURN
      END
```

Algorithm 428

Hu-Tucker Minimum Redundancy Alphabetic Coding Method [Z]

J.M. Yohe* [Recd. 2 January 1970, 12 February 1971, and 21 June 1971]
Mathematics Research Center, University of Wisconsin, Madison, WI 53706

Key Words and Phrases: information theory, coding theory, Hu-Tucker method, minimum redundancy coding
CR Categories: 5.6

Description

This algorithm implements the Hu-Tucker method of variable length, minimum redundancy alphabetic binary encoding [1]. The symbols of the alphabet are considered to be an ordered forest of n terminal nodes. Two nodes in an ordered forest are said to be tentative-connecting if the sequence of nodes between the two given nodes is either empty or consists entirely of nonterminal nodes.

An interval of nodes each pair of which is a tentative-connecting pair is called a tentative connecting string.

Given an ordered forest, we create a new ordered forest with one less tree by combining a pair of tentative-connecting nodes i_1, i_2 such that $Q[i_1] + Q[i_2]$ is minimal. Such a pair is said to have minimal weight sum. The old nodes i_1 and i_2 are eliminated, and the new node replaces the first of the former nodes in the ordering. Its weight is the sum of the weights of the former nodes.

The original forest will, after a finite number of steps, be connected into a single tree. This tree will not, in general, satisfy the order-preserving requirement. However, it is shown in [1] that the path lengths are feasible for the construction of a tree which does satisfy this requirement and is, moreover, minimal in cost.

The present procedure finds a minimal cost tree whose longest path length and total path length are minimal. This was done for the nonalphabetic case by Schwartz [3], and his work carries over directly to the alphabetic case by virtue of the fact that any optimal alphabetic encoding can be constructed by the Hu-Tucker method, simply by modifying the choice of which tentative-connecting nodes are combined. This procedure therefore represents a modification of the Hu-Tucker algorithm to incorporate these ideas of Schwartz.

During the procedure, the array L is used to determine which roots are tentative-connecting. If L is initially filled with 1's instead of 0's, any pair of nodes will be considered tentative-connecting, and the procedure will implement Huffman's method [2], giving the same results as the "bottom merging" method of Schwartz and Kallick [4]. This is because this procedure picks, among those pairs with minimal weight sum, the first pair having minimal length sum.

Modifying the procedure to pick the first pair having maximal length sum would be equivalent to the "top merging" method of Schwartz and Kallick, and would maximize the total number of digits and the maximal length of the code in alphabetic case (and in the nonalphabetic case, if the L-array is initially filled with 1's).

The decision tree may be obtained from the branch lengths by combining the first node of maximal path length with the second node of maximal path length to form a new node with path length one less than that of the original nodes, iterating the procedure until only one node (the root) remains. The code can then be constructed by assigning the value 0 to the first node on the next level from the root and 1 to the second node, appending 0 or respectively 1 to the ith level encoding of a node to obtain the encoding for the first or second son on the $(i + 1)$-th level.

References

1. Hu, T.C., and Tucker, A.C. Optimal computer search trees and variable-length alphabetical codes. *SIAM J. Appl. Math.* (to appear).
2. Huffman, David A. A method for the construction of minimum-redundancy codes. *Proc. I.R.E. 40* (1952), 1098–1101.
3. Schwartz, Eugene S. An optimum encoding with minimum longest code and total number of digits. *Inform. Contr.* 7 (1964), 37–44.
4. Schwartz, Eugene S., and Kallick, Bruce. Generating a canonical prefix encoding. *Comm. ACM* 7 (1964), 166–169.

Algorithm

```
procedure Hutree (n, Q, L);
  value n; integer n; integer array Q, L;
comment n is the number of symbols in the alphabet, and Q is a
  vector of length n. Q[i] is the weight to be attached to the ith
  symbol in the alphabet.
    The output of the procedure is the vector L of length n. L[i] is
  the length of the path to the ith symbol of the alphabet in a tree of
  minimal cost (i.e. the sum of the Q[i] × L[i] is minimal) which has
  the further property that, subject to minimality of cost, the sum of
  the L[i] and max L[i] are minimal;
begin
  integer maxn, m, i;
  integer array P[1 : n], s[1 : n − 1], d[1 : n − 1];
  comment P is used to hold the weights of the trees in the ordered
    forest, beginning with the alphabet at the start of the procedure
    and ending with the tree at the conclusion of the procedure. L is
    used during the procedure to hold information relating to the
    length sums. At the conclusion of the procedure, L is used to
    return the path lengths.
      If i1 < i2 and nodes i1 and i2 are connected on the mth pass
    through the body of the algorithm, then P[i1] will be set equal
    to P[i1] + P[i2], which is the weight of the new node, and P[i2]
    will be set to zero to indicate that node i2 is no longer a partici-
    pating node. L[i1] is set equal to L[i1] + L[i2] + 1, which is one
    less than the number of terminal nodes which are descended
    from the new node. This number is also one less than the incre-
    ment to the total path length which would result from connect-
    ing the new node i1 in a subsequent pass through the body of
    the algorithm. The value of L[i2] is irrelevant during the re-
    mainder of the procedure. The s and d vectors are used to
    record connections of tentative-connecting nodes. s[m] is set to
    i1 , which is both the ordered position of the leftmost node and
```

* Sponsored by the United States Army under Contract No.: DA-31-124-ARO-D-462.

the ordered position of the new node, and *d*[*m*] is set to *i*2, which is the ordered position of the rightmost node.

The variable *maxn* is set to a number which is larger than the sum of the elements of *Q*.

The following simple example should be of some assistance in understanding the procedure. Assume the procedure is called with $n = 5$ and $Q = (3, 1, 1, 1, 3)$. The evolution of the vectors *P*, *L*, *s*, and *d* is shown in the following table. Values which are not relevant are indicated by dashes.

m	0	1	2	3	4
P[1]	3	3	3	6	9
P[2]	1	2	3	3	0
P[3]	1	0	0	0	0
P[4]	1	1	0	0	0
P[5]	3	3	3	0	0
L[1]	0	0	0	1	4
L[2]	0	1	2	2	–
L[3]	0	–	–	–	–
L[4]	0	0	–	–	–
L[5]	0	0	0	–	–
s[*m*]		2	2	1	1
d[*m*]		3	4	5	2;

```
maxn := 1;
for i := 1 step 1 until n do
begin
  L[i] := 0; P[i] := Q[i];
  maxn := maxn + Q[i];
end
comment Since there are n terminal nodes in the original forest, we
  must make exactly n − 1 connections. On each pass through
  the body of this procedure we will determine the next optimal
  connection. We initialize by setting the minimum weight to a
  large value to insure that any valid connection chosen will
  replace the bogus connection initially indicated;
for m := 1 step 1 until n − 1 do
begin
  integer j, j1, min1, minL1, j2, min2, minL2, pt, pmin, sumLt,
    sumL, i1, i2;
  i := 0;
  pmin := maxn;
B1:
  i := i + 1;
  comment At B2 we begin our scan of the next tentative-connect-
    ing string to find the most desirable pair in the string. If
    necessary, we skip over any previously absorbed nodes. We
    initialize the most desirable node to the first in the tentative-
    connecting string, and the record of the second most desirable
    node is initialized to reflect a very large minimum. This
    insures that any participating node will be more desirable and
    that valid information will replace the bogus information as
    soon as the next participating node is encountered. If the
    first participating node is the last node in the forest, or if no
    further nodes are participating nodes, then we have com-
    pleted our scan for the next tentative-connecting pair and we
    go to E1 to make the optimal connection;
B2:
  if i1 ≥ n then go to E1 else
  if P[i] = 0 then go to B1;
  min2 := maxn;
  j1 := i;
  minL1 := L[i]; min1 := P[i];
  comment We now begin our scan of all remaining nodes in the
    current tentative-connecting string. The string will end as
    soon as we have examined a participating node which has not
    previously been combined. The purpose of this scan is to
    locate the optimal tentative-connecting pair in the tentative-
    connecting string. The optimal pair is defined to be that pair
    with minimal weight and minimal length sum which occurs
    first in the tentative-connecting string;
  for j := i + 1 step 1 until n do
  begin
  comment We check for P[j] > 0 to see whether the jth node is a
    participating node. If P[j] = 0, the node has previously been
    absorbed and we pass over the empty space;
  if P[j] > 0 then
    begin
      if P[j] < min1 ∨ (P[j] = min1 ∧ L[j] < minL1) then
      begin
      comment If the jth node is "more desirable" than either of
        the previously most desirable tentative-connecting nodes,
        we record the previous most desirable node as the second
        most desirable node and record the jth node as being
        most desirable;
        min2 := min1; j2 := j1; minL2 := minL1;
        min1 := P[j]; j1 := j; minL1 := L[j];
      end
      else if P[j] < min2 ∨ (P[j] = min2 ∧ L[j] < minL2) then
      begin
      comment If the jth node was not more desirable than the
        previous most desirable node, but is more desirable
        than the previous second most desirable node, we record
        the jth node as being second most desirable;
        min2 := P[j]; j2 := j; minL2 := L[j];
      end;
      if L[j] = 0 then go to E2;
      comment If L[j] = 0 then we have reached the end of the
        current tentative-connecting string, and we have found
        the most desirable pair in that string. We now go to
        compare it with the previous most desirable pair in the
        forest;
    end
  end;
E2:
  pt := P[j1] + P[j2];
  sumLt := L[j1] + L[j2];
  comment We have now found the next tentative-connecting
    pair, namely the j1 and j2 nodes. Here, we test this new pair
    against the previous minimal pair to see whether the new pair
    is more desirable. The new pair is more desirable if its weight is
    less than that of the previous pair, or if its weight is equal to
    that of the previous pair and its length sum is smaller;
  if pt < pmin ∨ (pt = pmin ∧ sumLt < sumL) then
  begin
    pmin := pt;
    i1 := j1; i2 := j2;
    sumL := sumLt;
  end;
  comment The next tentative-connecting string begins with the
    last participating node in the current tentative-connecting
    string. Hence we replace i by j and return to B2 to begin
    processing the next tentative-connecting string;
  i := j;
  go to B2;
  comment Upon reaching E1 the procedure has scanned all
    tentative-connecting pairs and the decision has been made to
    connect nodes in order positions i1 and i2. We switch i1 and
    i2 if necessary to insure that i1 < i2. We record the connec-
    tion by setting s[m] := i1 and d[m] := i2. The weight of the
    new node is placed in the weight table in position i1 (the
    order position of the new node). The weight in the order
    position of the second combined node is set to zero to indi-
    cate that the node has now been absorbed and no longer
    participates in the scan. L[i1] is set to one less than the incre-
    ment to the path length sum which would result from con-
    necting the new node;
```

```
E1:
    if i1 > i2 then
    begin
      j1 := i1; i1 := i2; i2 := j1;
    end;
    s[m] := i1; d[m] := i2;
    P[i1] := pmin; P[i2] := 0;
    L[i1] := sumL + 1;
  end;
  comment s[n − 1] gives the ordered location of the root of the
    tentative tree. We now generate the path lengths as follows:
    the path length to the root is zero, and if the path length to any
    node is i, then the path length to each of its sons is i + 1. The
    two sons of the node whose order position is given in s[m] lie in
    the order positions given in s[m] and d[m]. Moreover, if an
    order position is given in s[m] for m < n − 1 then that order
    position must be listed in s[j] or d[j] for some j > m, so the path
    lengths obtained by this algorithm are well defined.
      Returning to our example, we now trace the construction of
    the vector of path lengths. This is shown in the following table.
    For the sake of clarity, the vectors s and d are now shown in
    reverse order.

    m          4  3  2  1
    L[1]   0   1  2  2  2
    L[2]   -   1  1  2  3
    L[3]   -   -  -  -  3
    L[4]   -   -  -  2  2
    L[5]   -   -  2  2  2
    s[m]       1  1  2  2
    d[m]       2  5  4  3

      Thus the final value of the vector L is (2, 3, 3, 2, 2);
  L[s[n − 1]] := 0;
  for m := n − 1 step −1 until 1 do
  L[s[m]] := L[d[m]] := L[s[m]] + 1;
end;
```

Remark on Algorithm 428 [Z]

Hu-Tucker Minimum Redundancy Alphabetic Coding Method [J.M. Yohe, *Comm. ACM 15* (May 1972), 360–362]

J.G. Byrne [Recd. 26 June 1972] Department of Computer Science, Trinity College, Dublin 2, Ireland

Algorithm 428 was translated into Basic Fortran IV and run on IBM System 360/44 running under *RAX*. When the line just after the label *B*2:

if $i1 > n$ **then go to** $E1$ **else**

was changed to

if $i > n$ **then go to** $E1$ **else**

the algorithm gave correct results for the example given and for the example in Gilbert and Moore [1]. In the latter case the cost defined as

$$\frac{\sum_{I=1}^{N} Q(I)*L(I)}{\sum_{I=1}^{N} Q(I)}$$

and code lengths were correct.

When the *L* array was set to 1's on entry, the optimum (Huffman) codes were obtained, and they were the same as those given by the Schwartz and Kallick [2] method as claimed in the author's description.

Table I.

Size of alphabet	10	27	60
Time to find optimum alphabetic codes (secs)	0·02	0·14	0·62
Time to find optimum codes (secs)	0·02	0·08	0·34

Table I, which gives the cpu time required, shows that the algorithm is very fast for small alphabets and that the time is approximately proportional to n^2, as expected.

References

1. Gilbert, E.N., and Moore, E.F. Variable length binary encodings. *Bell Systems Tech. J. 38* (1959), 933–968.
2. Schwartz, E.S., and Kallick, B. Generating a canonical prefix encoding. *Comm. ACM 7* (Mar. 1964), 166–169.

Algorithm 429

Localization of the Roots of a Polynomial [C2]

W. Squire (Recd. 16 Mar. 1970, 2 June 1971, and 4 Oct. 1971)
College of Engineering, Dept. of Aerospace Engineering, West Virginia University, Morgantown, WV 26506

Key Words and Phrases: polynomials, roots of polynonials, theory of equations, Routh-Hurwitz criterion
CR Categories: 5.15
Language: Fortran

Description

This algorithm provides information about the roots of the polynomial

$$x^n + c_1x^{n-1} + c_2x^{n-2} + \cdots + c_n\,. \qquad (1)$$

The theorem [1] that the roots of (1) are all inside a ring of radius

$$1 + \max_{1\leq i\leq n} |C_i|$$

is embodied in the Fortran function *RADIUS*. By applying this to the original polynomial and to the polynomial

$$y^n + \frac{c_{n-1}}{c_n}\, y^{n-1} + \frac{c_{n-2}}{c_n}\, y^{n-2} + \cdots + \frac{1}{c_n} \qquad (2)$$

the inner and outer radii of an annulus containing all the roots are determined.

The theorem [1] that the positive real roots of (1) are less than

$$1 + \left[\max_{1\leq i\leq n} |C_i| \right]^{1/m}$$

where m is the subscript of the first negative coefficient is embodied in *RADIUS*. If there are no negative coefficients there cannot be any positive roots and *RADIUS* returns zero in this case. By applying *RADIUS* to both (1) and (2) upper and lower bounds are obtained for the positive roots. In some cases (all coefficients positive) it is possible to say that there are no real positive roots, but the converse does not hold so that the determination of bounds does not guarantee the existence of a real root between those bounds. *RADIUS* is also applied to the equations whose roots are the negatives and negative reciprocal of the roots of (1) to obtain similar results for the negative real roots.

The Fortran function *HRWTZR* employs a modification of the Routh-Hurwitz criterion [2] to determine whether (1) and the equation whose roots are the negatives of those of (1) have any roots with positive real parts. Unfortunately a zero real part is considered positive so that this test will not determine if an equation nas purely imaginary roots.

The subroutine POLYAN, which computes the coefficients for the modified polynomials, calls the functions, and prints out suitable messages, has for its arguments:

1. An N element array C which contains the coefficients of the polynomial except for the leading 1.
2. An auxiliary N element array CM in which the coefficients of the modified polynomials are stored as needed.
3. N is equal to the degree of the polynomial.

If desired the argument list can be extended to include the various bounds so that they can be transmitted back to the main program for use.

References

1. Berezin, I.S., and Zhidkov, N.P. *Computing Methods*. Vol. II, Ch. 7. Pergamon Press, New York, 1965.
2. Sherman, S., Di Paola, J., and Frissel, H.F. The simplification of flutter calculation by the use of an extended form of the Routh discriminant. *J. Aeronaut. Sci. 12* (1945), 385–392.

Algorithm

```
      SUBRØUTINE PØLYAN(C,CM,N)
C     PØLYAN ØBTAINS INFØRMATIØN ABØUT THE LØCATIØN
C     ØF THE RØØTS ØF A PØLYNØMIAL BY USING
C     BØUND,RADIUS,AND HRWTZR
C     C IS A N ELEMENT ARRAY CONTAINING THE CØEFICIENTS
C     NØRMALIZED  SØ THAT THE LEADING CØEFFICIENT(WHICH
C     IS NØT INCLUDED IN C) IS +1.0
C     CM IS A WØRKING ARRAY THE SAME SIZE AS C
C     N=DEGREE ØF PØLYNØMIAL
C
      DIMENSIØN C(N),CM(N)
      LØGICAL HRWTZR
C      TEST FØR ZERØ RØØT
      IF(C(N).EQ.0.0) GO TØ 50
C     CØEFFICIENTS FØR RECIPRØCAL PØLYNØMIAL ARE PUT IN CM
      CM(N)=1./C(N)
      NM1=N-1
      DØ 5 I=1,NM1
      NI=N-I
 5    CM(I)=CM(N)*C(NI)
      RØUT=RADIUS(C,N)
      RIN=1./RADIUS(CM,N)
      WRITE(6,201) RIN, RØUT
 201  FØRMAT(40H RØØTS ARE IN AN ANNULUS ØF INNER RADIUS,
     1E10.3,17H AND ØUTER RADIUS,E10.3)
      RPU=BØUND(C,N)
      IF(RPU.NE.0.0) GØ TØ 10
      WRITE(6,202)
 202  FØRMAT(33H THERE ARE NØ REAL PØSITIVE RØØTS)
      GØ TØ 20
 10   RPL=1./BØUND(CM,N)
      WRITE(6,203) RPL, RPU
 203  FØRMAT
     1(40H THE PØSITIVE RØØTS(IF ANY) ARE BETWEEN,
     2E10.3,4H AND,E10.3)
C     CØEFFICIENTS FØR NEGATIVE RECIPRØCAL ARE PUT IN CM
 20   DØ 25 I=1,N,2
 25   CM(I)=-CM(I)
      RNU=BØUND(CM,N)
      IF(RNU.NE.0.0)GO TØ 30
      WRITE(6,204)
 204  FORMAT
     1(33H THERE ARE NØ NEGATIVE REAL RØØTS)
      GØ TØ 40
C     COEFFICIENTS FØR NEGATIVE RØØTS ARE PUT IN CM
 30   X=-1.0
      DØ 35 I=1,N
      CM(I)=X*C(I)
  35  X=-X
      RNU=-1./RNU
      RNL=-BØUND(CM,N)
      WRITE(6,205) RNU, RNL
 205  FØRMAT
     1(44H THE REAL NEGATIVE RØØTS(IF ANY)ARE BETWEEN,
     2E10.3,4H AND,E10.3)
 40   IF(HRWTZR (C,N)) WRITE(6,206)
 206  FØRMAT
     1(44H THERE ARE NØ RØØTS WITH PØSITIVE REAL PARTS)
      IF(HRWTZR (CM,N)) WRITE(6,207)
 207  FØRMAT
     1 (44H THERE ARE NØ RØØTS WITH NEGATIVE REAL PARTS)
      RETURN
 50   WRITE(6,208)
 208  FØRMAT (41H PØLYNØMIAL HAS A ZERO RØØT.REDUCE DEGREE)
      RETURN
      END
      FUNCTIØN RADIUS(C,N)
C     RADIUS RETURNS AN UPPER LIMIT FØR THE  MØDULUS
C     ØF THE RØØTS ØF AN N DEGREE PØLYNØMIAL.
C
      DIMENSIØN C(N)
      RADIUS=ABS(C(1))
      DØ 10 I=2,N
 10   IF(ABS(C(I)).GT.RADIUS) RADIUS=ABS(C(I))
      RADIUS=1.+RADIUS
      RETURN
      END
      FUNCTIØN BØUND(C,N)
C     BØUND RETURNS AN UPPER LIMIT FØR THE
C     PØSITIVE REAL RØØTS ØF AN N DEGREE PØLYNØMIAL
C
      DIMENSIØN C(N)
      M=0
      BØUND=0.0
      DØ 10 I=1,N
      IF(M.GT.0) GØ TØ 10
      IF(C(I).LT.0.0) M=I
 10   IF(C(I).LT.BØUND) BØUND=C(I)
      IF(M.EQ.0) RETURN
      BØUND=1.+(-BØUND)**(1./FLØAT(M))
      RETURN
      END
      LØGICAL FUNCTIØN HRWTZR(C,N)
C     HRWTZR RETURNS .TRUE. IF ALL THE RØØTS HAVE
C     NEGATIVE REAL PARTS,ØTHERWISE.FALSE.IS RETURNED.
C     IF A REAL PART IS ZERØ,THEN .FALSE. IS RETURNED.
C
      DIMENSIØN C(N)
      HRWTZR=.FALSE.
      C1=C(1)
      IF(C1.LE.0.0)RETURN
      M=N-1
      DØ 30 I=1,M
      KM=N-I
      DØ 20 K=1,KM
      C(K)=C1*C(K+1)
      IF(K.EQ.KM.ØR.2*(K/2).EQ.K) GØ TØ 20.
      C(K)=C(K)-C(K+2)
 20   C(K)=C(K)/C1
      C1=C(1)
      IF(C1.LE.0.0) RETURN
 30   CONTINUE
      HRWTZR=.TRUE.
      RETURN
      END
```

Remark on Algorithm 429 [C2]

Localization of the Roots of a Polynomial [W. Squire, *Comm. ACM 15* (Aug. 1972), 776]

Edward J. Williams [Recd. 15 Sept. 1972] Computer Science Department, Ford Motor Company, P.O. Box 2053, Dearborn, MI 48121

Corrections are needed in the third paragraph. The theorem that the positive real roots of (1) are less than

$1 + [\max_{1 \leq i \leq n} | Ci |]^{1/m}$. . . should read

$1 + [\max_{1 \leq i \leq n\ ci<0} | Ci |]^{1/m}$

Further, the four words "*RADIUS*" in this paragraph should be replaced by "*BOUND*".

References

1. Zaguskin, O.O. *Solution of Algebraic and Transcendental Equations*, Pergamon Press, New York, 1961, p. 21.

Remark on Algorithm 429 [C2]

Localization of the Roots of a Polynomial [C2]
[W. Squire, *Comm. ACM 15* (Aug. 1972), 776–777]

H.B. Driessen and E.W. LeM. Hunt [Recd. 13 Oct. 1972, 29 Jan. 1973]

Supreme Headquarters Allied Powers of Europe, Technical Center, P.O. Box 174, The Hague, The Netherlands

There seems to be an error in this algorithm. If we take the polynomial:

$z^4 + a_2z^2 + a_3z^3 + a_4z + a_5 = 0,$

then after the second pass through the *K*-loop of the logical function *HRWTZR*(*C*, *N*), the term $(a_2a_3-a_4)a_4 - a_5a_2$ is tested for a minus sign. However, the term which should be tested according to the Routh-Hurwitz criterion is $(a_2a_3-a_4)a_4 - a_5a_2^2$. If this term is negative then there are no roots with positive real parts.

As an example, if the polynomial

$z^4 + 5.6562\,z^3 + 5.8854\,z^2 + 7.3646\,z + 6.1354 = 0$

is studied with the help of Algorithm 429 one will find as output:

Roots are in an annulus of inner radius .454 $E + 00$ and outer radius .836 $E + 01$;

There are no real positive roots;

The negative roots (if any) are between $-.454\ E + 00$ and $-.836\ E + 01$;

There are no roots with positive real parts.

However, if one calculates the roots of this equation, one will find approximately:

$z_1 \;\; = -1.0001$

$z_2 \;\; = -4.7741$

$z_{3,4} = +0.0089 \pm 1.1457\,i$

Statement 20 + 1 in the logical function *HRWTZR*(*C*,*N*), which was originally "*C*1 = *C*(1)", should be amended to read "*C*1 = *C*(1)/*C*1".

As a by-product of our investigation, it turns out that the structure of the logical function *HRWTZR* can be simplified by abandoning the logically redundant steps *C*(*K*) = *C*(*K*+1).

The following listing incorporates both the correction and the simplifications. The function has been parameter tested on a CDC-6400.

```
      LOGICAL FUNCTION HRWTZR (C,N)
      DIMENSION C(N)
      HRWTZR = .FALSE.
      IF (C(1) .LE.0..OR.C(N).LE.0.) RETURN
      C1 = C(1)
      M = N − 1
      DO 30 I = 2,M
      DO 20 K = I,M,2
 20   C(K) = C(K) − C(K+1)/C1
      C1 = C(I)/C1
      IF (C1.LE.0.) RETURN
 30   CONTINUE
      HRWTZR = .TRUE.
      RETURN
      END
```

Algorithm 430

Immediate Predominators in a Directed Graph [H]

Paul W. Purdom Jr.* and Edward F. Moore [Recd. 14 Aug. 1970 and 13 July 1971]
Computer Sciences Department, University of Wisconsin, Madison, WI 53706

Key Words and Phrases: predominator, immediate predominator, graph theory, directed graph, shortest path, articulation, connectivity, program optimization, optimizing compiler

CR Categories: 4.12, 5.32

Language: Algol

Description

We assume a directed graph whose nodes are labeled by integers between 1 and n. The arcs of this graph correspond to the flow of control between blocks of a computer program. The initial node of this graph (corresponding to the entry point of the program) is labeled by the integer 1. For optimizing the object code generated by a compiler, the relationship of immediate predominator has been used by Lowry and Medlock [3]. We say that node i predominates node k if every path from node 1 to node k passes through (i.e. both into and out of) node i. Node j is an immediate predominator of node k if node j predominates node k and if every other node i which predominates node k also predominates node j. It can easily be proved that if $k \neq 1$ and node k is reachable from node 1t hen node k has exactly one immediate predominator. In case $k = 1$, or node k is not reachable from node 1, the immediate predominator of node k is undefined, and the value 0 will be given by the procedure *PREDOMINATOR*.

The input to this procedure is described for clarity of exposition as the adjacency matrix M of the directed graph.

It is assumed that there is a known bound a such that the number q of arcs in the directed graph satisfies $q \leq a$.

Both the machine time and the memory required to perform this procedure are related in a simple way to the number n of nodes and the number q of arcs of the given graph. If T is the length of time required to perform the procedure *PREDOMINATOR*, then T is bounded by

$$T \leq k_1 n^2 + k_2 nq + k_3 n + k_4 q + k_5 ,$$

where the k_i are constants depending on the machine used for the procedure. If S is the memory required to perform the procedure *PREDOMINATOR*, then S is given exactly by

$$S = k_6 n^2 + k_7 a + k_8 n + k_9 .$$

The $k_6 n^2$ term is merely the memory required to store the adjacency matrix M which is used to give the input description of the graph. The description of the graph is first transformed into a linked list, and no further use is made of the Boolean array M. If this procedure were incorporated into an optimizing compiler, the adjacency matrix should be eliminated, going directly from the source program into the list form, saving the memory used to store the adjacency matrix M, which would remove the $k_6 n^2$ term from the memory required, as well as decreasing the computing time required. The precise details of the list representation can be expressed in a more brief and unambiguous manner by a few lines of Algol than by an English description. The predominators of any given node can be computed as in [3] from the immediate predominators, and the articulation points of a graph are the predominators of the exit node.

In an article on program optimization, Allen [1] gives an algorithm for computing articulation points (which are the predominators of the exit node). To test if node i is an articulation point, he removes node i, from the graph, and computes the transitive closure to see if the exit node is connected to the entry node. By successively considering each node as an exit node, his algorithm can be adopted to computing the predominators (from which immediate predominators can be quickly computed) in a time proportional to n^2 times the time required to compute the transitive closure. Since the transitive closure takes between n^2 and n^3 operations to compute [4, 5, and 6], Allen's algorithm would be slower than the one presented here by at least a factor of n for large problems.

The procedure *PREDOMINATOR* depends for its speed on the use of an algorithm first proposed by Dijkstra [2] for finding the shortest path between two points in a graph. The basic idea of the method is that a tree is found which is rooted on the entry node and which includes each node in the graph which can be reached from the entry node. Any node which cannot be reached from the entry node does not have an immediate predominator. Each node which can be reached from the entry node has the entry node as a predominator. It is the immediate predominator unless the node has a predominator which is closer to it along the path which was used to reach it. To test if a node i, other than the entry node, is a predominator of some nodes, a test is made to see which nodes below (further from the root) i cannot be reached from the remaining nodes in the tree without going through i. The nodes which cannot be reached without going through i have i as a predominator. Using this method the entry *Immediate*[i] is set to the various predominators of node j. The calculation is, however, organized to start at the root of the tree and proceed to the leaves, so that the last value of *Immediate*[j] contains the immediate predominator of j.

The program was tested on 38 graphs including one with 36 nodes and 49 arcs which represents the flowchart of the algorithm and one with 82 nodes and 125 arcs which represents the flowchart of a Fortran program. The running time of the program on a Burroughs B5500 was 0.6 sec for the 36 node graph and 3.8 sec for the 82 node graph. The longest time for the remaining graphs was 0.5 sec for a graph with 18 nodes and 48 arcs. The shortest time was 0.07 sec for graphs with two nodes and one arc, five nodes and 25 arcs, and five nodes and 21 arcs. While these numbers are useful for estimating the average running time of the program, they are of limited use in calculating the constants in the formula for the running time, because the formula gives only an upper limit on the running time.

References

1. Allen, F.E. Program optimization. *Annual Rev. in Automatic Programming 5* (1969), 239–307.
2. Dijkstra, E.W. A note on two problems in connexion with graphs, *Numerische Mathematik 1*, 5 (Oct. 1959), 269–271.
3. Lowry,. Edward S. and Medlock, C. W. Object code optimization, *Comm. ACM 12*, 1 (Jan. 1969), 13–22.

*Present address: University of Indiana, Department of Computer Science, Bloomington, IN 47401.

4. Munro, Jan. Efficient determination of the transitive closure of a directed graph. To be published.
5. Purdom, Paul Jr. A transitive closure algorithm, *BIT 10*, 1 (1970), 76–95.
6. Warshall, S. A theorem on Boolean matrices. *J.ACM 9* (Jan. 1962), 11–12.

Algorithm

```
procedure PREDOMINATOR( Immediate, M, n, a);
  value n, a; integer n, a;
  integer array Immediate; Boolean array M;
comment The procedure sets Immediate[i] to the immediate predomi-
  nator of i or to 0 if i has no immediate predominator. The inci-
  dence matrix of the graph is given by M, where M[i, j] is true if
  there is an arc from node i to node j. The number of nodes in the
  graph, which must be at least 1, is n, and a is (an upper limit on)
  the number of arcs in the graph. The start node is assumed to be
  node 1;
begin
  integer node, j, avail, k, stp, new, oldnode, down;
  integer array First, Last, St[1:n], Next[1:n+a], Suc[n+1:n+a];
  Boolean array Mark[1:n];
  comment This section initializes various variables and forms a
    linked list representation of the graph. The head of the list of
    arcs out of node i is Next[i] (for 1≤i≤n). The arcs are put on a
    list linked by the array Next where the corresponding entry in
    the array Suc gives the node to which the arc goes. In the array
    Next 0 indicates the end of the list. For most uses of the proce-
    dure the graph will already be available as a linked list and in
    such cases the procedure should be modified so that it starts
    from the list and does not use the array M.;
  avail := n;
  for j := 1 step 1 until n do
  begin
      Mark[j] := false; Next[i] := Immediate[j] := 0;
    for k := step 1 until n do if M[j, k] then
    begin
      avail := avail+1; Suc[avail] := k;
      Next[avail] := Next[j]; Next[j] := avail;
    end;
  end;
  down := Last[1] := 0; St[1] := stp .= oldnode := 1;
  Mark[1] := true; new := Next[1];
  comment newp1 is the start of Dijkstra's[2] algorithm for the
    shortest path, modified for the case where all distances are 0 or
    infinity. In addition the array First is set to link the nodes in the
    order they are traversed by Dijkstra's algorithm. Last[i] is set to
    the next node after node i on the list First which cannot be
    reached from node i by those arcs of the graph which are tra-
    versed by Dijkstra's algorithm. Node 1 is set as the tentative im-
    mediate predominator of each node that can be reached from
    node 1;
 newp1:
  if new ≠ 0 then
  begin
    node := Suc[new];
    if ¬ Mark[node]then
    begin
      for j := 1 step 1 until down do Last[Suc[St[stp+j]]] := node;
      down := 0; stp := stp+1; St[stp] := new;
        Mark[node] := true;
      Immediate[node] := 1; First[oldnode] := node;
      oldnode := node; new := Next[node];
      go to newp1;
    end;
    new := Next[new];
    go to newp1;
  end;
  down := down+1; new := Next[St[stp]]; stp := stp−1;
  if stp≠0 then go to newp1;
  for j := 2 step 1 until down do Last[Suc[St[j]]] := 0;
  First[oldnode] := 0; j := 1;
  if First[1]=0 then go to exit;
nextdom:
  oldnode := j; j := First[j]; k := First[j];
  if k=0 then go to exit;
  comment The nodes that the above version of Dijkstra's algorithm
    reached by going through node j will now be unmarked;
unmark:
  if k≠Last[j] then
  begin Mark[k] := false; k := First[k]; go to unmark; end;
  First[oldnode] := Last[j]; k := 1;
trace:
  if k≠0 then
  begin
    new := Next[k]; stp := 1;
    comment newp2 starts a second modification of Dijkstra's
      algorithm to find which unmarked nodes can be reached
      from the marked nodes without using node j;
newp2:
    if new≠0 then
    begin
      node := Suc[new];
      if ¬ Mark[node] then
      begin
        stp := stp+1; St[stp] := new; Mark[node] := true;
        new := Next[node];
        go to newp2;
      end;
      new := Next[new];
      go to newp2;
    end;
    new := Next[St[stp]]; stp := stp−1;
    if stp≠0 then go to newp2;
    k := First[k];
    go to trace;
  end;
  k := First[j]; First[oldnode] := j;
  comment Each unmarked node will now be remarked and have
    j set to be its tentative immediate predominator. The last
    tentative immediate predominator is the actual one;
marker:
  if k≠Last[j] then
  begin
    if ¬ Mark [k] then
    begin Immediate[k] := j; Mark[k] := true; end;
    k := First[k];
    go to marker;
  end;
  go to nextdom;
exit:
end of PREDOMINATOR;
```

Algorithm 431

A Computer Routine for Quadratic and Linear Programming Problems [H]

Arunachalam Ravindran [Recd. 24 Aug. 1970, 11 June 1971, and 1 Nov. 1971]
School of Industrial Engineering, Purdue University, Lafayette, IN 47907

Abstract. A computer program based on Lemke's complementary pivot algorithm is presented. This can be used to solve linear and quadratic programming problems. The program has been extensively tested on a wide range of problems and the results have been extremely satisfactory.

Key Words and Phrases: linear program, quadratic program, complementary problem, Lemke's algorithm, simplex method
CR Categories: 5.41
Language: Fortran

Description

Introduction. The computer routine given below is based on Lemke's complementary pivot algorithm [2] to solve the complementary problem of the form:

$$\begin{aligned}&\text{Find } w, z \geqq 0\\ &\text{such that } w = Mz + q \qquad (1)\\ &\qquad w'z = 0\end{aligned}$$

where M is an $(N \times N)$ square matrix; w, z and q are $(N \times 1)$ column vectors. ("Prime" denotes the transpose of a vector or matrix.)

A solution to the above problem will be called a complementary solution, and Lemke's algorithm is guaranteed to find a complementary solution to system (1) only if the matrix M satisfies one of the following:

1. M has all positive elements.
2. M is a positive semidefinite matrix or $x'Mx \geqq 0$ for all x.
3. M has positive principal determinants.

Applications. The two important applications of the complementary problem (1) are to solve linear and quadratic programming problems by converting them to an equivalent complementary problem.

Quadratic Programming. Consider the quadratic program:

$$\begin{aligned}&\text{Minimize } Z = c'x + x'Qx\\ &\text{subject to } Ax \geqq b\\ &\qquad\qquad x \geqq 0\end{aligned}$$

where A is an $(m \times n)$ matrix, Q is an $(n \times n)$ matrix of the quadratic form, c and x are $(n \times 1)$ column vectors, and b is an $(m \times 1)$ column vector.

An optimum solution to the above problem may be obtained by solving a complementary problem of the form:

$$\begin{pmatrix} v \\ u \end{pmatrix} = \begin{pmatrix} Q + Q' & -A' \\ A & 0 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} c \\ -b \end{pmatrix} \qquad (2)$$

$$\begin{aligned}&u, v, x, y \geqq 0\\ &v'x + u'y = 0\end{aligned}$$

where u denotes the slack variables of the given quadratic program and (y, v) denotes the variables of the dual problem. Comparing the above system (2) with the original complementary problem (1), we note that

$$w = \begin{pmatrix} v \\ u \end{pmatrix}, z = \begin{pmatrix} x \\ y \end{pmatrix}, M = \begin{pmatrix} Q + Q' & A' \\ A & 0 \end{pmatrix} \text{ and } q = \begin{pmatrix} c \\ -b \end{pmatrix}.$$

System (2) can be solved by the given computer routine and then an optimum solution to the given quadratic program may be obtained by reading off the values of $(z_1, z_2, \dots, z_n, w_{n+1}, \dots, w_{n+m})$ from the complementary solution. It should be remarked here that the matrix M in this case is positive semidefinite if and only if the matrix Q is positive semidefinite. Hence, the computer routine is guaranteed to find an optimum solution to the given quadratic program only if the objective function Z is a convex function.

Linear Programming. Consider the linear program:

$$\begin{aligned}&\text{Minimize } Z = c'x\\ &\text{subject to } Ax \geqq b\\ &\qquad\qquad x \geqq 0.\end{aligned}$$

The only difference between a linear program and a quadratic program is in the objective function. Hence, by setting $Q = 0$ in system (2), we get the equivalent complementary problem for a linear program.

Program. A detailed description of Lemke's algorithm to solve the complementary problem, on which the computer routine is based, is given in [3]. The program consists of six subroutines and a main program which calls these subroutines in proper order. The various input data to the program are the number of problems to be solved in succession, the size of the problem and the elements of matrix M and vector q. The original Lemke's algorithm [2] was modified by the author along the lines of the revised simplex method [1] for a linear program to take advantage of the fact that for solving linear and quadratic programs, the M matrix in system (1) has many zero entries. This led to a greater efficiency of the computer routine.

In an experimental study conducted by the author [4], this computer routine was extensively used to compare the relative efficiencies of the simplex method [1] and Lemke's algorithm to solve linear programs. The study revealed the superiority of Lemke's algorithm over the simplex method in a number of problems both with regard to the number of iterations and computation time. Also in [3], another modification of Lemke's algorithm for solving linear programs has been proposed which may save a considerable storage and computation time.

References

1. Dantzig, G.B. *Linear Programming and Extensions.* Princeton U. Press, Princeton, N.J. 1963.
2. Lemke, C.E. Bimatrix equilibrium points and mathematical programming. *Management Sci. 11* (1965), 681–689.
3. Ravindran, A. Computational aspects of Lemke's complementary algorithm applied to linear programs. *Opsearch* 7 (1970), 241–262.

4. Ravindran, A. A comparison of the primal-simplex and complementary pivot methods for linear programming. Rep. No. 70–9 (July 1970), School of Industrial Engineering, Purdue U., Lafayette, Ind.

Algorithm

```
C REMARKS
C SINCE THIS PROGRAM IS COMPLETE IN ALL RESPECTS,IT CAN BE
C RUN AS IT IS WITHOUT ANY ADDITIONAL MØDIFICATIØN OR
C INSTRUCTIØN.IN SUCH CASE FØLLØW THE INPUT FØRMAT AS GIVEN
C
C PROGRAM FØR SØLVING LINEAR AND QUADRATIC PRØGRAMMING
C PRØBLEMS IN THE FØRM W=M*Z+Q, W.Z=0, W AND Z NONNEGATIVE
C BY LEMKE/S ALGØRITHM.
C
C MAIN PRØGRAM WHICH CALLS THE SIX SUBRØUTINES-MATRIX,
C INITIA,NEWBAS,SØRT,PIVØT AND PPRINT IN PRØPER ØRDER.
C
      CØMMØN AM,Q,L1,B,NL1,NL2,A,NE1,NE2,IR,MBASIS,W,Z
      DIMENSIØN AM(50,50), Q(50), B(50,50), A(50)
      DIMENSIØN W(50), Z(50), MBASIS(100)
C DESCRIPTIØN ØF PARAMETERS IN CØMMØN
C   AM      A TWØ DIMENSIØNAL ARRAY CØNTAINING THE
C           ELEMENTS ØF MATRIX M.
C   Q       A SINGLY SUBSCRIPTED ARRAY CØNTAINING THE
C           ELEMENTS ØF VECTØR Q.
C   L1      AN INTEGER VARIABLE INDICATING THE NUMBER ØF
C           ITERATIØNS TAKEN FØR EACH PRØBLEM.
C   B       A TWØ DIMENSIØNAL ARRAY CØNTAINING THE
C           ELEMENTS ØF THE INVERSE ØF THE CURRENT BASIS.
C   W       A SINGLY SUBSCRIPTED ARRAY CØNTAINING THE VALUES
C           ØF W VARIABLES IN EACH SØLUTIØN.
C   Z       A SINGLY SUBSCRIPTED ARRAY CØNTAINING THE VALUES
C           ØF Z VARIABLES IN EACH SØLUTIØN.
C   NL1     AN INTEGER VARIABLE TAKING VALUE 1 ØR 2 DEPEND-
C           ING ØN WHETHER VARIABLE W ØR Z LEAVES THE BASIS
C   NE1     SIMILAR TØ NL1 BUT INDICATES VARIABLE ENTERING
C   NL2     AN INTEGER VARIABLE INDICATING WHAT CØMPØNENT
C           ØF W ØR Z VARIABLE LEAVES THE BASIS.
C   NE2     SIMILAR TØ NL2 BUT INDICATES VARIABLE ENTERING
C   A       A SINGLY SUBSCRIPTED ARRAY CØNTAINING THE
C           ELEMENTS ØF THE TRANSFØRMED CØLUMN THAT IS
C           ENTERING THE BASIS.
C   IR      AN INTEGER VARIABLE DENØTING THE PIVØT RØW AT
C           EACH ITERATIØN. ALSØ USED TØ INDICATE TERMINA-
C           TIØN ØF A PRØBLEM BY GIVING IT A VALUE ØF 1000.
C   MBASIS  A SINGLY SUBSCRIPTED ARRAY-INDICATØR FØR THE
C           BASIC VARIABLES. TWØ INDICATØRS ARE USED FØR
C           EACH BASIC VARIABLE-ØNE INDICATING WHETHER
C           IT IS A W ØR Z AND ANØTHER INDICATING WHAT
C           CØMPØNENT ØF W ØR Z.
C
C READ IN THE VALUE ØF VARIABLE IP INDICATING THE
C NUMBER ØF PRØBLEMS TØ BE SØLVED.
      READ(5,3) IP
C VARIABLE NØ INDICATES THE CURRENT PRØBLEM BEING SØLVED
      NØ=0
    1 NØ=NØ+1
      IF (NØ.GT.IP) GØ TØ 5
      WRITE(6,2) NØ
    2 FØRMAT (1H1,10X,11HPRØBLEM NØ.,I2)
C
C READ IN THE SIZE ØF THE MATRIX M
      READ(5,3) N
    3 FØRMAT (I2)
C PRØGRAM CALLING SEQUENCE
      CALL MATRIX (N)
C PARAMETER N INDICATES THE PRØBLEM SIZE
      CALL INITIA (N)
C SINCE FØR ANY PRØBLEM TERMINATIØN CAN ØCCUR IN INITIA,
C NEWBAS ØR SØRT SUBRØUTINE,THE VALUE ØF IR IS MATCHED WITH
C 1000 TØ CHECK WHETHER TØ CØNTINUE ØR GØ TØ NEXT PRØBLEM.
      IF (IR.EQ.1000) GØ TØ 1
    4 CALL NEWBAS (N)
      IF (IR.EQ.1000) GØ TØ 1
      CALL SØRT (N)
      IF (IR.EQ.1000) GØ TØ 1
      CALL PIVØT (N)
      GØ TØ 4
    5 STØP
      END
      SUBRØUTINE MATRIX (N)
C PURPØSE - TØ INITIALIZE AND READ IN THE VARIØUS INPUT DATA
C
      CØMMØN AM,Q,L1,B,NL1,NL2,A,NE1,NE2,IR,MBASIS,W,Z
      DIMENSIØN AM(50,50), Q(50), B(50,50), A(50)
      DIMENSIØN W(50), Z(50), MBASIS(100)
C READ THE ELEMENTS ØF M MATRIX CØLUMN BY CØLUMN
      DØ 1 J=1,N
    1   READ(5,2) (AM(I,J),I=1,N)
    2 FØRMAT (7F10.5)
C READ THE ELEMENTS ØF Q VECTØR
      READ(5,2) (Q(I),I=1,N)
C IN ITERATIØN 1,BASIS INVERSE IS AN IDENTITY MATRIX.
      DØ 5 J=1,N
        DØ 4 I=1,N
          IF (I.EQ.J) GØ TØ 3
          B(I,J)=0.0
          GØ TØ 4
    3     B(I,J)=1.0
    4     CØNTINUE
    5   CØNTINUE
      RETURN
      END
      SUBRØUTINE INITIA (N)
C PURPØSE-TØ FIND THE INITIAL ALMØST CØMPLEMENTARY SØLUTIØN
C           BY ADDING AN ARTIFICIAL VARIABLE ZØ.
C
      CØMMØN AM,Q,L1,B,NL1,NL2,A,NE1,NE2,IR,MBASIS,W,Z
      DIMENSIØN AM(50,50), Q(50), B(50,50), A(50)
      DIMENSIØN W(50), Z(50), MBASIS(100)
C SET ZØ EQUAL TØ THE MØST NEGATIVE Q(I)
      I=1
      J=2
    1 IF (Q(I).LE.Q(J)) GØ TØ 2
      I=J
    2 J=J+1
      IF (J.LE.N) GØ TØ 1
C UPDATE Q VECTØR
      IR=I
      T1=-Q(IR)
      IF (T1.LE.0.0) GØ TØ 9
      DØ 3 I=1,N
        Q(I)=Q(I)+T1
    3   CØNTINUE
      Q(IR)=T1
C UPDATE BASIS INVERSE AND INDICATØR VECTØR
C ØF BASIC VARIABLES.
      DØ 4 J=1,N
        B(J,IR)=-1.0
        W(J)=Q(J)
        Z(J)=0.0
        MBASIS(J)=1
        L=N+J
        MBASIS(L)=J
    4   CØNTINUE
      NL1=1
      L=N+IR
      NL2=IR
      MBASIS(IR)=3
      MBASIS(L)=0
      W(IR)=0.0
      ZØ=Q(IR)
      L1=1
C PRINT THE INITIAL ALMØST CØMPLEMENTARY SØLUTIØN
      WRITE(6,5)
    5 FØRMAT (3(/),5X,29HINITIAL ALMØST CØMPLEMENTARY ,
     1   8HSØLUTIØN)
      DØ 7 I=1,N
        WRITE(6,6) I,W(I)
    6   FØRMAT (10X,2HW(,I4,2H)=,F15.5)
    7   CØNTINUE
      WRITE(6,8) ZØ
    8 FØRMAT (10X,3HZØ=,F15.5)
      RETURN
    9 WRITE (6,10)
   10 FØRMAT (5X,36HPRØBLEM HAS A TRIVIAL CØMPLEMENTARY ,
     1   23HSØLUTIØN WITH W=Q, Z=0.)
      IR=1000
      RETURN
      END
      SUBRØUTINE NEWBAS (N)
C PURPØSE - TØ FIND THE NEW BASIS CØLUMN TØ ENTER IN
C           TERMS ØF THE CURRENT BASIS.
C
      CØMMØN AM,Q,L1,B,NL1,NL2,A,NE1,NE2,IR,MBASIS,W,Z
      DIMENSIØN AM(50,50), Q(50), B(50,50), A(50)
      DIMENSIØN W(50), Z(50), MBASIS(100)
C IF NL1 IS NEITHER 1 NØR 2 THEN THE VARIABLE ZØ LEAVES THE
C BASIS INDICATING TERMINATIØN WITH A CØMPLEMENTARY SØLUTIØN
      IF (NL1.EQ.1) GØ TØ 2
      IF (NL1.EQ.2) GØ TØ 5
      WRITE(6,1)
    1 FØRMAT (5X,22HCØMPLEMENTARY SØLUTIØN)
      CALL PPRINT (N)
      IR=1000
      RETURN
    2 NE1=2
      NE2=NL2
C UPDATE NEW BASIC CØLUMN BY MULTIPLYING BY BASIS INVERSE.
      DØ 4 I=1,N
        T1=0.0
        DØ 3 J=1,N
    3     T1=T1-B(I,J)*AM(J,NE2)
        A(I)=T1
    4   CØNTINUE
      RETURN
    5 NE1=1
      NE2=NL2
      DØ 6 I=1,N
        A(I)=B(I,NE2)
    6   CØNTINUE
      RETURN
      END
      SUBRØUTINE SØRT (N)
C PURPØSE - TØ FIND THE PIVØT RØW FØR NEXT ITERATIØN BY THE
C           USE ØF (SIMPLEX-TYPE) MINIMUM RATIØ RULE.
C
      CØMMØN AM,Q,L1,B,NL1,NL2,A,NE1,NE2,IR,MBASIS,W,Z
      DIMENSIØN AM(50,50), Q(50), B(50,50), A(50)
      DIMENSIØN W(50), Z(50), MBASIS(100)
      I=1
    1 IF (A(I).GT.0.0) GØ TØ 2
      I=I+1
      IF (I.GT.N) GØ TØ 6
      GØ TØ 1
    2 T1=Q(I)/A(I)
      IR=I
    3 I=I+1
      IF (I.GT.N) GØ TØ 5
      IF (A(I).GT.0.0) GØ TØ 4
      GØ TØ 3
    4 T2=Q(I)/A(I)
      IF (T2.GE.T1) GØ TØ 3
      IR=I
      T1=T2
      GØ TØ 3
    5 RETURN
C FAILURE ØF THE RATIØ RULE INDICATES TERMINATIØN WITH
C NØ CØMPLEMENTARY SØLUTIØN.
    6 WRITE(6,7)
    7 FØRMAT (5X,37HPRØBLEM HAS NØ CØMPLEMENTARY SØLUTIØN)
      WRITE(6,8) L1
    8 FØRMAT (10X,13HITERATIØN NØ.,I4)
      IR=1000
      RETURN
      END
      SUBRØUTINE PIVØT (N)
C PURPØSE - TØ PERFØRM THE PIVØT ØPERATIØN BY UPDATING THE
C           INVERSE ØF THE BASIS AND Q VECTØR.
C
      CØMMØN AM,Q,L1,B,NL1,NL2,A,NE1,NE2,IR,MBASIS,W,Z
      DIMENSIØN AM(50,50), Q(50), B(50,50), A(50)
      DIMENSIØN W(50), Z(50), MBASIS(100)
      DØ 1 I=1,N
    1   B(IR,I)=B(IR,I)/A(IR)
      Q(IR)=Q(IR)/A(IR)
      DØ 3 I=1,N
        IF (I.EQ.IR) GØ TØ 3
        Q(I)=Q(I)-Q(IR)*A(I)
        DØ 2 J=1,N
          B(I,J)=B(I,J)-B(IR,J)*A(I)
    2     CØNTINUE
    3   CØNTINUE
C UPDATE THE INDICATØR VECTØR ØF BASIC VARIABLES
      NL1=MBASIS(IR)
      L=N+IR
      NL2=MBASIS(L)
      MBASIS(IR)=NE1
      MBASIS(L)=NE2
      L1=L1+1
      RETURN
      END
      SUBRØUTINE PPRINT (N)
C PURPØSE - TØ PRINT THE CURRENT SØLUTIØN TØ CØMPLEMENTARY
C           PRØBLEM AND THE ITERATIØN NUMBER.
C
      CØMMØN AM,Q,L1,B,NL1,NL2,A,NE1,NE2,IR,MBASIS,W,Z
      DIMENSIØN AM(50,50), Q(50), B(50,50), A(50)
      DIMENSIØN W(50), Z(50), MBASIS(100)
      WRITE(6,1) L1
    1 FØRMAT (10X,13HITERATIØN NØ.,I4)
      I=N+1
      J=1
    2 K1=MBASIS(I)
      K2=MBASIS(J)
      IF (Q(J).GE.0.0) GØ TØ 3
      Q(J)=0.0
    3 IF (K2.EQ.1) GØ TØ 5
      WRITE(6,4) K1,Q(J)
    4 FØRMAT (10X,2HZ(,I4,2H)=,F15.5)
      GØ TØ 7
    5 WRITE(6,6) K1,Q(J)
    6 FØRMAT (10X,2HW(,I4,2H)=,F15.5)
    7 I=I+1
      J=J+1
      IF (J.LE.N) GØ TØ 2
      RETURN
      END
```

Editor's note: *Algorithm 432 described here is available on magnetic tape from the Department of Computer Science, University of Colorado, Boulder, CO 80302. The cost for the tape is $16.00 (U.S. and Canada) or $18.00 (elsewhere). If the user sends a small tape (wt. less than 1 lb.) the algorithm will be copied on it and returned to him at a charge of $10.00 (U.S. only). All orders are to be prepaid with checks payable to ACM Algorithms. The algorithm is re corded as one file of BCD 80 character card images at 556 B.P.I·, even parity, on seven track tape. We will supply the algorithm at a density of 800 B.P.I. if requested. The cards for the algorithm are sequenced starting at 10 and incremented by 10. The sequence number is right justified in colums 80. Although we will make every attempt to insure that the algorithm conforms to the description printed here, we cannot guarantee it, nor can we guarantee that the algorithm is correct.—L.D.F.*

Remark on Algorithm 431 [H]

A Computer Routine for Quadratic and Linear Programming Problems [H] [Arunachalam Ravindran, *Comm. ACM 15* (Sept., 1972), 818]

Arunachalam Ravindran [Recd. 12 Mar. 1973]
School of Industrial Engineering, Purdue University,
West Lafayette, IN 47907

A small error has been brought to my notice in this algorithm. The error is in defining the matrix M. It should read as

$$M = \begin{pmatrix} Q+Q' & -A' \\ A & 0 \end{pmatrix}.$$

Remark on Algorithm 431 [H]

A Computer Routine for Quadratic and Linear Programming Problems [A. Ravindran, *Comm. ACM 15* (Sept. 1972), 818–820]

L.G. Proll (Recd. 13 Aug. 1973)
Centre for Computer Studies, University of Leeds,
Leeds LS2 9JT, England

Algorithm 431 is a Fortran implementation of Lemke's complementary pivot algorithm [1]. This algorithm has recently received a considerable amount of attention in the literature; in particular, there is some evidence that the algorithm is an attractive means of solving linear programs [2, 3] and can readily be modified to find stationary points of nonconvex quadratic programs [4].

Eaves [5] has shown that, in principle, degeneracy causes no problems in Lemke's algorithm and that it will always be possible to pivot the artificial variable out of the basis. In the presence of rounding error, however, this may no longer be true, and further pivoting may not be possible despite the presence of the artificial variable with a value close to zero. In such a case Algorithm 431 may incorrectly arrive at the conclusion that the problem has no complementary solution because it only recognizes a complementary solution when the artificial variable leaves the basis.

The difficulty can be avoided by: (a) testing whether the value assumed by the artificial variable is acceptably "small" if no further pivoting is possible; and (b) not pivoting on "small" elements. The problem of deciding what is meant by "small" in this context is one for which there is no adequate theory. Clasen [6] has, however, proposed some empirical rules for dealing with similar problems in the revised simplex algorithm, and an adaptation of these has proved satisfactory. The modifications of Algorithm 431 given below incorporate Clasen's pivot tolerance to deal with point (b) above and also use this value as the upper limit on the acceptable value of the artificial variable.

(i) In the subroutine *INITIA*, add *IZR* to the end of the *COMMON* list and insert after the statement labeled 4, the statement

```
IZR = IR
```

(ii) In the subroutine *SORT*, add *IZR* to the end of the *COMMON* list and

(a) after the second *DIMENSION* statement, insert

```
         AMAX=ABS(A(1))
         DO 10 I=2,N
              IF (AMAX.GE.ABS(A(I))) GOTO 10
              AMAX=ABS(A(I))
10       CONTINUE
         TOL=AMAX*2.0**(-NB)
C     IN ANY ACTUAL IMPLEMENTATION NB SHOULD BE RE-
C     PLACED BY B-11 WHERE B IS THE NUMBER OF BITS IN
C     THE FLOATING POINT MANTISSA AS CLASEN SUGGESTS
```

(b) Replace 0.0 by *TOL* in the statement labeled 1 and in the statement two lines before that labeled 4.

(c) Replace the label 6, occurring two lines before the statement labeled 2, by 9.

(d) Immediately after *RETURN*, insert the statements,

```
9        IF(Q(IZR).GT.TOL) GOTO 6
         WRITE(6,11)
11       FORMAT(5X,22HCOMPLEMENTARY SOLUTION)
         CALL PPRINT(N)
         IR=1000
         RETURN
```

References

1. Lemke, C.E. Bimatrix equilibrium points and mathematical programming. *Management Sci. 11* (1965), 681–689.
2. Ravindran, A. Computational aspects of Lemke's complementary algorithm applied to linear programs. *Opsearch* 7 (1970), 241–262.
3. Ravindran, A. A comparison of the primal simplex and complementary pivot methods for linear programming. *Naval Res. Log. Q. 20* (1972), 95–100.
4. Eaves, B.C. On quadratic programming. *Management Sci. 17* (1971), 698–711.
5. Eaves, B.C. The linear complementarity problem. *Management Sci. 17* (1971), 612–634.
6. Clasen, R.J. Techniques for automatic tolerance control in linear programming. *Comm. ACM 9* (1966), 802–803.

Algorithm 432
Solution of the Matrix Equation AX + XB = C [F4]

R.H. Bartels and G.W. Stewart [Recd. 21 Oct. 1970
and 7 March 1971]
Center for Numerical Analysis, The University of Texas
at Austin, Austin, TX 78712

Key Words and Phrases: linear algebra, matrices, linear equations
CR Categories: 5.14
Language: Fortran

Description

The following programs are a collection of Fortran IV subroutines to solve the matrix equation

$$AX + XB = C \tag{1}$$

where A, B, and C are real matrices of dimensions $m \times m$, $n \times n$, and $m \times n$, respectively. Additional subroutines permit the efficient solution of the equation

$$A^T X + XA = C, \tag{2}$$

where C is symmetric. Equation (1) has applications to the direct solution of discrete Poisson equations [2].

It is well known that (1) has a unique solution if and only if the eigenvalues $\alpha_1, \alpha_2, \ldots, \alpha_m$ of A and $\beta_1, \beta_2, \ldots, \beta_n$ of B satisfy

$$\alpha_i + \beta_j \neq 0 \quad (i = 1, 2, \ldots, m; j = 1, 2, \ldots, n).$$

One proof of the result amounts to constructing the solution from complete systems of eigenvalues and eigenvectors of A and B, when they exist. This technique has been proposed as a computational method (e.g. see [1]); however, it is unstable when the eigensystem is ill conditioned. The method proposed here is based on the Schur reduction to triangular form by orthogonal similarity transformations.

Equation (1) is solved as follows. The matrix A is reduced to lower real Schur form A' by an orthogonal similarity transformation U; that is A is reduced to the real, block lower triangular form.

$$A' = U^T A U = \begin{bmatrix} A'_{11} & & & \bigcirc \\ A'_{21} & A'_{22} & & \\ \cdot & \cdot & \cdot & \\ \cdot & \cdot & \cdot & \\ \cdot & \cdot & \cdot & \\ A'_{p1} & A'_{p2} & \cdots & A'_{pp} \end{bmatrix},$$

where each matrix A'_{ii} is of order at most two. Similarly B is reduced to upper real Schur form by the orthogonal matrix V:

$$B' = V^T B V = \begin{bmatrix} B'_{11} & B'_{12} & \cdots & B'_{1q} \\ & B'_{22} & \cdots & B'_{2q} \\ & & \cdot & \cdot \\ & & \cdot & \cdot \\ & \bigcirc & \cdot & \cdot \\ & & & B'_{qq} \end{bmatrix},$$

where again each B'_{ii} is of order at most two. If

$$C' = U^T C V = \begin{bmatrix} C'_{11} & \cdots & C'_{1q} \\ \cdot & & \\ \cdot & & \\ \cdot & & \\ C'_{p1} & \cdots & C'_{pq} \end{bmatrix}$$

and

$$X' = U^T X V = \begin{bmatrix} X'_{11} & \cdots & X'_{1q} \\ \cdot & & \\ \cdot & & \\ \cdot & & \\ X'_{p1} & \cdots & X'_{pq} \end{bmatrix},$$

then eq. (1) is equivalent to

$$A'X' + X'B' = C'.$$

If the partitions of A', B', C', and X' are conformal, then

$$A'_{kk}X'_{kl} + X'_{kl}B'_{ll} = C'_{kl} - \sum_{j=1}^{k-1} A'_{kj}X'_{jl} - \sum_{i=1}^{l-1} X'_{ki}B'_{il} \qquad (k = 1, 2, \cdots, p; \; l = 1, 2, \cdots, q). \tag{3}$$

These equations may be solved successively for $X'_{11}, X'_{21}, \ldots, X'_{p1}, X'_{12}, X'_{22}, \ldots$ The solution of (1) is then given by $X = UX'V^T$.

The reduction of A and B to real Schur form is accomplished by standard techniques. The matrix B is reduced to upper Hessenberg form by Householder's method [4, p. 34], and the upper Hessenberg matrix is in turn reduced to real Schur form by the *QR* algorithm [3]. The product of the transformations used in the reductions is accumulated to form the matrix V. The reduction of A to lower real Schur form is accomplished by reducing the transpose of A to upper real Schur form and transposing back.

Since the *QR* algorithm is an iterative method that, as used here, reduces the subdiagonal elements of an upper Hessenberg matrix to zero, some criterion must be adopted for determining when an element is negligible. In these programs an element of H is considered negligible if it is less than or equal to $\epsilon_H \| H \|_\infty$ where ϵ_H is a constant supplied by the user. This criterion is appropriate if the elements of H are all of roughly the same size. A different criterion may be required if the elements vary widely and the small elements are significant, as when the elements decrease greatly in size as one passes from the upper left to the lower right corners of A (see, for example, the criterion in [3]).

The solution for X'_{kl} in (3) still requires the solution of a matrix equation of the form (1). However, in this case the matrices A'_{kk} and B'_{ll} are of order at most two; hence the solution of (3) can be obtained by solving a linear system of order at most four. For example, if A'_{kk} and B'_{ll} are both of order two, then

$$\begin{bmatrix} a'_{11} + b'_{11} & a'_{12} & b'_{21} & 0 \\ a'_{21} & a'_{22} + a'_{11} & 0 & b'_{21} \\ b'_{12} & 0 & a'_{11} + b'_{22} & a'_{12} \\ 0 & b'_{12} & a'_{21} & a'_{22} + b'_{22} \end{bmatrix} \begin{bmatrix} x'_{11} \\ x'_{21} \\ x'_{12} \\ x'_{22} \end{bmatrix} = \begin{bmatrix} d_{11} \\ d_{21} \\ d_{12} \\ d_{22} \end{bmatrix},$$

This research was supported in part by Grant DA-ARO(D)-31-124-G721, Army Research Office, Durham, and by National Science Foundation Grant GP-5253 awarded to The University of Texas at Austin.

where a'_{ij}, b'_{ij}, and x'_{ij} denote the elements of A'_{kk}, B'_{ll}, and X'_{lk} and d_{ij} denotes the elements of the right-hand side of (3). The systems arising from (3) are solved using the Crout reduction. Once calculated, the solution X'_{kl} may be stored in the locations occupied by C_{kl}, which is no longer needed.

The programs contain provisions for skipping the reduction of A to real Schur form, so that once A' and U have been calculated they may be used to solve new systems with different matrices B and C. Likewise, the reduction of B may be skipped. These provisions may be used to advantage in the iterative refinement of the computed solution X_1 of (1). Namely, let the residual matrix $R_1 = C - AX_1 - X_1B$ be computed in double precision and rounded to single precision (on many computers this may be done with single-precision multiplications and double-precision additions). Use the programs to solve the system $AY_1 + Y_1B = R_1$. Then $X_2 = X_1 + Y_1$ will in general be a more accurate approximate solution. This process may be iterated, no step after the computation of X_1 requiring reductions of A and B. This iteration is perfectly analogous to the iterative refinement of approximate solutions of linear systems described by Wilkinson [4, p. 255].

The following trick enables one to use an upper rather than a lower real Schur form of A in the solution of (1). Let D be the matrix with ones on the secondary diagonal and zeros elsewhere. Then

$$(DAD)DX + DXB = DC. \tag{4}$$

Moreover, if $A' = U^TAU$ is an upper real Schur form for A, then $DA'D = (DUD)^T(DAD)(DUD)$ is a lower real Schur form for DAD. Hence to calculate DX, which is X with its rows written in reverse order, one may use the above algorithm with $DA'D$ and DUD to solve (4). A similar trick enables one to use a lower real Schur form for B.

In principle, the algorithm described above can be used to solve the symmetric problem (2). However, it is possible to take advantage of the symmetry. Let U be orthogonal and $A' = U^TAU$ be in upper real Schur form. Partition A', $C' = U^TCU$, and $X' = U^TXU$ in the form

$$A' = \begin{bmatrix} A'_{11} & A'_{12} \\ 0 & A'_{22} \end{bmatrix},$$

$$X' = \begin{bmatrix} X'_{11} & X'^T_{21} \\ X'_{21} & X'_{22} \end{bmatrix},$$

$$C' = \begin{bmatrix} C'_{11} & C'^T_{21} \\ C'_{21} & C'_{22} \end{bmatrix},$$

where A'_{11}, X'_{11}, and C'_{11} are at most of order 2. Then from the equation $A'^TX' + X'A' = C'$, it follows that

$$A'^T_{22}X'_{22} + X'_{22}A'_{22} = C'_{22} - X'_{21}A'_{12} - A'^T_{12}X'_{21}.$$

Hence, once X'_{11} and X'_{21} have been calculated, the size of the problem can be reduced.

The matrix X'_{21} is computed as described above for the general case. The matrix X'_{11} satisfies the symmetric equation

$$A'^T_{11}X'_{11} + X'_{11}A'_{11} = C'_{11}, \tag{5}$$

whose solution is trivial when A'_{11} is of order unity. When A'_{11} is of order two, equation (5) gives a new linear system of order three for the three distinct elements of X'_{11}.

A mild saving in operations may be realized in the computation of $C' = U^TCU$ and $X = UX'U^T$. Let $C = T + T^T$, where T is upper triangular. Then

$$C' = U^TCU = U^TTU + (U^TTU)^T.$$

Thus one need calculate only U^TTU, and, since T is upper triangular, the product TU can be computed with about half the operations required for the computation of CU.

The number of multiplications required for the solution of (1) is probably overestimated by

$$(2 + 4\sigma)(m^3 + n^3) + \frac{5}{2}(mn^2 + nm^2)$$

where σ is the average number of QR steps required to make a subdiagonal element negligible. The first term is due to the reduction of A and B to real Schur form. A like estimate for the solution of (2) is given by

$$(2 + 4\sigma)n^3 + \frac{7}{2}n^3;$$

the first term is again due to the reduction of A to real Schur form.

To solve the nonsymmetric problem, the user must furnish $2m^2 + 2n^2 + mn$ storage locations to hold the matrices A, U, B, V, and C. If A, B, and C are required for later use, they must be stored elsewhere, since the programs overwrite A and B with their real Schur forms and C with the solution. The symmetric problem requires $3n^2$ locations to hold A, U, and C.

In assessing the effects of rounding error on the algorithm, we should consider the algorithm stable if the computed solution were near a matrix $\overline{X}$ that satisfied

$$(A+E)\overline{X} + \overline{X}(B+F) = C + G$$

for some small E, F, and G. We are unable to establish such a result. However, an elementary rounding error analysis, combined with the known properties of the other algorithms used in the method, shows that the residual matrix is small compared with the larger of $\| A \| \| X \|$ and $\| B \| \| X \|$.

Here follows a brief description of the programs listed below. Detailed information on their use will be found in the program listings themselves. The casual user need only familiarize himself with the programs *AXPXB* and *ATXPXA*, which coordinate the other programs for the solutions of (1) and (2), respectively.

AXPXB. The coordinating program for the solution of (1). Given A, B and C the program overwrites C with the solution X. The lower real Schur form of A overwrites A, and the upper real Schur form of B overwrites B. The user may furnish the real Schur forms and skip the reductions. The subroutine requires the subroutines *HSHLDR*, *BCKMLT*, *SCHUR*, *SHRSLV*, and *SYSSLV*.

ATXPXA. The coordinating program for the solution of (2). Given A and C the program overwrites C with the solution X. The upper real Schur form of A overwrites A. The user may furnish the real Schur form and skip the reduction. The subroutine requires the subroutines *HSHLDR*, *BCKMLT*, *SCHUR*, *SYMSLV*, and *SYSSLV*.

HSHLDR. Reduces a matrix A to upper Hessenberg form. The upper Hessenberg form and a history of the transformations overwrite A.

BCKMLT. Takes the output A of *HSHLDR* and computes the orthogonal matrix U that reduces the original matrix A to upper Hessenberg form. At the user's option the elements of U can overwrite A.

SCHUR. Computes an upper real Schur form of an upper Hessenberg matrix A. *SCHUR* is an adaptation of thel Agol procedure *hqr* by Martin, Peters, and Wilkinson [1]. The product of the transformations used in the reduction is accumulated. *SCHUR* leaves undisturbed the elements below the third subdiagonal of the array containing A. (N.b. The modifications made in *hqr* to find a real Schur form make *SCHUR* an inefficient program for calculating the eigenvalues of an upper Hessenberg matrix.)

SHRSLV. Solves an equation of the form (1), where A is in lower real Schur form and B is in upper real Schur form.

SYMSLV. Solves an equation of the form (2), where A is in upper real Schur form.

SYSSLV. Solves a system of linear equations.

When $m \geq n$, *AXPXB* can be modified so that the real Schur forms of A and B share the storage originally allocated to A and the matrix V occupies the locations occupied by B. The modifications are as follows. Replace the section labeled "*IF REQUIRED, REDUCE B TO UPPER REAL SCHUR FORM*" with

```
35 IF(EPSB .LT. 0.) GO TO 45
   CALL HSHLDR (B, N, NB)
```

```
    DO 40 I = 1, N
      IF (I .NE. 1) A(I, I+4) = B(I-1, N1)
      DO 40 J = I, N
        A(I, J+5) = B(I, J)
40  CONTINUE
    CALL BCKMLT(B,B,N,NB,NB)
    CALL SCHURA(1,6),B,N,NA,NB,EPSB,FAIL)
    FAIL = -FAIL
    IF(FAIL .NE. 0) RETURN
```

In the sections labeled *"TRANSFORM C"* and *"TRANSFORM C BACK TO THE SOLUTION"* replace all occurrences of the variable *V* with *B* and all references to *A(I,M1)* with *A(M1,I)*. Change the call to *SHRSLV* to

CALL SHRSLV(A,A(1,6),C,M,N,NA,NA,NC).

Note that in this modification the reduction of *B* to real Schur form cannot be skipped without also skipping the reduction of *A*. When $m \leq n$ a similar modification can be made to store the Schur form of *A* and *B* together in *B*.

References

1. Bickley, W.G. and McNamee, J. Matrix and other direct methods for the solution of systems of linear difference equations. *Philos. Trans. Roy. Soc.* (London) Ser. A, *252* (1960), 69–131.
2. Dorr, Fred W. The direct solution of the discrete Poisson equation on a rectangle. *SIAM Rev. 12* (1970), 248–263.
3. Martin, R.S., Peters, G., and Wilkinson, J.H. The QR algorithm for real Hessenberg matrices. (Handbook series linear algebra.) *Numer. Math. 14* (1970), 219–231.
4. Wilkinson, J.H. *The Algebraic Eigenvalue Problem.* Clarendon, Oxford, 1965.

Algorithm

```
      SUBRØUTINE AXPXB(A,U,M,NA,NU,B,V,N,NB,NV,C,NC,EPSA,
     1EPSB,FAIL)
C
C AXPXB IS A FØRTRAN IV SUBRØUTINE TØ SØLVE THE REAL MATRIX
C EQUATIØN AX + XB - C.  THE MATRICES A AND B ARE TRANS-
C FØRMED INTØ REAL SCHUR FØRM, AND THE TRANSFØRMED SYSTEM IS
C SØLVED BY BACK SUBSTITUTIØN.  THE PRØGRAM REQUIRES THE
C AUXILIARY SUBRØUTINES HSHLDR, BCKMLT, SCHUR, AND SHRSLV.
C THE PARAMETERS IN THE ARGUMENT LIST ARE
C
C           A       A DØUBLY SUBSCRIPTED ARRAY CØNTAINING THE
C                   MATRIX A.  ØN RETURN, THE LØWER TRIANGLE
C                   AND SUPERDIAGØNAL ØF THE ARRAY A CØNTAIN
C                   A LØWER REAL SCHUR FØRM ØF A.  THE ARRAY
C                   A MUST BE DIMENSIØNED AT LEAST M+1 BY
C                   M+1.
C           U       A DØUBLY SUBSCRIPTED ARRAY THAT, ØN
C                   RETURN, CØNTAINS THE ØRTHØGØNAL MATRIX
C                   THAT REDUCES A TØ REAL SCHUR FØRM.
C           M       THE ØRDER ØF THE MATRIX A.
C           NA      THE FIRST DIMENSIØN ØF THE ARRAY A.
C           NU      THE FIRST DIMENSIØN ØF THE ARRAY U.
C           B       A DØUBLY SUBSCRIPTED ARRAY CØNTAINING THE
C                   MATRIX B.  ØN RETURN, THE UPPER TRIANGLE
C                   AND SUBDIAGØNAL ØF THE ARRAY B CØNTAIN AN
C                   UPPER REAL SCHUR FØRM ØF B.  THE ARRAY B
C                   MUST BE DIMENSIØNED AT LEAST M+1 BY M+1.
C           V       A DØUBLY SUBSCRIPTED ARRAY THAT, ØN
C                   RETURN, CØNTAINS THE ØRTHØGØNAL MATRIX
C                   THAT REDUCES B TØ REAL SCHUR FØRM.
C           N       THE ØRDER ØF THE MATRIX B.
C           NB      THE FIRST DIMENSIØN ØF THE ARRAY B.
C           NV      THE FIRST DIMENSIØN ØF THE ARRAY V.
C           C       A DØUBLY SUBSCRIPTED ARRAY CØNTAINING THE
C                   MATRIX C.  ØN RETURN, C CØNTAINS THE
C                   SØLUTIØN MATRIX X.
C           NC      THE FIRST DIMENSIØN ØF THE ARRAY C.
C           EPSA    A CØNVERGENCE CRITERIØN FØR THE REDUCTIØN
C                   ØF A TØ SCHUR FØRM.  EPSA SHØULD BE SET
C                   SLIGHTLY SMALLER THAN 10.**(-N), WHERE N
C                   IS THE NUMBER ØF SIGNIFICANT DIGITS IN
C                   THE ELEMENTS ØF THE MATRIX A.
C           EPSB    A CØNVERGENCE CRITERIØN FØR THE REDUCTIØN
C                   ØF B TØ REAL SCHUR FØRM.
C           FAIL    AN INTEGER VARIABLE THAT, ØN RETURN,
C                   CØNTAINS AN ERRØR SIGNAL.  IF FAIL IS
C                   PØSITIVE (NEGATIVE) THEN THE PRØGRAM WAS
C                   UNABLE TØ REDUCE A (B) TØ REAL SCHUR
C                   FØRM.  IF FAIL IS ZERØ, THE REDUCTIØNS
C                   PRØCEEDED WITHØUT MISHAP.
C
C WHEN EPSA IS NEGATIVE THE REDUCTIØN ØF A TØ REAL SCHUR
C FØRM IS SKIPPED AND THE ARRAYS A AND U ARE ASSUMED TØ
C CØNTAIN THE SCHUR FØRM AND ACCØMPANYING ØRTHØGØNAL MATRIX.
C THIS PERMITS THE EFFICIENT SØLUTIØN ØF SEVERAL EQUATIØNS
C ØF THE FØRM AX + BX = C WHEN A DØES NØT CHANGE.  LIKEWISE,
C IF EPSB IS NEGATIVE, THE REDUCTIØN ØF B TO REAL SCHUR FØRM
C IS SKIPPED.
C
      REAL
     1A(NA,1),U(NU,1),B(NB,1),V(NV,1),C(NC,1),EPSA,EPSB,TEMP
      INTEGER
     1M,NA,NU,N,NB,NV,NC,FAIL,M1,MM1,N1,NM1,I,J,K
      M1 = M+1
      MM1 = M-1
      N1 = N+1
      NM1 = N-1
C
C IF REQUIRED, REDUCE A TØ UPPER REAL SCHUR FØRM.
C
      IF(EPSA .LT. 0.) GØ TØ 35
      DØ 10 I=1,M
        DØ 10 J=I,M
          TEMP = A(I,J)
          A(I,J) = A(J,I)
          A(J,I) = TEMP
   10 CØNTINUE
      CALL HSHLDR(A,M,NA)
      CALL BCKMLT(A,U,M,NA,NU)
      IF(MM1 .EQ. 0) GØ TØ 25
      DØ 20 I=1,MM1
        A(I+1,I) = A(I,M1)
   20 CØNTINUE
      CALL SCHUR(A,U,M,NA,NU,EPSA,FAIL)
       IF(FAIL .NE. 0) RETURN
   25 DØ 30 I=1,M
        DØ 30 J=I,M
          TEMP = A(I,J)
          A(I,J) = A(J,I)
          A(J,I) = TEMP
   30 CØNTINUE
C
C IF REQUIRED, REDUCE B TØ UPPER REAL SCHUR FØRM.
C
   35 IF(EPSB .LT. 0.) GØ TØ 45
      CALL HSHLDR(B,N,NB)
      CALL BCKMLT(B,V,N,NB,NV)
      IF(NM1 .EQ. 0) GØ TØ 45
      DØ 40 I=1,NM1
        B(I+1,I) = B(I,N1)
   40 CØNTINUE
      CALL SCHUR(B,V,N,NB,NV,EPSB,FAIL)
      FAIL = -FAIL
      IF(FAIL .NE. 0) RETURN
C
C TRANSFØRM C.
C
   45 DØ 60 J=1,N
        DØ 50 I=1,M
          A(I,M1) = 0.
          DØ 50 K=1,M
            A(I,M1) = A(I,M1) + U(K,I)*C(K,J)
   50   CØNTINUE
        DØ 60 I=1,M
          C(I,J) = A(I,M1)
   60 CØNTINUE
      DØ 80 I=1,M
        DØ 70 J=1,N
          B(N1,J) = 0.
          DØ 70 K=1,N
            B(N1,J) = B(N1,J) + C(I,K)*V(K,J)
   70   CØNTINUE
        DØ 80 J=1,N
          C(I,J) = B(N1,J)
   80 CØNTINUE
C
C SØLVE THE TRANSFØRMED SYSTEM.
C
      CALL SHRSLV(A,B,C,M,N,NA,NB,NC)
C
C TRANSFØRM C BACK TØ THE SØLUTIØN.
C
      DØ 100 J=1,N
        DØ 90 I=1,M
          A(I,M1) = 0.
          DØ 90 K=1,M
            A(I,M1) = A(I,M1) + U(I,K)*C(K,J)
   90   CØNTINUE
        DØ 100 I=1,M
          C(I,J) = A(I,M1)
  100 CØNTINUE
      DØ 120 I=1,M
        DØ 110 J=1,N
          B(N1,J) = 0.
          DØ 110 K=1,N
            B(N1,J) = B(N1,J) + C(I,K)*V(J,K)
  110   CØNTINUE
        DØ 120 J=1,N
          C(I,J) = B(N1,J)
  120 CØNTINUE
      RETURN
      END
      SUBRØUTINE SHRSLV(A,B,C,M,N,NA,NB,NC)
C SHRSLV IS A FØRTRAN IV SUBRØUTINE TØ SØLVE THE REAL MATRIX
C EQATIØN AX + XB = C, WHERE A IS IN LØWER REAL SCHUR FØRM
C AND B IS IN UPPER REAL SCHUR FØRM.  SHRSLV USES THE AUX-
C ILIARY SUBRØUTINE SYSSLV, WHICH IT CØMMUNICATES WITH
C THRØUGH THE CØMMØN BLØCK SLVBLK.  THE PARAMETERS IN THE
C ARGUMENT LIST ARE
C           A       A DØUBLY SUBSCRIPTED ARRAY CØNTAINING THE
C                   MATRIX A IN LØWER REAL SCHUR FØRM.
C           B       A DØUBLY SUBSCRIPTED ARRAY CØNTAINING THE
C                   MATRIX B IN UPPER REAL SCHUR FØRM.
C           C       A DØUBLY SUBSCRIPTED ARRAY CØNTAINING THE
C                   MATRIX C.
C           M       THE ØRDER ØF THE MATRIX A.
```

```
C              N        THE ØRDER ØF THE MATRIX B.
C              NA       THE FIRST DIMENSIØN ØF THE ARRAY A.
C              NB       THE FIRST DIMENSIØN ØF THE ARRAY B.
C              NC       THE FIRST DIMENSIØN ØF THE ARRAY C.
C
      REAL
     1A(NA,1),B(NB,1),C(NC,1),T,P
      INTEGER
     1M,N,NA,NB,NC,K,KM1,DK,KK,L,LM1,DL,LL,I,IB,J,JA,NSYS
      CØMMØN/SLVBLK/T(5,5),P(5),NSYS
      L = 1
   10   LM1 = L-1
        DL = 1
        IF(L .EQ. N) GØ TØ 15
        IF(B(L+1,L) .NE. 0.) DL = 2
   15   LL = L+DL-1
        IF(L .EQ. 1) GØ TØ 30
        DØ 20 J=L,LL
          DØ 20 I=1,M
            DØ 20 IB=1,LM1
              C(I,J) = C(I,J) - C(I,IB)*B(IB,J)
   20   CØNTINUE
   30   K = 1
   40     KM1 = K-1
          DK = 1
          IF(K .EQ. M) GØ TØ 45
          IF(A(K,K+1) .NE. 0.) DK = 2
   45     KK = K+DK-1
          IF(K .EQ. 1) GØ TØ 60
          DØ 50 I=K,KK
            DØ 50 J=L,LL
              DØ 50 JA=1,KM1
                C(I,J) = C(I,J) - A(I,JA)*C(JA,J)
   50     CØNTINUE
   60     IF(DL .EQ. 2) GØ TØ 80
          IF(DK .EQ. 2) GØ TØ 70
          T(1,1) = A(K,K) + B(L,L)
          IF(T(1,1) .EQ. 0.) STØP
          C(K,L) = C(K,L)/T(1,1)
          GØ TØ 100
   70     T(1,1) = A(K,K) + B(L,L)
          T(1,2) = A(K,KK)
          T(2,1) = A(KK,K)
          T(2,2) = A(KK,KK) + B(L,L)
          P(1) = C(K,L)
          P(2) = C(KK,L)
          NSYS = 2
          CALL SYSSLV
          C(K,L) = P(1)
          C(KK,L) = P(2)
          GØ TØ 100
   80     IF(DK .EQ. 2) GØ TØ 90
          T(1,1) = A(K,K) + B(L,L)
          T(1,2) = B(LL,L)
          T(2,1) = B(L,LL)
          T(2,2) = A(K,K) + B(LL,LL)
          P(1) = C(K,L)
          P(2) = C(K,LL)
          NSYS = 2
          CALL SYSSLV
          C(K,L) = P(1)
          C(K,LL) = P(2)
          GØ TØ 100
   90     T(1,1) = A(K,K) + B(L,L)
          T(1,2) = A(K,KK)
          T(1,3) = B(LL,L)
          T(1,4) = 0.
          T(2,1) = A(KK,K)
          T(2,2) = A(KK,KK) + B(L,L)
          T(2,3) = 0.
          T(2,4) = T(1,3)
          T(3,1) = B(L,LL)
          T(3,2) = 0.
          T(3,3) = A(K,K) + B(LL,LL)
          T(3,4) = T(1,2)
          T(4,1) = 0.
          T(4,2) = T(3,1)
          T(4,3) = T(2,1)
          T(4,4) = A(KK,KK) + B(LL,LL)
          P(1) = C(K,L)
          P(2) = C(KK,L)
          P(3) = C(K,LL)
          P(4) = C(KK,LL)
          NSYS = 4
          CALL SYSSLV
          C(K,L) = P(1)
          C(KK,L) = P(2)
          C(K,LL) = P(3)
          C(KK,LL) = P(4)
  100   K = K + DK
          IF(K .LE. M) GØ TØ 40
      L = L + DL
      IF(L .LE. N) GØ TØ 10
      RETURN
      END
      SUBRØUTINE ATXPXA(A,U,C,N,NA,NU,NC,EPS,FAIL)
C
C ATXPXA IS A FØRTRAN IV SUBRØUTINE TØ SØLVE THE REAL MATRIX
C EQUATIØN TRANS(A)*X + X*A = C, WHERE C IS SYMMETRIC AND
C TRANS(A) DENØTES THE TRANSPØSE ØF A.  THE EQUATIØN IS
C TRANSFØRMED SØ THAT A IS IN UPPER REAL SCHUR FØRM, AND THE
C TRANSFØRMED EQUATIØN IS SØLVED BY A RECURSIVE PRØCEDURE.
C THE PRØGRAM REQUIRES THE AUXILIARY SUBRØUTINES HSHLDR,
C BCKMLT, SCHUR, AND SYMSLV.  THE PARAMETERS IN THE ARGUMENT
C LIST ARE
C              A        A DØUBLY SUBSCRIPTED ARRAY CØNTAINING THE
C                       MATRIX A.  ØN RETURN, THE UPPER TRIANGLE
C                       AND THE FIRST SUBDIAGØNAL ØF THE ARRAY A
C                       CØNTAIN AN UPPER REAL SCHUR FØRM ØF A.
C                       THE ARRAY A MUST BE DIMENSIØNED AT LEAST
C                       N+1 BY N+1.
C              U        A DØUBLY SUBSCRIPTED ARRAY THAT, ØN
C                       RETURN, CØNTAINS THE ØRTHØGØNAL MATRIX
C                       THAT REDUCES A TØ UPPER REAL SCHUR FØRM.
C              C        A DØUBLY SUBSCRIPTED ARRAY CØNTAINING THE
C                       MATRIX C.  ØN RETURN, C CØNTAINS THE
C                       SØLUTIØN MATRIX X.
C              N        THE ØRDER ØF THE MATRIX A.
C              NA       THE FIRST DIMENSIØN ØF THE ARRAY A.
C              NU       THE FIRST DIMENSIØN ØF THE ARRAY U.
C              NC       THE FIRST DIMENSIØN ØF THE ARRAY C.
C              EPS      A CØNVERGENCE CRITERIØN FØR THE REDUCTIØN
C                       ØF A TØ REAL SCHUR FØRM.  EPS SHØULD BE
C                       SET SLIGHTLY SMALLER THAN 10.**(-N),
C                       WHERE N IS THE NUMBER ØF SIGNIFICANT
C                       DIGITS IN THE ELEMENTS ØF THE MATRIX A.
C              FAIL     AN INTEGER VARIABLE THAT, ØN RETURN,
C                       CØNTAINS AN ERRØR SIGNAL.  IF FAIL IS
C                       NØNZERØ, THEN THE PRØGRAM WAS UNABLE TØ
C                       REDUCE A TØ REAL SCHUR FØRM.  IF FAIL IS
C                       ZERØ, THE REDUCTIØN PRØCEEDED WITHØUT
C                       MISHAP.
C
C WHEN EPS IS NEGATIVE, THE REDUCTIØN ØF A TØ REAL SCHUR
C FØRM IS SKIPPED AND THE ARRAYS A AND U ARE ASSUMED TØ
C CØNTAIN THE SCHUR FØRM AND ACCØMPANYING ØRTHØGØNAL MATRIX.
C THIS PERMITS THE EFFICIENT SØLUTIØN ØF SEVERAL EQUATIØNS
C WITH DIFFERENT RIGHT HAND SIDES.
C
      REAL
     1A(NA,1),U(NU,1),C(NC,1),EPS
      INTEGER
     1N,NA,NU,NC,FAIL,N1,NM1,I,J,K
      N1 = N+1
      NM1 = N-1
C
C IF REQUIRED, REDUCE A TØ LØWER REAL SCHUR FØRM.
C
      IF(EPS .LT. 0.) GØ TØ 15
      CALL HSHLDR(A,N,NA)
      CALL BCKMLT(A,U,N,NA,NU)
      DØ 10 I=1,NM1
        A(I+1,I) = A(I,N1)
   10 CØNTINUE
      CALL SCHUR(A,U,N,NA,NU,EPS,FAIL)
      IF(FAIL .NE. 0) RETURN
C
C TRANSFØRM C.
C
   15 DØ 20 I=1,N
        C(I,I) = C(I,I)/2.
   20 CØNTINUE
      DØ 40 I=1,N
        DØ 30 J=1,N
          A(N1,J) = 0.
          DØ 30 K=I,N
            A(N1,J) = A(N1,J) + C(I,K)*U(K,J)
   30   CØNTINUE
        DØ 40 J=1,N
          C(I,J) = A(N1,J)
   40 CØNTINUE
      DØ 60 J=1,N
        DØ 50 I=1,N
          A(I,N1) = 0.
          DØ 50 K=1,N
            A(I,N1) = A(I,N1) + U(K,I)*C(K,J)
   50   CØNTINUE
        DØ 60 I=1,N
          C(I,J) = A(I,N1)
   60 CØNTINUE
      DØ 70 I=1,N
        DØ 70 J=I,N
          C(I,J) = C(I,J) + C(J,I)
          C(J,I) = C(I,J)
   70 CØNTINUE
C
C SØLVE THE TRANSFØRMED SYSTEM.
C
      CALL SYMSLV(A,C,N,NA,NC)
C
C TRANSFØRM C BACK TØ THE SØLUTIØN.
C
      DØ 80 I=1,N
        C(I,I) = C(I,I)/2.
   80 CØNTINUE
      DØ 100 I=1,N
        DØ 90 J=1,N
          A(N1,J) = 0.
          DØ 90 K=I,N
            A(N1,J) = A(N1,J) + C(I,K)*U(J,K)
   90   CØNTINUE
        DØ 100 J=1,N
          C(I,J) = A(N1,J)
  100 CØNTINUE
      DØ 120 J=1,N
        DØ 110 I=1,N
          A(I,N1) = 0.
          DØ 110 K=1,N
            A(I,N1) = A(I,N1) + U(I,K)*C(K,J)
  110   CØNTINUE
        DØ 120 I=1,N
          C(I,J) = A(I,N1)
  120 CØNTINUE
      DØ 130 I=1,N
        DØ 130 J=I,N
          C(I,J) = C(I,J) + C(J,I)
          C(J,I) = C(I,J)
  130 CØNTINUE
      RETURN
      END
      SUBRØUTINE SYMSLV(A,C,N,NA,NC)
C
C SYMSLV IS A FØRTRAN IV SUBRØUTINE TØ SØLVE THE REAL MATRIX
C EQUATIØN TRANS(A)*X + X*A = C, WHERE C IS SYMMETRIC, A IS
C IN UPPER REAL SCHUR FØRM, AND TRANS(A) DENØTES THE TRANS-
C PØSE ØF A.  SYMSLV USES THE AUXILIARY SUBRØUTINE SYSSLV,
C WHICH IT CØMMUNICATES WITH THRØUGH THE CØMMØN BLØCK
```

```
C SLVBLK.  THE PARAMETERS IN THE ARGUMENT LIST ARE
C          A       A DØUBLY SUBSCRIPTED ARRAY CØNTAINING THE
C                  MATRIX A IN UPPER REAL SCHUR FØRM.
C          C       A DØUBLY SUBSCRIPTED ARRAY CØNTAINING THE
C                  MATRIX C.
C          N       THE ØRDER ØF THE MATRIX A.
C          NA      THE FIRST DIMENSIØN ØF THE ARRAY A.
C          NC      THE FIRST DIMENSIØN ØF THE ARRAY C.
C
      REAL
     1A(NA,1),C(NC,1),T,P
      INTEGER
     1N,NA,NC,K,KK,DK,KM1,L,LL,DL,LDL,I,IA,J,NSYS
      CØMMØN/SLVBLK/T(5,5),P(5),NSYS
      L = 1
   10   DL = 1
        IF(L .EQ. N) GØ TØ 20
        IF(A(L+1,L) .NE. 0.) DL = 2
   20   LL = L+DL-1
        K = L
   30     KM1 = K-1
          DK = 1
          IF(K .EQ. N) GØ TØ 35
          IF(A(K+1,K) .NE. 0.) DK = 2
   35     KK = K+DK-1
          IF(K .EQ. L) GØ TØ 45
          DØ 40 I=K,KK
            DØ 40 J=L,LL
              DØ 40 IA=L,KM1
                C(I,J) = C(I,J) - A(IA,I)*C(IA,J)
   40     CØNTINUE
   45     IF(DL .EQ. 2) GØ TØ 60
          IF(DK .EQ. 2 ) GØ TØ 50
          T(1,1) = A(K,K) + A(L,L)
          IF(T(1,1) .EQ. 0.) STØP
          C(K,L) = C(K,L)/T(1,1)
          GØ TØ 90
   50     T(1,1) = A(K,K) + A(L,L)
          T(1,2) = A(KK,K)
          T(2,1) = A(K,KK)
          T(2,2) = A(KK,KK) + A(L,L)
          P(1) = C(K,L)
          P(2) = C(KK,L)
          NSYS = 2
          CALL SYSSLV
          C(K,L) = P(1)
          C(KK,L) = P(2)
          GØ TØ 90
   60     IF(DK .EQ. 2) GØ TØ 70
          T(1,1) = A(K,K) + A(L,L)
          T(1,2) = A(LL,L)
          T(2,1) = A(L,LL)
          T(2,2) = A(K,K) + A(LL,LL)
          P(1) = C(K,L)
          P(2) = C(K,LL)
          NSYS = 2
          CALL SYSSLV
          C(K,L) = P(1)
          C(K,LL) = P(2)
          GØ TØ 90
   70     IF(K .NE. L) GØ TØ 80
          T(1,1) = A(L,L)
          T(1,2) = A(LL,L)
          T(1,3) = 0.
          T(2,1) = A(L,LL)
          T(2,2) = A(L,L) + A(LL,LL)
          T(2,3) = T(1,2)
          T(3,1) = 0.
          T(3,2) = T(2,1)
          T(3,3) = A(LL,LL)
          P(1) = C(L,L)/2.
          P(2) = C(LL,L)
          P(3) = C(LL,LL)/2.
          NSYS = 3
          CALL SYSSLV
          C(L,L) = P(1)
          C(LL,L) = P(2)
          C(L,LL) = P(2)
          C(LL,LL) = P(3)
          GØ TØ 90
   80     T(1,1) = A(K,K) + A(L,L)
          T(1,2) = A(KK,K)
          T(1,3) = A(LL,L)
          T(1,4) = 0.
          T(2,1) = A(K,KK)
          T(2,2) = A(KK,KK) + A(L,L)
          T(2,3) = 0.
          T(2,4) = T(1,3)
          T(3,1) = A(L,LL)
          T(3,2) = 0.
          T(3,3) = A(K,K) + A(LL,LL)
          T(3,4) = T(1,2)
          T(4,1) = 0.
          T(4,2) = T(3,1)
          T(4,3) = T(2,1)
          T(4,4) = A(KK,KK) + A(LL,LL)
          P(1) = C(K,L)
          P(2) = C(KK,L)
          P(3) = C(K,LL)
          P(4) = C(KK,LL)
          NSYS = 4
          CALL SYSSLV
          C(K,L) = P(1)
          C(KK,L) = P(2)
          C(K,LL) = P(3)
          C(KK,LL) = P(4)
   90   K = K + DK
        IF(K .LE. N) GØ TØ 30
        LDL = L + DL
        IF(LDL .GT. N) RETURN
        DØ 120 J=LDL,N
          DØ 100 I=L,LL
            C(I,J) = C(J,I)
  100     CØNTINUE
          DØ 120 I=J,N
            DØ 110 K=L,LL
              C(I,J) = C(I,J) - C(I,K)*A(K,J) - A(K,I)*C(K,J)
  110       CØNTINUE
            C(J,I) = C(I,J)
  120   CØNTINUE
      L = LDL
      GØ TØ 10
      END
      SUBRØUTINE HSHLDR(A,N,NA)
C
C HSHLDR IS A FØRTRAN IV SUBRØUTINE TØ REDUCE A MATRIX TØ
C UPPER HESSENBERG FØRM BY ELEMENTARY HERMITIAN TRANSFØRMA-
C TIØNS (THE METHØD ØF HØUSEHØLDER).  THE PARAMETERS IN THE
C ARGUMENT LIST ARE
C          A       A DOUBLY SUBSCRIPTED ARRAY CØNTAINING THE
C                  MATRIX A.  ØN RETURN, THE UPPER TRIANGLE
C                  ØF THE ARRAY A MATRIX AND THE (N+1)-TH
C                  CØLUMN CØNTAIN THE SUBDIAGØNAL ELEMENTS
C                  ØF THE TRANSFORMED MATRIX.  ØN RETURN,
C                  THE LØWER TRIANGLE AND THE (N+1)-TH RØW
C                  ØF THE ARRAY A CØNTAIN A HISTØRY ØF THE
C                  TRANSFØRMATIØNS.
C          N       THE ØRDER ØF THE MATRIX A.
C          NA      THE FIRST DIMENSIØN ØF THE ARRAY A.
C
      REAL
     1A(NA,1),MAX,SUM,S,P
      INTEGER
     1N,NA,NM2,N1,L,L1,I,J
      NM2 = N-2
      N1 = N+1
      IF(N .EQ. 1) RETURN
      IF(N .GT. 2) GØ TØ 5
      A(1,N1) = A(2,1)
      RETURN
    5 DØ 80 L=1,NM2
        L1 = L+1
        MAX = 0.
        DØ 10 I=L1,N
          MAX = AMAX1(MAX,ABS(A(I,L)))
   10   CØNTINUE
        IF(MAX .NE. 0.) GØ TØ 20
        A(L,N1) = 0.
        A(N1,L) = 0.
        GØ TØ 80
   20   SUM = 0.
        DØ 30 I=L1,N
          A(I,L) = A(I,L)/MAX
          SUM = SUM + A(I,L)**2
   30   CØNTINUE
        S = SIGN(SQRT(SUM),A(L1,L))
        A(L,N1) = -MAX*S
        A(L1,L) = S + A(L1,L)
        A(N1,L) = S*A(L1,L)
        DØ 50 J=L1,N
          SUM = 0.
          DØ 40 I=L1,N
            SUM = SUM + A(I,L)*A(I,J)
   40     CØNTINUE
          P = SUM/A(N1,L)
          DØ 50 I=L1,N
            A(I,J) = A(I,J) - A(I,L)*P
   50   CØNTINUE
        DØ 70 I=1,N
          SUM = 0.
          DØ 60 J=L1,N
            SUM = SUM + A(I,J)*A(J,L)
   60     CØNTINUE
          P = SUM/A(N1,L)
          DØ 70 J=L1,N
            A(I,J) = A(I,J) - P*A(J,L)
   70   CØNTINUE
   80 CØNTINUE
      A(N-1,N1) = A(N,N-1)
      RETURN
      END
      SUBRØUTINE BCKMLT(A,U,N,NA,NU)
C
C BCKMLT IS A FØRTRAN IV SUBRØUTINE THAT, GIVEN THE ØUTPUT
C ØF THE SUBRØUTINE HSHLDR, CØMPUTES THE ØRTHØGØNAL MATRIX
C THAT REDUCES A TØ UPPER HESSENBERG FØRM.  THE PARAMETERS
C IN THE ARGUMENT LIST ARE
C          A       A DØUBLY SUBSCRIPTED ARRAY CØNTAINING THE
C                  ØUTPUT FRØM HSHLDR.
C          U       A DØUBLY SUBSCRIPTED ARRAY THAT, ØN
C                  RETURN, CØNTAINS THE ØRTHØGØNAL MATRIX.
C          N       THE ØRDER ØF THE MATRIX A IN HSHLDR.
C          NA      THE FIRST DIMENSIØN ØF THE ARRAY A.
C          NU      THE FIRST DIMENSIØN ØF THE ARRAY U.
C
C THE ARRAYS A AND U MAY BE IDENTIFIED IN THE CALLING
C SEQUENCE.  IF THIS IS DØNE, THE ELEMENTS ØF THE ØRTHØGØNAL
C MATRIX WILL ØVERWRITE THE ØUTPUT ØF HSHLDR.
C
      REAL
     1A(NA,1),U(NU,1),SUM,P
      INTEGER
     1N,NA,N1,NM1,NM2,LL,L,L1,I,J
      N1 = N+1
      NM1 = N-1
      NM2 = N-2
      U(N,N) = 1.
      IF(NM1 .EQ. 0) RETURN
      U(NM1,N) = 0.
      U(N,NM1) = 0.
      U(NM1,NM1) = 1.
      IF(NM2 .EQ. 0) RETURN
      DØ 40 LL=1,NM2
        L = NM2-LL+1
        L1 = L+1
        IF(A(N1,L) .EQ. 0.) GØ TØ 25
      DØ 20 J=L1,N
        SUM = 0.
        DØ 10 I=L1,N
          SUM = SUM + A(I,L)*U(I,J)
 10     CØNTINUE
        P = SUM/A(N1,L)
        DØ 20 I=L1,N
          U(I,J) = U(I,J) - A(I,L)*P
 20   CØNTINUE
 25   DØ 30 I=L1,N
        U(I,L) = 0.
        U(L,I) = 0.
 30   CØNTINUE
      U(L,L) = 1.
 40 CØNTINUE
    RETURN
    END
    SUBRØUTINE SCHUR(H,U,NN,NH,NU,EPS,FAIL)
C
C SCHUR IS A FØRTRAN IV SUBRØUTINE TØ REDUCE AN UPPER
C HESSENBERG MATRIX TØ REAL SCHUR FØRM BY THE QR METHØD WITH
C IMPLICIT ØRIGIN SHIFTS.  THE PRØDUCT ØF THE TRANSFØRMA-
C TIØNS USED IN THE REDUCTIØN IS ACCUMULATED.  SCHUR IS AN
C ADAPTATIØN ØF THE ALGØL PRØGRAM HQR BY MARTIN, PETERS, AND
C WILKINSØN (NUMER. MATH. 14 (1970) 219-231).  THE PARA-
C METERS IN THE ARGUMENT LIST ARE
C           H       A DØUBLY SUBSCRIPTED ARRAY CØNTAINING THE
C                   UPPER HESSENBERG MATRIX H.  ØN RETURN, H
C                   CØNTAINS AN UPPER REAL SCHUR FØRM ØF H.
C                   THE ELEMENTS ØF THE ARRAY H BELØW THE
C                   THIRD SUBDIAGØNAL ARE UNDISTURBED.
C           U       A DØUBLY SUBSCRIPTED ARRAY CØNTAINING ANY
C                   MATRIX.  ØN RETURN, U CØNTAINS THE MATRIX
C                   U*R(1)*R(2)..., WHERE R(I) ARE THE TRANS-
C                   FØRMATIØNS USED IN THE REDUCTIØN ØF H.
C           NN      THE ØRDER ØF THE MATRICES H AND U.
C           NH      THE FIRST DIMENSIØN ØF THE ARRAY H.
C           NU      THE FIRST DIMENSIØN ØF THE ARRAY U.
C           EPS     A NUMBER USED IN DETERMINING WHEN AN
C                   ELEMENT ØF H IS NEGLIGIBLE.  H(I,J) IS
C                   NEGLIGIBLE IF ABS(H(I,J)) IS LESS THAN ØR
C                   EQUAL TØ EPS TIMES THE INFINITY NØRM ØF
C                   H.
C           FAIL    AN INTEGER VARIABLE THAT, ØN RETURN,
C                   CØNTAINS AN ERRØR SIGNAL. IF FAIL IS
C                   PØSITIVE, THEN THE PRØGRAM FAILED TØ MAKE
C                   THE FAIL-1 ØR FAIL-2 SUBDIAGØNAL ELEMENT
C                   NEGLIGIBLE AFTER 30 ITERATIØNS.
C
     REAL
    1H(NH,1),U(NU,1),EPS,HN,RSUM,TEST,P,Q,R,S,W,X,Y,Z
     INTEGER
    1NN,NA,NH,FAIL,I,ITS,J,JL,K,L,LL,M,MM,M2,M3,N,NA
     LØGICAL
    1LAST
     N = NN
     HN = 0.
     DØ 20 I=1,N
       JL = MAX0(1,I-1)
       RSUM = 0.
       DØ 10 J=JL,N
         RSUM = RSUM + ABS(H(I,J))
 10    CØNTINUE
       HN = AMAX1(HN,RSUM)
 20  CØNTINUE
     TEST = EPS*HN
     IF(HN .EQ. 0.) GØ TØ 230
 30  IF(N .LE. 1) GØ TØ 230
     ITS = 0
     NA = N-1
     NM2 = N-2
 40  DØ 50 LL=2,N
     L = N-LL+2
       IF(ABS(H(L,L-1)) .LE. TEST) GØ TØ 60
 50  CØNTINUE
     L = 1
     GØ TØ 70
 60  H(L,L-1) = 0.
 70  IF(L .LT. NA) GØ TØ 72
     N = L-1
     GØ TØ 30
 72  X = H(N,N)/HN
     Y = H(NA,NA)/HN
     R = (H(N,NA)/HN)*(H(NA,N)/HN)
     IF(ITS .LT. 30) GØ TØ 75
     FAIL = N
     RETURN
 75  IF(ITS.EQ.10 .ØR. ITS.EQ.20) GØ TØ 80
     S = X + Y
     Y = X*Y - R
     GØ TØ 90
 80  Y = (ABS(H(N,NA)) + ABS(H(NA,NM2)))/HN
     S = 1.5*Y
     Y = Y**2
 90  ITS = ITS + 1
     DØ 100 MM=L,NM2
       M = NM2-MM+L
       X = H(M,M)/HN
       R = H(M+1,M)/HN
       Z = H(M+1,M+1)/HN
       P = X*(X-S) + Y + R*(H(M,M+1)/HN)
       Q = R*(X+Z-S)
       R = R*(H(M+2,M+1)/HN)
       W = ABS(P) + ABS(Q) + ABS(R)
       P = P/W
       Q = Q/W
       R = R/W
       IF(M .EQ. L) GØ TØ 110
     IF(ABS(H(M,M-1))*(ABS(Q)+ABS(R)) .LE. ABS(P)*TEST)
    1GØ TØ 110
100  CØNTINUE
110  M2 = M+2
     M3 = M+3
     DØ 120 I=M2,N
       H(I,I-2) = 0.
120  CØNTINUE
     IF(M3 .GT. N) GØ TØ 140
     DØ 130 I=M3,N
       H(I,I-3) = 0.
130  CØNTINUE
140  DØ 220 K=M,NA
       LAST = K.EQ.NA
       IF(K .EQ. M) GØ TØ 150
       P = H(K,K-1)
       Q = H(K+1,K-1)
       R = 0.
       IF(.NØT.LAST) R = H(K+2,K-1)
       X = ABS(P) + ABS(Q) + ABS(R)
       IF(X .EQ. 0.) GØ TØ 220
       P = P/X
       Q = Q/X
       R = R/X
150    S = SQRT(P**2 + Q**2 + R**2)
       IF(P .LT. 0.) S = -S
       IF(K .NE. M) H(K,K-1) = -S*X
       IF(K.EQ.M .AND. L.NE.M) H(K,K-1) = -H(K,K-1)
       P = P + S
       X = P/S
       Y = Q/S
       Z = R/S
       Q = Q/P
       R = R/P
       DØ 170 J=K,NN
         P = H(K,J) + Q*H(K+1,J)
         IF(LAST) GØ TØ 160
         P = P + R*H(K+2,J)
         H(K+2,J) = H(K+2,J) - P*Z
160      H(K+1,J) = H(K+1,J) - P*Y
         H(K,J) = H(K,J) - P*X
170    CØNTINUE
       J = MIN0(K+3,N)
       DØ 190 I=1,J
         P = X*H(I,K) + Y*H(I,K+1)
         IF(LAST) GØ TØ 180
         P = P + Z*H(I,K+2)
         H(I,K+2) = H(I,K+2) - P*R
180      H(I,K+1) = H(I,K+1) - P*Q
         H(I,K) = H(I,K) - P
190    CØNTINUE
       DØ 210 I=1,NN
         P = X*U(I,K) + Y*U(I,K+1)
         IF(LAST) GØ TØ 200
         P = P + Z*U(I,K+2)
         U(I,K+2) = U(I,K+2) - P*R
200      U(I,K+1) = U(I,K+1) - P*Q
         U(I,K) = U(I,K) - P
210    CØNTINUE
220  CØNTINUE
     GØ TØ 40
230  FAIL = 0
     RETURN
     END
     SUBRØUTINE SYSSLV
C
C SYSSLV IS A FØRTRAN IV SUBRØUTINE THAT SØLVES THE LINEAR
C SYSTEM AX = B ØF ØRDER N LESS THAN 5 BY CRØUT REDUCTIØN
C FØLLØWED BY BACK SUBSTITUTIØN.  THE MATRIX A, THE VECTØR
C B, AND THE ØRDER N ARE CØNTAINED IN THE ARRAYS A,B, AND
C THE VARIABLE N ØF THE CØMMØN BLØCK SLVBLK.  THE SØLUTIØN
C IS RETURNED IN THE ARRAY B.
C
      CØMMØN/SLVBLK/A(5,5),B(5),N
      REAL MAX
    1 NM1 = N-1
      N1 = N+1
C
C CØMPUTE THE LU FACTØRIZATIØN ØF A.
C
      DØ 80 K=1,N
        KM1 = K-1
        IF(K.EQ.1) GØ TØ 20
        DØ 10 I=K,N
          DØ 10 J=1,KM1
            A(I,K) = A(I,K) - A(I,J)*A(J,K)
  10    CØNTINUE
  20    IF(K.EQ.N) GØ TØ 100
        KP1 = K+1
      MAX = ABS(A(K,K))
        INTR = K
        DØ 30 I=KP1,N
          AA = ABS(A(I,K))
          IF(AA .LE. MAX) GØ TØ 30
          MAX = AA
          INTR = I
  30    CØNTINUE
        IF(MAX .EQ. 0.) STØP
        A(N1,K) = INTR
        IF(INTR .EQ. K) GØ TØ 50
        DØ 40 J=1,N
          TEMP = A(K,J)
          A(K,J) = A(INTR,J)
          A(INTR,J) = TEMP
  40    CØNTINUE
  50    DØ 80 J=KP1,N
          IF(K.EQ.1) GØ TØ 70
          DØ 60 I=1,KM1
            A(K,J) = A(K,J) - A(K,I)*A(I,J)
  60      CØNTINUE
  70      A(K,J) = A(K,J)/A(K,K)
  80  CØNTINUE
C
C INTERCHANGE THE CØMPØNENTS ØF B.
C
 100  DØ 110 J=1,NM1
        INTR = A(N1,J)
        IF(INTR .EQ. J) GØ TØ 110
        TEMP = B(J)
        B(J) = B(INTR)
        B(INTR) = TEMP
 110  CØNTINUE
```

```
C
C SØLVE LX = B.
C
  200 B(1) = B(1)/A(1,1)
      DØ 220 I=2,N
        IM1 = I-1
        DØ 210 J=1,IM1
          B(I) = B(I) - A(I,J)*B(J)
  210   CØNTINUE
        B(I) = B(I)/A(I,I)
  220 CØNTINUE
C
C SØLVE UX = B.
C
  300 DØ 310 II=1,NM1
        I = NM1-II+1
        I1 = I+1
        DØ 310 J=I1,N
           B(I) = B(I) - A(I,J)*B(J)
  310 CØNTINUE
      RETURN
      END
```

Editor's note: Algorithm 433 described here is available on magnetic tape from the Department of Computer Science, University of Colorado, Boulder, CO 80302. The cost for the tape is $16.00 (U.S. and Canada) of $18.00 (elsewhere). If the user sends a small tape (wt. less than 1 lb.) the algorithm will be copied on it and returned to him at a charge of $10.00 (U.S. only). All orders are to be prepaid with checks payable to ACM Algorithms. The algorithm is recorded as one file of BCD 80 character card images at 556 B.P.I., even parity, on seven track tape. We will supply the algorithm at a density of 800 B.P.I. if requested. The cards for the algorithm are sequenced starting at 10 and incremented by 10. The sequence number is right justified in columns 80. Although we will make every attempt to insure that the algorithm conforms to the description printed here, we cannot guarantee it, nor can we guarantee that the algorithm is correct.—L.D.F.

Algorithm 433

Interpolation and Smooth Curve Fitting Based on Local Procedures [E2]

Hiroshi Akima [3 Nov. 1970, 9 Apr. 1971, and 1 Mar. 1972]
U.S. Department of Commerce, Office of Telecommunications, Institute for Telecommunication Sciences, Boulder, CO 80302

Key Words and Phrases: interpolation, polynomial, slope of curve, smooth curve fitting
CR Categories: 5.13
Language: Fortran

Description

Introduction. User information and Fortran listings are given on two subroutines, *INTRPL* and *CRVFIT*. Each subroutine implements the method of interpolation and smooth curve fitting based on local procedures [1]. These subroutines are written in ANSI Standard Fortran [2].

Outline of the Method. The method is devised in such a way that the resulting curve will pass through all the given data points and appear smooth and natural. It is based on a piecewise function; a portion of the curve between a pair of given points is represented by a third-degree polynomial for a single-valued function and by two third-degree polynomials for a multiple-valued function. In this method, the slope of the curve is determined at each given data point locally by the coordinates of five data points, with the data point in question as a center point and two data points on each side of it. Each piece of the function representing a portion of the curve between a pair of given data points is determined by the coordinates of and the slopes at the points.

When interpolation is made near the end points of the curve, two more points estimated at each end point are used to determine the slope of the curve. In this method, this estimation is based on three data points, the end point in question and two adjacent given data points.

The resulting curve of this method for a single-valued function is invariant under a linear-scale transformation of the coordinate system; different scalings of the coordinates result in equivalent curves. The resulting curve of this method for a multiple-valued function, on the other hand, is variant under a linear-scale transformation of the coordinate system; both the abscissa and the ordinate should be scaled with their respective units having an equal length on the graph.

This method requires only straightforward procedures, not iterative solutions of equations with preassigned error tolerances, which are required by some methods. No problem concerning computational stability or convergence exists in application of this method.

The INTRPL Subroutine. This subroutine interpolates, from values of the function given as ordinates of input data points in an x-y plane and for a given set of x values (abscissas of desired points), the values of a single-valued function $y = y(x)$.

The entrance to this subroutine is achieved by

CALL INTRPL(IU,L,X,Y,N,U,V)

where the input parameters are

IU = logical unit number of standard output unit,
L = number of input data points (must be two or greater),
X = array of dimension L storing the x values (abscissas) of input data points in ascending order,
Y = array of dimension L storing the y values (ordinates) of input data points,
N = number of points at which interpolation of the y value (ordinate) is desired (must be one or greater),
U = array of dimension N storing the x values (abscissas) of desired points,

and the output parameter is

V = array of dimension N where the interpolated y values (ordinates) are to be displayed.

This subroutine occupies 515 locations on the CDC-3800 computer. Computation time required for this subroutine on the same computer is approximately equal to

$1 + 0.2\,N$ msec for $L = 10$,
$3 + 0.5\,N$ msec for $L = 100$,

when the elements of the U array are given in ascending order; and

$1 + 0.5\,N$ msec for $L = 10$,
$3 + 0.7\,N$ msec for $L = 100$,

when they are given in random order.

When the function to be interpolated represents a periodic function and a set of L_p data points covers a whole period, two additional data points should be added at each end and a set of $L_p + 4$ data points should be given as the input data points to this subroutine.

The CRVFIT Subroutine. This subroutine fits a smooth curve to a given set of input data points in an x-y plane. It interpolates points in each interval between a pair of data points and generates a set of output points consisting of the input data points and the

Fig. 1. Curve fitted to the input data points given in Table I (a). (Encircled points are given data points.)

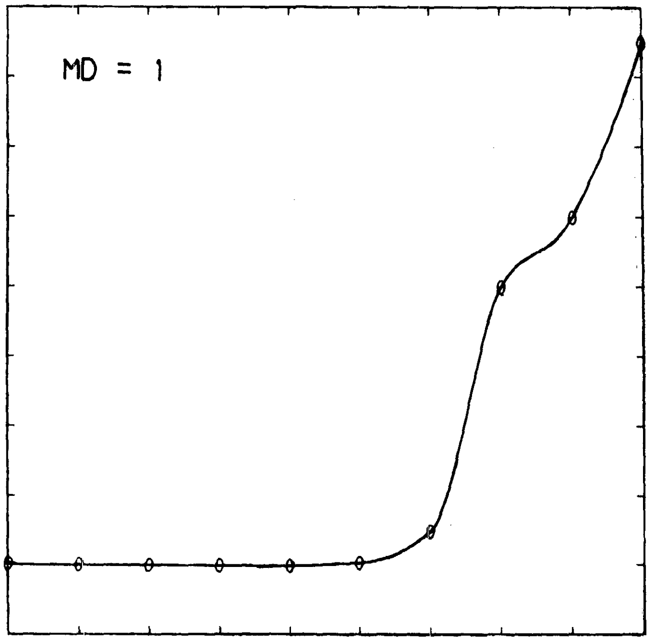


Fig. 2. Curve fitted to the input data points given in Table II (a). (Encircled points are given data points.)

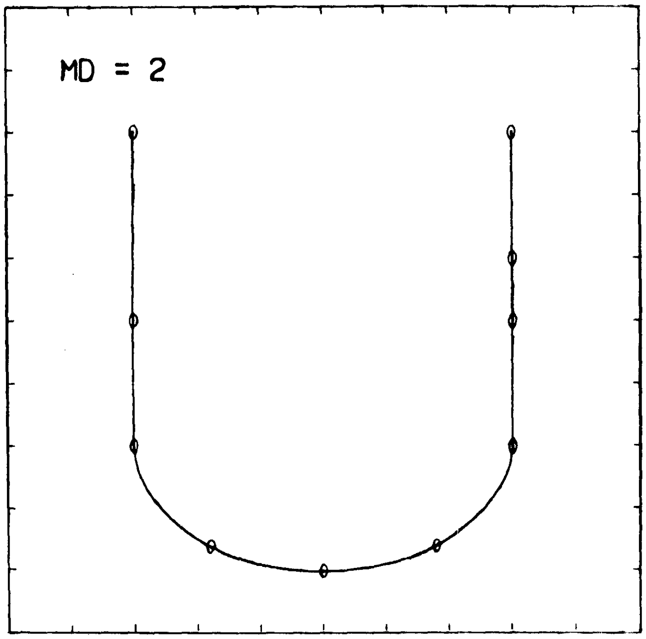


Table I. An Example of *CRVFIT* (*MD* = 1)

(a) Input data points

I	*X*(*I*)	*Y*(*I*)	*I*	*X*(*I*)	*Y*(*I*)	*I*	*X*(*I*)	*Y*(*I*)
1	0.000	0.000	4	3.000	0.000	7	6.000	10.000
2	1.000	0.000	5	4.000	0.000	8	7.000	80.000
3	2.000	0.000	6	5.000	1.000	9	8.000	100.000
4	3.000	0.000	7	6.000	10.000	10	9.000	150.000

(b) Output points

K	*U*(*K*)	*V*(*K*)	*K*	*U*(*K*)	*V*(*K*)	*K*	*U*(*K*)	*V*(*K*)
1	0.000	0.000	16	3.000	0.000	31	6.000	10.000
2	0.200	0.000	17	3.200	0.000	32	6.200	18.341
3	0.400	0.000	18	3.400	0.000	33	6.400	33.645
4	0.600	0.000	19	3.600	0.000	34	6.600	51.778
5	0.800	0.000	20	3.800	0.000	35	6.800	68.607
6	1.000	0.000	21	4.000	0.000	36	7.000	80.000
7	1.200	0.000	22	4.200	0.068	37	7.200	85.510
8	1.400	0.000	23	4.400	0.244	38	7.400	88.574
9	1.600	0.000	24	4.600	0.485	39	7.600	90.882
10	1.800	0.000	25	4.800	0.751	40	7.800	94.127
11	2.000	0.000	26	5.000	1.000	41	8.000	100.000
12	2.200	0.000	27	5.200	1.523	42	8.200	108.080
13	2.400	0.000	28	5.400	2.659	43	8.400	116.940
14	2.600	0.000	29	5.600	4.433	44	8.600	126.760
15	2.800	0.000	30	5.800	6.871	45	8.800	137.720
16	3.000	0.000	31	6.000	10.000	46	9.000	150.000

Table II. An Example of *CRVFIT* (*MD* = 2)

(a) Input data points

I	*X*(*I*)	*Y*(*I*)	*I*	*X*(*I*)	*Y*(*I*)	*I*	*X*(*I*)	*Y*(*I*)
1	−30.000	70.000	4	−18.000	4.000	7	30.000	20.000
2	−30.000	40.000	5	0.000	0.000	8	30.000	40.000
3	−30.000	20.000	6	18.000	4.000	9	30.000	50.000
4	−18.000	4.000	7	30.000	20.000	10	30.000	70.000

(b) Output points

K	*U*(*K*)	*V*(*K*)	*K*	*U*(*K*)	*V*(*K*)	*K*	*U*(*K*)	*V*(*K*)
1	−30.000	70.000	16	−18.000	4.000	31	30.000	20.000
2	−30.000	64.000	17	−14.641	2.463	32	30.000	24.000
3	−30.000	58.000	18	−11.097	1.331	33	30.000	28.000
4	−30.000	52.000	19	−7.433	0.567	34	30.000	32.000
5	−30.000	46.000	20	−3.713	0.136	35	30.000	36.000
6	−30.000	40.000	21	0.000	0.000	36	30.000	40.000
7	−30.000	36.000	22	3.713	0.136	37	30.000	42.000
8	−30.000	32.000	23	7.433	0.567	38	30.000	44.000
9	−30.000	28.000	24	11.097	1.331	39	30.000	46.000
10	−30.000	24.000	25	14.641	2.463	40	30.000	48.000
11	−30.000	20.000	26	18.000	4.000	41	30.000	50.000
12	−29.315	16.080	27	21.501	6.240	42	30.000	54.000
13	−27.466	12.400	28	24.758	9.080	43	30.000	58.000
14	−24.758	9.080	29	27.466	12.400	44	30.000	62.000
15	−21.501	6.240	30	29.315	16.080	45	30.000	66.000
16	−18.000	4.000	31	30.000	20.000	46	30.000	70.000

interpolated points. It can handle either a single-valued function or a multiple-valued function.

The entrance to this subroutine is achieved by

CALL CRVFIT(IU,MD,L,X,Y,M,N,U,V)

where the input parameters are

IU = logical unit number of standard output unit,
MD = mode of the curve (must be 1 or 2)
= 1 for a single-valued function
= 2 for a multiple-valued function,
L = number of input data points (must be two or greater),
X = array of dimension *L* storing the abscissas of input data points (in ascending or descending order for *MD* = 1),
Y = array of dimension *L* storing the ordinates of input data points,
M = number of subintervals between each pair of input data points (must be two or greater),
N = number of output points
= $(L-1)M + 1$,

and the output parameters are

U = array of dimension *N* where the abscissas of output points are to be displayed,
V = array of dimension *N* where the ordinates of output points are to be displayed.

This subroutine may also be entered by

CALL CRVFIT(IU,MD,L,X,Y,M,N,X,Y)

but the input data *X* and *Y* are not preserved in this case.

This subroutine occupies 711 locations on the CDC-3800 computer. Computation time required for this subroutine on the same computer is approximately

$500 + 300\,L + 50\,(L-1)\,(M-1)\ \mu\,\text{sec for}\ MD = 1,$
$500 + 600\,L + 75\,(L-1)\,(M-1)\ \mu\,\text{sec for}\ MD = 2.$

When the curve exhibits periodicity (that includes a closed curve) and a set of L_p data points covers a whole period, two additional data points should be added at each end, a set of L_p + 4 data points be given as the input data points to this subroutine, and two intervals on each side be discarded from the set of output points.

Test Results. All tests were performed on a CDC-3800 computer. An example of smooth curve fitting by the *CRVFIT* subroutine for a single-valued function (*MD* = 1) is shown in Table I, and for a multiple-valued function (*MD* = 2) in Table II. In each table, input data shown in (a) were given to *CRVFIT* with *L* = 10, *M* = 5, and *N* = 46, and values shown in (b) were obtained. Also, the data in Table I (a) together with the *U* values in Table I (b) were given to the *INTRPL* subroutine with *L* = 10 and *N* = 46, and the *V* values in Table I (b) were obtained. Figure 1 depicts the curve fitted to the input data points given in Table I (a) by the *CRVFIT* subroutine with *MD* = 1, and Figure 2, Table II (a) with *MD* = 2; both curves are fitted with *L* = 10, *M* = 20, and *N* = 181. These examples demonstrate one of the properties of this method, that the resulting curves are free from unnatural wiggles.

Acknowledgments. The author expresses his deep appreciation to Rayner K. Rosich of Office of Telecommunications and Jeanne M. Tucker of National Oceanic and Atmospheric Administration, both in Boulder, Colorado, for their critical review of this paper.

References

1. Akima, Hiroshi. A new method of interpolation and smooth curve fitting based on local procedures. *J. ACM 17*, 4 (Oct. 1970), 589–602.

2. ANSI Standard Fortran, Pub. X3.9-1966. American National Standards Institute, New York, N.Y. Also reproduced in Heising, W.P. History and summary of FORTRAN standardization development for the ASA. *Comm. ACM* 7 (Oct. 1964), 590–625.

Algorithm

```
      SUBRØUTINE  INTRPL(IU,L,X,Y,N,U,V)
C INTERPØLATIØN ØF A SINGLE-VALUED FUNCTIØN

C THIS SUBRØUTINE INTERPØLATES, FRØM VALUES ØF THE FUNCTIØN
C GIVEN AS ØRDINATES ØF INPUT DATA PØINTS IN AN X-Y PLANE
C AND FØR A GIVEN SET ØF X VALUES (ABSCISSAS), THE VALUES ØF
C A SINGLE-VALUED FUNCTIØN Y = Y(X).

C THE INPUT PARAMETERS ARE

C     IU = LØGICAL UNIT NUMBER ØF STANDARD ØUTPUT UNIT
C     L  = NUMBER ØF INPUT DATA PØINTS
C          (MUST BE 2 ØR GREATER)
C     X  = ARRAY ØF DIMENSIØN L STØRING THE X VALUES
C          (ABSCISSAS) ØF INPUT DATA PØINTS
C          (IN ASCENDING ØRDER)
C     Y  = ARRAY ØF DIMENSIØN L STØRING THE Y VALUES
C          (ØRDINATES) ØF INPUT DATA PØINTS
C     N  = NUMBER ØF PØINTS AT WHICH INTERPØLATIØN ØF THE
C          Y VALUE (ØRDINATE) IS DESIRED
C          (MUST BE 1 ØR GREATER)
C     U  = ARRAY ØF DIMENSIØN N STØRING THE X VALUES
C          (ABSCISSAS) ØF DESIRED PØINTS

C THE ØUTPUT PARAMETER IS

C     V  = ARRAY ØF DIMENSIØN N WHERE THE INTERPØLATED Y
C          VALUES (ØRDINATES) ARE TØ BE DISPLAYED

C DECLARATIØN STATEMENTS

      DIMENSIØN    X(L),Y(L),U(N),V(N)
      EQUIVALENCE  (P0,X3),(Q0,Y3),(Q1,T3)
      REAL         M1,M2,M3,M4,M5
      EQUIVALENCE  (UK,DX),(IMN,X2,A1,M1),(IMX,X5,A5,M5),
     1             (J,SW,SA),(Y2,W2,W4,Q2),(Y5,W3,Q3)

C PRELIMINARY PRØCESSING

   10 L0=L
      LM1=L0-1
      LM2=LM1-1
      LP1=L0+1
      N0=N
      IF(LM2.LT.0)          GØ TØ 90
      IF(N0.LE.0)           GØ TØ 91
      DØ 11  I=2,L0
        IF(X(I-1)-X(I))     11,95,96
   11   CØNTINUE
      IPV=0

C MAIN DØ-LØØP

      DØ 80  K=1,N0
        UK=U(K)

C RØUTINE TØ LØCATE THE DESIRED PØINT

   20   IF(LM2.EQ.0)        GØ TØ 27
        IF(UK.GE.X(L0))     GØ TØ 26
        IF(UK.LT.X(1))      GØ TØ 25
        IMN=2
        IMX=L0
   21   I=(IMN+IMX)/2
        IF(UK.GE.X(I))      GØ TØ 23
   22   IMX=I
        GØ TØ 24
   23   IMN=I+1
   24   IF(IMX.GT.IMN)      GØ TØ 21
        I=IMX
        GØ TØ 30
   25   I=1
        GØ TØ 30
   26   I=LP1
        GØ TØ 30
   27   I=2

C CHECK IF I = IPV

   30   IF(I.EQ.IPV)        GØ TØ 70
        IPV=I

C RØUTINES TØ PICK UP NECESSARY X AND Y VALUES AND
C           TØ ESTIMATE THEM IF NECESSARY

   40   J=I
        IF(J.EQ.1)          J=2
        IF(J.EQ.LP1)        J=L0
        X3=X(J-1)
        Y3=Y(J-1)
        X4=X(J)
        Y4=Y(J)
        A3=X4-X3
        M3=(Y4-Y3)/A3
        IF(LM2.EQ.0)        GØ TØ 43
        IF(J.EQ.2)          GØ TØ 41
        X2=X(J-2)
        Y2=Y(J-2)
        A2=X3-X2
        M2=(Y3-Y2)/A2
        IF(J.EQ.L0)         GØ TØ 42
   41   X5=X(J+1)
        Y5=Y(J+1)
        A4=X5-X4
        M4=(Y5-Y4)/A4
        IF(J.EQ.2)          M2=M3+M3-M4
        GØ TØ 45
   42   M4=M3+M3-M2
        GØ TØ 45
   43   M2=M3
        M4=M3
   45   IF(J.LE.3)          GØ TØ 46
        A1=X2-X(J-3)
        M1=(Y2-Y(J-3))/A1
        GØ TØ 47
   46   M1=M2+M2-M3
   47   IF(J.GE.LM1)        GØ TØ 48
        A5=X(J+2)-X5
        M5=(Y(J+2)-Y5)/A5
        GØ TØ 50
   48   M5=M4+M4-M3

C NUMERICAL DIFFERENTIATIØN

   50   IF(I.EQ.LP1)        GØ TØ 52
        W2=ABS(M4-M3)
        W3=ABS(M2-M1)
        SW=W2+W3
        IF(SW.NE.0.0)       GØ TØ 51
        W2=0.5
        W3=0.5
        SW=1.0
   51   T3=(W2*M2+W3*M3)/SW
        IF(I.EQ.1)          GØ TØ 54
   52   W3=ABS(M5-M4)
        W4=ABS(M3-M2)
        SW=W3+W4
        IF(SW.NE.0.0)       GØ TØ 53
        W3=0.5
        W4=0.5
        SW=1.0
   53   T4=(W3*M3+W4*M4)/SW
        IF(I.NE.LP1)        GØ TØ 60
        T3=T4
        SA=A2+A3
        T4=0.5*(M4+M5-A2*(A2-A3)*(M2-M3)/(SA*SA))
        X3=X4
        Y3=Y4
        A3=A2
        M3=M4
        GØ TØ 60
   54   T4=T3
        SA=A3+A4
        T3=0.5*(M1+M2-A4*(A3-A4)*(M3-M4)/(SA*SA))
        X3=X3-A4
        Y3=Y3-M2*A4
        A3=A4
        M3=M2

C DETERMINATIØN ØF THE CØEFFICIENTS

   60   Q2=(2.0*(M3-T3)+M3-T4)/A3
        Q3=(-M3-M3+T3+T4)/(A3*A3)

C CØMPUTATIØN ØF THE PØLYNØMIAL

   70   DX=UK-P0
   80   V(K)=Q0+DX*(Q1+DX*(Q2+DX*Q3))
      RETURN

C ERRØR EXIT

   90 WRITE (IU,2090)
      GØ TØ 99
   91 WRITE (IU,2091)
      GØ TØ 99
   95 WRITE (IU,2095)
      GØ TØ 97
   96 WRITE (IU,2096)
   97 WRITE (IU,2097)  I,X(I)
   99 WRITE (IU,2099)  LO,NO
      RETURN

C FØRMAT STATEMENTS

 2090 FØRMAT(1X/22H  ***   L = 1 ØR LESS./)
 2091 FØRMAT(1X/22H  ***   N = 0 ØR LESS./)
 2095 FØRMAT(1X/27H  ***   IDENTICAL X VALUES./)
 2096 FØRMAT(1X/33H  ***   X VALUES ØUT ØF SEQUENCE./)
 2097 FØRMAT(6H    I =,I7,10X,6HX(I) =,E12.3)
 2099 FØRMAT(6H    L =,I7,10X,3HN =,I7/
     1        36H ERRØR DETECTED IN RØUTINE    INTRPL)
      END



      SUBRØUTINE  CRVFIT(IU,MD,L,X,Y,M,N,U,V)
C SMØØTH CURVE FITTING

C THIS SUBRØUTINE FITS A SMØØTH CURVE TØ A GIVEN SET ØF IN-
C PUT DATA PØINTS IN AN X-Y PLANE.  IT INTERPØLATES PØINTS
C IN EACH INTERVAL BETWEEN A PAIR ØF DATA PØINTS AND GENER-
C ATES A SET ØF ØUTPUT PØINTS CØNSISTING ØF THE INPUT DATA
C PØINTS AND THE INTERPØLATED PØINTS.  IT CAN PRØCESS EITHER
C A SINGLE-VALUED FUNCTIØN ØR A MULTIPLE-VALUED FUNCTIØN.

C THE INPUT PARAMETERS ARE

C     IU = LØGICAL UNIT NUMBER ØF STANDARD ØUTPUT UNIT
C     MD = MØDE ØF THE CURVE (MUST BE 1 ØR 2)
C        = 1 FØR A SINGLE-VALUED FUNCTIØN
C        = 2 FØR A MULTIPLE-VALUED FUNCTIØN
C     L  = NUMBER ØF INPUT DATA PØINTS
C          (MUST BE 2 ØR GREATER)
C     X  = ARRAY ØF DIMENSIØN L STØRING THE ABSCISSAS ØF
C          INPUT DATA PØINTS (IN ASCENDING ØR DESCENDING
C          ØRDER FØR MD = 1)
C     Y  = ARRAY ØF DIMENSIØN L STØRING THE ØRDINATES ØF
C          INPUT DATA PØINTS
C     M  = NUMBER ØF SUBINTERVALS BETWEEN EACH PAIR ØF
C          INPUT DATA PØINTS (MUST BE 2 ØR GREATER)
C     N  = NUMBER ØF ØUTPUT PØINTS
C        = (L-1)*M+1

C THE ØUTPUT PARAMETERS ARE

C     U  = ARRAY ØF DIMENSIØN N WHERE THE ABSCISSAS ØF
C          ØUTPUT PØINTS ARE TØ BE DISPLAYED
C     V  = ARRAY ØF DIMENSIØN N WHERE THE ØRDINATES ØF
C          ØUTPUT PØINTS ARE TØ BE DISPLAYED


C DECLARATIØN STATEMENTS

      DIMENSIØN     X(L),Y(L),U(N),V(N)
      EQUIVALENCE   (M1,B1),(M2,B2),(M3,B3),(M4,B4),
     1              (X2,P0),(Y2,Q0),(T2,Q1)
      REAL          M1,M2,M3,M4
      EQUIVALENCE   (W2,Q2),(W3,Q3),(A1,P2),(B1,P3),
     1              (A2,DZ),(SW,R,Z)

C PRELIMINARY PRØCESSING

   10 MDO=MD
      MDM1=MDO-1
      LO=L
      LM1=LO-1
      MO=M
      MM1=MO-1
      NO=N
      IF(MDO.LE.0)        GØ TØ 90
      IF(MDO.GE.3)        GØ TØ 90
      IF(LM1.LE.0)        GØ TØ 91
      IF(MM1.LE.0)        GØ TØ 92
      IF(NO.NE.LM1*MO+1)  GØ TØ 93

      GØ TØ (11,16), MDO
   11 I=2
      IF(X(1)-X(2))       12,95,14
   12 DØ 13  I=3,LO
        IF(X(I-1)-X(I))   13,95,96
   13   CØNTINUE
      GØ TØ 18
   14 DØ 15  I=3,LO
        IF(X(I-1)-X(I))   96,95,15
   15   CØNTINUE
      GØ TØ 18
   16 DØ 17  I=2,LO
        IF(X(I-1).NE.X(I))  GØ TØ 17
        IF(Y(I-1).EQ.Y(I))  GØ TØ 97
   17   CØNTINUE

   18 K=NO+MO
      I=LO+1
      DØ 19  J=1,LO
        K=K-MO
        I=I-1
        U(K)=X(I)
   19   V(K)=Y(I)
      RM=MO
      RM=1.0/RM

C MAIN DØ-LØØP

   20 K5=MO+1
      DØ 80  I=1,LO

C RØUTINES TØ PICK UP NECESSARY X AND Y VALUES AND
C            TØ ESTIMATE THEM IF NECESSARY

        IF(I.GT.1)          GØ TØ 40
   30   X3=U(1)
        Y3=V(1)
        X4=U(MO+1)
        Y4=V(MO+1)
        A3=X4-X3
        B3=Y4-Y3
        IF(MDM1.EQ.0)       M3=B3/A3
        IF(LO.NE.2)         GØ TØ 41
        A4=A3
        B4=B3
   31   GØ TØ (33,32), MDO
   32   A2=A3+A3-A4
        A1=A2+A2-A3
   33   B2=B3+B3-B4
        B1=B2+B2-B3
        GØ TØ (51,56), MDO
 40     X2=X3
        Y2=Y3
        X3=X4
        Y3=Y4
        X4=X5
        Y4=Y5
        A1=A2
        B1=B2
        A2=A3
        B2=B3
        A3=A4
        B3=B4
        IF(I.GE.LM1)          GØ TØ 42
 41     K5=K5+MO
        X5=U(K5)
        Y5=V(K5)
        A4=X5-X4
        B4=Y5-Y4
        IF(MDM1.EQ.0)         M4=B4/A4
        GØ TØ 43
 42     IF(MDM1.NE.0)         A4=A3+A3-A2
        B4=B3+B3-B2
 43     IF(I.EQ.1)            GØ TØ 31
        GØ TØ (50,55), MDO

C NUMERICAL DIFFERENTIATIØN

 50     T2=T3
 51     W2=ABS(M4-M3)
        W3=ABS(M2-M1)
        SW=W2+W3
        IF(SW.NE.0.0)         GØ TØ 52
        W2=0.5
        W3=0.5
        SW=1.0
 52     T3=(W2*M2+W3*M3)/SW
        IF(I-1) 80,80,60

 55     CØS2=CØS3
        SIN2=SIN3
 56     W2=ABS(A3*B4-A4*B3)
        W3=ABS(A1*B2-A2*B1)
        IF(W2+W3.NE.0.0)  GØ TØ 57
        W2=SQRT(A3*A3+B3*B3)
        W3=SQRT(A2*A2+B2*B2)
 57     CØS3=W2*A2+W3*A3
        SIN3=W2*B2+W3*B3
        R=CØS3*CØS3+SIN3*SIN3
        IF(R.EQ.0.0)          GØ TØ 58
        R=SQRT(R)
        CØS3=CØS3/R
        SIN3=SIN3/R
 58     IF(I-1) 80,80,65

C DETERMINATIØN ØF THE CØEFFICIENTS

 60     Q2=(2.0*(M2-T2)+M2-T3)/A2
        Q3=(-M2-M2+T2+T3)/(A2*A2)
        GØ TØ 70

 65     R=SQRT(A2*A2+B2*B2)
        P1=R*CØS2
        P2=3.0*A2-R*(CØS2+CØS2+CØS3)
        P3=A2-P1-P2
        Q1=R*SIN2
        Q2=3.0*B2-R*(SIN2+SIN2+SIN3)
        Q3=B2-Q1-Q2
        GØ TØ 75

C CØMPUTATIØN ØF THE PØLYNØMIALS

 70     DZ=A2*RM
        Z=0.0
        DØ 71  J=1,MM1
          K=K+1
          Z=Z+DZ
          U(K)=P0+Z
 71       V(K)=Q0+Z*(Q1+Z*(Q2+Z*Q3))
        GØ TØ 79

 75     Z=0.0
        DØ 76  J=1,MM1
          K=K+1
          Z=Z+RM
          U(K)=P0+Z*(P1+Z*(P2+Z*P3))
 76       V(K)=Q0+Z*(Q1+Z*(Q2+Z*Q3))

 79     K=K+1
 80     CØNTINUE
      RETURN

C ERRØR EXIT

 90 WRITE (IU,2090)
    GØ TØ 99
 91 WRITE (IU,2091)
    GØ TØ 99
 92 WRITE (IU,2092)
    GØ TØ 99
 93 WRITE (IU,2093)
    GØ TØ 99
 95 WRITE (IU,2095)
    GØ TØ 98
 96 WRITE (IU,2096)
    GØ TØ 98
 97 WRITE (IU,2097)
 98 WRITE (IU,2098)  I,X(I),Y(I)
 99 WRITE (IU,2099)  MDO,LO,MO,NO
    RETURN

C FØRMAT STATEMENTS

2090 FØRMAT(1X/31H  ***   MD ØUT ØF PRØPER RANGE./)
2091 FØRMAT(1X/22H  ***   L = 1 ØR LESS./)
2092 FØRMAT(1X/22H  ***   M = 1 ØR LESS./)
2093 FØRMAT(1X/25H  ***   IMPRØPER N VALUE./)
2095 FØRMAT(1X/27H  ***   IDENTICAL X VALUES./)
2096 FØRMAT(1X/33H  ***   X VALUES ØUT ØF SEQUENCE./)
2097 FØRMAT(1X/33H  ***   IDENTICAL X AND Y VALUES./)
2098 FØRMAT(7H   I   =,I4,10X,6HX(I) =,E12.3,
   1                      10X,6HY(I) =,E12.3)
2099 FØRMAT(7H   MD =,I4,8X,3HL =,I5,8X,
   1        3HM =,I5,8X,3HN =,I5/
   2        36H ERRØR DETECTED IN RØUTINE    CRVFIT)
    END
```

ACM Transactions on Mathematical Software, Vol. 2, No. 2, June 1976, Page 208

REMARK ON ALGORITHM 433

Interpolation and Smooth Curve Fitting Based on Local Procedures [E2]
[H. Akima, *Comm. ACM 15*, 10 (Oct. 1972), 914–918]

Michael R. Anderson [Recd 8 Dec. 1975]
Gettysburg College, Gettysburg, PA 17325

Subroutine *CRVFIT* is not written in ANSI Standard Fortran as referenced in [2]. In particular, [2, 7.1.2.8] states that the initial value of a DO statement must be less than or equal to the value represented by the terminal parameter. DO statements numbered 12 and 14 violate this rule when *L* is input as 2, which the limitations of the program allow. Error conditions of IDENTICAL X VALUES or X VALUES OUT OF SEQUENCE may improperly result from the IF tests within these two DO statement loops.

The subroutine may be corrected as follows. Delete the statement numbered 12 and replace it with the following two statements:

```
12 IF(L0.EQ.2)   GO TO 18
   DO 13 I=3,L0
```

Delete the statement numbered 14 and replace it with the following two statements:

```
14  IF (L0.EQ.2)   GO TO 18
    DO 15 I=3,L0
```

The subroutine, if tested for the case $L = 2$, would have performed correctly because of the implementation of DO statements in Fortran for the CDC-3800, which would not have executed the range if $L < 3$. However, for the IBM System/360 Fortran compilers, the subroutine produces the erroneous messages mentioned.

With the preceding corrections, the subroutine has been used with much success on a wide variety of problems.

Algorithm 434

Exact Probabilities for R × C Contingency Tables [G2]

David L. March [Recd. 24 Nov. 1970 and 7 Mar. 1971]
School of Education, Lehigh University,
Bethlehem, PA. 18015

Key Words and Phrases: probability, contingency table, test of significance

CR Categories: 3.5, 5.5

Language: Fortran

Description

Freeman and Halton [1] derive a general method for computing exact probabilities for contingency tables that result if a sample is subjected to k different and independent classifications. The following algorithm is limited to the case where $k = 2$.

If a sample of size N is subjected to two different and independent classifications, A and B, with R and C classes respectively, the probability P_x of obtaining the observed array of cell frequencies $X(x_{ij})$, under the conditions imposed by the arrays of marginal totals $A(r_i)$ and $B(c_j)$ is given by

$$P_x = \frac{\prod_{i=1}^{R} (r_i!) \prod_{j=1}^{C} (c_j!)}{N! \prod_{i=1}^{R} \prod_{j=1}^{C} (x_{ij}!)} \tag{1}$$

Expression (1) is exact and holds if (a) the parent population is infinite or the sampling is done with replacement of the sampled items, (b) the sampling is random, (c) the population is homogeneous, and (d) the marginal totals are considered fixed in repeated sampling.

To test the null hypothesis that A and B are independent against the indefinite two-sided alternative, the probability P_s of obtaining an array as probable as, or less probable than, the observed array is needed. P_s is found as follows: (a) the probability P_t of the observed array is computed; (b) the probabilities for all other possible arrays of cell frequencies, subject to the conditions imposed by the fixed marginal totals, are computed; and (c) P_s is then obtained by summing all of the probability values found in (b) that are less than, or equal to, the probability P_t.

Method. The method of the subroutine uses the fact that expression (1) can be rewritten as

$$P_x = Q_x/R_x$$

where

$$Q_x = \frac{\prod_{i=1}^{R} (r_i!) \prod_{j=1}^{C} (c_j!)}{N!}$$

which is constant for the given set of marginal totals (r_i) and (c_j) and

$$R_x = \prod_{i=1}^{R} \prod_{j=1}^{C} (x_{ij}!)$$

which varies depending on the array of cell frequencies (x_{ij}). In order to avoid machine overflow and roundoff error, these computations are performed using logarithms.

The observed $R \times C$ contingency table is specified by the $NR \times NC$ matrix which is partitioned as follows:

$$\begin{array}{cccc|c}
x_{11} & \cdots & \cdots & x_{1C} & r_1 \\
\vdots & & & & \vdots \\
x_{R1} & \cdots & \cdots & x_{RC} & r_R \\
\hline
c_1 & \cdots & \cdots & c_C & N
\end{array}$$

After computing the constant term $QXLOG$ and the probability of the given table PT, the subroutine assigns to each of the lower right $(R - 1) \times (C - 1)$ cells the minimum of its corresponding row and column totals which is the maximum possible number for the cell. These cells are then varied in all possible combinations with each cell varied between its maximum number and zero.

Starting with cell (2,2), the variation is accomplished by subtraction of 1. When the subtraction yields a zero or positive result the routine goes to compute the remainder of the cell frequencies. When a negative result is obtained, the cell in question, say cell (i, j), is reset to the minimum of the corresponding row and column totals, 1 is subtracted from cell $(i, j + 1)$ or, if $j + 1$ is greater than C, cell $(i + 1, 2)$, and the count down resumes at cell (2,2). If none of the lower right $(R - 1) \times (C - 1)$ cells yield a zero or positive result, the computations are complete and the subroutine returns to the caller. For example, if the top line (below) is the cell maximum ordered left to right from the (2,2) to the (R, C) cell, the combinations generated will be

$$\begin{array}{llll}
2 & 1 & 1 & \cdots \\
1 & 1 & 1 & \cdots \\
0 & 1 & 1 & \cdots \\
2 & 0 & 1 & \cdots \\
1 & 0 & 1 & \cdots \\
0 & 0 & 1 & \cdots \\
2 & 1 & 0 & \cdots \\
 & \vdots & & \\
0 & 0 & 0 & \cdots
\end{array}$$

The column 1 and row 1 cells are filled by subtraction of the generated cell numbers from the marginal totals. Since the method described above yields illegal as well as legal partitions, it is possible to obtain a negative result for one of these cells. When this occurs, the routine goes back to get a new set of cell frequencies. Otherwise $RXLOG$ is computed. Then, the probability PX is computed and added to the cumulative sum PC. If PX is less than, equal to, or, to avoid missing one due to computational inaccuracy, slightly larger than PT, PX is also added to the significance probability PS.

Since PC is the probability of obtaining some of the tables possible within the constraints of the marginal totals, PC should equal 1.0.. The accuracy of the result can be estimated from the amount of deviation of PC from 1.0..

The floating point logarithms (base 10) of the integer factorials are obtained from function $FACLOG$. For arguments less than or equal to 100, the result is obtained from a table that is computationally filled on the first reference to $FACLOG$. Stirling's approximation is used for arguments greater than 100.

Results. The algorithm was tested on a CDC 6400 (60 bit word) using 2 × 3 (N = 30), 2 × 4 (N = 7), and 3 × 3 (N = 7) contingency tables. Results for the 2 × 3 tables were verified against values separately computed using programs developed by March [2]. In several cases PC deviated from 1.0. by 1.0×10^{-12}. Results for the 2 × 4 and 3 × 3 tests were verified by hand computation.

The author is indebted to the referees for their valuable comments and suggestions.

References

1. Freeman, G.H., and Halton, J.H. Note on an exact treatment of contingency, goodness of fit, and other problems of significance. *Biometrika 38* (1951), 141–149.
2. March, D.L. Accuracy of the chi-square approximation for 2 × 3 contingency tables with small expectations. An unpublished D.Ed. Diss., School of Education, Lehigh U., Bethlehem, Pa., 1970.

Algorithm

```
      SUBRØUTINE CØNP(MATRIX,NR,NC,PT,PS,PC)
C
C INPUT ARGUMENTS.
C
C     MATRIX = SPECIFICATIØN ØF THE CØNTINGENCY TABLE.
C         THIS MATRIX IS PARTITIØNED AS FØLLØWS
C
C              X(11).....X(1C)   R(1)
C                .  .....  .      .
C                .  .....  .      .
C              X(R1).....X(RC)   R(R)
C               C(1)..... C(C)    N
C
C         WHERE X(IJ) ARE THE ØBSERVED CELL FREQUENCIES,
C         R(I) ARE THE RØW TØTALS, C(J) ARE THE CØLUMN
C         TØTALS, AND N IS THE TØTAL SAMPLE SIZE.
C         NØTE THAT THE ØRIGINAL CELL FREQUENCIES ARE
C         DESTRØYED BY THIS SUBRØUTINE.
C
C     NR = THE NUMBER ØF RØWS IN MATRIX (R=NR-1).
C
C     NC = THE NUMBER ØF CØLUMNS IN MATRIX (C=NC-1).
C
C ØUTPUT ARGUMENTS.
C
C     PT = THE PRØBABILITY ØF ØBTAINING THE GIVEN TABLE.
C
C     PS = THE PRØBABILITY ØF ØBTAINING A TABLE AS PRØBABLE
C         AS, ØR LESS PRØBABLE THAN, THE GIVEN TABLE.
C
C     PC = THE PRØBABILITY ØF ØBTAINING SØME ØF THE
C         TABLES PØSSIBLE WITHIN THE CØNSTRAINTS ØF THE
C         MARGINAL TØTALS. (THIS SHØULD BE 1.0.  DEVIATIØNS
C         FRØM 1.0 REFLECT THE ACCURACY ØF THE CØMPUTATIØN.)
C
C EXTERNALS.
C
C     FACLØG(N) = FUNCTIØN TØ RETURN THE FLØATING PØINT
C         VALUE ØF LØG BASE 10 ØF N FACTØRIAL.
C
      DIMENSIØN MATRIX(NR,NC)
      INTEGER R,C,TEMP
C
      R=NR-1
      C=NC-1
C
C CØMPUTE LØG ØF CØNSTANT NUMERATØR
C
      QXLØG=-FACLØG(MATRIX(NR,NC))
      DØ 10 I=1,R
   10   QXLØG=QXLØG+FACLØG(MATRIX(I,NC))
      DØ 20 J=1,C
   20   QXLØG=QXLØG+FACLØG(MATRIX(NR,J))
C
C CØMPUTE PRØBABILITY ØF GIVEN TABLE
C
      RXLØG=0.0
      DØ 50 I=1,R
      DØ 50 J=1,C
   50   RXLØG=RXLØG+FACLØG(MATRIX(I,J))
      PT=10.0**(QXLØG-RXLØG)
C
      PS=0.0
      PC=0.0
C
C FILL LØWER RIGHT (R-1) X (C-1) CELLS WITH
C MINIMUM ØF RØW AND CØLUMN TØTALS
C
      DØ 100 I=2,R
      DØ 100 J=2,C
  100   MATRIX(I,J)=MINO(MATRIX(I,NC),MATRIX(NR,J))
      GØ TØ 300
C
C ØBTAIN A NEW SET ØF FREQUENCIES IN
C LØWER RIGHT (R-1) X (C-1) CELLS
C
  200 DØ 220 I=2,R
      DØ 220 J=2,C
        MATRIX(I,J)=MATRIX(I,J)-1
        IF(MATRIX(I,J).GE.0) GØ TØ 300
  220   MATRIX(I,J)=MINO(MATRIX(I,NC),MATRIX(NR,J))
      RETURN
C
C FILL REMAINDER ØF ØBSERVED CELLS
C .....CØMPLETE CØLUMN 1
C
  300 DØ 320 I=2,R
        TEMP=MATRIX(I,NC)
        DØ 310 J=2,C
  310     TEMP=TEMP-MATRIX(I,J)
        IF(TEMP.LT.0) GØ TØ 200
  320   MATRIX(I,1)=TEMP
C
C .....CØMPLETE RØW 1
C
      DØ 340 J=1,C
        TEMP=MATRIX(NR,J)
        DØ 330 I=2,R
  330     TEMP=TEMP-MATRIX(I,J)
        IF(TEMP.LT.0) GØ TØ 200
  340   MATRIX(1,J)=TEMP
C
C CØMPUTE LØG ØF THE DENØMINATØR
C
      RXLØG=0.0
      DØ 350 I=1,R
      DØ 350 J=1,C
  350   RXLØG=RXLØG+FACLØG(MATRIX(I,J))
C
C CØMPUTE PX.  ADD TØ PS IF PX .LE. PT
C (ALLØW FØR RØUND-ØFF ERRØR)
C
      PX=10.0**(QXLØG-RXLØG)
      PC=PC+PX
      IF((PT/PX).GT.0.99999) PS=PS+PX
      GØ TØ 200
      END
      FUNCTIØN FACLØG(N)
C
C INPUT ARGUMENT.
C
C     N = AN INTEGER GREATER THAN ØR EQUAL TØ ZERØ.
C
C FUNCTIØN RESULT.
C
C     FACLØG = THE LØG TØ THE BASE 10 ØF N FACTØRIAL.
C
      DIMENSIØN TABLE(101)
      DATA TPILØG/0.39908 99342/
      DATA ELØG  /0.43429 44819/
      DATA IFLAG/0/
C
C USE STIRLINGS APPRØXIMATIØN IF N GT 100
C
      IF(N.GT.100) GØ TØ 50
C
C LØØK UP ANSWER IF TABLE WAS GENERATED
C
      IF(IFLAG.EQ.0) GØ TØ 100
   10 FACLØG=TABLE(N+1)
      RETURN
C
C HERE FØR STIRLINGS APPRØXIMATIØN
C
   50 X=FLØAT(N)
      FACLØG=(X+0.5)*ALØG10(X) - X*ELØG + TPILØG
     1    + ELØG/(12.0*X) - ELØG/(360.0*X*X*X)
      RETURN
C
C HERE TØ GENERATE LØG FACTØRIAL TABLE
C
  100 TABLE(1)=0.0
      DØ 120 I=2,101
        X=FLØAT(I-1)
  120   TABLE(I)=TABLE(I-1)+ALØG10(X)
      IFLAG=1
      GØ TØ 10
      END
```

Remark on Algorithm 434 [G2]

Exact Probabilities for R × C Contingency Tables [D.L. March, *Comm. ACM 15* (Nov. 1972), 991]

D.M. Boulton [Recd. 5 Mar. 1973 and 30 July 1973]
Department of Information Science, Monash University, Melbourne, Australia

Algorithm 434 calculates the exact probability of a two-dimensional contingency table by generating all possible cell frequency combinations which satisfy the marginal sum constraints, and summing the probabilities of all combinations as likely or less likely than the observed combination. The method used to generate all the cell frequency combinations is rather inefficient as it operates by generating all combinations which satisfy a weakened set of constraints and then rejecting those combinations which violate the actual marginal sum constraints. As the number of combinations rejected very often far exceeds the actual number accepted, the process is very wasteful.

A more efficient combination generating algorithm is described in Boulton and Wallace [1]. It generates explicitly only those combinations which satisfy the marginal sum constraints. In addition, because the combinations are generated by a set of nested *DO* loops each with a different cell frequency as its controlled variable, the order of generation is such that one combination usually only differs from the next in the values of a few cell frequencies in the lower right corner of the table. This ordering can be used to reduce the time taken to obtain the logarithm of the probability of each combination. Instead of always summing over all cells, an array of partial sums of logarithms of cell frequencies is maintained, and for each new combination only that part of the logarithm which has changed is evaluated and then added to the relevant partial sum.

March's algorithm has been modified to use the combination generating algorithm of Boulton and Wallace and to take advantage of the order in which the combinations are generated. A series of comparison tests were run on a CDC 3200, and the results of a few are shown in Table I. The modified algorithm was always faster, and as can be seen in Table I, the speed improvement can be quite large.

Table I. Times for Evaluating Probabilities

Contingency table	Probability	Time (sec)	
		Original	Improved
8 12, (20) 8, 2, (10) (16) (14) (30)	.05767116	.026	.013
5, 3, 3, 0 (11) 2, 3, 1, 2 (8) (7) (6) (4) (2) (19)	.35262364	.290	.095
5, 1, 0, 0 (6) 1, 1, 2, 1 (5) 0, 1, 1, 1 (3) (6) (3) (3) (2) (14)	.10625089	3.31	.510
2, 0, 0, 0 (2) 0, 1, 0, 1 (2) 0, 0, 2, 0 (2) 0, 1, 0, 1 (2) (2) (2) (2) (2) (8)	.12380952	13.9	.693

Finally, it is worth noting that the combination generating algorithm of Boulton and Wallace can be systematically extended for contingency tables of more than two dimensions. It can thus be used as the basis of a subroutine for calculating exact probabilities in more than two dimensions.

References

1. Boulton, D.M., and Wallace, C.S. Occupancy of a rectangular array. *Comp. J. 16*, 1 (1973), 57–63.

Remark on Algorithm 434 [G2]

Exact Probabilities for *R* × *C* Contingency Tables [D.L. March, *Comm. ACM 15* (Nov. 1972), 991]

T.W. Hancock [Recd 16 Nov. 1973, 11 Feb 1974] Waite Agricultural Research Institute, The University of Adelaide, Glen Osmond, South Australia 5064.

The above algorithm was presented for computing exact probabilities of *R* × *C* contingency tables by the method described by Freeman and Halton [1]. Clearly inefficient for small matrices, this algorithm becomes impracticable for 4 × 4 matrices or larger. For this reason the subroutine presented below is suggested. Every effort has been made to minimize the number of coding changes so that (a) the original work of March can be recognized; and (b) the important differences are apparent to anyone wishing to compare the two approaches. Row and column dimensions have been added to the formal parameters, so that the elements of the contingency table do not have to be stored in a contiguous manner. (Both are included to ensure compatibility with any type of compiler.) Function *FACLOG*(*N*) is exactly as presented by March.

Acknowledgment. I thank Dr. O. Mayo, Waite Agricultural Research Institute, University of Adelaide, for suggesting that I investigate March's algorithm.

Differences in Method Comment cards have been included in the listing to locate and describe the differences discussed below. These can be identified by an asterisk in column three. Also where appropriate this is followed by a number which relates to the order in the list below.

1. All cell frequencies are set to zero initially.
2. The jump indicator *KEY* is equivalenced to 1, and cell (2,2) (*MATRIX*(2,2) in the subroutine) is set to −1.
3. The generation process is accomplished by addition of 1 to the appropriate (*I*,*J*) cell frequency (where *I* and *J* proceed from 2, . . ., *R* and 2, . . ., *C* respectively).
4. The valueof row marginal *I* is checked against $\sum_{K=J}^{C} MATRIX$ (*I*,*K*). Similarly column marginal *J* is checked. If either marginal is less than the appropriate sum, control returns via 8 below to 3 above.
5. If indicator *KEY* equals 2 the cell frequencies preceding cell (*I*,*J*) are set to zero and the addition sequence recommences from cell (2,2) (i.e. 2 above).
6. However, if *KEY* equals 1, subroutine *INIT* is called to generate the "next" matrix of cell frequencies satisfying the marginals. *INIT* first adjusts the marginals for the cell values in *MATRIX*. Then beginning at the lower left hand corner matrix (i.e. cell (*R*,1)), each cell in turn is increased to its maximum value and its marginals reduced. Once the row marginal is reduced to zero the sequence jumps to the first cell in the row above. Using this process it is possible to progress from one valid set of frequencies to the next, thus saving considerable time.
7. After the probability calculations have been computed, for the matrix returned from *INIT*, a sequence of matrices is generated if the frequencies of cells (1,2) and (2,1) are both greater than zero. (As explained by Freeman and Halton the probabilities of the members of this sequence of matrices are related and recognition of this simplifies their calculation.)
8. *KEY* is equivalent to 2, and control returns to 3 above via the loop terminator causing cell (*I*,*J*)′ to be increased by 1.

Results The two methods were compared on a Control Data Corporation CYBER 73 using contingency tables over a range of sizes and cell frequencies. Table I summarizes the CP times. Clearly the original method becomes unquestionably slow; in fact for a 4 × 4 matrix, with all frequencies one, this method would attempt $5^9 = 1,953,125$ matrices before it reached a result. For the same matrix the revised method calculates probabilities for 10147 matrices, all of which are compatible with the marginals. Obviously this improved method would be impracticable for contingency

Table I. Comparison of Subroutines
(CP time required in seconds to compute exact probabilities for RXC contingency tables; where all cell frequencies are chosen equal to one. These are presented to illustrate the relative improvement of *RXCPRB* over *CONP*. Obviously the actual times will depend on the machine used.)

R × *C*	*CONP* (by March)	*RXCPRB*
2 × 2	.019 (3†)	.018 (3†)
2 × 3	.012 (9)	.010 (7)
3 × 2	.018 (9)	.016 (7)
2 × 5	.073 (8)	.054 (51)
5 × 2	.093 (81)	.055 (51)
3 × 3	.110 (256)	.055 (55)
3 × 4	1.279 (4096)	.509 (415)
4 × 3	1.344 (4096)	.514 (415)
4 × 4	Unknown*	15.495 (10147)

† Number of matrices attempted in the calculation
* Computation was still incomplete after 500 seconds

tables with more degrees of freedom and/or larger total sample size, but grouping of classes and alternative statistical tests are available in this area (see Goodman [2] or Sugiura and Ôtake [3]). Further it is generally trivial to continue once the tail probability becomes large, so that insertion of a statement of the form,

IF(*PS*. *GT*. 0.1. *AND*. *PC*. *LT*. 0.9)*RETURN*

in subroutine *RXCPROB* prior to statement numbered 32 would increase efficiency.

In all cases, *RXCPROB* and *CONP* produced correct probabilities. (For smaller matrices, the computed probabilities were checked by hand; for the larger ones, agreement between the methods was taken to indicate the correctness of *RXCPROB*, since March had already tested his subroutine.) The maximum deviation of *PC* from 1.0 was 1.0 × 10^{-10}. Although slightly larger than reported by March this is a direct result of the increased complexity of the tables solved, and in fact *CONP* gave a similar deviation.

References

1. Freeman, G.H., and Halton, J.H. Note on an exact treatment of contingency, goodness of fit, and other problems of significance. *Biometrika* 38 (1951), 141–149.
2. Goodman, L.A. On methods for comparing contingency tables. *Journal of Royal Statistical Society* Series A 126 (1963), 94–105.
3. Sugiura, N., and Ôtake, M. Numerical comparison of Improved methods of testing in contingency tables with small frequencies. *Annals of the Institute of Statistical Mathematics* 20 (1968), 505–517.

Algorithm

```
      SUBROUTINE RXCPRB(MATRIX, NRD, NCD, NR, NC,
     * PT, PS, PC)
C *        THIS SUBROUTINE COMPUTES EXACT
C *        PROBABILITIES FOR R X C CONTINGENCY TABLES
C *INPUT VIA FORMAL PARAMETERS
C *   NRD = THE ROW DIMENSION
C *   NCD = THE COLUMN DIMENSION
C      NR = THE NUMBER OF ROWS IN MATRIX (R=NR-1).
C      NC = THE NUMBER OF COLUMNS IN MATRIX (C=NC-1).
C      MATRIX = SPECIFICATION OF THE CONTINGENCY
C          TABLE. THIS MATRIX IS PARTITIONED AS
C          FOLLOWS
C *       X(1,1),X(1,2),.............,X(1,C)    X(1,NC)
C *          .      .  ,............,    .          .
C *          .      .  ,............,    .          .
C *       X(R,1),X(R,2),.............,X(R,C)    X(R,NC)
C *       X(NR,1),X(NR,2),..........,X(NR,C)   X(NR,NC)
C *        WHERE X(I,J) ARE THE OBSERVED CELL
C *        FREQUENCIES, X(I,NC) ARE THE ROW TOTALS,
C *        X(NR,J) ARE THE COLUMN TOTALS, AND X(NR,NC)
C *        IS THE TOTAL SAMPLE SIZE.
C          NOTE THAT THE ORIGINAL CELL FREQUENCIES ARE
C          DESTROYED BY THIS SUBROUTINE.
C OUTPUT ARGUMENTS.
C      PT = THE PROBABILITY OF OBTAINING THE GIVEN
C      TABLE.
C      PS = THE PROBABILITY OF OBTAINING A TABLE AS
C           PROBABLE AS, OR LESS PROBABLE THAN, THE
C           GIVEN TABLE.
C *    PC = THE PROBABILITY OF OBTAINING ALL OF THE
C           TABLES POSSIBLE WITHIN THE CONSTRAINTS OF
C           THE MARGINAL TOTALS. (THIS SHOULD BE 1.Ø.
C           DEVIATIONS FROM 1.Ø REFLECT THE ACCURACY OF
C           THE COMPUTATION.)
C EXTERNALS.
C *    INIT(MATRIX,NRD,NCD,NR,NC) = SUBROUTINE WHICH
C *             RETURNS THE *NEXT* MATRIX TO SATISFY
C *             THE MARGINALS.
C      FACLOG(N) = FUNCTION TO RETURN THE FLOATING
C           POINT VALUE OF LOG BASE 1Ø OF N FACTORIAL.
      DIMENSION MATRIX(NRD,NCD)
      INTEGER R, C
      R = NR - 1
      C = NC - 1
C COMPUTE LOG OF CONSTANT NUMERATOR
      QXLOG = -FACLOG(MATRIX(NR,NC))
      DO 1Ø I=1,R
        QXLOG = QXLOG + FACLOG(MATRIX(I,NC))
   1Ø CONTINUE
      DO 2Ø J=1,C
        QXLOG = QXLOG + FACLOG(MATRIX(NR,J))
   2Ø CONTINUE
C COMPUTE PROBABILITY OF GIVEN TABLE
      RXLOG = Ø.Ø
      DO 4Ø I=1,R
        DO 3Ø J=1,C
          RXLOG = RXLOG + FACLOG(MATRIX(I,J))
   3Ø   CONTINUE
   4Ø CONTINUE
      PT = 1Ø.Ø**(QXLOG-RXLOG)
      PS = Ø.Ø
      PC = Ø.Ø
C * 1. ALL CELL VALUES INITIALLY SET TO ZERO
      DO 6Ø I=1,R
        DO 5Ø J=1,C
          MATRIX(I,J) = Ø
   5Ø   CONTINUE
   6Ø CONTINUE
C * 2. EACH CYCLE STARTS HERE
   7Ø KEY = 1
      MATRIX(2,2) = -1
C * 3. GENERATING SET OF FREQUENCIES PROGRESSIVELY IN
C *    LOWER RIGHT (R-1) * (C-1) CELLS.
      DO 16Ø I=2,R
        DO 15Ø J=2,C
          MATRIX(I,J) = MATRIX(I,J) + 1
C * 4. CHECKING SUMMATIONS .LE. RESPECTIVE MARGINALS
C *     I.E. (SUM OF ELTS. J TO C IN ROW I) .LE.
C *     MATRIX(I,NC) AND  (SUM OF ELTS. I TO R IN COL.
C *     J).LE. MATRIX(NR,J)
          ISUM = Ø
          JSUM = Ø
          DO 8Ø M=J,C
            ISUM = ISUM + MATRIX(I,M)
   8Ø     CONTINUE
          IF (ISUM.GT.MATRIX(I,NC)) GO TO 13Ø
          DO 9Ø K=I,R
            JSUM = JSUM + MATRIX(K,J)
   9Ø     CONTINUE
          IF (JSUM.GT.MATRIX(NR,J)) GO TO 13Ø
C * 5. JUMP TO STATEMENT 17Ø WHERE ALL CELLS PRIOR TO
C *   MATRIX(I,J) ARE SET TO ZERO.
          IF (KEY.EQ.2) GO TO 17Ø
          IP = I
          JP = J
C * 6. CALL SUBROUTINE INIT TO FIND THE NEXT BALANCED
C *   MATRIX
          CALL INIT(MATRIX, NRD, NCD, NR, NC)
C COMPUTE LOG OF THE DENOMINATOR
          RXLOG = Ø.Ø
          DO 11Ø K=1,R
            DO 1ØØ M=1,C
              RXLOG = RXLOG + FACLOG(MATRIX(K,M))
  1ØØ       CONTINUE
  11Ø     CONTINUE
C * COMPUTE PX. ADD TO PC AND ALSO PS IF PX .LE. PT
C    (ALLOW FOR ROUND-OFF ERROR)
          PX = 1Ø.Ø**(QXLOG-RXLOG)
          PC = PC + PX
          IF ((PT/PX).GT.Ø.99999) PS = PS + PX
C * 7. IF POSSIBLE A SEQUENCE OF MATRICES AND
C *     ASSOCIATED PROBABILITIES (PX,PC AND PS) ARE
C *     GENERATED BY MANIPULATING CELLS (1,1), (1,2),
C *     (2,1) AND (2,2)  (SIMILARLY ALLOWING
C *     FOR ROUND-OFF ERROR)
  12Ø     IF (MATRIX(1,2).LT.1 .OR.
     *     MATRIX(2,1).LT.1) GO TO 14Ø
          MATRIX(1,1) = MATRIX(1,1) + 1
          MATRIX(2,2) = MATRIX(2,2) + 1
          PX = PX*FLOAT(MATRIX(1,2))*FLOAT(MATRIX(2,1))
     *         /FLOAT(MATRIX(1,1))/FLOAT(MATRIX(2,2))
          PC = PC + PX
          IF ((PT/PX).GT.Ø.99999) PS = PS + PX
          MATRIX(1,2) = MATRIX(1,2) - 1
          MATRIX(2,1) = MATRIX(2,1) - 1
          GO TO 12Ø
```

```
 130        IP = I
            JP = J
C * 8. KEY SET TO 2 AS CYCLE COMPLETED
 140        KEY = 2
 150      CONTINUE
 160 CONTINUE
     PETURN
C * ALL CELLS OF MATRIX PRIOR TO THE (I,J)TH. ARE
C * SET TO ZERO.
 170 DO 180 M=2,JP
       MATRIX(IP,M) = 0
 180 CONTINUE
     IP = IP - 1
     DO 200 K=1,IP
       DO 190 M=2,C
         MATRIX(K,M) = 0
 190   CONTINUE
 200 CONTINUE
     GO TO 70
     END
```

```
     SUBROUTINE INIT(MATRIX, NRD, NCD, NR, NC)
C * THIS SUBROUTINE RETURNS THE *NEXT* MATRIX TO
C *  SATISFY (1) THE MARGINALS AND (2) THE SEQUENCE
C *  OF GENERATION DEFINED IN SUBROUTINE RXCPRB.
     DIMENSION MATRIX(NRD,NCD), MROW(50), MCOL(50)
     INTEGER R, C
     R = NR - 1
     C = NC - 1
C * EQUIVALENCE MROW AND MCOL TO ROW AND COLUMN
C * MARGINALS RESPECITVELY.
     DO 10 K=1,R
       MATRIX(K,1) = 0
       MROW(K) = MATRIX(K,NC)
  10 CONTINUE
     DO 20 M=1,C
       MCOL(M) = MATRIX(NR,M)
  20 CONTINUE
C * FOR EACH ROW, SUBTRACT ELEMENTS 2 TO C FROM MROW
     DO 40 K=2,R
       DO 30 M=2,C
         MROW(K) = MROW(K) - MATRIX(K,M)
  30   CONTINUE
  40 CONTINUE
C * FOR EACH COLUMN, SUBTRACT ELEMENTS 2 TO R FROM
C * MCOL
     DO 60 M=2,C
       DO 50 K=2,R
         MCOL(M) = MCOL(M) - MATRIX(K,M)
  50   CONTINUE
  60 CONTINUE
C * FORMING *NEXT BALANCED* ARRAY
     DO 90 I=1,R
       IR = NR - I
       DO 80 J=1,C
         MIN = MIN0(MROW(IR),MCOL(J))
         IF (MIN.EQ.0) GO TO 70
         MATRIX(IR,J) = MATRIX(IR,J) + MIN
         MROW(IR) = MROW(IR) - MIN
         MCOL(J) = MCOL(J) - MIN
  70     IF (MROW(IR).EQ.0) GO TO 90
  80   CONTINUE
  90 CONTINUE
     RETURN
     END
```

```
     FUNCTION FACLOG(N)
C INPUT ARGUMENT.
C      N = AN INTEGER GREATER THAN OR EQUAL TO ZERO.
C FUNCTION RESULT.
C      FACLOG = THE LOG TO THE BASE 10 OF N FACTORIAL.
     DIMENSION TABLE(101)
     DATA TPILOG /0.3990899342/
     DATA ELOG /0.4342944819/
     DATA IFLAG /0/
C USE STIRLINGS APPROXIMATION IF N GT 100
     IF (N.GT.100) GO TO 20
C LOOK UP ANSWER IF TABLE WAS GENERATED
     IF (IFLAG.EQ.0) GO TO 30
  10 FACLOG = TABLE(N+1)
     RETURN
C HERE FOR STIRLINGS APPROXIMATION
  20 X = FLOAT(N)
     FACLOG = (X+0.5)*ALOG10(X) - X*ELOG + TPILOG +
    * ELOG/(12.0*X) - ELOG/(360.0*X*X*X)
     RETURN
C HERE TO GENERATE LOG FACTORIAL TABLE
  30 TABLE(1) = 0.0
     DO 40 I=2,101
       X = FLOAT(I-1)
       TABLE(I) = TABLE(I-1) + ALOG10(X)
  40 CONTINUE
     IFLAG = 1
     GO TO 10
     END
```

ACM Transactions on Mathematical Software, Vol. 2, No. 1, March 1976, Page 108

REMARK ON ALGORITHM 434

Exact Probabilities for R × C Contingency Tables [G2]
[D.L. March, *Comm. ACM* 15, 11 (Nov. 1972), 991]

D.M. Boulton [Recd 25 June 1975]
Department of Computer Science, Monash University, Clayton, 3168, Victoria, Australia

Two previous Remarks, by Boulton (1974) [1] and by Hancock (1975) [2], have shown that Algorithm 434 by March (1972) is rather inefficient, especially for contingency tables with many degrees of freedom. The inefficiency lies in the method

Table I. Times in Seconds for the Contingency Tables in Boulton [1]

$R \times C$	Hancock	Boulton
2 × 2	0.024	0.018
2 × 4	0.16	0.10
3 × 4	1.37	0.68
4 × 4	2.21	1.05

Table II. Times in Seconds for the Contingency Tables in Hancock[2]

$R \times C$	Hancock	Boulton
2 × 2	0.008	0.007
2 × 3	0.023	0.016
3 × 2	0.023	0.016
2 × 5	0.21	0.11
5 × 2	0.21	0.10
3 × 3	0.22	0.12
3 × 4	2.08	0.98
4 × 3	2.08	0.99
4 × 4	63.5	25.5

of generating all those cell frequency combinations that satisfy the marginal sum constraints.

The purpose of this remark is to compare directly the speeds of the above two more recent algorithms (in the Remarks). The comparisons were carried out on a Hewlett-Packard HP2100A computer with fully extended arithmetic and microprogrammed array referencing and subroutine entry. In Table I, times are given for the four examples originally used in Boulton. In Table II, times are given for the examples presented in Hancock.

The algorithm by Boulton is always faster, and for all but 2 × 2 tables the improvement is quite significant, being more than a factor of 2 for contingency tables with several degrees of freedom.

The same set of tests were run again on the HP2100A with standard firmware, i.e. without microprogrammed array referencing and subroutine entry. The times were then found to be even more in favor of Boulton's algorithm. The speed ratio increased to 3 for Hancock's 4 × 4 table.

REFERENCES

[1] Boulton, D.M. Remark on Algorithm 434. *Comm. ACM 17*, 6(June 1974), 326.
[2] Hancock, T.W. Remark on Algorithm 434. *Comm. ACM 18*, 2(Feb. 1975), 117–119.

Algorithm 435

Modified Incomplete Gamma Function [S 14]

Wayne Fullerton [Recd. 30 Dec. 1970 and 12 April 1971]
Department of Astronomy, University of Michigan, Ann Arbor, MI 48104

Key Words and Phrases: modified incomplete Gamma function, incomplete Gamma function, chi-square distribution function, Poisson distribution function
CR Categories: 5.13
Language: Fortran

Description

The incomplete Gamma function is defined by

$$\gamma(a, x) = \int_0^x y^{a-1.} \cdot e^{-y}\, dy, \qquad x \geq 0. \tag{1}$$

If x is allowed to assume negative values and if the absolute value of y is substituted for y in the term $y^{a-1.}$, then a modified incomplete Gamma function may be defined by

$$\gamma'(a, x) = \int_0^x |y|^{a-1.} \cdot e^{-y}\, dy, \qquad -\infty < x \leq \infty. \tag{2}$$

Note that if x is less than zero, the above is equivalent to

$$\gamma'(a, x) = -\int_0^{|x|} y^{a-1.} \cdot e^{+y}\, dy, \qquad x \leq 0. \tag{3}$$

The function subprogram *GAMINC* given below computes the more general function

$$\begin{aligned} GAMINC\,(a, x_1, x_2) &\cong e^{x_1} \int_{x_1}^{x_2} |y|^{a-1.} \cdot e^{-y}\, dy \\ &= e^{x_1}[\gamma'(a, x_2) - \gamma'(a, x_1)]. \end{aligned} \tag{4}$$

For x_1 equal to zero, *GAMINC* is just a modified incomplete Gamma function. And if x_2 is also greater than or equal to zero, then *GAMINC* is simply an incomplete Gamma function.

The need for the function *GAMINC* arises in the calculation of

$$I \equiv \int_{Z_1}^{Z_2} e^{a+bZ} \exp\left\{-\int_0^Z e^{\alpha+\beta Z'} \frac{dZ'}{-sin(\vartheta)}\right\} \frac{dZ}{-sin(\vartheta)}, \tag{5}$$

where ϑ is an angle between $-\pi$ and $+\pi$ not equal to zero. The two constants b and β are of the same sign. The integral in the exponent can be done explicitly to yield

$$I = \frac{e^{a-\tau_0+X_1}}{-sin\,(\vartheta)} \int_{Z_1}^{Z_2} e^{a+bZ} \exp\left\{\frac{-e^{\alpha+\beta Z}}{-sin\,(\vartheta)}\right\} dZ, \tag{6}$$

where

$$X_i = \frac{e^{a+bZ_i}}{-b\ sin\,(\vartheta)}$$

and

$$\tau_0 = \int_0^{Z_1} e^{\alpha+\beta Z'} \frac{dZ'}{-sin\,(\vartheta)}$$

A change of variables finally reduces the above integral to

$$I = e^{\alpha-\tau_0} |b\ sin\,(\vartheta)|^{\beta/b-1.} \cdot e^{-a\beta/b} \left[e^{x_1} \int_{x_1}^{x_2} |y|^{\beta/b-1.} \cdot e^{-y}\, dy\right] \tag{7}$$

The quantity in brackets is $GAMINC(\beta/b, X_1, X_2)$.

The approximations of $\gamma'(a, x)$ used in *GAMINC* are valid only for $1. \lesssim a \lesssim 2.$ (See Table I.) The user may compute *GAMINC* for other values of a with the aid of the recurrence relation (m is a positive integer such that $1. \lesssim a \lesssim 2$).

$$\begin{aligned} &GAMINC(m + a, x_1, x_2) \\ &= (m + a - 1)\ GAMINC(m + a - 1, x_1, x_2) + [|x_1|^{m+a-1} \\ &\qquad - |x_2|^{m+a-1} e^{x_1-x_2}] \end{aligned} \tag{8}$$

In general for $x_1 \geq 0$ and $x_2 \geq 0$,

$$\begin{aligned} &GAMINC(m + a, x_1, x_2) \\ &= (m + a - 1) \cdot (m + a - 2) \cdots (a) \cdot GAMINC(a, x_1, x_2) \\ &\quad + |x_1|^a \Big[|x_1|^{m-1.} + \sum_{i=1}^{m-1} (m + a - 1) \\ &\qquad\qquad \cdots (m + a - i)\, |x_1|^{m-1-i}\Big] \\ &\quad - |x_2|^a \Big[|x_2|^{m-1.} + \sum_{i=1}^{m-1} (m + a - 1) \\ &\qquad\qquad \cdots (m + a - i)\, |x_2|^{m-1-i}\Big] e^{x_1-x_2}. \end{aligned} \tag{9}$$

The recurrence relation should be applied in the other direction if $m + a$ is less than 1.

For large values of a ($a \gtrsim 15.$) in the incomplete Gamma function, the user is referred to the algorithm by Takenaga [5].

In all cases we use approximations which are functions of both a and x, so that it is not necessary to compute and store an economized polynomial for each value of a. The overhead in execution time for doing this is not significant since many-term expressions would result anyway. Also exponentiation and real numbers raised to a real power require 30 percent of the total computing time. Multiplying $\gamma'(a, x_2) - \gamma'(a, x_1)$ by e^{x_1} saves two exponentiations and greatly extends the range over which the difference can be represented without over- or underflows occurring. Four separate approximations are used to compute $\gamma'(a, x)$.

Region 1. For $x \geq 5.0$, the complimentary incomplete Gamma function is computed by using a continued fraction approximation [1]

$$\Gamma(a) - \gamma'(a, x) = \frac{e^{-x} x^a}{x + T_1}, \tag{10}$$

The facilities of the University of Michigan Computing Center were used for this work and the research was supported by the National Science Foundation through a Traineeship granted to the author. Author's present address: Los Alamos Scientific Laboratory, Los Alamos, NM 87544.

where

$$T_i = \frac{i+a}{1 + i/(x + T_{i+1})},$$

and where $\Gamma(a)$ is the complete Gamma function of a. Only terms through T_3 are used explicitly. T_4 is taken into account in an approximate way by setting $T_4 = 1.7$, which is its approximate value when $x \sim 5.0$. If both argument values are greater than 5.0, then significance is maintained by subtracting the complementary functions, not the functions themselves.

Region 2A. For $-12. < x < -1.$ and $1. < x < 5.$, the continued fraction approximation given by Luke [3] is valid. We rewrite the approximation in the form

$$\gamma'(a, x) = \frac{x \cdot |x|^{a-1} \cdot e^{-x}}{a \cdot T_1}, \tag{11}$$

where

$$T_n = 1. - \frac{(a+n-1) \cdot x}{(a+2n-2) \cdot [a + 2n - 1 + (n \cdot x)/((a+2n) \cdot T_{n+1})]}.$$

Only terms through T_7 are used explicitly, and T_8 is computed by using the approximate expression

$$\begin{aligned} T_8 \cong\ & 1.00150 - 8.95 \cdot 10^{-5} \cdot a + x \\ & \cdot (-0.0337062 + .0004182 \cdot a + x \\ & \cdot (.000999294 - .000104103 \cdot a)). \end{aligned} \tag{12}$$

On a computer with 32 bit words, eq. (11) must be evaluated in double precision in order to maintain approximately six significant figures of accuracy. On an IBM 360 double precision evaluation can be forced by including more than seven digitsin a constant as is done in eq. (12). Of course, double precision evaluation is unnecessary if there are somewhat more than 32 bits per word. Because the calculation of the approximation of $\gamma'(a, x)$ is a relatively time consuming operation, a separate approximation is used when $|x| \leq 1.$

Region 2B. For $-1.0 \leq x \leq 1.0$, a change of variables is made so that

$$\gamma'(a, x) = |x|^{a-1} \cdot e^{-x} \int_0^x \left(\frac{y}{x}\right)^{a-1.} e^{-y+x}\, dy, \tag{13}$$

or

$$\gamma'(a, x) = x \cdot |x|^{a-1} \cdot e^{-x} \int_0^1 (1-p)^{a-1} \cdot e^{xp}\, dp. \tag{14}$$

Because $-1.0 \leq xp \leq 1.0$, e^{xp} may be adequately approximated with a polynomial. A Chebyshev approximation of nine terms yields a maximum absolute error less than 10^{-7}, which is adequate to insure that the maximum relative error of the *integral* ordinarily be much less than about 10^{-6}. Since the relative error in the single precision evaluation of $|x|^a e^{-x}$ is usually $\sim 1 \cdot 10^{-6}$ for a machine with a 32 bit word length, the above error bound seems entirely reasonable. Write

$$e^Z \cong \sum_{i=0}^{M} b_i Z^i, \quad -1.0 \leq Z \leq 1.0. \tag{15}$$

Then

$$\gamma'(a, x) \cong x \cdot |x|^{a-1} \cdot e^{-x} \sum_{i=0}^{M} \frac{i!\, b_i\, x^i}{(i+a)(i+a-1) \cdots (a)}. \tag{16}$$

Finally we may define $b_i' = b_i \cdot i!$, and write

$$\gamma'(a, x) \cong x \cdot |x|^{a-1} \cdot e^{-x} \sum_{i=0}^{M} \frac{b_i' x^i}{(i+a) \cdot (i+a-1) \cdots (a)}. \tag{17}$$

Note that if the series was not economized, all the b_i' would be unity. But because a finite Chebyshev economized series is employed, the b_i' are only approximately unity.

Of course, it would be possible to extend the Chebyshev approximation to include the entire range $-12. < x < 5.0$; however the many-term result would have to be evaluated in double precision in order to insure a relative error $< 10^{-6}$. It would also be possible to decrease the range of validity of the ascending continued fraction approximation; however the other approximations would then have to be more complicated and would require an accordingly longer time to evaluate. Such a change was judged inadvisable since the function is used predominantly with arguments whose absolute values are large. Also, the present choice of ranges and approximations provides for the accurate representation of $\gamma'(a, x)$ further beyond $a = 2.$ than would many other choices.

Table I. Relative Errors of *GAMINC*(*A*,0.,*X*) in Units of the Sixth Decimal Place

X	*A*								
	0.5	0.8	1.1	1.4	1.7	2.0	2.3	2.9	3.5
−14.	16.57	3.37	0.27	0.79	1.11	0.81	1.87	1.35	1.03
−12.	1.03	1.61	0.42	0.47	1.23	0.95	2.70	1.69	2.70
− 8.	0.09	1.04	0.34	0.69	1.69	1.07	2.01	2.11	2.24
− 4.	0.38	0.82	0.15	0.79	0.67	0.19	0.51	1.59	0.75
− 2.	1.22	0.78	0.50	0.11	0.07	0.01	0.09	0.11	0.51
− 0.5	0.65	0.43	0.21	0.32	0.23	0.15	0.18	0.65	0.78
+ 0.5	1.03	0.42	0.66	1.19	1.25	0.77	0.29	0.06	0.35
2.	0.53	1.57	0.77	0.26	0.04	0.22	0.05	0.23	0.40
6.	0.44	0.38	0.06	0.01	0.06	0.06	1.21	0.36	2.21
10.	0.63	0.73	0.03	0.02	0.08	0.03	0.60	0.04	0.26

Table II. Execution Times of *GAMINC*(*A*, X_1, X_2) in Milliseconds

	$X_2 \leq -12.$	$-12. < X_2 < 5$		$X_2 \geq 5.$
		$\lvert X_2 \rvert \leq 1.$	$\lvert X_2 \rvert > 1.$	
$X_1 \leq -12.$	1.1	1.4	2.0	0.6*
$-12. < X_1 < 5.$ $\lvert X_1 \rvert \leq 1.$	1.4	1.3	2.0	1.4
$-12. < X_1 < 5.$ $\lvert X_1 \rvert > 1.$	2.0	2.0	2.4	2.0
$X_1 \geq 5.$	0.6*	1.4	2.0	1.1
$X_1 = 0.$	0.8	0.9	1.4	0.8

* Only the modified incomplete Gamma function for $X = X_1$ was calculated, because $|X_2 - X_1|$ was greater than *EXPLIM*.

Region 3. For $x \leq -12.$, the asymptotic expansion

$$\gamma'(a, x) \approx \Gamma(a) - |x|^{a-1} \cdot e^{-x} \left[1 + \frac{a-1}{x} + \frac{(a-1) \cdot (a-2)}{x^2} + \cdots \right] \tag{18}$$

is used. Shank's e_1 process [4] is applied once to the six-term series in order to accelerate convergence.

The function subprogram is invoked by a reference of the form

GAMINC(*A*, *X*1, *X*2, *GAM*),

where *GAM* is the user-supplied value of the complete Gamma function of *A*. $\Gamma(a)$ is now commonly a part of the standard Fortran library of functions. If it is not, one of the several algorithms described in this department may be used, or *GAMMA* given in IBM's Scientific Subroutine Package (cf. Hastings [2]) may be used.

Table I presents the absolute value of the relative errors (multiplied by 10^6) of $\gamma'(a, x)$ for selected values of a and x. Because $|x|^a e^{-x}$ was not calculated in double precision, these errors are the total errors and not the errors in the approximations. The "exact" values were found by directly summing the series

$$\gamma'(a, x) \cong |x|^a \sum_{i=0}^{N} \frac{(-x)^i}{(a+i)i!}$$

in double precision. N was chosen so that the contribution of the Nth term was less than $2 \cdot 10^{-9}$ times the sum of the previous N terms. Single precision approximations were used to represent a and x in order to insure that the series and the subprogram gave $\gamma'(a, x)$ for the same parameter values. The subroutine has been used extensively to compute a three-fold integral which includes numerous cases of eq. (5) as a part of the integral. Independent numerical integration results are in agreement with subroutine results to within three significant figures—the accuracy of the numerical integration. Table II gives the average execution times in milliseconds of the subroutine for various argument combinations. The times are for an IBM 360/67, which, for comparison, exponentiates in approximately 0.1 milliseconds.

Acknowledgments. It is a pleasure to thank Dr. Carl deBoor for commenting on a draft of this paper.

References

1. Abromowitz, M., and Stegun, I.A. *Handbook of Mathematical Functions.* National Bureau of Standards, U.S. Gov. Print. Off., Washington, D.C., 1967, p. 263.
2. Hastings, C. *Approximations for Digital Computers.* Princeton University Press, Princeton, N.J., 1955, p. 155.
3. Luke, Y.L. *The Special Functions and Their Approximations Vol II.* Academic Press, New York and London, 1969, p. 196.
4. Shanks, D. Non-linear transformations of divergent and slowly convergent sequences. *J. Math. Phys. 34* (1955), 1.
5. Takenaga, R. On the evaluation of the incomplete gamma function. *Math. Computation 20* (Oct. 1966), 606.

Algorithm

```
      FUNCTIØN GAMINC (A,X1,X2,GAM)
C
C COMPUTE  THE  DIFFERENCE BETWEEN TWØ  MØDIFIED  INCØMPLETE
C GAMMA  FUNCTIØNS FØR (A,X1)  AND  (A,X2) THEN  MULTIPLY BY
C EXP(X1).  THAT IS, CØMPUTE THE INTEGRAL ØF  ABS(X)**(A-1.)
C *EXP(X1-X) FRØM X1 TØ X2.   IF X1 .GT. X2, THEN X1-X2 MUST
C BE .LE. EXPLIM.
C EXPLIM CAN BE A MACHINE  DEPENDENT CØNSTANT WHICH PREVENTS
C EXPØNENTIATIØN  ØVER- AND  UNDERFLØWS.  IT IS USED HERE TØ
C SUPPRESS THE CALCULATIØN ØF  MIGAM(A,X2) WHEN THE VALUE ØF
C MIGAM(A,X2) IS  INSIGNIFICANT.   THIS USAGE  REQUIRES X2 +
C EXPLIM .GE. X1. (MIGAM IS AN ABBREVIATIØN FØR MØDIFIED IN-
C CØMPLETE GAMMA FUNCTIØN.)
C GAM IS THE CØMPLETE  GAMMA  FUNCTIØN ØF A SUPPLIED  BY THE
C CALLING PRØGRAM.
C
C FØR X .GT. 5., GAM-MIGAM(A,X) IS CØMPUTED WITH A CØNTINUED
C FRACTIØN APPRØXIMATIØN.  FØR ABS(X) .LE. 1.0, THE INTEGRAL
C IS TRANSFØRMED AND EXP(Q) IS APPRØXIMATED WITH A CHEBYSHEV
C SERIES SØ THAT THE NEW INTEGRAL MAY BE DØNE  ANALYTICALLY.
C FØR X .GT. -12. AND X .LT. 5. (ABS(X) .GT. 1.0), A CØNTIN-
C UED  FRACTIØN  APPRØXIMATIØN IS USED.  FINALLY  FØR X .LE.
C -12., THE ASYMPTØTIC EXPANSIØN IS USED.
C
C SGN IS A SWITCH  WHICH, IF NØNZERØ, INDICATES  WHETHER GAM
C SHØULD BE ADDED ØR SUBTRACTED FRØM AN INTERMEDIATE RESULT.
C
      DATA EXPLIM/20./
      Z=X1
      SGN=0.
      TIM=-1.
      EXPDIF=1.0
 5    IF (Z .NE. 0.) GØ TØ 10
      GAM1=0.
      SGN=SGN+TIM
      GØ TØ 40
 10   IF (Z .LE. 5.) GØ TØ 20
C USE EQUATIØN 10.
      GAM1=-EXPDIF*Z**A/(Z+(1.-A)/(1.+1./(Z+(2.-A)/(1.+2.
     1  /(Z+(3.-A)/(1.+3./(Z+1.7)))))))
      GØ TØ 40
 20   AZ=ABS(Z)
      IF (Z .LE. -12.) GØ TØ 30
      SGN=SGN+TIM
C USE EQUATIØN 17.
      IF (AZ .LE. 1.) GAM1=EXPDIF*Z/A *AZ**(A-1.)
     1  *(1.        +Z/(A+1.) *(.9999999+Z/(A+2.)
     2  *(.9999999 +Z/(A+3.) *(1.000008+Z/(A+4.)
     3  *(1.000005 +Z/(A+5.) *(.9994316+Z/(A+6.)
     4  *(.9995587 +Z/(A+7.) *(1.031684+Z/(A+8.)
     5  *1.028125))))))))
C USE EQUATIØNS 11 AND 12.  EVALUATIØN MUST BE DØNE
C IN DØUBLE PRECISIØN IF CØMPUTER HAS 32 ØR FEWER BITS
C PER WØRD.  ØN AN IBM 360, D. P. EVALUATIØN IS FØRCED
C BY THE D. P. CØNSTANTS IN CØNTINUATIØN CARD 9.
      IF (AZ .GT. 1.) GAM1=EXPDIF*Z/A *AZ**(A-1.)
     1  /(1.- A     *Z/( A      *(A+ 1.+    Z/((A+ 2.)
     2  *(1.-(A+1.)*Z/((A+ 2.)*(A+ 3.+2.*Z/((A+ 4.)
     3  *(1.-(A+2.)*Z/((A+ 4.)*(A+ 5.+3.*Z/((A+ 6.)
     4  *(1.-(A+3.)*Z/((A+ 6.)*(A+ 7.+4.*Z/((A+ 8.)
     5  *(1.-(A+4.)*Z/((A+ 8.)*(A+ 9.+5.*Z/((A+10.)
     6  *(1.-(A+5.)*Z/((A+10.)*(A+11.+6.*Z/((A+12.)
     7  *(1.-(A+6.)*Z/((A+12.)*(A+13.+7.*Z/((A+14.)
     8  *(1.00150-A*8.95E-5 +Z*(-.0337062+A*.0004182
     9  +Z*(.00999294-A*.000104103))) )))) )))) ))))
     A  )))) )))) )))) ))))
      GØ TØ 40
C USE EQUATIØN 18 AND SHANK-S E1 PRØCESS ØNCE.
 30   GAM1=-EXPDIF*AZ**(A-1.)*(1.+(A-1.)*(1.+(A-2.)*
     1  (1.+(A-3.)*(1.+(A-4.)*(1.+(A-5.)/(Z-A+6.))
     2  /Z)/Z)/Z)/Z)
 40   IF (TIM .GT. 0.) GØ TØ 55
      GAMINC=GAM1
      IF (ABS(X1-X2) .GT. EXPLIM) GØ TØ 50
C IF TRUE, CØNTRIBUTIØN AT X2 IS .LT. 1.E-7 *(CØNTR AT X1),
C PRØVIDED X2 .GT. X1.
      Z=X2
      EXPDIF=EXP(X1-X2)
      TIM=1.
      GØ TØ 5
 50   GAM1=0.
 55   GAMINC=GAM1-GAMINC
      IF (SGN .NE. 0.) GAMINC=GAMINC-SIGN(GAM*EXP(X1),SGN)
      RETURN
      END
```

ACM Transactions on Mathematical Software, Vol. 4, No. 3, September 1978, Pages 296-304.

REMARK ON ALGORITHM 435

Modified Incomplete Gamma Function [S14]
[Wayne Fullerton, *Comm. ACM 15,* 11 (Nov. 1972), 993-995]

Andrew Y. Schoene [Recd 18 May 1977 and 13 October 1977]
Research Laboratories, General Motors Technical Center, General Motors Corporation, Warren, MI 48090

The following changes were made to ACM Algorithm 435:

(1) .LE. in the line labeled 10 was changed to .LT. to conform with the algorithm presented in the text.

(2) .LE. in the line following the line labeled 20 was changed to .LT. This change is recommended because the continued fraction [eq. (11)] is more accurate than the asymptotic expansion [eq. (18)] at $X = -12$.[1]

[1] Equation numbers in this Remark refer to those in ACM Algorithm 435, referenced above.

Note, also, that the expression for T_i following eq. (10) contains a misprint: the numerator should read $i - a$ rather than $i + a$.

With changes (1) and (2) the algorithm was executed on an IBM 370/168 using the Fortran H extended (Opt = 2) compiler, and Table I of Algorithm 435 was approximately reproduced (see Table I of this Remark).

The proposed method for extending the range of applicability of *GAMINC* is, however, not entirely satisfactory. It is the purpose of this Remark to show how Fullerton's methods may be successfully employed to compute his modified incomplete Gamma function for an extended parameter range. A Fortran function. subprogram *GAMDRV* which accomplishes this is included here; it serves partly as a driver for *GAMINC* and should be a useful companion to it.

To compute *GAMINC(A, X1, X2)* for $2 < a < 15$, Fullerton suggests the use of forward recursion. However, satisfactory accuracy cannot be maintained for all values of the parameters due to numerical instability of the recursion. For simplicity we consider only the modified incomplete Gamma function defined by Fullerton as $G(a, x) = \int_0^x |y|^{a-1} \exp(-y)\, dy$. Using the methods of Gautschi [1], forward recursion for $G(a, x)$ can be shown to be numerically unstable for $x > 0$ and for $x < 0$ with $a > |x|$. For example, computing $G(12.5, 2)$ by double-precision forward recursion starting from $G(1.5, 2)$ yields a value with the *incorrect sign*.

While recursion cannot be used indiscriminately, it is possible to extend $G(a, x)$ to the range $2 < a < 15$ while maintaining approximately six-significant-digit accuracy. This can be done most simply by dividing the x-axis into three regions and using a different extension in each region. This task can be appreciably simplified by evaluating the term $|x|^{a-1}$ in eqs. (11) and (17) of Algorithm 435 in double precision. The Fortran function subprogram *GAMDRV*(A, X), when used in conjunction with a version of *GAMINC* modified as suggested above, will compute $G(a, x)$ for $1 \le a < 15$ and $-EXPLIM \le x < \infty$ to an accuracy of approximately six significant digits. *EXPLIM* is a machine-dependent constant (with the value 20. for the IBM 360/370 series) used in *GAMINC* to prevent exponent overflow. The extensions employed by *GAMDRV* are sketched by region as follows.

Region 1: $x \ge 5$. *GAMDRV* also makes use of the complementary incomplete Gamma function denoted by $CG(a, x) = \int_x^\infty y^{a-1} \exp(-y)\, dy$ and its continued fraction approximation [eq. (10)] from Algorithm 435. Since up to three digits of accuracy may be lost in the subtraction $G(a, x) = \text{Gamma}(x) - CG(a, x)$ for values of a near 15, it is necessary to use double precision exclusively in this region. Five terms of eq. (10) are used, with T_5 represented by a linear function of a, selected to fit for $x = 5$, $2 \le a \le 3$. After subtraction from Gamma(x) this basic approximation yields six-digit accuracy in the region $x \ge 5$, $a \le .5\ (x + 4)$ (this bound is slightly conservative to simplify the code). For larger a, the recurrence relation $CG(a + 1, x) = a \cdot CG(a, x) + x^a \cdot \exp(-x)$ is employed after first reducing a to get a sufficiently accurate starting value.

Region 2: $-12 \le x < 5$. If $|x|^{a-1}$ in eqs. (11) and (17) of *GAMINC* is evaluated in double precision as suggested above, then *GAMINC* achieves approximately six-digit accuracy for $2 < a < 15$. On the assumption that this has been done, *GAMDRV* calls *GAMINC* to obtain the value. If *GAMINC* is not so modified, then as a very rough approximation the relative error increases linearly with a, reaching levels of 20×10^{-6} for $a > 10$.

We consider further the evaluation of eqs. (11) and (12). On the IBM 370 series the double-precision constants in eq. (12) cause some subexpressions of eq. (11) to be evaluated in double precision while others involving only a and z are evaluated in single precision and the results subsequently extended to double precision. If double precision is used for the entire expression (including $|x|^{a-1}$), then only six terms of the continued fraction are required to achieve six-digit accuracy throughout the entire range $1 \le a < 15$. A slight complication in the coding is that different approximations to T_6 must be used for $x < 0$ and $x > 0$. The following were obtained by a least squares fit to computed values of T_6 for

Table I. Relative Errors of GAMDRV (× 10^6)

	a				
x	1.50	5.50	8.00	11.00	14.50
−18.00	0.61	2.57	0.87	1.19	3.63
−14.00	0.10	1.40	0.85	1.03	2.79
−12.00	0.14	0.07	0.12	0.12	0.30
− 8.00	0.65	0.12	0.01	0.57	0.00
− 4.00	0.21	0.36	0.12	0.15	0.18
− 2.00	0.12	0.10	0.05	0.01	0.25
− 0.50	0.15	0.32	0.12	0.14	0.25
0.50	0.84	0.75	0.59	0.29	0.63
2.00	0.17	0.36	0.29	0.15	0.18
5.00	0.03	0.62	0.18	0.00	0.33
7.00	0.03	1.64	0.05	0.02	0.02

Table II. Execution Times of *GAMDRV* in Milliseconds on the IBM 370/168.
(The numbers in parentheses represent the original *GAMINC* values.)

	a			
x	1.5	5.	10.	15.
$X < -12.$	.21 (.20)	.23	.36	.39
$(-12. \leq X < -1.\ 1. < X < 5.)$	.27 (.28)	.28	.28	.28
$\lvert X \rvert \leq 1.$	.20 (.17)	.21	.21	.21
$X = 5.$	.21 (.20)	.27	.34	.38
$X = 10.$	.21	.25	.33	.37
$X = 15.$	.21	.25	.31	.34

the critical regions $-12 \leq x \leq -10$, $1 \leq a \leq 2$ and $4 \leq x \leq 5$, $1 \leq a \leq 2$, respectively:

$$\begin{aligned} T_6 &\doteq .92391 + x \cdot (-.065094 + .00073933 \cdot x) \\ &\quad + a \cdot (.020541 + .0020402 \cdot a + .0060327 \cdot x), \quad x < 0 \\ &\doteq .96410 - x \cdot (.029325 + .0012057 \cdot a) \\ &\quad + .0034758 \cdot a, \qquad x > 0. \end{aligned}$$

This six-term double-precision approximation executes slightly faster than the original eight-term approximation on the 370/168. Double-precision arithmetic is only modestly slower than single precision on this computer.

Region 3: $x < -12$. *GAMINC* yields approximately six-digit accuracy for $2 \leq a \leq 6$ and is called by *GAMDRV* for such a. Larger a are reduced to the range $1 < a \leq 2$ (it is necessary to start the recursion with as accurate an a value as possible) and the forward recursion relation $G(a + 1, x) = -x^a \cdot \exp(-x) - a \cdot G(a, x)$ employed. This recursion is essentially stable in the above range, although accuracy deteriorates slightly for a near 15 where the maximum observed relative error of 5×10^{-6} occurs.

Table I presents the absolute value of the relative errors (multiplied by 10^6) for selected values of a and x using *GAMDRV* in conjunction with the modified version of *GAMINC* described above. The exact values were found as described in Algorithm 435. For $x > 8$ the observed relative errors were always less than 1. Execution times of *GAMDRV* for various arguments are given in Table II.

REFERENCES

1. Gautschi, W., and Klein, B.J. Recursive computation of certain derivatives—A study of error propagation. *Comm. ACM 13*, 1 (Jan. 1970), 7-9.

ALGORITHM

```
C                                                                         SCH00400
C  TEST DRIVER FOR FUNCTION SUBPROGRAM GAMDRV                             SCH00450
C  FOR -EXPLIM .LE. X .LT. 10. THE "EXACT" ANSWER IS COMPUTED BY          SCH00500
C  FUNCTION SUBPROGRAM SUMSER.                                            SCH00550
C  FOR X .GE. 10. THE "EXACT" ANSWER IS COMPUTED BY FUNCTION              SCH00600
C  SUBPROGRAM COMGAM.                                                     SCH00650
C                                                                         SCH00700
      DOUBLE PRECISION AA,XX,GAM,SERIES,DELTA                             SCH00750
      DOUBLE PRECISION SUMSER,COMGAM                                      SCH00800
      DIMENSION A(30),X(30),T(30,30)                                      SCH00850
   50 READ(5,9) NA,NX                                                     SCH00900
      IF (NA .EQ. 0) GO TO 400                                            SCH00950
      READ(5,10) (A(I),I=1,NA)                                            SCH01000
      READ(5,10) (X(J), J=1,NX)                                           SCH01050
      WRITE(6,1) NA                                                       SCH01100
      WRITE(6,2) (A(I), I=1,NA)                                           SCH01150
      WRITE(6,3) NX                                                       SCH01200
      WRITE(6,2) (X(J), J=1,NX)                                           SCH01250
      WRITE(6,4)                                                          SCH01300
      DO 200 J=1,NX                                                       SCH01350
         DO 100 I=1,NA                                                    SCH01400
            XX = X(J)                                                     SCH01450
            AA = A(I)                                                     SCH01500
            GAM = GAMDRV(A(I),X(J),IER)                                   SCH01550
            IF (IER .NE. 0) WRITE(6,7) IER                                SCH01600
            IF (X(J) .LT. 10.) SERIES = SUMSER(A(I),X(J))                 SCH01650
            IF (X(J) .GE. 10.) SERIES = COMGAM(A(I),X(J))                 SCH01700
            RELDEL = 0.                                                   SCH01750
            IF (SERIES .EQ. 0.D0) GO TO 20                                SCH01800
             DELTA = (GAM-SERIES)/SERIES                                  SCH01850
            RELDEL = ABS(SNGL(DELTA))                                     SCH01900
            T(I,J) = 1.D6*(DABS(SERIES-GAM)/SERIES)                       SCH01950
   20       WRITE(6,5) A(I),X(J),GAM,SERIES,RELDEL                        SCH02000
  100     CONTINUE                                                        SCH02050
  200  CONTINUE                                                           SCH02100
      WRITE(6,8)                                                          SCH02150
      WRITE(6,6) (A(I), I=1,NA)                                           SCH02200
      DO 300 J=1,NX                                                       SCH02250
  300    WRITE(6,2) X(J),(T(I,J), I=1,NA)                                 SCH02300
    1 FORMAT(///30H NUMBER OF INPUT VALUES OF A =,I3)                     SCH02350
    2 FORMAT(/16F8.2)                                                     SCH02400
    3 FORMAT(/30H NUMBER OF INPUT VALUES OF X =,I3)                       SCH02450
    4 FORMAT(/45H     A         X        GAMDRV           EXACT,          SCH02500
     *        18H           REL ERR/)                                     SCH02550
    5 FORMAT(2F8.2,2D18.8,E12.3)                                          SCH02600
    6 FORMAT(//9X,15F8.2/)                                                SCH02650
    7 FORMAT(/5H IER=,I3/)                                                SCH02700
    8 FORMAT(//10X,41HTABLE OF RELATIVE ERRORS OF GAMDRV X 1.E6)          SCH02750
    9 FORMAT(2I3)                                                         SCH02800
   10 FORMAT(12F6.2)                                                      SCH02850
      GO TO 50                                                            SCH02900
  400  STOP                                                               SCH02950
C LAST CARD OF TEST DRIVER PROGRAM FOR FUNCTION SUBPROGRAM GAMDRV         SCH03000
      END                                                                 SCH03050
```

```
      FUNCTION GAMDRV(A,X,IER)                                            SCH03450
      REAL A,X                                                            SCH03500
      INTEGER IER                                                         SCH03550
C                                                                         SCH03600
C PURPOSE:  COMPUTES A MODIFIED INCOMPLETE GAMMA FUNCTION DEFINED         SCH03650
C AS THE INTEGRAL OF ABS(Y)**(A-1.) * EXP(-Y) FROM 0 TO X, WHERE          SCH03700
C X MAY BE NEGATIVE.  GAMDRV IS AN EXTENSION OF GAMINC (ALGORITHM         SCH03750
C 435, CACM), AND USES IT AS AN AUXILIARY FUNCTION SUBPROGRAM.            SCH03800
C                                                                         SCH03850
C PRECISION:  SINGLE                                                      SCH03900
C                                                                         SCH03950
C ARGUMENT RESTRICTIONS:  1. .LE. A .LT. 15.                              SCH04000
C                        -EXPLIM .LE. X .LT. INFINITY                     SCH04050
C     EXPLIM IS A MACHINE DEPENDENT CONSTANT USED BY GAMINC TO PREVENT    SCH04100
C     EXPONENT OVERFLOW.  IT HAS THE VALUE 20. FOR THE 360/370 SERIES.    SCH04150
C                                                                         SCH04200
C ERROR RETURNS:  IER = -1   A .LE. 0. OR X .LT. -EXPLIM                  SCH04250
```

```
C                  IER =  Ø   NORMAL RETURN                             SCHØ43ØØ
C                  IER =  1   Ø. .LT. A .LT 1.                          SCHØ435Ø
C                  IER = 15   A .GE. 15.                                SCHØ44ØØ
C  FOR IER = -1 GAMDRV RETURNS THE VALUE Ø., WHILE FOR Ø. .LT. A .LT. 1.SCHØ445Ø
C  OR A .GE. 15. THE (INACCURATE) APPROXIMATION IS RETURNED.            SCHØ45ØØ
C                                                                       SCHØ455Ø
C  SUBROUTINES REQUIRED:  DGAMMA (DOUBLE PRECISION GAMMA FUNCTION)      SCHØ46ØØ
C  AND GAMINC, WHICH IN TURN REQUIRES GAMMA (SINGLE PRECISION GAMMA     SCHØ465Ø
C  FUNCTION).  BOTH ARE COMMONLY INCLUDED IN THE FORTRAN LIBRARY OF     SCHØ47ØØ
C  FUNCTIONS.  OTHER SOURCES ARE THE IMSL (INTERNATIONAL MATHEMATICAL   SCHØ475Ø
C  AND STATISTICAL LIBRARIES, INC.) AND NAG (NOTTINGHAM ALGORITHMS      SCHØ48ØØ
C  GROUP) FORTRAN LIBRARIES.  ALGORITHM 54, COMM. ACM 4 (APRIL 1961),   SCHØ485Ø
C  P. 18Ø, IS ALSO SATISFACTORY FOR THE SINGLE PRECISION GAMMA          SCHØ49ØØ
C  FUNCTION.  ALGORITHM 221, COMM. ACM 7 (MARCH 1964), P.143,           SCHØ495Ø
C  ACHIEVES 1Ø SIGNIFICANT DIGIT ACCURACY WHICH IS SUFFICIENT FOR       SCHØ5ØØØ
C  THE DGAMMA REQUIRED BY GAMDRV.                                       SCHØ5Ø5Ø
C                                                                       SCHØ51ØØ

      DOUBLE PRECISION DA,DX,DEXPXA,DGAM1                               SCHØ515Ø
      DATA EXPLIM/2Ø.Ø/                                                 SCHØ52ØØ
      IER=Ø                                                             SCHØ525Ø
      ASAVE = A                                                         SCHØ53ØØ
      IF (X .NE. Ø.) GO TO 1Ø                                           SCHØ535Ø
         GAMDRV = Ø.                                                    SCHØ54ØØ
         RETURN                                                         SCHØ545Ø
C                                                                       SCHØ55ØØ
  1Ø  IF (X .LT. -EXPLIM) GO TO 5Ø                                      SCHØ555Ø
C                                                                       SCHØ56ØØ
      IF (A .GT. Ø.) GO TO 1ØØ                                          SCHØ565Ø
C  A IS NOT POSITIVE OR X IS LESS THAN -EXPLIM                          SCHØ57ØØ
  5Ø     IER = -1                                                       SCHØ575Ø
         GAMDRV = Ø.                                                    SCHØ58ØØ
         RETURN                                                         SCHØ585Ø
C                                                                       SCHØ59ØØ
 1ØØ  IF (A .GT. 2.) GO TO 11Ø                                          SCHØ595Ø
C  A IS LESS THAN OR EQUAL TO 2.                                        SCHØ6ØØØ
          IF (A .LT. 1.) IER = 1                                        SCHØ6Ø5Ø
          GAMDRV = GAMINC(A,Ø.,X)                                       SCHØ61ØØ
          RETURN                                                        SCHØ615Ø
C                                                                       SCHØ62ØØ
 11Ø  IF (A .LT. 15.) GO TO 21Ø                                         SCHØ625Ø
C  A .GE. 15.  SET IER = 15 AND CONTINUE                                SCHØ63ØØ
      IER = 15                                                          SCHØ635Ø
C                                                                       SCHØ64ØØ
 21Ø  IF ((X .LT. -12.) .AND. (A .GT. 6.)) GO TO 22Ø                    SCHØ645Ø
      IF (X .GE. 5.) GO TO 23Ø                                          SCHØ65ØØ
C  -12 .LE. X .LT. 5.  GET REQUIRED VALUE FROM GAMINC.                  SCHØ655Ø
         GAMDRV = GAMINC(A,Ø.,X)                                        SCHØ66ØØ
         RETURN                                                         SCHØ665Ø
C                                                                       SCHØ67ØØ
C   REDUCE A TO RANGE 1. .LT. A .LE. 2. AND USE FORWARD RECURSION       SCHØ675Ø
C                                                                       SCHØ68ØØ
 22Ø  NRECUR = INT(A)-2                                                 SCHØ685Ø
C    IF A IS NOT INTEGRAL, ONE MORE RECURSION WILL BE NEEDED            SCHØ69ØØ
      IF (A-FLOAT(NRECUR) .GT. 2.) NRECUR = NRECUR+1                    SCHØ695Ø
      A = A-FLOAT(NRECUR)                                               SCHØ7ØØØ
      SIGNX = SIGN(1.Ø,X)                                               SCHØ7Ø5Ø
      EXPXA = EXP(-X) * ABS(X)**A                                       SCHØ71ØØ
C                                                                       SCHØ715Ø
C  CALL GAMINC TO GET INITIAL VALUE FOR RECURSION                       SCHØ72ØØ
C                                                                       SCHØ725Ø
      GAM1 = GAMINC(A,Ø.,X)                                             SCHØ73ØØ
      DO 225 K = 1,NRECUR                                               SCHØ735Ø
         GAM1 = SIGNX * (EXPXA + GAM1*A)                                SCHØ74ØØ
         A = A+1.Ø                                                      SCHØ745Ø
         EXPXA = EXPXA*ABS(X)                                           SCHØ75ØØ
 225  CONTINUE                                                          SCHØ755Ø
      GAMDRV = GAM1                                                     SCHØ76ØØ
      A = ASAVE                                                         SCHØ765Ø
      RETURN                                                            SCHØ77ØØ
C                                                                       SCHØ775Ø
C  CALCULATE THE COMPLEMENTARY INCOMPLETE GAMMA FUNCTION.  IF A IS      SCHØ78ØØ
C  TOO LARGE, REDUCE A AND USE FORWARD RECURSION.  DOUBLE PRECISION     SCHØ785Ø
C  IS REQUIRED SINCE SIGNIFICANT CANCELLATION OF LEADING DIGITS         SCHØ79ØØ
C  MAY OCCUR IN SUBTRACTION FROM GAMMA(A) WHEN A IS LARGE.              SCHØ795Ø
```

```
C                                                                       SCHØ8ØØØ
  23Ø NRECUR = Ø                                                        SCHØ8Ø5Ø
      RANGE = .5*(X+4.)                                                 SCHØ81ØØ
C                                                                       SCHØ815Ø
C  TEST TO SEE IF FORWARD RECURSION IS NECESSARY.                       SCHØ82ØØ
      IF (A .LE. RANGE .OR. X .GE. 22.) GO TO 235                       SCHØ825Ø
C                                                                       SCHØ83ØØ
      NRECUR = INT(A-RANGE) + 2                                         SCHØ835Ø
      A = A-FLOAT(NRECUR)                                               SCHØ84ØØ
  235 DA = A                                                            SCHØ845Ø
      DX = X                                                            SCHØ85ØØ
      DEXPXA = DEXP(-DX) * DABS(DX)**DA                                 SCHØ855Ø
      DGAM1 = DEXPXA/(DX+(1.DØ-DA)/                                     SCHØ86ØØ
     1          (1.DØ + 1.DØ/(DX+(2.DØ-DA)/                             SCHØ865Ø
     2          (1.DØ + 2.DØ/(DX+(3.DØ-DA)/                             SCHØ87ØØ
     3          (1.DØ + 3.DØ/(DX+(4.DØ-DA)/                             SCHØ875Ø
     4          (1.DØ + 4.DØ/(DX+1.78DØ - .64DØ*(DA-2.DØ)))))))))       SCHØ88ØØ
      IF (NRECUR .EQ. Ø) GO TO 25Ø                                      SCHØ885Ø
C                                                                       SCHØ89ØØ
C   DO FORWARD RECURSION FOR COMPLEMENTARY INCOMPLETE GAMMA             SCHØ895Ø
      DO 24Ø K=1,NRECUR                                                 SCHØ9ØØØ
         DGAM1 = DGAM1*DA + DEXPXA                                      SCHØ9Ø5Ø
         DA = DA+1.DØ                                                   SCHØ91ØØ
         DEXPXA = DEXPXA*DX                                             SCHØ915Ø
  24Ø CONTINUE                                                          SCHØ92ØØ
  25Ø GAMDRV = DGAMMA(DA) - DGAM1                                       SCHØ925Ø
      A = ASAVE                                                         SCHØ93ØØ
      RETURN                                                            SCHØ935Ø
C  LAST CARD OF FUNCTION SUBPROGRAM GAMDRV                              SCHØ94ØØ
      END                                                               SCHØ945Ø
C                                                                       SCHØ985Ø


      FUNCTION GAMINC(A,X1,X2)                                          SCHØ99ØØ
C                                                                       SCHØ995Ø
C  MODIFIED VERSION OF ALGORITHM 435, MODIFIED INCOMPLETE GAMMA         SCH1ØØØØ
C  FUNCTION, TO BE USED WITH FUNCTION SUBPROGRAM GAMDRV.  THE           SCH1ØØ5Ø
C  MODIFICATIONS ARE DESCRIBED IN THE ACCOMPANYING TEXT.                SCH1Ø1ØØ
C                                                                       SCH1Ø15Ø
      DOUBLE PRECISION DA,DZ,T6                                         SCH1Ø2ØØ
      DATA EXPLIM/2Ø./                                                  SCH1Ø25Ø
      DATA ZERO/Ø./,ONE/1./,FIVE/5./,TWELVE/12./                        SCH1Ø3ØØ
      Z = X1                                                            SCH1Ø35Ø
      SGN = Ø.                                                          SCH1Ø4ØØ
      TIM = -1.                                                         SCH1Ø45Ø
      EXPDIF = 1.                                                       SCH1Ø5ØØ
    5 IF (Z .NE. ZERO) GO TO 1Ø                                         SCH1Ø55Ø
      GAM1 = Ø.                                                         SCH1Ø6ØØ
      SGN = SGN + TIM                                                   SCH1Ø65Ø
      GO TO 4Ø                                                          SCH1Ø7ØØ
   1Ø IF (Z .LT. FIVE) GO TO 2Ø                                         SCH1Ø75Ø
C  USE EQUATION 1Ø (SEE REFERENCE)                                      SCH1Ø8ØØ
      GAM1 = -EXPDIF * Z**A/(Z+(1.-A)/(1.+1./(Z+(2.-A)/(1.+2.           SCH1Ø85Ø
     1       /(Z+(3.-A)/(1.+3./(Z+1.7)))))))                            SCH1Ø9ØØ
      GO TO 4Ø                                                          SCH1Ø95Ø
   2Ø AZ =  ABS(Z)                                                      SCH11ØØØ
      IF (Z .LT. -TWELVE) GO TO 3Ø                                      SCH11Ø5Ø
      SGN = SGN + TIM                                                   SCH111ØØ
      IF (AZ .GT. ONE) GO TO 25                                         SCH1115Ø
C  USE EQUATION 17                                                      SCH112ØØ
       GAM1 = EXPDIF*Z/A* DBLE(AZ)**(A-1.)                              SCH1125Ø
     1  *(1.ØØØØØØ +Z/(A+1.) *(.9999999+Z/(A+2.)                        SCH113ØØ
     2  *(.9999999 +Z/(A+3.) *(1.ØØØØØ8+Z/(A+4.)                        SCH1135Ø
     3  *(1.ØØØØØ5 +Z/(A+5.) *(.9994316+Z/(A+6.)                        SCH114ØØ
     4  *(.9995587 +Z/(A+7.) *(1.Ø31684+Z/(A+8.)                        SCH1145Ø
     5  * 1.Ø28125))))))))                                              SCH115ØØ
      GO TO 4Ø                                                          SCH1155Ø
C                                                                       SCH116ØØ
C  USE EQUATIONS 11 AND 12.  EVALUATION IS DONE IN DOUBLE PRECISION.    SCH1165Ø
C                                                                       SCH117ØØ
   25 DA = A                                                            SCH1175Ø
      DZ = Z                                                            SCH118ØØ
      IF (Z .LT. Ø.) T6 = .92391DØ + DZ*(-.65Ø94D-1 + .73933D-3*DZ)     SCH1185Ø
     1                  + DA*(.2Ø541D-1 + .2Ø4Ø2D-2*DA + .6Ø327D-2*DZ)  SCH119ØØ
      IF (Z .GT. Ø.) T6 = .9641ØDØ - DZ*(.29325D-1 + .12Ø57D-2*DA)      SCH1195Ø
     1                  + .34758D-2*DA                                  SCH12ØØØ
      GAM1 = EXPDIF*DZ/DA * DBLE(AZ)**(DA-1.DØ)                         SCH12Ø5Ø
     1  /(1.DØ- DA      *DZ/( DA       *(DA+ 1.DØ+1.DØ*DZ/((DA+ 2.DØ)   SCH121ØØ
     2  *(1.DØ-(DA+1.DØ)*DZ/((DA+ 2.DØ)*(DA+ 3.DØ+2.DØ*DZ/((DA+ 4.DØ)   SCH1215Ø
     3  *(1.DØ-(DA+2.DØ)*DZ/((DA+ 4.DØ)*(DA+ 5.DØ+3.DØ*DZ/((DA+ 6.DØ)   SCH122ØØ
     4  *(1.DØ-(DA+3.DØ)*DZ/((DA+ 6.DØ)*(DA+ 7.DØ+4.DØ*DZ/((DA+ 8.DØ)   SCH1225Ø
     5  *(1.DØ-(DA+4.DØ)*DZ/((DA+ 8.DØ)*(DA+ 9.DØ+5.DØ*DZ/((DA+1Ø.DØ)   SCH123ØØ
     6  * T6 )))) )))) )))) )))) ))))                                    SCH1235Ø
      GO TO 4Ø                                                           SCH124ØØ
C  USE EQUATION 18 AND SHANKS E1 PROCESS ONCE                            SCH1245Ø
  3Ø  GAM1 = -EXPDIF*AZ**(A-1.)*(1.+(A-1.)*(1.+(A-2.)*                   SCH125ØØ
     1       (1.+(A-3.)*(1.+(A-4.)*(1.+(A-5.)/(Z-A+6.))/Z)/Z)/Z)/Z)      SCH1255Ø
  4Ø  IF (TIM .GT. ZERO) GO TO 55                                        SCH126ØØ
      GAMINC = GAM1                                                      SCH1265Ø
      IF (ABS(X1-X2) .GT. EXPLIM) GO TO 5Ø                               SCH127ØØ
C                                                                        SCH1275Ø
C  IF TRUE, CONTRIBUTION AT X2 IS .LT. 1.E-7 * CONTRIBUTION AT X1,       SCH128ØØ
C  PROVIDED X2 .GT. X1.                                                  SCH1285Ø
      Z = X2                                                             SCH129ØØ
      EXPDIF = EXP(X1-X2)                                                SCH1295Ø
      TIM = 1.                                                           SCH13ØØØ
      GO TO 5                                                            SCH13Ø5Ø
  5Ø  GAM1 = Ø.                                                          SCH131ØØ
  55  GAMINC = GAM1 - GAMINC                                             SCH1315Ø
      IF (SGN .NE. ZERO) GAMINC = GAMINC - SIGN(GAMMA(A)*EXP(X1),SGN)    SCH132ØØ
      RETURN                                                             SCH1325Ø
C  LAST CARD OF FUNCTION SUBPROGRAM GAMINC                               SCH133ØØ
      END                                                                SCH1335Ø
C                                                                        SCH1375Ø


      DOUBLE PRECISION FUNCTION COMGAM(A,X)                              SCH138ØØ
C                                                                        SCH1385Ø
C  COMPUTES THE INCOMPLETE GAMMA FUNCTION BY SUBTRACTING A               SCH139ØØ
C  CONTINUED FRACTION EXPANSION FOR THE COMPLEMENTARY INCOMPLETE         SCH1395Ø
C  GAMMA FUNCTION FROM DGAMMA(X).                                        SCH14ØØØ
C  REFERENCE:  ABROMOWITZ, M.,AND STEGUN, I.A.. HANDBOOK OF MATHEMATICALSCH14Ø5Ø
C  FUNCTIONS. NATIONAL BUREAU OF STANDARDS, U.S. GOV. PRINT. OFF.,       SCH141ØØ
C  WASHINGTON D.C., 1967, P. 263, FORMULA 6.5.31                         SCH1415Ø
C                                                                        SCH142ØØ
      DOUBLE PRECISION DA,DX,TK                                          SCH1425Ø
      DA = A                                                             SCH143ØØ
      DX=X                                                               SCH1435Ø
      TK=Ø.DØ                                                            SCH144ØØ
      LAST=2Ø                                                            SCH1445Ø
      DO 1Ø K=1,LAST                                                     SCH145ØØ
         FK = FLOAT(LAST+1-K)                                            SCH1455Ø
         TK = (DBLE(FK)-DA)/(1.DØ+DBLE(FK)/(DX+TK))                      SCH146ØØ
  1Ø   CONTINUE                                                          SCH1465Ø
      TK = DEXP(-DX)*DX**DA/(DX+TK)                                      SCH147ØØ
      COMGAM = DGAMMA(DA) - TK                                           SCH1475Ø
      RETURN                                                             SCH148ØØ
C  LAST CARD OF FUNCTION SUBPROGRAM COMGAM                               SCH1485Ø
      END                                                                SCH149ØØ

      DOUBLE PRECISION FUNCTION SUMSER(A,X)                              SCH153ØØ
C                                                                        SCH1535Ø
C  COMPUTES THE INCOMPLETE GAMMA FUNCTION FOR -EXPLIM .LE. X .LT. 1Ø.    SCH154ØØ
C  THE SERIES IS TRUNCATED AS DESCRIBED BY FULLERTON.                    SCH1545Ø
C  REFERENCE:  ABROMOWITZ, M.,AND STEGUN, I.A.. HANDBOOK OF MATHEMATICALSCH155ØØ
C  FUNCTIONS. NATIONAL BUREAU OF STANDARDS, U.S. GOV. PRINT. OFF.,       SCH1555Ø
C  WASHINGTON D.C., 1967, P. 262, FORMULA 6.5.29                         SCH156ØØ
C                                                                        SCH1565Ø
      DOUBLE PRECISION SUM,TERM,X2,ISIGN                                 SCH157ØØ
      DOUBLE PRECISION XX,AA                                             SCH1575Ø
      IF (X .NE. Ø.) GO TO 5                                             SCH158ØØ
         SUMSER = Ø.DØ                                                   SCH1585Ø
         RETURN                                                          SCH159ØØ
   5  XX = X                                                             SCH1595Ø
      AA = A                                                             SCH16ØØØ
      TERM = 1.DØ/AA                                                     SCH16Ø5Ø
      SUM = TERM - XX/(AA+1.DØ)                                          SCH161ØØ
      ISIGN=1.DØ                                                         SCH1615Ø
      X2 = XX*XX                                                         SCH162ØØ
      DO 1Ø N=2,1ØØ                                                      SCH1625Ø
         FN = FLOAT(N)                                                   SCH163ØØ
         TERM = X2*ISIGN/(2.DØ*(AA+DBLE(FN)))                            SCH1635Ø
         SUM = SUM + TERM                                               SCH164ØØ
         IF (DABS(TERM) .LT. 2.D-9 * DABS(SUM)) GO TO 2Ø                SCH1645Ø
         ISIGN = -ISIGN                                                 SCH165ØØ
         X2 = X2*XX/(DBLE(FN)+1.DØ)                                     SCH1655Ø
   1Ø CONTINUE                                                          SCH166ØØ
   2Ø SUMSER = DABS(XX)**AA * SUM                                       SCH1665Ø
      IF (X .LT. Ø.) SUMSER = -SUMSER                                   SCH167ØØ
      RETURN                                                            SCH1675Ø
C  LAST CARD OF FUNCTION SUBPROGRAM SUMSER                              SCH168ØØ
      END                                                               SCH1685Ø

  5 11                                                                  SCH1725Ø
   1.5   5.5   8.Ø  11.Ø  14.5                                          SCH173ØØ
  -18.  -14.  -12.   -8.   -4.   -2.   -.5    .5    2.    5.    7.      SCH1735Ø
  1  1                                                                  SCH174ØØ
  1.5                                                                   SCH1745Ø
 -25.                                                                   SCH175ØØ
  1  1                                                                  SCH1755Ø
 -1.2                                                                   SCH176ØØ
 4.                                                                     SCH1765Ø
  1  1                                                                  SCH177ØØ
 .5                                                                     SCH1775Ø
 2.                                                                     SCH178ØØ
  1  1                                                                  SCH1785Ø
 16.5                                                                   SCH179ØØ
 2.                                                                     SCH1795Ø
  Ø  Ø                                                                  SCH18ØØØ
```

Algorithm 436

Product Type Trapezoidal Integration [D1]

W. Robert Boland [Recd. 10 Dec. 1970 and 14 May 1971]
Department of Mathematics, Clemson University, Clemson, SC 29631

Key Words and Phrases: numerical integration, product type quadrature, trapezoidal integration
CR Categories: 5.16
Language: Fortran

Description

This subroutine uses the product type trapezoidal rule compounded n times to approximate the value of the integral

$$\int_a^b f(x)g(x)\,dx.$$

The approximating sum is

$$\frac{h}{6}\sum_{j=1}^{n}(f(a+(j-1)h), f(a+jh))\begin{pmatrix}2 & 1\\ 1 & 2\end{pmatrix}\begin{pmatrix}g(a+(j-1)h)\\ g(a+jh)\end{pmatrix},$$

where $h = (b-a)/n$. Note that if $g(x) \equiv 1$ (or $f(x) \equiv 1$), the rule reduces to the regular trapezoidal rule. The procedure was proposed and discussed by Boland and Duris in [1].

The subroutine was written in Fortran using double precision arithmetic and was checked on an IBM 360 Model 50. The calling parameters for the routine are as follows. A is the name for the lower limit of integration, and B is the name for the upper limit. N is the number of times the formula is to be compounded. The basic interval $[A, B]$ is subdivided into N subintervals each of length $(B - A)/N$ and the rule is applied to each subinterval. *FN* and *GN* are names of double precision *FUNCTION* subprograms which evaluate the functions $f(x)$ and $g(x)$, respectively. These are to be supplied by the user. The result is stored in *VINT*.

There are no machine dependent parameters.

References

1. Boland, W.R., and Duris, C.S. Product type quadrature formulas. *BIT 11*, 2 (1971), 139–158.

Algorithm

```
      SUBRØUTINE  PTRAP(A, B, N, FN, GN, VINT)
C
C THIS SUBRØUTINE USES THE PRØDUCT TYPE TRAPEZØIDAL RULE
C CØMPØUNDED N TIMES TØ APPRØXIMATE THE INTEGRAL FRØM A TØ B
C ØF THE FUNCTIØN FN(X) * GN(X). FN AND GN ARE FUNCTIØN
C SUBPRØGRAMS WHICH MUST BE SUPPLIED BY THE USER. THE
C RESULT IS STØRED IN VINT.
C
      DØUBLE PRECISIØN A, AG, AM(2,2), B, F(2), FN, G(2),
     *                GN, H, VINT, X, DBLE
      DATA AM(1,1), AM(2,2) /2 * 2.DO/, AM(1,2), AM(2,1)
     *     /2 * 1.DO/
      H = (B - A) / DBLE(FLØAT(N))
      VINT = 0.DO
      X = A
      F(2) = FN(A)
      G(2) = GN(A)
      DØ 2 I = 1, N
        F(1) = F(2)
        G(1) = G(2)
        X = X + H
        F(2) = FN(X)
        G(2) = GN(X)
        DØ 2 J = 1, 2
          AG = 0.DO
          DØ 1 K = 1, 2
1           AG = AG + AM(J,K) * G(K)
2         VINT = VINT + F(J) * AG
      VINT = H * VINT / 6.DO
      RETURN
      END
```

Algorithm 437

Product Type Simpson's Integration [D1]

W. Robert Boland [Recd. 10 Dec. 1970 and 14 May 1971]
Department of Mathematics, Clemson University, Clemson, SC 29631

Key Words and Phrases: numerical integration, product type quadrature, Simpson's rule
CR Categories: 5.16
Language: Fortran

Description

This subroutine uses the product type Simpson's rule compounded n times to approximate the value of the integral

$$\int_a^b f(x)g(x)\,dx.$$

The approximating sum is

$$\frac{h}{30}\sum_{j=1}^{n}(f(a+(j-1)h), f(a+(j-\tfrac{1}{2})h), f(a+jh)) \cdot \begin{pmatrix} 4 & 2 & -1 \\ 2 & 16 & 2 \\ -1 & 2 & 4 \end{pmatrix} \begin{pmatrix} g(a+(j-1)h) \\ g(a+(j-\tfrac{1}{2})h) \\ g(a+jh) \end{pmatrix},$$

where $h = (b-a)/n$. Note that if $g(x) \equiv 1$ (or $f(x) \equiv 1$), the rule reduces to the regular Simpson's rule. The procedure was proposed and discussed by Boland and Duris in [1].

The subroutine was written in Fortran using double precision arithmetic and was checked on an IBM 360 Model 50. The calling parameters for the routine are as follows. *A* is the name for the lower limit of integration and *B* is the name for the upper limit. *N* is the number of times the formula is to be compounded. The basic interval [*A*, *B*] is subdivided into *N* subintervals each of length $(B - A)/N$ and the rule is applied to each subinterval. *FN* and *GN* are names of double precision *FUNCTION* subprograms which evaluate the functions $f(x)$ and $g(x)$, respectively. These are to be supplied by the user. The result is stored in *VINT*.

There are no machine dependent parameters.

References

1. Boland, W.R., and Duris, C.S. Product type quadrature formulas. *BIT 11*, 2 (1971), 139–158.

Algorithm

```
      SUBRØUTINE PSIMP(A,B,N,FN,GN,VINT)
C
C THIS SUBRØUTINE USES THE PRØDUCT TYPE SIMPSØN RULE
C CØMPØUNDED N TIMES TØ APPRØXIMATE THE INTEGRAL FRØM A TØ B
C ØF THE FUNCTIØN FN(X) * GN(X). FN AND GN ARE FUNCTIØN
C SUBPRØGRAMS WHICH MUST BE SUPPLIED BY THE USER. THE
C RESULT IS STØRED IN VINT.
C
      DØUBLE PRECISIØN A, AG, AM(3,3), B, F(3), FN, G(3),
     *                 GN, H, VINT, X(2), DBLE
      DATA AM(1,1), AM(3,3) /2 * 4.DO/, AM(1,2), AM(2,1),
     *     AM(2,3), AM(3,2) /4 * 2.DO/, AM(1,3), AM(3,1)
     *     /2 * -1.DO/, AM(2,2) /16.DO/
      H = (B - A) / DBLE(FLØAT(N))
      X(1) = A + H / 2.DO
      X(2) = A + H
      VINT = 0.DO
      F(3) = FN(A)
      G(3) = GN(A)
      DØ 3 I = 1, N
        F(1) = F(3)
        G(1) = G(3)
        DØ 1 J = 1,2
          F(J+1) = FN(X(J))
          G(J+1) = GN(X(J))
1         X(J) = X(J) + H
        DØ 3 J = 1, 3
          AG = 0.DO
          DØ 2 K = 1, 3
2           AG = AG + AM(J,K) * G(K)
3         VINT = VINT + F(J) * AG
      VINT = H * VINT / 30.DO
      RETURN
      END
```

Algorithm 438

Product Type Two-point Gauss-Legendre-Simpson's Integration [D1]

W. Robert Boland [Recd. 10 Dec. 1970 and 14 May 1971]
Department of Mathematics, Clemson University, Clemson, SC 29631

Key Words and Phrases: numerical integration, product type quadrature, Gaussian quadrature, Simpson's rule
CR Categories: 5.16
Language: Fortran

Description

This subroutine uses the product type two-point Gauss-Legendre-Simpson's rule compounded n times to approximate the value of the integral

$$\int_a^b f(x)g(x)\,dx.$$

The approximating sum is

$$\frac{h}{12}\sum_{j=1}^{n}(f(a+(j-\tfrac{1}{2}-3^{1/2}/6)h), f(a+(j-\tfrac{1}{2}+3^{1/2}/6)h)) \cdot \begin{pmatrix} 1+3^{1/2} & 4 & 1-3^{1/2} \\ 1-3^{1/2} & 4 & 1+3^{1/2} \end{pmatrix} \begin{pmatrix} g(a+(j-1)h) \\ g(a+(j-\tfrac{1}{2})h) \\ g(a+jh) \end{pmatrix},$$

where $h = (b-a)/n$. Note that if $g(x) \equiv 1$, the rule reduces to the regular two-point Gauss-Legendre rule, while if $f(x) \equiv 1$, it reduces to the regular Simpson's rule. The procedure was proposed and discussed by Boland and Duris in [1].

The subroutine was written in Fortran using double precision arithmetic and was checked on an IBM 360 Model 50. The calling parameters for the routine are as follows. A is the name for the lower limit of integration and B is the name for the upper limit. N is the number of times the formula is to be compounded. The basic interval $[A, B]$ is subdivided into N subintervals each of length $(B-A)/N$ and the rule is applied to each subinterval. *FN* and *GN* are names of double precision *FUNCTION* subprograms which evaluate the functions $f(x)$ and $g(x)$, respectively. These are to be supplied by the user. The result is stored in *VINT*.

There are four machine dependent constants. These are:

(i) $1 + 3^{1/2} \approx 2.732050807568877$,
(ii) $1 - 3^{1/2} \approx -0.7320508075688773$,
(iii) $\frac{1}{2} - 3^{1/2}/6 \approx 0.2113248654051871$, and
(iv) $\frac{1}{2} + 3^{1/2}/6 \approx 0.7886751345948129$.

The first constant is assigned to $AM(1, 1)$ and $AM(2, 3)$, the second to $AM(1, 3)$ and $AM(2, 1)$, while the third and fourth are used in the calculation of $X(1)$ and $X(2)$, respectively.

References

1. Boland, W.R., and Duris, C.S. Product type quadrature formulas. *BIT 11*, 2 (1971), 139–158.

Algorithm

```
      SUBRØUTINE  P2PGS(A, B, N, FN, GN, VINT)
C
C THIS SUBRØUTINE USES THE PRØDUCT TYPE TWØ-PØINT GAUSS-
C LEGENDRE-SIMPSØN RULE CØMPØUNDED N TIMES TØ APPRØXIMATE
C THE INTEGRAL FRØM A TØ B ØF THE FUNCTIØN FN(X) * GN(X).
C FN AND GN ARE FUNCTIØN SUBPRØGRAMS WHICH MUST BE SUPPLIED
C BY THE USER.  THE RESULT IS STØRED IN VINT.
C
      DØUBLE PRECISIØN A, AG, AM(2,3), B, F(2), FN, G(3),
     *                  GN, H, VINT, X(2), Y(2), DBLE
      DATA AM(1,1), AM(2,3) /2 * 2.732050807568877D0/,
     *     AM(1,2), AM(2,2) /2 * 4.D0/, AM(1,3), AM(2,1)
     *     /2 * -.7320508075688773D0/
      H = (B - A) / DBLE(FLØAT(N))
      X(1) = A + .2113248654051871D0 * H
      X(2) = A + .7886751345948129D0 * H
      Y(1) = A + H / 2.D0
      Y(2) = A + H
      VINT = 0.D0
      G(3) = GN(A)
      DØ 3 I = 1, N
        G(1) = G(3)
        DØ 1 J = 1, 2
          F(J) = FN(X(J))
          G(J+1) = GN(Y(J))
          X(J) = X(J) + H
1         Y(J) = Y(J) + H
        DØ 3 J = 1, 2
          AG = 0.D0
          DØ 2 K = 1, 3
2           AG = AG + AM(J,K) * G(K)
3         VINT = VINT + F(J) * AG
      VINT = H * VINT / 12.D0
      RETURN
      END
```

Algorithm 439

Product Type Three-point Gauss-Legendre-Simpson's Integration [D1]

W. Robert Boland [Recd. 10 Dec. 1970 and 14 May 1971]

Department of Mathematics, Clemson University, Clemson, SC 29631

Key Words and Phrases: numerical integration, product type quadrature, Gaussian quadrature, Simpson's rule
CR Categories: 5.16
Language: Fortran

Description

This subroutine uses the product type three-point Gauss-Legendre-Simpson's rule compounded n times to approximate the value of the integral

$$\int_a^b f(x)g(x)\,dx.$$

The approximating sum is

$$\frac{h}{9}\sum_{j=1}^{n}\big(f(a+(j-\tfrac{1}{2}-\tfrac{1}{2}(3/5)^{1/2})h),f(a+(j-\tfrac{1}{2})h),$$
$$f(a+(j-\tfrac{1}{2}+\tfrac{1}{2}(3/5)^{1/2})h)\big)$$
$$\cdot\begin{pmatrix}\tfrac{3}{4}(1+(5/3)^{1/2}) & 1 & \tfrac{3}{4}(1-(5/3)^{1/2})\\ 0 & 4 & 0\\ \tfrac{3}{4}(1-(5/3)^{1/2}) & 1 & \tfrac{3}{4}(1+(5/3)^{1/2})\end{pmatrix}\begin{pmatrix}g(a+(j-1)h)\\ g(a+(j-\tfrac{1}{2})h)\\ g(a+jh)\end{pmatrix},$$

where $h = (b-a)/n$. Note that if $g(x) \equiv 1$, the rule reduces to the regular three-point Gauss-Legendre rule, while if $f(x) \equiv 1$, it reduces to the regular Simpson's rule. The procedure was proposed and discussed by Boland and Duris in [1].

The subroutine was written in Fortran using double precision arithmetic and was checked on a IBM 360 Model 50. The calling parameters for the routine are as follows. *A* is the name for the lower limit of integration and *B* is the name for the upper limit. *N* is the number of times the formula is to be compounded. The basic interval $[A, B]$ is subdivided into *N* subintervals each of length $(B-A)/N$ and the rule is applied to each subinterval. *FN* and *GN* are names of double precision *FUNCTION* subprograms which evaluate the functions $f(x)$ and $g(x)$, respectively. These are to be supplied by the user. The result is stored in *VINT*.

There are four machine dependent constants. These are:

(i) $\frac{3}{4}(1+(5/3)^{1/2}) \approx 1.718245836551854$,
(ii) $\frac{3}{4}(1-(5/3)^{1/2}) \approx -0.2182458365518542$,
(iii) $\frac{1}{2}(1-(3/5)^{1/2}) \approx 0.1127016653792583$, and
(iv) $\frac{1}{2}(1+(3/5)^{1/2}) \approx 0.8872983346207417$.

The first constant is assigned to $AM(1, 1)$ and $AM(2, 3)$, the second to $AM(1, 3)$ and $AM(2, 1)$, while the third and fourth are used in the calculation of $X(1)$ and $X(2)$, respectively.

References

1. Boland, W.R., and Duris, C.S. Product type quadrature formulas. *BIT 11*, 2 (1971), 139–158.

Algorithm

```
      SUBRØUTINE  P3PGS ( A, B, N, FN, GN, VINT)
C
C THIS SUBRØUTINE USES THE PRØDUCT TYPE THREE-PØINT GAUSS-
C LEGENDRE-SIMPSØN RULE CØMPØUNDED N TIMES TØ APPRØXIMATE
C THE INTEGRAL FRØM A TØ B ØF THE FUNCTION FN(X) * GN(X).
C FN AND GN ARE FUNCTIØN SUBPRØGRAMS WHICH MUST BE SUPPLIED
C BY THE USER.  THE RESULT IS STØRED IN VINT.
C
      DØUBLE PRECISIØN A, AG, AM(2,3), B, F(2), FN, G(3),
     *                  GN, H, VINT, X(2) , Y(2), DBLE
      DATA AM(1,1), AM(2,3) /2 * 1.718245836551854D0/,
     *     AM(1,2), AM(2,2) /2 * 1.D0/, AM(1,3), AM(2,1)
     *     /2 * -.2182458365518542D0/
      H = (B - A) / DBLE(FLØAT(N))
      X(1) = A + .1127016653792583D0 * H
      X(2) = A + .8872983346207417D0 * H
      Y(1) = A + H / 2.D0
      Y(2) = A + H
      VINT = 0.D0
      G(3) = GN(A)
      DØ 3 I = 1, N
        AG = FN(Y(1))
        G(1) = G(3)
        DØ 1 J = 1, 2
          F(J) = FN(X(J))
          G(J+1) = GN(Y(J))
          X(J) = X(J) + H
1         Y(J) = Y(J) + H
        VINT = VINT + AG * 4.D0 * G(2)
        DØ 3 J = 1,2
          AG = 0.D0
          DØ 2 K = 1, 3
2           AG = AG + AM(J,K) * G(K)
3         VINT = VINT + F(J) * AG
      VINT = H * VINT / 9.D0
      RETURN
      END
```

Algorithm 440

A Multidimensional Monte Carlo Quadrature with Adaptive Stratified Sampling [Dl]

L.J. Gallaher (Recd. 10 Dec. 1970, 20 July 1971)
Rich Electronic Computer Center, Georgia Institute of Technology, Atlanta, GA 30319

Key Words and Phrases: Monte Carlo quadrature, stratified sampling, adaptive quadrature, sequential stratification
CR Categories: 5.16, 5.5
Language: Algol

Description

This procedure evaluates the n-dimensional integral

$$\int_{V(a,b)} v(\mathrm{x})\,d\mathrm{x} = \int_{a_1}^{b_1}\int_{a_2}^{b_2}\cdots\int_{a_n}^{b_n} v(x_1, x_2, \cdots, x_n)\,dx_n \cdots dx_2\,dx_1$$

by the Monte Carlo method. The variance reduction scheme used here is a form of stratified sampling.

The advantages of stratified sampling are well known [1], and the concept of optimum stratification is discussed in most text books on Monte Carlo methods [2, 3, 4]. The advantages of adaptive quadrature are also well known, and many such algorithms have been published in Communications and elsewhere [5, 6, 7]. Combining adaptive quadrature with stratified sampling is a straightforward process [8, 9].

The workings of this procedure are somewhat similar to Algorithm 303 [6]. Algorithm 303 is one-dimensional, and while it can be used for multidimension integrals by recursive calls, for more than approximately six dimensions the number of evaluations of the integrand becomes intolerable. The goal of the algorithm given here is to try to overcome this defect of Algorithm 303 and other algorithms like it.

The procedure works as follows:

1. A set of samples is taken, uniformly stratified throughout the entire volume being integrated.
2. Based on the variance in these samples, a decision is made as to whether more samples are needed.
3. If more samples are needed, the volume is cut in half and the entire procedure (but with fewer samples) is repeated on each half, recursively, the halvings being repeated as required. The choice of axis for the halving is based on samples of the gradient.

The result of this process is that the overall stratification is not uniform, but approaches optimum as more and more samples are taken, since more halvings (thus more samples) are taken in the regions of high variance.

A certain amount of caution must be used in the choice of the input parameter m ($m + n$ is the number of samples taken initially). If the function being integrated is reasonably smooth, relatively low values of m (say 5 to 10) are satisfactory. If $v(\mathrm{x})$ is known to have sharp peaks, ridges, valleys, or pits, then large values of m will be necessary in order to avoid missing these high and low spots. A rough rule is that m should be inversely proportional to the error tolerance and proportional to the logarithm of volume of anomalous regions. If V_A is the fractional volume of the anomalous regions and E_r is the relative error tolerance, then the empirical rule $m \gtrsim (-2\ln(V_A))/E_r$ has proved satisfactory. For this quadrature algorithm tc be useful, the results should be insensitive to the users choice of m, and this has been observed provided m is not chosen too small. (This difficulty about the occasional need to choose m shrewdly is characteristic of all adaptive quadrature schemes, whether Monte Carlo or "exact" methods such as Romberg, Simpson, or others.)

As a test of this procedure, 100 evaluations were made of the volume of 1/32 of a hypersphere in five dimensions (in rectangular coordinates), i.e.

$$\int_0^R\int_0^R\int_0^R\int_0^R \left\{\begin{array}{l}\text{if } \sum\limits_{1\le i\le 4} x_i^2 \ge R^2 \text{ then } 0\\ \text{else } (R^2 - \sum\limits_{1\le i\le 4} x_i^2)^{1/2}\end{array}\right\} dx_1\,dx_2\,dx_3\,dx_4\,,$$

with 3% accuracy requested. A histogram is given below of the values obtained.

Number of occurrences		4		0		8		14		18		16		20		14		5		1	
I_{obs}/I_{exact}	0.94		0.95		0.96		0.97		0.98		0.99		1.00		1.01		1.02		1.03		1.04

Here I_{obs} is the value observed, I_{exact} is the correct value. The initial value of m was 120, and the average number of function evaluations per integral was 1427. The standard error for the 100 evaluations was approximately 2%. For corresponding accuracy, about 4.5 times as many samples would have been needed by unstratified uniform sampling.

Finally it should be pointed out that the results given by adaptive stratification are not entirely unbiased in the usual sense of the Monte Carlo method. There is, in fact, a biasing in favor of regions having low values of the magnitude of the gradient. However, this bias should normally be expected to be much smaller than the requested error tolerance.

Acceptable random number generators for this algorithm may be found in [10].

References

1. Cochran, William G. *Sampling Techniques*. Wiley, New York, 1953 (2nd ed. 1963).
2. Kahn, Herman. Applications of Monte Carlo. RM-1237-AEC, Rand Corp., Santa Monica, Calif., Apr. 1954 (revised Apr. 1956).
3. Hammersley, J.M., and Handscomb, D.C. *Monte Carlo Methods*. J. Wiley, New York, 1964.
4. Spanier, Jerome, and Gedbard, Ely M. *Monte Carlo Principles and Neutron Transport Problems*. Addison-Wesley, Reading, Mass., 1969.
5. McKeeman, William M. Algorithm 145: Adaptive numerical integration by Simpson's rule. *Comm. ACM 5* (Dec. 1962), 604.
6. Gallaher, L.J. Algorithm 303: An adaptive quadrature procedure with random panel sizes. *Comm. ACM 10* (June 1967), 373-374.
7. Lyness, J.N. Algorithm 379: SQUANK (Simpson quadrature used adaptively—noise killed). *Comm. ACM 13* (Apr. 1970),

260–263.

8. Halton, J.H., and Zeidman, E.A. Monte Carlo integration and sequential stratification. Comput. Sci. Tech. Rep. 13, U. of Wisconsin, Madison, Wis. 1968.

9. Zeidman, E.A. The Evaluation of multidimensional integrals by sequential stratification. (to be published).

10. Halton, John H. A retrospective and prospective survey of the Monte Carlo method. *SIAM Rev. 12*, 1 (Jan. 1970), 1–63.

Algorithm

```
real procedure quadmc (n, a, x, b, vx, esq, m, Vab, rn);
  value n, esq, m, Vab;
  integer n, m; real vx, esq, Vab, rn;
  array a, x, b;
comment The procedure parameters are:
  n – number of dimensions, n ≥ 1
  a – array of n lower bounds
  x – array of n position coordinates of which v(x1 , x2 , ... , xn) is
    a function, x is called by name
  b – array of n upper bounds (it is not required that bi > ai)
  vx – function to be integrated, vx must be a function of the array
    x (Jensen's device) and be called by name
  esq – square of the absolute error tolerance for the quadrature
  m – the number of samples to be taken at the first level is m + n,
    m ≥ n
  Vab – volume being integrated, i.e. Vab = ∏1≤i≤n | (bi − ai) |
  rn – procedure to give a new random number uniform on the
    open interval zero to one (0 < rn < 1) each time referenced,
    called by name.
  All of these parameters are input parameters to be supplied by
    the user.
  Some of the local variables of this procedure are:
  vbar – average value of v(x) for m + n samples, i.e.
```

$$\bar{v} \equiv \frac{1}{m+n} \sum_{1 \leq i \leq m+n} v(\mathbf{x}_i)$$

```
  vsqbar – average value of v(x)² for m + n samples, i.e.
```

$$\overline{v^2} \equiv \frac{1}{m+n} \sum_{1 \leq i \leq m+n} v(\mathbf{x}_i)^2$$

```
  ssq – the square of the standard error of the mean (of the integral)
    for m + n samples, i.e.
```

$$\sigma^2 = \frac{(\overline{v^2} - \bar{v}^2)}{(m+n-1)} V_{ab}^2$$

```
  vi – value of v(x) at ith sample, i.e. v(xi)
  vip – a value of v(x) such that 2 | vip − vi | is a sample of the
    magnitude of the ith component of the average normalized
    gradient, 1 ≤ i ≤ n
  it – vector of shuffled integers 1 to m
  j – array of n different vectors of shuffled integers 1 to m used in
    constructing the (uniform) stratification
  cl – point on the lth axis that divides the volume of integration in
    half for the next recursive level, i.e. cl = (b[l] − a[l])/2,
  l – index of the axis having the largest in magnitude sample of
    the component of the average normalized gradient.
  end of comment;
begin
  integer l; real vbar, ssq;
  if m < n then m := n;
  begin
    real gm, vi, vip, vsqbar;
    integer itemp, ir, k, i;
    array h[1:n];
    integer array j[1:n, 1:m], it[1:m];
    for i := 1 step 1 until m do it[i] := i;
    for k := 1 step 1 until n do
    begin
      h[k] := (b[k] − a[k])/m;
      for i := 1 step 1 until m do
      begin
        ir := entier (rn × m) + 1;
        comment 0 < rn < 1;
        itemp := it[i]; it[i] := it[ir]; it[ir] := itemp;
      end;
      for i := 1 step 1 until m do j[k, i] := it[i];
    end;
    l := 1;
    vsqbar := vbar := gm := 0;
    for i := 1 step 1 until m do
    begin
      for k := 1 step 1 until n do
      x[k] := a[k] + (j[k, i] − rn) × h[k];
      vi := vx;
      vbar := vbar + vi;
      vsqbar := vsqbar + vi ↑ 2;
      if i ≤ n then
      begin
        comment Sample the gradients;
        x[i] := x[i] + abs(b[i] − a[i])/2×
        (if x[i] < (b[i] + a[i])/2 then 1 else −1);
        vip := vx;
        vbar := vbar + vip;
        vsqbar := vsqbar + vip ↑ 2;
        if gm < abs(vip − vi) then
        begin
          l := i; gm := abs(vip − vi);
        end;
      end;
    end;
    vbar := vbar/(m + n);
    vsqbar := vsqbar/(m+n);
    ssq := Vab ↑ 2 × (vsqbar − vbar ↑ 2)/(m+n−1);
  end;
  if ssq ≤ 2 × esq then quadmc := vbar × Vab else
  begin
    real temp, cl, al, bl;
    m := m × 0.707;
    if m < ssq/esq then m := ssq/esq;
    comment The author is indebted to the referee's
      discussions pointing out the significance of maintaining
      m ≳ ssq/esq;
    esq := esq × ssq/(ssq − esq);
    al := a[l]; bl := b[l];
    b[l] := cl := (bl + al)/2;
    temp := quadmc(n, a, x, b, vx, esq/2, m, Vab/2, rn);
    b[l] := bl; a[l] := cl;
    temp := quadmc(n, a, x, b, vx, esq/2, m, Vab/2, rn) + temp;
    a[l] := al;
    quadmc := (temp × ssq + esq × vbar × Vab)/(ssq+esq);
  end;
end of quadmc
```

Algorithm 441

Random Deviates from the Dipole Distribution [G5]

Robert E. Knop [Recd. 12 Jan. 1971, 7 May 1971, 23 Aug. 1971, and 8 Mar. 1972]
Department of Physics, The Florida State University, Tallahassee, FL 32306

Key Words and Phrases: random number, probability density, probability distribution, Dipole distribution, Cauchy distribution, simulation, Monte Carlo
CR Categories: 3.17, 5.5
Language: Fortran

Description

The function subprogram *DIPOLE* returns a random deviate $-\infty < z < \infty$ sampled from the two parameter ($R^2 < 1$, α arbitrary) family of density functions:

$$f(z) = 1/(\pi(1+z^2))$$

$$+ R^2 \times ((1-z^2)\times cos(2\alpha) + 2\times z \times sin(2\alpha))/(\pi\times(1+z^2)^2)$$

The cumulative distribution function is:

$$F(z) = (1/2) + (1/\pi)\times tan^{-1}(z)$$

$$+ R^2\times(z\times co(s(2\alpha - sin(2\alpha))/(\pi\times(1+z^2))$$

Densities of this type commonly occur in the analysis of resonant scattering of elementary particles. We note that when $R = 0$ we have the Cauchy [1] or Breit-Wigner [2] density. When $R = 1$ and $\alpha = 0$ we have the single channel dipole density.[1] The dipole density with free parameters has been proposed to describe multichannel resonant scattering [3].

The algorithm begins by sampling the random vector (x, y) from a density uniform over the unit disk. The center of the unit disk is then displaced from the origin by the transformations $u = x + R\times cos(\alpha)$ and $v = y + R\times sin(\alpha)$. Letting $u = r\times cos(\theta)$ and $v = r\times sin(\theta)$ we can find the marginal density of θ:

$$f(\theta) = 1/(2\pi) \times \left(\int_0^{r_+^2} ds + \int_0^{r_-^2} ds\right)$$

where the limits of integration for r are given by:

$$r_\pm(\theta) = R\times cos(\theta-\alpha) \pm (1-R^2\times sin^2(\theta-\alpha))^{\frac{1}{2}}.$$

The marginal density of θ is thus:

$$f(\theta) = (1 + R^2cos(2\times(\theta-\alpha)))/\pi$$

for $-\pi/2 < \theta < \pi/2$. The transformation $z = tan(\theta) = v/u$ then yields the dipole density function. Other densities which could be easily sampled by computing rational functions of u and v are suggested by transformations such as $z = tan^2(\theta)$, $sin^2(\theta)$, $sin(2\times\theta)$, or $1/|sin(2\times\theta)| - 1$.

Function *DIPOLE* has two arguments which must be calculated by the calling program, $A = R\times cos(\alpha)$ and $B = R\times sin(\alpha)$. *DIPOLE* calls the function $R11(D)$ which must return a random deviate from the uniform distribution over the interval $(-1,1)$. D represents a dummy argument.

The author wishes to express his gratitude to Prof. R.G. Glasser of the University of Maryland for comments concerning this algorithm.

References

1. von Neumann, J. Various techniques used in connection with random digits. In Nat. Bur. Standards Appl. Math. Ser. 12, U.S. Gov. Print. Off., Washington, D.C., 1951, p. 36.
2. Goldberger, M.L., and Watson, K.M. *Collision Theory*. J Wiley, New York, 1964, Chap. 8.
3. Rebbi, C., and Slansky, R. Doubled resonances and unitarity. *Phys. Rev. 185*, 1838 (1969).

Algorithm

```
      FUNCTIØN DIPØLE(A,B)
10    X = R11(D)
      Y = R11(D)
      IF(1.0-X*X-Y*Y) 10,10,20
20    DIPØLE = (Y+B)/(X+A)
      RETURN
      END
```

This work was supported in part by the U.S. Atomic Energy Commission.

[1] The density is named after the analytic property of having poles of order 2 in the complex plane. See [2].

Algorithm 442 Normal Deviate [S14]

G.W. Hill and A.W. Davis [Recd. 20 Jan. 1971 and 2 Aug. 1971]
C.S.I.R.O. Division of Mathematical Statistics, Glen Osmond, Sth. Australia

Key Words and Phrases: normal distribution inverse, probit transform, Taylor series approximation
CR Categories: 5.12, 5.5
Language: Algol

Description

This procedure evaluates the inverse of the cumulative normal distribution, i.e. the normal deviate $u(p)$, corresponding to the probability level p, where

$$p = P(u) = \int_{-\infty}^{u} \phi(t)\, dt, \qquad \phi(t) = \frac{1}{(2\pi)^{\frac{1}{2}}} exp(-t^2/2).$$

An initial approximation to $u(p)$, such as $x(p)$, may be improved by using an expansion of $u(z)$, defined as the inverse of

$$z = p - P(x) = \int_{x}^{u} \phi(t)\, dt.$$

$u(z)$ may be developed in a Taylor series about $z=0$, where $u(0)=x$, see ref. [1],

$$u_n = x + \sum_{r=1}^{n} c_r(x) \left(\frac{z}{\phi(x)}\right)^r \bigg/ r!,$$

and

$$c_1(x)=1,\ c_2(x)=x,\ c_3(x)=2x^2+1,\ c_4(x)=6x^3+7x,$$
$$c_{r+1}(x)=(rx+d/dx)c_r(x).$$

An error $\epsilon(x)$ in the initial approximation, $u_0=x(p)$, entails an error $\epsilon_n(x)$ in u_n of the order of $\epsilon^{n+1}c_{n+1}(x)/(n+1)$! In order to minimize the maximum relative error $R_n = max \mid \epsilon_n/u_n \mid$ in the result obtained from n terms of the Taylor series, several sets of coefficients in an initial rational approximation styled after Hastings [2]

$$x(p) = s - \frac{a + bs + cs^2}{d + es + fs^2 + s^3}, \quad s = (-2ln(p))^{\frac{1}{2}}, \quad 0 < p < 0.5,$$

have been obtained such that $\mid [\epsilon(x)]^{n+1}c_{n+1}(x)/x \mid$ is minimax for $\mid x \mid < 40$. For odd n the minimized expression is an even function of ϵ and x, so that the relative error level may be halved when n is odd by adding $\frac{1}{2}xR_n$. The resulting precision is shown below as S_n, i.e.

$$10^{-S_n} = max \left| \frac{error(result)}{result} \right|$$

	a	b	c	d	e	f	R_n	S_n
u_1	1271.059	450.636	7.45551	500.756	750.365	110.4212	$0.622_{10}-7$	7.50
u_2	1484.397	494.327	7.61067	589.557	855.441	119.4733	$0.644_{10}-10$	10.19
u_3	1251.789	444.751	7.51005	493.187	739.156	109.3967	$0.743_{10}-13$	13.43
u_4	1637.720	494.877	7.47395	659.935	908.401	117.9407	$0.111_{10}-15$	15.95
u_5	1488.369	460.200	7.38458	598.957	831.379	110.7527	$0.940_{10}-19$	19.37
u_6	1269.225	448.718	7.49755	499.171	749.275	110.0194	$0.759_{10}-21$	21.12
u_7	1266.846	448.047	7.49101	498.003	748.189	109.8371	$0.166_{10}-23$	24.07

According to the precision required, one set of coefficients and the corresponding labeled statement, selected from the following list, should be incorporated in the procedure body as illustrated for the case of u_7.

```
u1: normdev := z + x×1.0000000311
u2: normdev := (x×z×0.5 + 1.0)×z + x
u3: normdev := (((s + 0.5)×z/3.0 + x×0.5)×z + 1.0)×z + x
               + 0.3713₁₀−13
u4: normdev := (((((s×0.75 + 0.875)×z + x)×x + 0.5)×z/3.0
               + x×0.5)×z + 1.0)×z + x
u5: normdev := ((((((s×0.6 + 1.15)×s + 0.175)×z
               + (s×0.75 + 0.875)×x)×z + s + 0.5)×z/3.0
               + x×0.5)×z + 1.0)×z + x
               + 0.42₁₀−19×x
u6: normdev := (((((((120×s + 326)×s + 127)×x×z/6
               + (24×s + 46)×s + 7)×z/40 + (0.75×s
               + 0.875)×x)×z
               + s + 0.5)×z/3.0 + x×0.5)×z + 1.0)×z + x
u7: normdev := (((((((((720 ×s + 2556)×s + 1740)×s
               + 127)×z/7
               + ((120×s + 326)×s + 127)×x)×z/6
               + (24×s  +  46)×s  +  7)×z/40  +  (0.75×s
               + 0.875)×x)×z
               + s + 0.5)×z/3.0 + x×0.5)×z + 1.0)×z + x
               + 0.832₁₀−24×x
```

Coefficients in a similar Taylor series in powers of $ln(P(x)/p)$, used in AS Algorithm *24* [*3*], require more computation than the $c_n(x)$ in these approximations.

The real procedure supplied by the user for *normal*(x,y) should return the value of the tail area to the left of x and, via the second parameter, y, should return the value of $\phi(x)$, which is often available in the process of computing the tail area. A procedure based on Algorithm 304 [4] is recommended since other algorithms such as Algorithm 209 [5] and *CDFN* [6] lose precision as p approaches their error levels (about 10^{-7}, 10^{-10} respectively), whereas Algorithm 304 maintains precision until calculations involving $\phi(x)$ exceed the capacity of floating point representation. The similar CJ Algorithm 39 [7] matches the precision of u_2 and may be readily modified to return also the value of $\phi(x)$.

The user-supplied real procedure *extreme* (p) should cater for the cases $p=0$, $p=1$, by returning suitable extreme values dependent on the floating point representation for the processor used, e.g. *extreme*$(0) = -37$ where binary exponents are ten bits, since $\phi(-37)$ is approximately $2^{-2^{10}}$ and *extreme*$(1) = +7$ for 36-bit precision, since $P(x>7)$ is approximately $1-2^{-36}$. If p lies outside (0,1) the procedure should provide a diagnostic warning and may terminate or return an extreme value such as +37 as an indication of error to the calling program.

Precision may be extended by using the D decimal digit result from one application of *normal* and the n-term Taylor series as an initial approximation for a second application, thus increasing precision to at least $(n+1)(D\text{-}log_{10}(x^2+1))$ decimal digits (as noted by the referee) or at most the precision of *normal*, e.g. $u_1(u_1)$ as in *CDFNI* [6] would have a relative error $O(10^{-14}(x^2+1))$, if not limited by the use of the lower precision *CDFN* for *normal*. For double precision calculations the more elaborate higher order terms of the Taylor series may be evaluated using single precision operations, enabling achievement of extended precision with relatively little increase in processor time. Calculations to 25 decimal digit precision and independently calculated check values to 18 significant digits [8] confirmed achievement of at least 10 significant decimal digits for Algorithm AS 24 and S_n significant digits for this procedure, except for limitations of representation of p near 1.0.

References

1. Hill, G.W., and Davis, A.W. Generalized asymptotic expansions of Cornish-Fisher type. *Ann. Math. Statist. 39*, 4 (Aug. 1968), 1264–1273.
2. Hastings, C. Jr. *Approximations for Digital Computers*. Princeton U. Press, Princeton, New Jersey, 1955, pp. 191–192.
3. Cunningham, S.W. Algorithm AS 24, From normal integral to deviate. *J.R. Statist. Soc. C. 18*, 3 (1969), 290–293.
4. Hill, I.D., and Joyce, S.A. Algorithm 304, Normal. *Comm. ACM 10*, 6 (June 1967), 374.
5. Ibbetson, D. Algorithm 209, Gauss. *Comm. ACM 6*, 10 (Oct. 1963), 616.
6. Milton, R.C., and Hotchkiss, R. Computer evaluation of the normal and inverse normal distribution functions. *Technometrics 11*, 4 (Nov. 1969), 817–822.
7. Adams, A.G. Algorithm 39. Areas under the normal curve. *Comp. J. 12*, 2 (May 1969), 197–198.
8. Strecok, A.J. The inverse of the error function. *Math. Comp. 22*, 101 (Jan. 1968), 144–158.

Algorithm

```
real procedure normdev(p, normal, extreme);
  value p; real p; real procedure normal, extreme;
  comment Input parameter p is the cumulative normal probability
     defined by

     $p = \int_{-\infty}^{u} \phi(t)\,dt, \qquad \phi(t) = \frac{1}{(2\pi)^{\frac{1}{2}}} exp(-t^2/2),$

     normal (x,y) is a procedure for evaluating the above integral
     for u=x and which returns y = φ(x), extreme(p) is a procedure
     designed to handle extreme values of p. On completion of ex-
     ecution of this procedure normdev is an approximation for u;
begin
  real s, x, z;
  x := if p > 0.5 then 1.0 − p else p;
  if x < 0.0 then normdev := extreme (p)
  else
  begin
    comment Initial rational approximation;
    s := sqrt(−2.0 × ln(x));
    x := ((−7.49101 × s − 448.047) × s − 1266.846)/
      (((s + 109.8371) × s + 748.189) × s + 498.003) + s;
    if p < 0.5 then x := −x;
    z := p − normal(x,s); z := z/s; s := x ↑ 2;
u7:
    normdev := ((((((((720 × s + 2556) × s + 1740) × s
      + 127) × z/7
      + ((120 × s + 326) × s + 127) × x) × z/6
      + (24 × s + 46) × s + 7) × z/40 + (0.75 × s + 0.875)
      × x) × z
      + s + 0.5) × z/3.0 + x × 0.5) × z + 1.0) × z + x
      + 0.832₁₀−24 × x
  end seven term Taylor series for 24 decimal precision
end normal deviate
```

Algorithm 443

Solution of the Transcendental Equation $we^w = x$ [C5]

F.N. Fritsch, R.E. Shafer, and W.P. Crowley [Recd. 11 Dec. 1970, and 15 Sept. 1971]
University of California, Lawrence Livermore Laboratory, Livermore, CA 94550

Key Words and Phrases: transcendental function evaluation, solution of transcendental equation
CR Categories: 5.12, 5.15
Language: Fortran

Description

Purpose. *WEW* solves the transcendental equation $we^w = x$ for w, given $x > 0$, by an iteration that converges much more rapidly than either Newton's method or fixed-point iteration. The user provides $x = X$. The routine returns $w = WEW$ and the last relative correction $e_n = EN$. Two versions are described here. Version A produces CDC 6600 machine accuracy (48 bits), and the relative error should be approximately e_n^3. Version B produces at least six significant figures, and the relative error should be approximately e_n^4.

Iteration. Assuming $x > 0$, we may rewrite the equation defining w as

$$w + log(w) = log(x). \qquad (1)$$

For a given approximation w_n to w, let $w_{n+1} = w_n + \delta_n$ be a much better approximation. Substitution into (1) yields

$$\delta_n + log\ (1 + \delta_n/w_n) = log\ x - log\ w_n - w_n$$
$$= z_n\text{, say.}$$

Using the approximation [1] $log\ (1 + \delta/w) \approx (\delta w + 1/6\ \delta^2)/(w^2 + 2/3\ \delta w)$ and clearing fractions yields the following quadratic equation for δ_n :

$$(2/3\ w_n + 1/6)\delta_n^2 + (w_n^2 + w_n - 2/3\ z_n w_n)\delta_n - z_n w_n^2 = 0.$$

Solving for the root that tends to zero as $z_n \to 0$ gives

$$\delta_n = \frac{2z_n w_n}{(1 + w_n - 2/3\ z_n) + ((1 + w_n + 2/3\ z_n)^2 - 2z_n)^{\frac{1}{2}}}\ .$$

This has a continued fraction expansion [3]

$$\delta_n = \cfrac{2w_n z_n}{2(1 + w_n) - \cfrac{2z_n}{2(1 + w_n + 2/3\ z_n) - \cfrac{2z_n}{2(1 + w_n + 2/3\ z_n) - \cdots}}}$$

for which the third convergent yields sufficient accuracy. If we ignore the quantity $2/3\ z_n$ in the third term, we obtain the iteration formula

$$w_{n+1} = w_n + \delta_n = w_n(1 + e_n), \qquad (2)$$

where

$$e_n = \frac{z_n}{1 + w_n} \frac{2(1 + w_n)(1 + w_n + 2/3\ z_n) - z_n}{2(1 + w_n)(1 + w_n + 2/3\ z_n) - 2z_n}\ , \qquad (3)$$

and the error term is $O(e_n^4)$. An iteration which is $O(e_n^3)$ is obtained by truncating the continued fraction at the second convergent:

$$e_n = \frac{z_n(1 + w_n + 2/3\ z_n)}{(1 + w_n)(1 + w_n + 2/3\ z_n) - 1/2\ z_n}\ . \qquad (4)$$

Initial guesses. For small values of x, the given equation has a series solution due to L. Euler [2]. A Padé rational fraction approximation to this series is

$$w_0 = \frac{x + 4/3\ x^2}{1 + 7/3\ x + 5/6\ x^2}\ . \qquad (5)$$

As computed from (5), $w_0(x) < w(x)$, good to within 5 percent if $x = 2.5$ and much better for smaller values of x. For larger values of x we may use

$$w_0 = log(x), \qquad (6)$$

which has a maximum relative error no greater than 37 percent for $x \geq e$. Version A actually switches from (5) to (6) at $x = 6.46$, the approximate location of the intersection of the two relative error curves. With these initial guesses, one iteration of (2) with e_n computed from (3) produces a maximum relative error of about 2.7×10^{-5} (see Figure 1), so that a second iteration using (4) produces CDC 6600 machine accuracy.

A much better initial guess for $x > 0.7$ can be derived by substituting $w_0 = log(x) + \delta$ into (1) to obtain

$$\delta + log\left(1 + \delta + log\left(\frac{x}{e}\right)\right) = 0.$$

Exponentiation yields

$$e^{-\delta} - 1 - \delta = log\left(\frac{x}{e}\right).$$

Using a Padé approximation to the series expansion of the left hand side, we have

$$e^{-\delta} - 1 - \delta \approx -\frac{2\delta - 1/2\ \delta^2}{1 - 1/12\ \delta^2}\ ,$$

so that (approximately)

$$\left(1/2 + 1/12\ log\left(\frac{x}{e}\right)\right)\delta^2 - 2\delta - log\left(\frac{x}{e}\right) = 0.$$

If this equation is solved approximately by the same procedure that was used to derive (3) and (4), the second convergent of the continued fraction yields the approximation

$$w_0 = log(x) - \frac{24(log^2(x) + 2log(x) - 3)}{7log^2(x) + 58log(x) + 127}\ . \qquad (7)$$

Version B switches from (5) to (7) at $x = 0.7385$. With these initial guesses, a single iteration of (2) with e_n computed from (3) yields at least six-figure accuracy (see Figure 2).

Testing. *WEW* has been tested for x in the range $0.01 \leq x \leq 1000$ against an algorithm that uses Newton's method for small to moderate values of x and fixed-point iteration for large values of x

Work performed under the auspices of the U.S. Atomic Energy Commission.

on both the CDC 6600 and 7600 computers. Measured computing times were about the same for small x ($\leq$ 1.), but the time required by *WEW* is better by a factor of 1.5 to 3.4 (depending on the required relative error) for moderate to large x. Some typical times (microseconds) obtained on the Livermore Time Sharing System are given in Table I.

Implementation Note. The section of coding preceding statement 20, labeled "set constants," provides a machine-independent means for setting the values of the constants *C*1, *C*2, *C*3, *C*4 on the first execution of *WEW*. Since the object of these algorithms is speed, it is recommended that the user compute these constants to the accuracy required for his particular machine and set them initially by means of a *DATA* statement.

Fig. 1. Relative error $| w - w_1 | / w$ with w_0 computed from (5) for $x \leq 6.46$ and from (6) for $x > 6.46$. The apparent cusp is due to the fact that the error curve $(w - w_1)$ has a zero near $x = 80.4$.

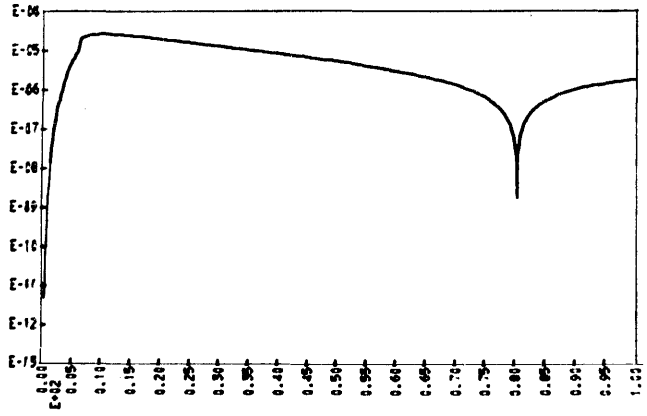


Fig. 2. Relative error $| w - w_1 | / w$ with w_0 computed from (5) for $x \leq 0.7385$ and from (7) for $x > 0.7385$. The strange appearance of the curve for small x is due to the fact that $w_0 = w$ for $x = e$ and several values of x between 0.7 and 1.1.

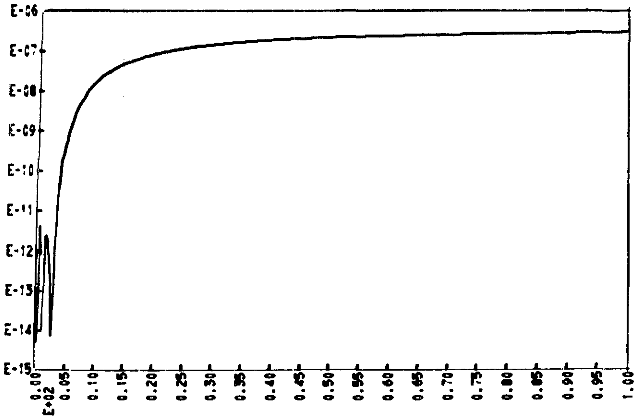


Table I. Execution Times for WEW (microsec)

	CDC 6600		CDC 7600	
	Version A	Version B	Version A	Version B
$X \leq X_c$*	118	88	25.8	17.8
$X > X_c$*	105	87	23.0	18.1

* X_c = 6.46 for Version A or 0.7385 for Version B

References

1. Abramowitz, M., and Stegun, I.A. *Handbook of Mathematical Functions.* National Bureau of Standards (AMS-55), Washington, D.C., 1964, Formula 4.1.39, p. 68.
2. Polya, G., and Szego, G. *Aufgaben und Lehrsätze aus der Analysis*, Vol. 1. Springer-Verlag, Berlin, 1954, Problem 209, p. 125.
3. Wall, H. *Analytic Theory of Continued Fractions.* Van Nostrand, New York, 1948.

Algorithms

```
      FUNCTIØN WEW (X, EN)
C
C   ITERATIVE SØLUTIØN ØF X=W*EXP(W) WHERE X IS GIVEN.   (NØVEMBER 1970)
C                                               (REVISED - SEPTEMBER 1971)
C   VERSIØN A -- CDC 6600 MACHINE ACCURACY.
C
C   INPUT PARAMETER]
C             X   ARGUMENT ØF W(X).
C
C   ØUTPUT PARAMETERS]
C           WEW   THE DESIRED SØLUTIØN.
C            EN   THE LAST RELATIVE CØRRECTIØN TØ W(X).
C
C
C   SET CØNSTANTS...
      DATA  NEWE/1/
      IF (NEWE)  10,20,10
 10   NEWE = 0
      C1=4./3.
      C2=7./3.
      C3=5./6.
      C4=2./3.
C
C   CØMPUTE INITIAL GUESS...
 20   FLØGX = ALØG(X)
      IF (X-6.46)  30,30,40
 30   WN = X*(1.+C1*X)/(1.+X*(C2+C3*X))
      ZN = FLØGX - WN - ALØG(WN)
      GØ TØ 50
 40   WN = FLØGX
      ZN = -ALØG(WN)
 50   CØNTINUE
C
C   ITERATIØN ØNE...
     TEMP = 1. + WN
      Y = 2.*TEMP*(TEMP+C4*ZN) - ZN
      WN = WN*(1. + ZN*Y/(TEMP*(Y-ZN)))
C
C   ITERATIØN TWØ...
      ZN = FLØGX - WN - ALØG(WN)
     TEMP = 1. + WN
     TEMP2 = TEMP + C4*ZN
      EN = ZN*TEMP2/(TEMP*TEMP2-.5*ZN)
      WN = WN*(1.+EN)
C
C   RETURN...
      WEW = WN
      RETURN
      END
```

```
      FUNCTIØN WEW (X, EN)
C
C   ITERATIVE SØLUTIØN ØF X=W*EXP(W) WHERE X IS GIVEN.   (NØVEMBER 1970)
C                                               (REVISED - SEPTEMBER 1971)
C   VERSIØN B -- MAXIMUM RELATIVE ERRØR 3.E-7 .
C
C   INPUT PARAMETER]
C             X   ARGUMENT ØF W(X).
C
C   ØUTPUT PARAMETERS]
C           WEW   THE DESIRED SØLUTIØN.
C            EN   THE LAST RELATIVE CØRRECTIØN TØ W(X).
C
C
C   SET CØNSTANTS...
      EQUIVALENCE (F, FLØGX)
      DATA  NEWE/1/
      IF (NEWE)  10,20,10
 10   NEWE = 0
      C1=4./3.
      C2=7./3.
      C3=5./6.
      C4=2./3.
C
C   CØMPUTE INITIAL GUESS...
 20   FLØGX = ALØG(X)
      IF (X-.7385) 30,30,40
 30   WN = X*(1.+C1*X)/(1.+X*(C2+C3*X))
      GØ TØ 50
 40   WN = F - 24.*((F+2.)*F-3.)/((.7*F+58.)*F+127.)
 50   CØNTINUE
C
C   ITERATIØN ØNE...
      ZN = FLØGX - WN - ALØG(WN)
     TEMP = 1. + WN
      Y = 2.*TEMP*(TEMP+C4*ZN) - ZN
      EN = ZN*Y/(TEMP*(Y-ZN))
      WN = WN*(1.+EN)
C
C   RETURN...
      WEW = WN
      RETURN
      END
```

Remark on Algorithm 443 [C5]

Solution of the Transcendental Equation $we^w = x$
[F.N. Fritsch, R.E. Shafer, and W.P. Crowley, *Comm. ACM 16* (Feb. 1973), 123–124]

Bo Einarsson [Recd. 5 Mar. 1973 and 4 June 1973]
Research Institute of National Defense, Box 98,
S-147 00 Tumba, Sweden

This algorithm contains a violation of the Fortran standard as defined in [1]. According to Section 10.2.6 of the standard, certain variables in a subprogram will be undefined at the execution of the RETURN statement, if they are not in a common block. This applies to the section in Algorithm 443 labeled "set constants" and commented in the Implementation Note. The IBM FORTRAN IV H Extended Compiler (Program Product) makes use of the standard in such a way that the variable NEWE does not have the value zero at a reentry to the subprogram, so that the variable NEWE does not fill its purpose. On the other hand this compiler performs the divisions and stores the quotients, so that no divisions are needed at the *execution* of the subprogram. The IBM FORTRAN IV G Compiler performs as the authors of Algorithm 443 take for granted. Other optimizing compilers may have the value of NEWE as zero at reentry but have undefined values of $C1$, $C2$, $C3$, and $C4$. In that case the subprogram would produce erroneous results.

The remark above is similar to the third paragraph of Remark on Algorithm 352 [2], where the consequences of Section 10.2.5 of the standard are discussed.

The problem with the local variables can be evaded without loss of computing efficiency by replacing statement 30 with

```
30      WN = X * (3. +4. * X) / (3. +X *(7. +2.5 *X)),
```

replacing C4 * ZN wherever it appears with ZN/1.5, and finally deleting the section "set constants" and the "Implementation Note". In version B the statement EQUIVALENCE (F, FLOGX) must be kept.

I have also certified the routine (version B) by testing it in single precision on an IBM 360/75 by performing some statistics on $R(x) = (we^w - x)/x$. The first test used $x = 0.01\ (0.01)\ 10.00$ and the second a thousand x values from a normal random distribution with mean value zero and variance 1, but if the obtained random value x was nonpositive, a new value of x was computed. The values of R were calculated in double precision.

The following results were obtained:

Test	Mean value of R	Standard deviation of R	Maximal value of $\mid R \mid$
Linear	$-1.7 \cdot 10^{-6}$	$1.2 \cdot 10^{-6}$	$4.2 \cdot 10^{-6}$
Random	$-0.4 \cdot 10^{-6}$	$0.5 \cdot 10^{-6}$	$3.3 \cdot 10^{-6}$

Since the relative error in a single precision value on IBM 360 may be as high as $0.5 \cdot 10^{-6}$, the above results appear reasonable.

References

1. American National Standard FORTRAN, ANSI X3.9—1966. American National Standards Institute, New York, 1966.
2. Sale, A.H.J. Remark on Algorithm 352. *Comm. ACM 13* (Dec. 1970), 750.

Algorithm 444

An Algorithm for Extracting Phrases in a Space-Optimal Fashion [Z]

R.A. Wagner [Recd. 5 Mar. 1971 and 30 Aug. 1971]
Department of Systems and Information Science,
Vanderbilt University, Nashville, TN 37203

Key Words and Phrases: information retrieval, coding, text compression

CR Categories: 3.70, 5.6

Language: PL/I

Description

Introduction. The algorithm *PARSE* computes and prints a minimum-space form of a textual message, *MS*. The minimization is performed over all possible "parses" of *MS* into sequences of phrase references and character strings. Each phrase reference represents one of a finite collection, *P*, of phrases. The collection, *P*, must be selected before *PARSE* is applied.

Assumptions and requirements. PARSE assumes that the unit of storage is the byte, defined such that one byte can hold either a single character of text or an integer i in the range $0 \leq i < W$. (For IBM 360 equipment, $W = 256 = 2{**}8$). *PARSE* also assumes that the number of different phrases in the collection *P* is no larger than *W**PHC*, and that each message to be parsed contains fewer than *W**CHC* characters of text. The parameter values $CHC = PHC = 1$ appear appropriate on IBM 360 equipment, when *PARSE* is applied to short messages, such as compiler error messages.

PARSE requires two arguments. The first is the message to be parsed; the second is the table of common phrases which may be used in the parse.

PARSE assumes that an external procedure *HASH* is present; *HASH(MS,I,K)* is defined as follows: Let $H_1, H_2, \ldots, H_m$ be a sequence of indices such that among them they exhaust all entries $P(H_i)$ such that

$$SUBSTR(MS,I,3) = SUBSTR(P(H_i),1,3).$$

(That is, the H_i's include indices for every phrase $P(H_i)$ which agrees with characters I, $I+1$, and $I+2$ of the given message. Other indices may occur among the H_i's, as well.) Then $HASH(MS,I,0) = H_1$, $HASH(MS,I,H_j) = H_{j+1}$, and $HASH(MS,I,H_m) = 0$.

A "hash table" procedure can easily be modified to yield this performance; an equally useable, although slower version returns $MOD(K+1, M+1)$ on every call. A procedure *HASH* is included below.

Methods. The method used to determine which phrases to extract from the given message is described in [1]. The resulting parsed message requires least space, assuming that messages are storable only as described in [1]—that is, as sequences of

C ⟨number⟩ ⟨character string⟩
| *P* ⟨number⟩

representing a literal string of characters, and a reference to a common phrase, respectively.

During the course of the computation, arrays *G* and *H* are filled with values of functions *g* and *h*, respectively, as defined in [1]. Just before label *BUILD* is reached,

$H(I)$ = length of the best parse of *SUBSTR(MS,I)*, and
$G(I)$ = length of the best parse of *SUBSTR(MS,I)* among those parses beginning with a character string,

both for $I = 1, \ldots, LENGTH(MS)$.

Internally, *PARSE* uses a single array, *Z*, paralleling the function arrays *G* and *H*, to retain the information needed for re-constructing the parsed form of the message.

$Z(I) = K$, if $G(I) > H(I)$, where *K* is the number of the "best" common phrase matching *MS* at *I*, or
$= J$, if $G(I) = H(I)$. ($G(I) < H(I)$ is impossible.)

J gives the index of the end (plus one) of the character string starting at *I*. In this case, the best parse at *I* begins with this character string. *J* satisfies: $G(J) > H(J)$ and for all k, $I \leq k < J$, $G(k) = H(k)$.

Results: To make the printed form of the parsed message more intelligible, *PARSE* prints:
'*C* ⟨number⟩' as '#*ddd*'
'*P* ⟨number⟩' as '%*ddd*'
where "*ddd*" is the 3-digit decimal representation of ⟨number⟩ + 1. In practice, a number representing a character count or phrase index can be stored as an integer, in place of *CHC* or *PHC* characters respectively. Thus, the character string '*ABC*' would be stored as '*C*$\underline{2}$*ABC*', where $\underline{2}$ is a *CHC*-byte integer whose value is 2. The same string would be printed by the *PARSE* algorithm as '#003*ABC*'.

The program *PARSE* returns the number of bytes needed to store *MS*, given the particular set of extractable phrases in *P*.

A sample driver, two sample input streams and associated output follow the procedures *PARSE* and *HASH*.

References

1. Wagner, R.A. Common phrases and minimum-space text storage. *Comm. ACM 16*, 3 (Mar. 1973), 148–152.
2. Bell, James R. The quadratic quotient method; a hash code eliminating secondary clustering. *Comm. ACM 13*, 2 (Feb. 1970), 107–109.

Algorithm (Figures 1–6 follow.)

Fig. 1. The PARSE Algorithm.

```
PARSE:    PROC(MS,P) RETURNS(FIXED BINARY);
          DCL  (MS,P(*)) CHAR(*) VARYING;
          DCL  N;
          DCL  HASH RETURNS(FIXED BINARY);
          DCL (CHC,   /* BYTES PER CHARACTER-COUNT  */
               PHC)   /* BYTES PER PHRASE-INDEX  */
               STATIC EXTERNAL FIXED BINARY;

          N=LENGTH(MS);
          BEGIN;
               DCL(G,H,Z)(N+1) FIXED BINARY;
               DCL(I,J,K,L,T)  FIXED BINARY;

               G(N+1)=3; H(N+1)=1; J,Z(N+1)=N+1;
          MSGP:     DO I=N BY -1 TO 1;
                    K=HASH(MS,I,0B);
                    H(I), G(I) = MIN( G(I+1)+1, H(I+1)+CHC+2 );
                    Z(I)=J;
               /* J HOLDS INDEX OF END+1 OF NEXT CHAR-STRING */
               M1:  DO WHILE (K>0);
                         L=LENGTH(P(K));
                         IF L ¬> N-I+1 THEN
                         IF L < N THEN
                         IF SUBSTR(MS,I,L)=P(K) THEN DO;
                              T=H(I+L)+PHC+1;
                              IF H(I)>T THEN DO;
                                   H(I)=T; Z(I)=K; J=I;
                                   END;
                              END;
                         K=HASH(MS,I,K);
                         END M1;
                    END MSGP;

               PUT SKIP EDIT(H(1),N+3,':  ')(2 F(4),A);
               I=1; GOTO B1;
          BUILD:
               IF H(I)<G(I) THEN DO;
                    PUT EDIT('%', Z(I))(A,P'999');
                    I=I+LENGTH(P( Z(I)));
                    END;
               ELSE DO;
                    J=Z(I)-I;
                    PUT EDIT('#',J,SUBSTR(MS,I,J))(A,P'999',A);
                    I=Z(I);
                    END;
          B1:  IF I¬>N THEN GOTO BUILD;
               PUT EDIT('.')(A);
               RETURN(H(1));
          END PARSE;
```

Fig. 2. An acceptable HASH procedure.

```
HASH:     PROC(MS,I,K) RETURNS(FIXED BINARY);
     DCL  MS CHAR(*), J FIXED BINARY(31,0),
          (HT (0:200)INIT((201)0),
                KJ, HP INIT(197),
                HX,HY,HZ) FIXED BINARY STATIC;
          DCL (CHC,    /* BYTES PER CHARACTER-COUNT   */
               PHC)    /* BYTES PER PHRASE-INDEX  */
               STATIC EXTERNAL FIXED BINARY;

          CALL HCMN(K);
          RETURN(HT(HZ));

     HCMN:     PROC(K);
          IF K = 0 THEN
               IF LENGTH(MS)-I < PHC+1 THEN HZ=-1;
               ELSE DO;
                    UNSPEC(J)=UNSPEC(SUBSTR(MS,I,PHC+2));
                    HZ=MOD(J,HP);
                    HY=J/HP;
                    HX=0;
                    END;
          ELSE DO;
               HX=MOD(HX+HY,HP);
               HZ=MOD(HX+HZ,HP);
               END;
          HZ=HZ+1;
          RETURN;
          END HCMN;

ENTER:    ENTRY(MS,I,K);
          IF LENGTH(MS) < PHC+2 THEN RETURN;
          KJ=0;
     E1:  CALL HCMN(KJ);
          KJ=HT(HZ);
          IF KJ > 0 THEN GOTO E1;
          HT(HZ)=K;
          RETURN;
          END HASH;
```

Fig. 3. A driver for the PARSE procedure.

```
DRIVER:   PROC OPTIONS(MAIN);
          DCL  MS CHAR(256) VARYING;
          DCL NP,M;
          DCL (HASH RETURNS(FIXED BINARY), ENTER)
               ENTRY(CHAR(256) VARYING, FIXED BINARY, FIXED BINARY);
          DCL PARSE RETURNS(FIXED BINARY);
          DCL (CHC,  /* BYTES PER CHARACTER-COUNT   */
               PHC)  /* BYTES PER PHRASE-INDEX   */
               STATIC EXTERNAL FIXED BINARY;

          CHC,PHC=1;    /* COUNT/INDEX SIZE=1 BYTE  */
          GET SKIP LIST(NP,M);
          BEGIN;
               DCL P(NP) CHAR(M) VARYING;
               DCL NB,NA,I,J;

               NB,NA=0;
               DO I=1 TO NP;
               GET SKIP LIST(P(I));
               CALL ENTER(P(I),1,I);
               END;

               PUT PAGE LIST('PHRASES, AND THEIR PARSED FORMS');
               DO I=1 TO NP;
                    PUT SKIP(2) EDIT(I,'  ''' || P(I) || '''')
                         (F(4),A);
                    NA=NA+PARSE(P(I),P);
                    END;

               PUT PAGE LIST('MESSAGES:');
          L1:  GET SKIP LIST(MS);
               PUT SKIP(2) LIST( '''' || MS || '''' );
               IF MS='' THEN GOTO L2;
               NB=NB+LENGTH(MS)+CHC+2;
                         /* ALLOW FOR STRING-OVERHEAD + END MARK  */
                    NA=NA+PARSE(MS,P);
                    GOTO L1;

          L2:  PUT SKIP EDIT('FINAL STATISTICS:',
               'WITHOUT PHRASE EXTRACTION:',NB,
               'AFTER PHRASE EXTRACTION:',NA,
               'SAVING:',NB-NA,
               '  (',(NB-NA)*100/NB,'%)')
                    (A,3(SKIP,A,F(5)),A,F(5,1),A);
               RETURN;
               END DRIVER;
```

Fig. 4. Sample input files.
(a) Two phrases, four messages. Illustrates heavily overlapping phrases.
(b) Five phrases, 23 messages. These messages are the first 23 numbered error messages from the syntactic analysis section of the PL/C compiler.

```
A
 CMS03 LISTING OF INPUT STREAM
  00001
  00002      2,10
  00003     'AAAAA'
  00004     'AAAAAAA'
  00005     'AAAAAAAAAA'
  00006     'AAAAAAAAAAAA'
  00007     'AAAAAAAAAAAAAA'
  00008     'AAAAAAAAAAAAAAA'
  00009     ''
```

```
B
 CMS03 LISTING OF INPUT STREAM
  00001
  00002            5,20
  00003     'EXTRA '
  00004     'MISSING '
  00005            'IMPROPER '
  00006     'SEMI-COLON'
  00007     'EXPRESSION'
  00008     'EXTRA ('
  00009     'MISSING ('
  00010     'EXTRA )'
  00011     'MISSING )'
  00012     'EXTRA COMMA'
  00013     'MISSING COMMA'
  00014     'EXTRA SEMI-COLON'
  00015     'MISSING SEMI-COLON'
  00016     'MISSING :'
  00017     'MISSING ='
  00018     'IMPROPER *'
  00019     'MISSING *'
  00020     'EXTRA END'
  00021            'MISSING END'
  00022            'MISSING KEYWORD'
  00023            'INCOMPLETE EXPRESSION'
  00024            'MISSING EXPRESSION'
  00025            'MISSING VARIABLE'
  00026            'MISSING ARGUMENT, 1 SUPPLIED'
  00027            'EMPTY LIST'
  00028            'IMPROPER NOT'
  00029            'IMPROPER ELEMENT'
  00030            'UNTRANSLATABLE STATEMENT'
  00031     ''
```

Fig. 5. Result of applying DRIVER to the cards listed in Figure 4(a). Note that phrase 2 is itself reduced in size by PARSE, while each of the messages are reduced to strings of phrase references alone.

```
PHRASES, AND THEIR PARSED FORMS

   1   'AAAAA'
   8    8:  #005AAAAA.

   2   'AAAAAAA'
   7   10:  #002AA%001.


MESSAGES:

'AAAAAAAAAA'
   5  13:  %001%001.

'AAAAAAAAAAAA'
   5  15:  %001%002.

'AAAAAAAAAAAAAA'
   5  17:  %002%002.

'AAAAAAAAAAAAAAA'
   7  18:  %001%001%001.

''
FINAL STATISTICS:
WITHOUT PHRASE EXTRACTION:   63
AFTER PHRASE EXTRACTION:   37
SAVING:   26  ( 41.3%)
```

Fig. 6. Result of applying DRIVER to the cards listed in Figure 4(b).

```
PHRASES, AND THEIR PARSED FORMS

   1   'EXTRA '
   9    9:  #006EXTRA .

   2   'MISSING '
  11   11:  #008MISSING .

   3   'IMPROPER '
  12   12:  #009IMPROPER .

   4   'SEMI-COLON'
  13   13:  #010SEMI-COLON.

   5   'EXPRESSION'
  13   13:  #010EXPRESSION.


MESSAGES:

'EXTRA ('
   6  10:  %001#001(.

'MISSING ('
   6  12:  %002#001(.

'EXTRA )'
   6  10:  %001#001).

'MISSING )'
   6  12:  %002#001).

'EXTRA COMMA'
  10  14:  %001#005COMMA.

'MISSING COMMA'
  10  16:  %002#005COMMA.

'EXTRA SEMI-COLON'
   5  19:  %001%004.

'MISSING SEMI-COLON'
   5  21:  %002%004.

'MISSING :'
   6  12:  %002#001:.

'MISSING ='
   6  12:  %002#001=.

'IMPROPER *'
   6  13:  %003#001*.

'MISSING *'
   6  12:  %002#001*.

'EXTRA END'
   8  12:  %001#003END.

'MISSING END'
   8  14:  %002#003END.

'MISSING KEYWORD'
  12  18:  %002#007KEYWORD.

'INCOMPLETE EXPRESSION'
  16  24:  #011INCOMPLETE %005.

'MISSING EXPRESSION'
   5  21:  %002%005.

'MISSING VARIABLE'
  13  19:  %002#008VARIABLE.

'MISSING ARGUMENT, 1 SUPPLIED'
  25  31:  %002#020ARGUMENT, 1 SUPPLIED.

'EMPTY LIST'
  13  13:  #010EMPTY LIST.

'IMPROPER NOT'
   8  15:  %003#003NOT.

'IMPROPER ELEMENT'
  12  19:  %003#007ELEMENT.

'UNTRANSLATABLE STATEMENT'
  27  27:  #024UNTRANSLATABLE STATEMENT.

''
FINAL STATISTICS:
WITHOUT PHRASE EXTRACTION:  376
AFTER PHRASE EXTRACTION:  283
SAVING:   93  ( 24.6%)
```

Algorithm 445

Binary Pattern Reconstruction from Projections [Z]

Shi-Kuo Chang [Recd. 4 Nov. 1970 and 12 May 1971]
School of Electrical Engineering, Cornell University
Ithaca, NY 14850.

Key Words and Phrases: pattern reconstruction, image reconstruction, data compression, picture processing
CR Categories: 3.63, 5.30
Language: Algol

Description

This procedure reconstructs a binary pattern from its horizontal and vertical projections [1]. The parameters are described as follows. *m*, *n* are the dimensions of the binary pattern *f*. *switch* is an integer variable. *fx* [1:*n*] is the projection of *f* on the horizontal axis. *fy* [1:*m*, 1] is initially set to (1, 2, . . . , *m*). *fy* [1:*m*, 2] is the projection of *f* on the vertical axis. *f* [1:*n*, 1:*m*] is the pattern to be reconstructed, initially set to 0.

The projections *fx* and *fy* are *inconsistent* if there is no pattern *f* having such projections. The pattern *f* is *unambiguous* if there is no other pattern having the same projections as *f*. Given the projections *fx* and *fy*, there are three possibilities: (1) *fx* and *fy* are inconsistent; (2) they are consistent but the pattern *f* is ambiguous; or (3) they are consistent and the pattern *f* is unambiguous.

(1) Inconsistent Projections. This procedure sets *switch* to −1 and reconstructs a pattern *f* having the correct horizontal projection *fx*. Its vertical projection will be different from *fy*.

(2) Ambiguous Pattern. This procedure sets *switch* to 0 and reconstructs a pattern *f* having projections *fx* and *fy*.

(3) Unambiguous Pattern. This procedure sets *switch* to 1 and reconstructs a pattern *f* having projections *fx* and *fy*. In this case *f* is unique.

References

1. Chang, S.-K. The reconstruction of binary patterns from their projections. *Comm. ACM 14*, 1 (Jan. 1971), 21–25.
2. Chang S.-K., and Shelton, G.L. Two algorithms for multiple-view binary pattern reconstruction. *IEEE Trans. Syst., Man, Cybern.* (Jan. 1971), 90–94.

Algorithm

```
procedure Pattern Reconstruction (switch, m, n, fx, fy, f);
  integer m, n, switch; integer array fx, fy, f;
  comment The parameters are defined as follows: switch is an
  output parameter with values −1, 0, or 1 according as the
  projections are inconsistent (switch = −1), the pattern is
  ambiguous (switch = 0), the pattern is unambiguous (switch =
  1). m is the column dimension of the binary pattern f, and n is
  the row dimension of the binary pattern f. m and n are input
  parameters. The array fx [1:n] is the projection of the binary
  pattern f on the x axis. fx is an input array. The array fy [1:m,
  1:2] contains 1, 2, . . . , m in column 1 initially, and column 2
  contains the projection of the binary pattern f on the y axis. fy
  is an input array, and it is modified by this procedure. The array
  f [1:n, 1:m] contains 0 initially and contains the reconstructed
  binary pattern finally;
begin
  integer ix, iy, j, number;
  procedure Sort;
  begin
    integer limit, ind, i;
    limit := m − 1;
S1:
    ind := 0;
    for i := 1 step 1 until limit do
    if fy [i, 2] < fy [i+1, 2] then
    begin
      integer t1, t2;
      ind := 1;
      t1 := fy [i+1, 1];  t2 := fy [i+1, 2];
      fy [i+1, 1] := fy [i, 1];
      fy [i+1, 2] := fy [i, 2];
      fy [i, 1] := t1;  fy [i, 2] := t2
    end;
    limit := limit −1;
    if (limit > 0) ∧ (ind = 1) then go to S1
  end Sort;
  procedure Merge;
  if fy [number, 2] < fy [number+1, 2] then
  begin
    integer n1, n2, t1, t2;
    n1 := number;
S2:
    if n1 > 1 then
    begin
      if fy [n1, 2] = fy [n1−1, 2] then
      begin n1 := n1 − 1;  go to S2 end
    end;
    n2 := number + 1;
S3:
    if n2 < m then
    begin
      if fy [n2+1, 2] = fy [n2, 2] then
      begin n2 := n2 + 1;  go to S3 end
    end;
S4:
    t1 := fy [n1, 1];  t2 := fy [n1, 2];
    fy [n1, 1] := fy [n2, 1];  fy [n1, 2] := fy [n2, 2];
    fy [n2, 1] := t1;  fy [n2, 2] := t2;
    if (n1 < number) ∧ (number+1 < n2) then
    begin n1 := n1 + 1;  n2 := n2 − 1;  go to S4 end
  end Merge;
  comment The procedure Sort orders fy, and the procedure Merge
    reorders fy. The main procedure now follows;
  switch := 1;
  Sort;
  for ix := 1 step 1 until n do
  begin
    number := fx [ix];
    if number > 0 then
```

Author's present address: Institute of Mathematics, Academia Sinica, 910 Nankang, Taiwan, Republic of China.

```
      begin
        for j := 1 step 1 until number do
        begin
          iy := fy [j, 1];
          fy [j, 2] := fy [j, 2] − 1;
          f [ix, iy] := 1
        end;
        comment One column of f is reconstructed;
        if number < m then
        begin
          if (switch=1) ∧ (fy[number, 2] < fy[number+1, 2])
          then switch := 0;
          comment The above condition indicates that the
            pattern f is ambiguous, and the switch is set to 0;
          Merge;
          comment fy is reordered before we start to reconstruct
            the next column;
        end
      end
    end;
    for j := 1 step 1 until m do
    if fy [j, 2] ≠ 0 then switch := −1;
    comment The above condition indicates that the projections are
      inconsistent, and the switch is set to −1;
  end Pattern Reconstruction
```

Remarks on Algorithm 445 [Z]
Binary Pattern Reconstruction from Projections [by Shi-Kuo Chang, *Comm. ACM 16* (Mar. 1973), 185–186]

John Lau [Recd. 22 July 1971]
Department of Computer Science, University of
British Columbia, Vancouver 8, B.C., Canada

Key Words and Phrases: pattern reconstruction, image reconstruction, data compression, picture processing
CR Categories: 3.63, 5.30
Language: Algol

The procedure works well for all consistent patterns, ambiguous or unambiguous. However, when *fx* and *fy* are inconsistent, the procedure can construct a pattern *f*[1:*n*, 1:*m*] with *fx* satisfied, only if all elements of *fx* have values between 0 and *m*. If any of these elements is greater than *m*, a program interrupt would usually be caused by "value of subscript outside declared bounds" when the program executes the lines

```
for j := 1 step 1 until number do
begin
  iy := fy[j, 1];
  fy[j, 2] := fy[j, 2] − 1;
  f[ix, iy] := 1
end;
```

and execution of program would then be terminated. Even if a pattern could be constructed in this case, it would not be able to satisfy *fx* entirely.

Algorithm 446 Ten Subroutines for the Manipulation of Chebyshev Series [C1]

R. Broucke [Recd. 17 May 1971 and 7 April 1972]
University of California, Los Angeles, CA 90024, and
Jet Propulsion Laboratory, Pasadena, Calif.

Key Words and Phrases: Chebyshev series, differentiation, integration, curve fitting, approximations, negative powers
CR Categories: 5.12, 5.13, 5.16
Language: Fortran

Description

Introduction. These subroutines deal with the manipulation of Chebyshev series. The operations performed are the construction of the Chebyshev approximation of functions, the evaluation of the series or their derivative, the integration or differentiation, and the construction of negative or fractional powers of such a series.

The subroutines are written in ANSI Fortran. They have been used without modification on such computers as the IBM-7094, IBM-360/91 (Fortran-IV-G compiler) and Univac 1108 (Fortran-V compiler).

The ten subroutines are considered as a single set, principally because they all use the same storage philosophy. All information is transmitted through the *CALL*-sequence rather than through the use of *COMMON* statements. Therefore, the user must provide storage for all the series in his main program, taking into account that all operations are performed in double precision. The coefficients of each series occupy a one-dimensional double-precision array according to the rules of ANSI Fortran. When several Chebyshev series are being manipulated, it is convenient to store all the series in a matrix. Each column of the matrix contains a single series, in order that the coefficients of each series occupy consecutive storage locations.

The first six subroutines contain no calls to other subroutines; in this sense they may be considered as independent. Each subroutine can be used separately.

In the present type of operations, it is extremely important to design and perform a large number of tests to certify all of the subroutines. We have tested the subroutines by generating some Chebyshev series which were published by Clenshaw [4], but we have also tested them with a number of additional methods; for instance:

a. The series for several elementary functions such as $\sin(x)$, $\cos(x)$, $\sin(2x)$, and $\cos(2x)$ have been constructed directly. These series have then been evaluated, and the values have been compared with the values of the functions.

b. The series for $\cos(2x)$ and $\sin(2x)$ have been derived from the series $\sin(x)$ and $\cos(x)$ by multiplication and addition of series.

c. The series for $\sin(x)$ and $\cos(x)$ have been derived from each other by integration and differentiation.

d. Many tests have been made by multiplying a series $f(x)$ by the series $1/f(x)$ or for instance by squaring the series for $f(x)^{\frac{1}{2}}$, or other similar operations.

The generation, evaluation and multiplication subroutines. The methods for the generation of a Chebyshev series have been taken from C.W. Clenshaw's papers [3, 4, 5]. The rule for the multiplication of Chebyshev series is also described by Clenshaw [3, p. 137], but the flowchart of our subroutine is from L. Carpenter [2].

We only consider the interval $(-1, +1)$ of the independent variable x, and we represent a truncated Chebyshev series of order n in the form:

$$f(x) = (c_0/2) + c_1T_1(x) + c_2T_2(x) + \cdots + c_nT_n(x). \tag{1}$$

We want to draw the user's attention to the fact that we use a factor $\frac{1}{2}$ in the zero-order term but not in the last term of the series. Some authors have used different conventions in relation to this factor $\frac{1}{2}$ for the first and last terms.

In the applications of the subroutines some caution is also necessary, because the independent variable x (the Chebyshev independent variable) is within the limits $(-1, +1)$. If the user's variable t (the physical independent variable) is within the limits (t_1, t_2), the conversions between t and x should be made with the linear relations

$$t = ((t_2 + t_1)/2) + ((t_2 - t_1)/2)x; \qquad x = ((2t - (t_2 + t_1))/(t_2 - t_1)). \tag{2}$$

The coefficients c_i in formula (1) are computed with the rule given by Clenshaw [4, p. 3]:

$$c_i = (2/n \sum_{j=0}^{n}{}'' f(\cos(\pi j/n)) \cos(\pi ij/n); \qquad i = 0, 1, \ldots, n. \tag{3}$$

The double accent means that the first and last terms of the sum are divided by two. It is seen that $n + 1$ special values of the function $f(x)$ are needed. In some applications, n has been as large as 1,500.

A large number of applications have shown that in most instances the user desires to construct the Chebyshev series for not just one function but for several functions simultaneously. For instance, in the study of the motion of a particle there will always be three coordinates, x_1, x_2, x_3, rather than just one. For this reason we programmed the subroutine *CHEBY* to efficiently construct several Chebyshev series simultaneously. In particular, the number of cosine calculations has been minimized. There will be only $2n$ cosine calculations, no matter how many functions are being analyzed simultaneously.

Besides the main program, the user will have to provide his own subroutine for the evaluation of the special values of the functions to be analyzed, as explained in the comments of the subroutine *CHEBY*. The user may choose any name for this subroutine; however, this name has to be transmitted through the *CALL CHEBY*-statement. This function subroutine will generally evaluate the function values either by using the appropriate formulas or by performing table lookup and interpolations if the data is only available in the form of a table with discrete points.

The subroutine *ECHEB* evaluates a Chebyshev series with the aid of Clenshaw's recurrence rule [4, p. 9]. The c_i's being the coefficients of the given series, we compute the values $b_{n+2}, b_{n+1}, b_n, \ldots, b_0$ with:

$$b_{n+2} = b_{n+1} = 0; \qquad b_i = 2xb_{i+1} - b_{i+2} + c_i, \tag{4}$$

where the subscript i runs from n to 0. The number of arithmetic operations involved is only $3n$, and the value of the function is then $f(x) = (b_0 - b_2)/2$.

The subroutine *EDCHB* evaluates the derivative of a Chebyshev series (without storing the coefficients of the differentiated series). It implements a combination of the evaluation formula (4) and the differentiation formula (6) given below.

The differentiation and integration subroutines. Clenshaw's formulas [4, p. 11] have again been used for the differentiation and integration operations. The coefficients a_i of the integrated Chebyshev series are derived from the input coefficients c_i by:

$$a_0 = 0; \quad a_n = c_{n-1}/2n; \quad a_i = (c_{i-1} - c_{i+1})/2i; \quad i = 1, 2, \ldots, n-1. \tag{5}$$

The coefficients d_i of the differentiated series are obtained by a set of recurrence equations:

$$d_n = 0; \quad d_{n-1} = 2nc_n\,; \quad d_{i-1} = d_{i+1} + 2ic_i\,; \quad i = n-1, n-2, \ldots, 1. \tag{6}$$

When using the differentiation and integration subroutines, the user should remember the relation between the differentials of t and x:

$$dt = ((t_2 - t_1)/2)\, dx = (\Delta t/2)dx. \tag{7}$$

This should be considered whenever differentiation or integration of Chebyshev series is performed. For instance we have for any Chebyshev series f:

$$\textstyle\int f\,dt = (\Delta t/2) \int f\,dx. \tag{8}$$

Negative and fractional powers. Our last four subroutines, dealing with expansion or iteration methods for the generation of noninteger powers of a Chebyshev series, are somewhat more sophisticated than the first six subroutines, but the theoretical basis of their operation has recently been described in detail [1]. For this reason, they will not be described in more detail here. All four subroutines use the multiplication subroutine *MLTPLY* but are otherwise independent. The subroutines *BINOM*, *XALFA*2, and *XALFA*3 all have the same purpose but operate with different methods and have different convergence properties. All three are given in order to allow the user to experiment and eventually select the one that is most efficient for his particular application.

Acknowledgments. I wish to thank Nancy Hamata at the Jet Propulsion Laboratory for her assistance in the programming and debugging of the present subroutines; also the two anonymous reviewers for their useful suggestions.

References

1. Broucke, R. Construction of national and negative powers of a formal series. *Comm. ACM 14*, 1 (Jan. 1971), 32–35.
2. Carpenter, L. Planetary perturbations in Chebyshev series. NASA Tech. Note TN-D-3168, Goddard Space Flight Center, Greenbelt, Md., Jan. 1966.
3. Clenshaw, C. W. The numerical solution of linear differential equations in Chebyshev series. *Proc. Cambr. Phil. Soc., 53* (1957), 134–149.
4. Clenshaw, C.W., Chebyshev series for mathematical functions. *Nat. Phys. Lab. Math. Tables, 5* (1962) London, HMSO.
5. Clenshaw, C.W., and Norton, H.J. The solution of nonlinear ordinary differential equations in Chebyshev series. *Computer J. 6* (1963), 88–92.

Algorithm

```
      SUBRØUTINE CHEBY(NF, NPL, NPLMAX, N2, FUNCTN, X, FXJ, GC)
C SIMULTANEØUS CHEBYSHEV ANALYSIS ØF NF FUNCTIØNS
C CØMPUTES A MATRIX, X, CØNTAINING ØNE CHEBYSHEV SERIES PER
C CØLUMN FØR A GIVEN NUMBER ØF FUNCTIØNS, NF. INPUT NPL,
C THE NUMBER ØF TERMS IN ALL SERIES, NPLMAX, THE RØW
C DIMENSIØN ØF X IN THE CALLING PRØGRAM (MUST BE.GE.NPL),
C N2, DIMENSIØN ØF GC (MUST BE.GE.2*(NPL-1)), AND FUNCTN,
C THE NAME ØF USER SUBRØUTINE WHICH DEFINES THE NF
C FUNCTIØNS. FXJ AND GC ARE WØRK SPACE.
C AN EXAMPLE ØF SUCH A SUBRØUTINE IS AS FØLLØWS
C SUBRØUTINE FUNCTN(A,VAL)
C DØUBLE PRECISIØN A,VAL(2)
C VAL(1)=DSIN(A)
C VAL(2)=DCØS(A)
C RETURN
C END
      DØUBLE PRECISIØN X(NPLMAX,NF), FXJ(NF), GC(N2), ENN, XJ,
     * FK, PEN, FAC
      DØ 20 K=1,NPL
        DØ 10 J=1,NF
          X(K,J) = 0.DO
   10   CØNTINUE
   20 CØNTINUE
      N = NPL - 1
      ENN = N
      PEN = 3.141592653589793200/ENN
      DØ 30 K=1,N2
        FK = K - 1
        GC(K) = DCØS(FK*PEN)
   30 CØNTINUE
      DØ 80 J=1,NPL
        XJ = GC(J)
        CALL FUNCTN(XJ, FXJ)
        IF (J.NE.1 .AND. J.NE.NPL) GO TO 50
        DØ 40 K=1,NF
          FXJ(K) = .5DO*FXJ(K)
   40   CØNTINUE
   50   DØ 70 L=1,NPL
          LM = MØD((L-1)*(J-1),N2) + 1
          DØ 60 K=1,NF
            X(L,K) = X(L,K) + FXJ(K)*GC(LM)
   60     CØNTINUE
   70   CØNTINUE
   80 CØNTINUE
      FAC = 2.0DO/ENN
      DØ 100 K=1,NPL
        DØ 90 J=1,NF
          X(K,J) = FAC*X(K,J)
   90   CØNTINUE
  100 CØNTINUE
      RETURN
      END


      SUBRØUTINE MLTPLY(XX, X2, NPL, X3)
C MULTIPLIES TWØ GIVEN CHEBYSHEV SERIES, XX AND X2, WITH
C NPL TERMS TØ PRØDUCE AN ØUTPUT CHEBYSHEV SERIES, X3.
      DØUBLE PRECISIØN XX(NPL), X2(NPL), X3(NPL), EX
      DØ 10 K=1,NPL
        X3(K) = 0.0DO
   10 CØNTINUE
      DØ 30 K=1,NPL
        EX = 0.0DO
        MM = NPL - K + 1
        DØ 20 M=1,MM
          L = M + K - 1
          EX = EX + XX(M)*X2(L) + XX(L)*X2(M)
   20   CØNTINUE
        X3(K) = 0.5DO*EX
   30 CØNTINUE
      X3(1) = X3(1) - 0.5DO*XX(1)*X2(1)
      DØ 50 K=3,NPL
        EX = 0.0DO
        MM = K - 1
        DØ 40 M=2,MM
          L = K - M + 1
          EX = EX + XX(M)*X2(L)
   40   CØNTINUE
        X3(K) = 0.5DO*EX + X3(K)
   50 CØNTINUE
      RETURN
      END


      SUBRØUTINE ECHEB(X, CØEF, NPL, FX)
C EVALUATES THE VALUE FX(X) ØF A GIVEN CHEBYSHEV SERIES,
C CØEF, WITH NPL TERMS AT A GIVEN VALUE ØF X BETWEEN
C -1. AND 1.
      DØUBLE PRECISIØN CØEF(NPL), X, FX, BR, BRPP, BRP2
      BR = 0.0DO
      BRPP = 0.0DO
      DØ 10 K=1,NPL
        J = NPL - K + 1
        BRP2 = BRPP
        BRPP = BR
        BR = 2.0DO*X*BRPP - BRP2 + CØEF(J)
   10 CØNTINUE
      FX = 0.5DO*(BR-BRP2)
      RETURN
      END


      SUBRØUTINE EDCHEB(X, CØEF, NPL, FX)
C EVALUATES THE VALUE FX(X) ØF THE DERIVATIVE ØF A
C CHEBYSHEV SERIES, CØEF, WITH NPL TERMS AT A GIVEN
C VALUE ØF X BETWEEN -1. AND 1.
      DØUBLE PRECISIØN CØEF(NPL), X, FX, XJP2, XJPL, XJ, BJP2,
     * BJPL, BJ, BF, DJ
      XJP2 = 0.0DO
      XJPL = 0.0DO
      BJP2 = 0.0DO
      BJPL = 0.0DO
      N = NPL - 1
      DØ 10 K=1,N
        J = NPL - K
        DJ = J
        XJ = 2.DO*CØEF(J+1)*DJ + XJP2
        BJ = 2.DO*X*BJPL - BJP2 + XJ
        BF = BJP2
        BJP2 = BJPL
        BJPL = BJ
        XJP2 = XJPL
        XJPL = XJ
   10 CØNTINUE
      FX = .5DO*(BJ-BF)
      RETURN
      END
      SUBRØUTINE DFRNT(XX, NPL, X2)
C COMPUTES THE DERIVATIVE CHEBYSHEV SERIES, X2, ØF A GIVEN
C CHEBYSHEV SERIES, XX, WITH NPL TERMS.
C TØ REPLACE A SERIES X BY ITS DERIVATIVE, USE
C CALL DFRNT(X,NPL,X)
      DØUBLE PRECISIØN XX(NPL), XXN, XXL, DN, DL, X2(NPL)
      DN = NPL - 1
      XXN = XX(NPL-1)
      X2(NPL-1) = 2.DO*XX(NPL)*DN
      X2(NPL) = 0.DO
      DØ 10 K=3,NPL
        L = NPL - K + 1
        DL = L
        XXL = XX(L)
        X2(L) = X2(L+2) + 2.DO*XXN*DL
        XXN = XXL
   10 CØNTINUE
      RETURN
      END
```

```
      SUBRØUTINE NTGRT(XX, NPL, X2)
C CØMPUTES THE INTEGRAL CHEBYSHEV SERIES, X2, ØF A GIVEN
C CHEBYSHEV SERIES, XX, WITH NPL TERMS.
C TØ REPLACE A SERIES X BY ITS INTEGRAL, USE
C CALL NTGRT(X,NPL,X)
      DØUBLE PRECISIØN XX(NPL), XPR, TERM, DK, X2(NPL)
      XPR = XX(1)
      X2(1) = 0.0DO
      N = NPL - 1
      DØ 10 K=2,N
        DK = K - 1
        TERM = (XPR-XX(K+1))/(2.DO*DK)
        XPR = XX(K)
        X2(K) = TERM
   10 CØNTINUE
      DK = N
      X2(NPL) = XPR/(2.DO*DK)
      RETURN
      END
```

```
      SUBRØUTINE INVERT(X, XX, NPL, NET, XNVSE, WW, W2)
C CØMPUTES THE INVERSE CHEBYSHEV SERIES, XNVSE, GIVEN A
C CHEBYSHEV SERIES, X, A FIRST APPRØXIMATIØN CHEBYSHEV
C SERIES, XX, WITH NPL TERMS, AND THE NUMBER ØF
C ITERATIØNS, NET. THE SUBRØUTINE USES THE EULER METHØD
C AND CØMPUTES ALL PØWERS EPS**K UP TC K=2**(NET+1),
C WHERE EPS=1-X*(XX INVERSE). WW AND W2 ARE WØRK SPACE.
C SUBRØUTINES USED - MLTPLY
      DØUBLE PRECISIØN X(NPL), XX(NPL), XNVSE(NPL), WW(NPL),
     * W2(NPL)
      CALL MLTPLY(X, XX, NPL, WW)
      WW(1) = 2.DO - WW(1)
      DØ 10 K=2,NPL
        WW(K) = -WW(K)
   10 CØNTINUE
      CALL MLTPLY(WW, WW, NPL, W2)
      WW(1) = 2.DO + WW(1)
      DØ 40 K=1,NET
        CALL MLTPLY(WW, W2, NPL, XNVSE)
        DØ 20 J=1,NPL
          WW(J) = WW(J) + XNVSE(J)
   20   CØNTINUE
        CALL MLTPLY(W2, W2, NPL, XNVSE)
        DØ 30 J=1,NPL
          W2(J) = XNVSE(J)
   30   CØNTINUE
   40 CØNTINUE
      CALL MLTPLY(WW, XX, NPL, XNVSE)
      RETURN
      END
```

```
      SUBRØUTINE BINØM(X, XX, NPL, M, NT, XA, WW, W2, W3)
C CØMPUTES THE BINØMIAL EXPANSIØN SERIES, XA, FØR (-1/M)
C PØWER ØF A GIVEN CHEBYSHEV SERIES, X, WITH NPL TERMS,
C WHERE M IS A PØSITIVE INTEGER. XX IS A GIVEN INITIAL
C APPRØXIMATIØN TØ X**(-1/M). NT IS A GIVEN NUMBER ØF
C TERMS IN BINØMIAL SERIES. WW, W2, AND W3 ARE WØRK SPACE
C SUBRØUTINES USED - MLTPLY
      DØUBLE PRECISIØN X(NPL), XX(NPL), XA(NPL), WW(NPL),
     * W2(NPL), W3(NPL), ALFA, CØEF, DM, DKMM, DKM2
      DM = M
      ALFA = -1.DO/DM
      DØ 10 J=1,NPL
        WW(J) = X(J)
   10 CØNTINUE
      DØ 30 K=1,M
        CALL MLTPLY(WW, XX, NPL, W2)
        DØ 20 J=1,NPL
          WW(J) = W2(J)
   20   CØNTINUE
   30 CØNTINUE
      WW(1) = WW(1) - 2.DO
      XA(1) = 2.DO
      DØ 40 J=2,NPL
        XA(J) = 0.0DO
        W3(J) = 0.DO
   40 CØNTINUE
      W3(1) = 2.DO
      DØ 60 K=2,NT
        DKMM = K - 1
        DKM2 = K - 2
        CØEF = (ALFA-DKM2)/DKMM
        CALL MLTPLY(W3, WW, NPL, W2)
        DØ 50 J=1,NPL
          W3(J) = W2(J)*CØEF
          XA(J) = XA(J) + W3(J)
   50   CØNTINUE
   60 CØNTINUE
      CALL MLTPLY(XA, XX, NPL, W2)
      DØ 70 J=1,NPL
        XA(J) = W2(J)
   70 CØNTINUE
      RETURN
      END
```

```
      SUBRØUTINE XALFA2(X, XX, NPL, M, MAXET, EPSLN, NET, WW,
     * W2)
C REPLACES A GIVEN INITIAL APPRØXIMATIØN CHEBYSHEV SERIES,
C XX, BY A GIVEN CHEBYSHEV SERIES, X, WITH NPL TERMS,
C RAISED TØ THE (-1/M) PØWER, WHERE M IS AN INTEGER.
C INPUT MAXET, MAXIMUM ALLØWED NUMBER ØF ITERATIØNS, AND
C EPSLN, REQUIRED PRECISIØN EPSILØN. ØUTPUT ARGUMENT,
C NET, IS NUMBER ØF ITERATIØNS PREFØRMED. IF MAXET=NET,
C REQUIRED PRECISIØN MAY NØT HAVE BEEN REACHED AND THERE
C MAY BE DIVERGENCE. WW AND W2 ARE WØRK SPACE.
C CØNVERGENCE IS QUADRATIC
C SUBRØUTINES USED - MLTPLY
      DØUBLE PRECISIØN X(NPL), XX(NPL), WW(NPL), W2(NPL),
     * EPSLN, DALFA, DM, S, TDMM
      DM = M
      DALFA = 1.DO/DM
      TDMM = 2.DO*(DM+1.DO)
      DØ 60 JX=1,MAXET
        DØ 10 L=1,NPL
          WW(L) = X(L)
   10   CØNTINUE
        DØ 30 K=1,M
          CALL MLTPLY(WW, XX, NPL, W2)
          DØ 20 L=1,NPL
            WW(L) = W2(L)
   20     CONTINUE
   30   CØNTINUE
        S = -2.DO
        DØ 40 L=1,NPL
          S = S + DABS(WW(L))
          WW(L) = -WW(L)
   40   CØNTINUE
        WW(1) = WW(1) + TDMM
        CALL MLTPLY(WW, XX, NPL, W2)
        DØ 50 L=1,NPL
          XX(L) = W2(L)*DALFA
   50   CØNTINUE
        NET = JX
        IF (DABS(S).LT.EPSLN) RETURN
   60 CØNTINUE
      RETURN
      END
```

```
      SUBRØUTINE XALFA3(X, XX, NPL, M, MAXET, EPSLN, NET, WW,
     * W2)
C REPLACES A GIVEN INITIAL APPRØXIMATIØN CHEBYSHEV SERIES,
C XX, BY A GIVEN CHEBYSHEV SERIES, X, WITH NPL TERMS,
C RAISED TØ THE (-1/M) PØWER, WHERE M IS AN INTEGER.
C INPUT MAXET, MAXIMUM ALLØWED NUMBER ØF ITERATIØNS, AND
C EPSLN, REQUIRED PRECISIØN EPSILØN. ØUTPUT ARGUMENT,
C NET, IS NUMBER ØF ITERATIØNS PREFØRMED. IF MAXET=NET,
C REQUIRED PRECISIØN MAY NOT HAVE BEEN REACHED AND THERE
C MAY BE DIVERGENCE. WW AND W2 ARE WØRK SPACE.
C CØNVERGENCE IS ØF ØRDER THREE
C SUBRØUTINES USED - MLTPLY
      DØUBLE PRECISIØN X(NPL), XX(NPL), WW(NPL), W2(NPL),
     * EPSLN, DALFA, DM, S, TDMM, P5DML
      DM = M
      DALFA = 1.DO/DM
      TDMM = 2.DO*(DM+1.DO)
      P5DML = .5DO*(DM+1.DO)
      DØ 90 JX=1,MAXET
        DØ 10 L=1,NPL
          WW(L) = X(L)
   10   CØNTINUE
        DØ 30 K=1,M
          CALL MLTPLY(WW, XX, NPL, W2)
          DØ 20 L=1,NPL
            WW(L) = W2(L)
   20     CØNTINUE
   30   CØNTINUE
        S = -2.DO
        DØ 40 L=1,NPL
          S = S + DABS(WW(L))
   40   CØNTINUE
        WW(1) = WW(1) - 2.DO
        DØ 50 L=1,NPL
          WW(L) = WW(L)*DALFA
   50   CØNTINUE
        CALL MLTPLY(WW, WW, NPL, W2)
        DC 60 L=1,NPL
          WW(L) = -WW(L)
          W2(L) = W2(L)*P5DML
   60   CØNTINUE
        WW(1) = WW(1) + 2.DO
        DØ 70 L=1,NPL
          W2(L) = W2(L) + WW(L)
   70   CØNTINUE
        CALL MLTPLY(W2, XX, NPL, WW)
        DØ 80 L=1,NPL
          XX(L) = WW(L)
   80   CØNTINUE
        NET = JX
        IF (DABS(S).LT.EPSLN) RETURN
   90 CØNTINUE
      RETURN
      END
```

Remark and Certification on Algorithm 446

Ten Subroutines for the Manipulation of Chebyshev Series [C1] [R. Broucke, *Comm. ACM 16* (Apr. 1973), 254–265]

Robert Piessens and Irene Mertens [Recd 11 Jan. 1974] Applied Mathematics and Programming Division, University of Leuven, B-3030, Heverlee (Belgium)

1. Two corrections are needed in the subroutine *CHEBY*:

(i) The statement after statement 50 must be changed into:

```
LM = MOD(L−1)*(J−1), 2*N) + 1
```

(ii) formulas (1) and (3) for the computation of Chebyshev series coefficients c_l do not agree with the exact formulas given by Fox and Parker [1, p. 66]. Indeed the last coefficient must be halved. This can be accomplished in the routine by replacing the five statement before *RETURN* by

```
      DO 100 J = 1, NF
        DO 90 K = 1, NPL
          X(K, J) = FAC*X(K, J)
 90     CONTINUE
        X(NPL, J) = 0.5 DO*X(NPL, J)
100   CONTINUE
```

2. Moreover, the number of cosine-evaluations in *CHEBY* can be reduced by a factor 4 if the *DO*-loop:

```
      DO 30 K = 1, N2
        ⋮
 30   CONTINUE
```

is replaced by

```
      NN = (NPL+1)/2
      DO 30 K = 1, NN
        FK = K − 1
        GC(K) = DCOS(FK*PEN)
        NPLK = NPL+1 − K
        GC(NPLK) = −GC(K)
 30   CONTINUE
      DO 35 K = 1, N
        NPLK = NPL + K
        GC(NPLK) = −GC(K+1)
 35   CONTINUE
```

3. In subroutine *MLTPLY*, the *DO*-loop

```
      DO 10 K = 1, NPL
        ⋮
 10   CONTINUE
```

may be deleted.

We have tested *INVERT* and *BINOM* by calculating

$$[T_0(x) + aT_1(x)]^{-1},$$

and *BINOM*, *XALFA*2 and *XALFA*3 by calculating

$$\left[\left(1+\frac{a^2}{2}\right)T_0(x) + 2aT_1(x) + \frac{a^2}{2}T_2(x)\right]^{-1/2}$$

The results are compared with the exact Chebyshev series expansion

$$(1+ax)^{-1} = \sum_{k=0}^{\infty}{}' a_k T_k(x)$$

where

$$a_k = \frac{2}{(1-a^2)^{\frac{1}{2}}}\left(\frac{(1-a^2)^{\frac{1}{2}} - 1}{a}\right)^k, \qquad |a| < 1.$$

The rate of convergence of this series depends strongly on the value of a. For this reason, we have given a the values 0.1, 0.2, . . . , 0.9.

We have noted that, especially in the case of slowly converging series, *INVERT*, *XALFA*2 and *XALFA*3 are more efficient than *BINOM*. Moreover, in order to have convergence, *BINOM* requires more accurate initial approximations than the other routines.

Reference

1. Fox, L., and Parker, I.B. *Chebyshev Polynomials in Numerical Analysis*. Oxford University Press, London, 1968.

Algorithm 447

Efficient Algorithms for Graph Manipulation [H]

John Hopcroft and Robert Tarjan [Recd. 24 March 1971 and 27 Sept. 1971]
Cornell University, Ithaca, NY 14850

Abstract: Efficient algorithms are presented for partitioning a graph into connected components, biconnected components and simple paths. The algorithm for partitioning of a graph into simple paths is iterative and each iteration produces a new path between two vertices already on paths. (The start vertex can be specified dynamically.) If V is the number of vertices and E is the number of edges, each algorithm requires time and space proportional to max (V, E) when executed on a random access computer.

Key Words and Phrases: graphs, analysis of algorithms, graph manipulation

CR Categories: 5.32

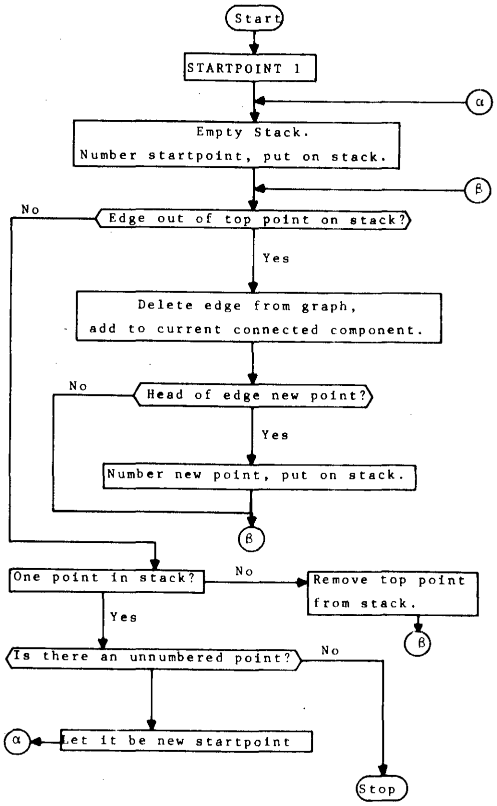
Language: Algol

Description

Graphs arise in many different contexts where it is necessary to represent interrelations between data elements. Consequently algorithms are being developed to manipulate graphs and test them for various properties. Certain basic tasks are common to many of these algorithms. For example, in order to test a graph for planarity, one first decomposes the graph into biconnected components and tests each component separately. If one is using an algorithm [4] with asymptotic growth of V log(V) to test for planarity, it is imperative that one use an algorithm for partitioning the graph whose asymptotic growth is linear with the number of edges rather than quadratic in the number of vertices. In fact, representing a graph by a connection matrix in the above case would result in spending more time in constructing the matrix than in testing the graph for planarity if it were represented by a list of edges. It is with this in mind that we present a structure for representing graphs in a computer and several algorithms for simple operations on the graph. These include dividing a graph into connected components, dividing a graph into biconnected components, and partitioning a graph into simple paths. The algorithm for division into connected components is well known [7]. The description of an algorithm similar to the biconnected components algorithm has just appeared [6]. For a graph with V vertices and E edges, each algorithm requires time and space proportional to max(V, E).

Standard graph terminology will be used throughout this discussion. See for instance [2]. We assume that the graph is initially given as a list of pairs of vertices, each pair representing an edge of the graph. The order of the vertices is unimportant; that is, the graph is unordered. Labels may be attached to some or all of the vertices and edges.

Fig. 1. Flowchart for connected components algorithm.


Start
STARTPOINT 1
α
Empty Stack.
Number startpoint, put on stack.
β
No
Edge out of top point on stack?
Yes
Delete edge from graph,
add to current connected component.
No
Head of edge new point?
Yes
Number new point, put on stack.
β
One point in stack?
No
Remove top point from stack.
β
Yes
Is there an unnumbered point?
No
α
Let it be new startpoint
Stop

Our model is that of a random-access computer with standard operations; accessing a number in storage requires unit time. We allow storage of numbers no larger than k max(V, E) where k is some constant. (If the labels are large data items, we assume that they are numbered with small integer codes and referred to by their codes; there are no more than k max(V, E) labels.) It is easy to see and may be proved rigorously that most interesting graph procedures require time at least proportional to E when implemented on any reasonable model of a computer, if the input is a list of edges. This follows the fact that each edge must be examined once.

It is very important to have an appropriate computer representation for graphs. Many researchers have described algorithms which use the matrix representation of a graph [1]. The time and space bounds for such algorithms generally are at least V^2 [3] which is not as small as possible if E is small. (In planar graphs for instance, $E \leq 3V - 3$.) We use a list structure representation of a graph. For each vertex, a list of vertices to which it is adjacent is made. Note that two entries occur for each edge, one for each of its end points. A cross-link between these two entries is often useful. Note also that a directed graph may be represented in this fashion;

This research was carried out while the authors were at Stanford University and was supported by the Hertz Foundation and by the Office of Naval Research under grant number N-00014-67-A-0112-0057 NR-44-402.

if vertex v_2 is on the list of vertices adjacent to v_1, then (v_1, v_2) is a directed edge of the graph. Vertex v_1 is called the *tail*, and vertex v_2 is called the *head* of the edge.

A directed representation of an undirected graph is a representation of this form in which each edge appears only once; the edges are directed according to some criterion such as the direction in which they are transversed during a search. Some version of this structure representation is used in all the algorithms.

One technique has proved to be of great value. That is the notion of search, moving from vertex to adjacent vertex in the graph in such a way that all the edges are covered. In particular depth-first search is the basis of all the algorithms presented here. In this pattern of search, each time an edge to a new vertex is discovered, the search is continued from the new vertex and is not renewed at the old vertex until all edges from the new vertex are exhausted. The search process provides an orientation for each edge, in addition to generating information used in the particular algorithms.

Detailed Description of the Algorithms

Algorithm for finding the connected components of a graph. This algorithm finds the connected components of a graph by performing depth-first search on each connected component. Each new vertex reached is marked. When no more vertices can be reached along edges from marked vertices, a connected component has been found. An unmarked vertex is then selected, and the process is repeated until the entire graph is explored.

The details of the algorithm appear in the flowchart (Figure 1). Since the algorithm is well known, and since it forms a part of the algorithm for finding biconnected components, we omit proofs of its correctness and time bound. These proofs may be found as part of the proofs for the biconnected components algorithm. The algorithm requires space proportional to $\max(V, E)$ and time proportional to $\max(V, E)$, where V is the number of vertices and E is the number of edges of the graph.

Algorithm for finding the biconnected components of a graph. This algorithm breaks a graph into its biconnected components by performing a depth-first search along the edges of the graph. Each new point reached is placed on a stack, and for each point a record is kept of the lowest point on the stack to which it is connected by a path of unstacked points. When a new point cannot be reached from the top of the stack, the top point is deleted, and the search is continued from the next point on the stack. If the top point does not connect to a point lower than the second point on the stack, then this second point is an articulation point of the graph. All edges examined during the search are placed on another stack, so that when an articulation point is found the edges of the corresponding biconnected component may be retrieved and placed in an output array.

When the stack is exhausted, a complete search of a connected component has been performed. If the graph is connected, the process is complete. Otherwise, an unreached node is selected as a new starting point and the process repeated until all of the graph has been exhausted. Isolated points are not listed as biconnected components, since they have no adjacent edges. They are merely skipped. The details of the algorithm are given in the flowchart (Figure 2). Note that this flowchart gives a nondeterministic algorithm, since any new edge may be selected in block A. The actual program is deterministic: the choice of an edge depends on the particular representation of the graph.

We will prove that the nondeterministic algorithm terminates on all simple graphs without loops, and we also derive a bound on the execution time. We will then prove the correctness of the algorithm, by induction on the number of edges in the graph. Note that the algorithm requires storage space proportional to $\max(V, E)$, where V is the number of vertices and E is the number of edges of the graph.

Let us consider applying the algorithm to a graph. Referring to the flowchart, every passage through the *YES* branch of block A causes an edge to be deleted from the graph. Each passage through the *NO* branch of block B causes a point to be deleted from the stack. Once a point is deleted from the stack it is never added to the stack again, since all adjacent edges have been examined. Each edge is deleted from the stack of edges once in block C. Thus the blocks directly below the *YES* branch of block A are executed at most E times, those below the *NO* branch of block B at most V times, and the total time spent in block C is proportional to E. Therefore there is some k such that for all graphs the algorithm takes no more than $k \max(V, E)$ steps. A more explicit time bound may be calculated by referring to the program.

Suppose the graph G contains no edges. By examining the flowchart we see that the algorithm, when applied to G, will terminate after examining each point once and listing no components. Thus the algorithm operates correctly in this case. Suppose the algorithm works correcly on all graphs with E-1 or fewer edges. Consider applying the algorithm to a graph G with E edges. Since the stack of points becomes empty at least once during the operation of the algorithm, and since the *YES* branch at block D must be taken when only two points are on the stack, every edge must not only be placed on the stack of edges but must be removed in block C. Consider the first time block C is reached when the algorithm is applied to graph G. Suppose not all the edges in the graph are removed from the stack of edges in this execution of block C. Then p, the second point on the stack, is an articulation point and separates the removed edges from the other edges in the graph.

Let E_1 be the set of removed edges, let E_2 be the set of edges still on the stack, and let E_3 be the set of remaining edges of G. Let G_1 be the subgraph of G made up of the edges from E_1, and let $G_2 = G - G_1$. Since G_1 and G_2 each have at most E-1 edges, the induction hypothesis implies that the algorithm operates correctly on both G_1 and G_2.

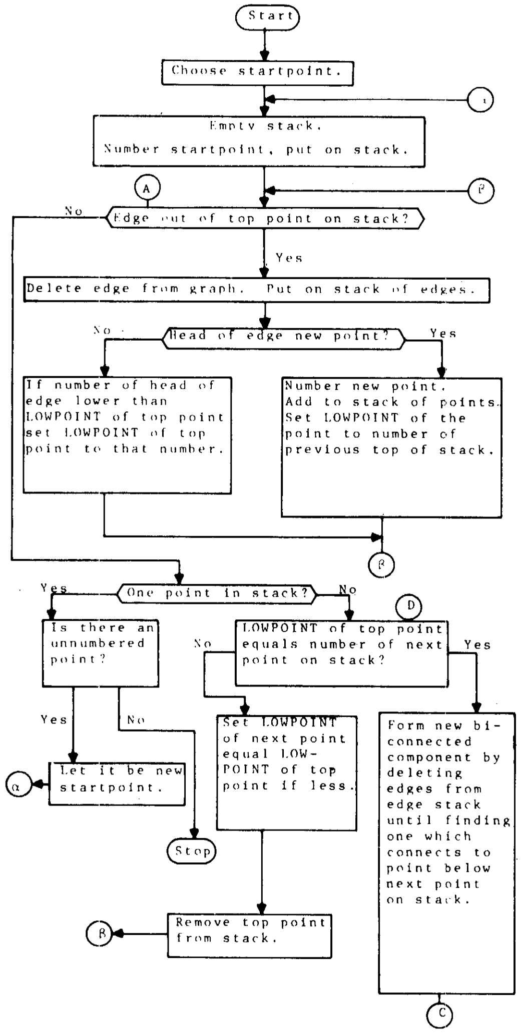
Assume that the edges for each vertex in G_1 and G_2 are listed in the same order as for G. Consider the sequence of steps taken when the algorithm is applied to G. The sequence of steps taken on G_2 can be divided into an initial sequence of steps which results in placing the edges E_1 on the stack, followed by the remaining sequence S_2. The sequence of steps taken on G consists of the sequence S_1, followed by the steps taken on G_2 with p as the start point, followed by S_2.

The behavior of the algorithm on G is simply the composite of its behavior on G_1 and G_2; thus the algorithm must operate correctly on G.

Now suppose that the first time block C is reached, all the edges of G are removed from the stack of edges. We want to show that in this case G is biconnected. Suppose that G is not biconnected. Then choose a biconnected component of G which may be separated by removing some one point p and which does not contain the start point of G. Let the edges making up this component be subgraph G_1 of G; let the remainder of G be G_2. The algorithm operates correctly on G_1 and on G_2 by assumption. The behavior of the algorithm on G is a composite of its behavior on G_1 and on G_2. Assume that the edges for each vertex in G_1 and G_2 are listed in the same order as for G. The sequence of steps on G is identical to the sequence of steps on G_1 until an edge of G_2 out of vertex p is selected. Then the sequence of steps of G is identical to the sequence on G_2 with start point p. The remaining steps on G are the same as the remaining steps on G_1. But the algorithm reaches block C once while processing G_1 and at least once while processing G_2. This contradicts the fact that the algorithm only reaches block C once while processing G. Thus G must be biconnected. and the algorithm operates correctly on G. By induction, the algorithm is correct for all simple graphs without loops.

Algorithm for finding simple paths in a graph. This algorithm may be used to partition a graph into simple paths, such that all the paths exhaust the edges of the graph. Each iteration of the algorithm produces a new path which contains no vertex twice, and which connects the chosen startpoint with some other vertex which already occurs in a path. Total running time is proportional to the number of edges in the graph. The starting point for each successive path may be selected arbitrarily. In fact, the initial edge of each

Fig. 2. Flowchart for biconnected components algorithm.


Start
Choose startpoint.
Empty stack.
Number startpoint, put on stack.
A
β
No
Edge out of top point on stack?
Yes
Delete edge from graph. Put on stack of edges.
No
Head of edge new point?
Yes
If number of head of edge lower than LOWPOINT of top point set LOWPOINT of top point to that number.
Number new point. Add to stack of points. Set LOWPOINT of the point to number of previous top of stack.
β
Yes
One point in stack?
No
D
Is there an unnumbered point?
No
LOWPOINT of top point equals number of next point on stack?
Yes
Yes
No
Set LOWPOINT of next point equal LOW-POINT of top point if less.
Form new biconnected component by deleting edges from edge stack until finding one which connects to point below next point on stack.
α
Let it be new startpoint.
Stop
β
Remove top point from stack.
C

successive path may be selected arbitrarily from the set of unused edges.

The algorithm is highly dependent on the graph being biconnected. (The biconnected components of a graph are found using the previously described algorithm.) In order to find a new path, the initial edge is selected and the head of the edge is checked. If this point has never been reached before, a depth-first search is begun which must end in a path since the graph is biconnected. The search generates a tree-like structure: specifically, it is a tree with edges connecting some vertices with their (not necessarily immediate) ancestors. (We will visualize the tree drawn so that the root, which is an ancestor of all points, is at the bottom of the tree.) Enough information is saved from this tree so that if a point in it is reached when building another path, the path may be completed without any further search.

The flowchart (Figures 3 and 4) gives the details of the algorithm. It is divided into two parts; one for the depth-first search process and one for path construction using previously gathered information. We shall prove the correctness of the algorithm and give a time bound for its operation. To derive the time bound, we assume that one point is marked old initially, and a different point

Fig. 3. Flowchart for pathfinding algorithm (I).

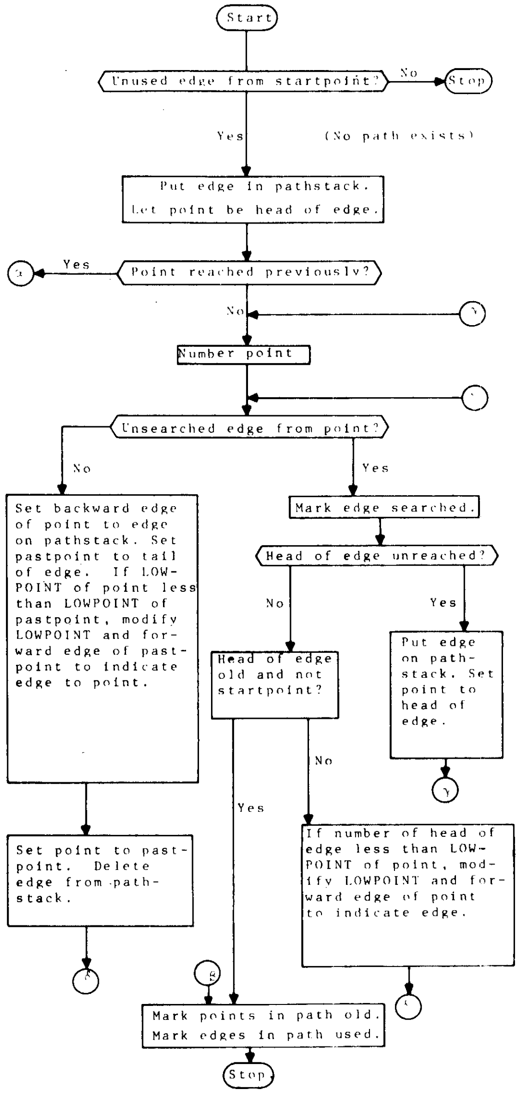
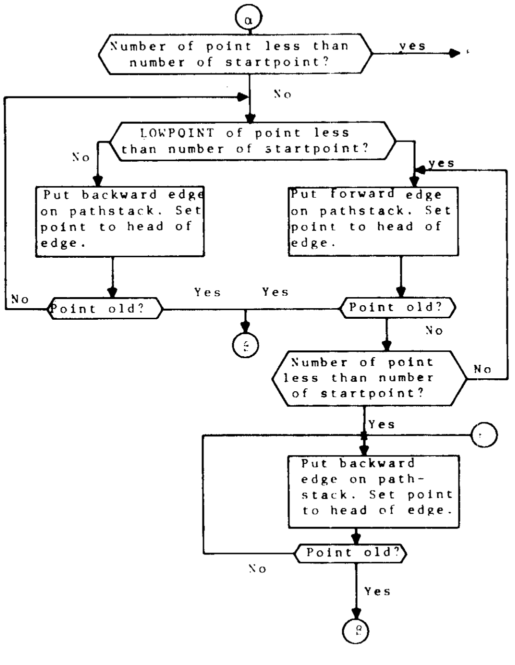

Start
Unused edge from startpoint?
No
Stop
Yes
(No path exists)
Put edge in pathstack.
Let point be head of edge.
α
Yes
Point reached previously?
No
γ
Number point
δ
Unsearched edge from point?
No
Yes
Set backward edge of point to edge on pathstack. Set pastpoint to tail of edge. If LOW-POINT of point less than LOWPOINT of pastpoint, modify LOWPOINT and forward edge of pastpoint to indicate edge to point.
Mark edge searched.
Head of edge unreached?
No
Yes
Head of edge old and not startpoint?
Put edge on pathstack. Set point to head of edge.
No
γ
Yes
Set point to pastpoint. Delete edge from pathstack.
If number of head of edge less than LOW-POINT of point, modify LOWPOINT and forward edge of point to indicate edge.
δ
β
δ
Mark points in path old.
Mark edges in path used.
Stop

Fig. 4. Flowchart for pathfinding algorithm (II).


α
Number of point less than number of startpoint?
yes
No
LOWPOINT of point less than number of startpoint?
No
yes
Put backward edge on pathstack. Set point to head of edge.
Put forward edge on pathstack. Set point to head of edge.
No
Point old?
Yes
Yes
Point old?
No
β
Number of point less than number of startpoint?
No
Yes
Put backward edge on path-stack. Set point to head of edge.
Point old?
No
Yes
β

is selected as the initial startpoint. The algorithm is then run repeatedly with arbitrary startpoints until all edges are used to form paths.

Let us consider path generation using depth-first search; that is, suppose the algorithm is applied and that the head of the first edge selected is previously unreached. Referring to the flowchart, we see that the search process is very similar to that used in the biconnectivity algorithm. A search tree is generated, and each edge examined is either part of the tree or connects a point to one of its predecessors in the tree. *LOWPOINT* is exactly the same as in the biconnectivity algorithm; it gives the number of the lowest point in the tree reachable from a given point by continuing out along the tree and taking one edge back toward the root. The forward edges point along this path, while the backward edges point back along the tree branches. We have shown in the correctness proof of the biconnectivity algorithm that, if the graph is biconnected, *LOWPOINT* of a given point must point to a node which is an ancestor of the immediate predecessor of the given point. In particular, *LOWPOINT* of the second point in the search tree must indicate an old point which is not the startpoint. Therefore the algorithm will find a path containing the initial edge. Note that all points encountered during the search process must either be old or unreached, since every point reached in a previous search either has had all its edges examined or has been included in a path.

Let us now suppose that the head of the first edge has been reached previously but is not marked old. Then the forward and backward pointers, along with the *LOWPOINT* values, allow the algorithm to construct a path without further search. First, if the number of the head of the edge is less than the number of the startpoint, then following backward pointers will certainly produce a simple path, since the root of a search tree must be old and each successive point along a backward path has a lower number and thus is distinct from the other points in the path. If the initial edge is part of a search tree and the startpoint is the predecessor of the second point, then *LOWPOINT* of the second point must be less than the number of the startpoint. Following forward edges until reaching a point numbered lower than the startpoint and then following backward edges will produce a simple path. This is true since the forward edges point through descendants of the tree, with the single exception of the edge whose head is a point below startpoint in the tree. The last case to consider occurs when the initial edge is not part of a search tree but points from a node to one of its descendants in a tree. In this case some node in the tree between the startpoint and the second point of the path must have a *LOWPOINT* value less than the number of the startpoint. If we follow backward edges until the first such point is reached, then follow forward edges until a point numbered less than the startpoint is reached, and finally follow backward edges until an old point is reached, we will generate a simple path. Note that the first forward edge taken cannot lead to the previous point because, if it did, the *LOWPOINT* value at the previous point would be less than the number of startpoint, and the forward edge from this point would have been chosen instead of the backward edge.

We thus see that each execution of the pathfinding algorithm produces a simple path, assuming that the algorithm is applied to a biconnected graph with at least one point which is not the first startpoint marked old initially. Since each edge is examined at most once in the search section of the algorithm, and since each edge is put into a path once, there is a constant k such that the time required to execute the algorithm until no edges are unused is less than kE steps, where E is the number of edges in the graph. (Note that the number of vertices, V, is less than E if the graph is biconnected.) Detailed examination of the program will produce a more exact time bound.

Another algorithm for finding simple paths exists. Lempel, Even, and Cederbaum [5] have described an algorithm for numbering the vertices of a biconnected graph such that: (i) each number is an integer in the range 1 to V, where V is the number of vertices on the graph; (ii) vertices 1 and V are jointed by an edge; (iii) for all $1 < i < V$, vertex i is joined to at least two vertices, one with a higher number and one with a lower number. We may use this algorithm to partition a graph into simple paths.

Given a start point and an adjacent end point, number the vertices so that the startpoint is 1, the endpoint is V, and the numbering satisfies the conditions above. Take edge $(1, V)$ as the first path. Given an arbitrary startpoint, find an edge to a higher numbered vertex. Continue to find edges to successively higher numbered vertices until an old vertex is reached.

This algorithm is clearly correct and looks conceptually simple. However, Lempel, Even, and Cederbaum present no efficient implementation of their numbering algorithm, and the only efficient way we have found to implement it requires using the previously described pathfinding algorithm in a more complicated form. Thus the new algorithm requires time and space proportional to $\max(V, E)$, but the constants of proportionality are larger than those for the implemented algorithm.

Implementation. The algorithms for finding connected components, biconnected components, and simple paths were originally implemented and tested in Algol W. The programs were then translated to Algol for publication and tested using the OS/360 Algol compiler. Auxiliary subroutines were also implemented. Brief descriptions of the procedures are provided below.

ADD2(*A*, *B*, *STACK*, *PTR*): This procedure adds value *A* followed by value *B* to the top of stack *STACK* and increments the pointer to the top of the stack (*PTR*). Stacks are represented as arrays; the top of the stack is the highest filled location.

NEXTLINK(*POINT*,*VALUE*): This procedure is used to build the structural representation of a graph. It adds *VALUE* to the list of vertices adjacent to *POINT*. (*POINT*, *VALUE*) is an edge (possibly directed) of the graph.

CONNECT(*V*, *E*, *EPTR*, *EDGELIST*, *COMPONENTS*): This procedure, given a graph with *V* vertices and *E* edges, whose edges are listed in *EDGELIST*, computes the connected components of the graph and places the edges of the components in *COMPONENTS*. Each component is preceded by an entry containing the number of edges *E*′ of the component. The edges are oriented for output according to the direction in which they were searched (head first, tail second).

BICONNECT(*V*,*E*,*EPTR*,*EDGELIST*,*COMPONENTS*):This procedure, given a graph with *V* vertices and *E* edges, whose edges are listed in *EDGELIST*, computes the biconnected components of the graph and places them in *BICOMPONENTS*. Each component is preceded by an entry containing the number of edges *E* of the component. The edges are oriented for output according to the direction in which they were searched (head first, tail second).

PATHFINDER(*STARTPT*,*PATHPT*,*CODEVALUE*,*PATH*): This procedure, given a list structure representation of a biconnected graph with certain vertices marked as old, constructs a simple path from *STARTPOINT* to some old vertex, saving information to be used in constructing succeeding paths. The new path is stored in array *PATH*. Calling *PATHFINDER* repeatedly may be used to partition the graph into simple paths.

The procedure *PATHFINDER* requires that the structural representation of the graph be stored as follows. Each edge is treated as a pair of directed edges each of which is represented by an integer between $v + 1$ and $v + 2 \times e$. If $i_1, i_2, \ldots, i_k$ are the integers corresponding to the edges out of vertex i, then initialize $NEXT(i)$ to i_1, $NEXT(i_j)$ to i_{j+1}, $1 \leq j < k$, and $NEXT(k)$ to 0. If the edge i_j terminates at vertex l, initialize $HEAD(i_j)$ to l. $LINK(i_j)$ is the integer corresponding to the edge in the other direction. For $1 \leq i \leq v$, $BACK(i)$, $FORWARD(i)$, $PATHOCDE(i)$ are initialized to 0, $LOWPOINT(i)$ is initialized to $v + 1$, $NODE(i)$ is initialized to $NEXT(i)$ and $OLD(i)$ is initialized to *FALSE*. For $v + 1 \leq i \leq v + 2 \times e$ $MARK(i)$ is initialized to *FALSE*. Before the first call of *PATHFINDER* some nonnull set of vertices must be marked as *OLD* and assigned successive *PATHCODE* values. *CODEVALUE* is set equal to the number of vertices marked as *OLD*. If this is not done the first path cannot end at an *OLD* vertex.

Further comments may be found in the program listings below.

References

1. Fisher, G.J. Computer recognition and extraction of planar graphs from the incidence matrix. *IEEE Trans. in Circuit Theory CT-13*, (June 1966), 154–163.
2. Harary, F. *Graph Theory*. Addison-Wesley, Reading, Mass., 1969.
3. Holt, R., and Reingold, E. On the time required to detect cycles and connectivity in directed graphs. Comput. Sci. TR 70-33, Cornell U. Ithaca, N.Y.
4. Hopcroft, J., and Tarjan, R. Planarity testing in *v* log *v* steps, extended abstract. Stanford U. CS 201, Mar. 1971.
5. Lempel, A., Even, S., and Cederbaum, I. An algorithm for planarity testing of graphs. Theory of Graphs: International Symposium: Rome, July 1966. P. Rosenstiehl (Ed.) Gordon and Breach, New York, 1967, pp. 215–232.
6. Paton, K. An algorithm for the blocks and cutnodes of a graph. *Comm. ACM 14*, 7(July 1971), 428–475.
7. Shirey, R.W. Implementation and analysis of efficient graph planarity testing. Ph.D. diss., Comput. Sci. Dep., U. of Wisconsin, Madison, Wis., 1969.

Algorithm

```
procedure add2 (a, b, stack, ptr);
  value a, b; integer a, b, ptr; integer array stack;
comment Procedure adds values a and b to stack stack and in-
  creases stack pointer ptr by 2;
begin
  ptr := ptr + 2; stack[ptr − 1] := a; stack[ptr] := b
end of add2;
procedure nextlink (point, val);
  value point, val; integer point, val;
comment Procedure adds directed edge (point, val) to structural
  representation of a graph. Global variables are described as fol-
  lows. head[v+1:v+2×e] and next[1:v+2×e] contain the struc-
  tural representation of the graph. freenext is the current last
  entry in next array;
begin
  freenext := freenext + 1; next[freenext] := next[point];
  next[point] := freenext; head[freenext] := val
end of nextlink;
integer procedure min(a, b);
  value a, b; integer a, b;
comment Procedure computes the minimum of two integers;
if a < b then min := a else min := b;
procedure connect (v, e, cptr, edgelist, components);
  value v, e; integer v, e, cptr;
  integer array edgelist, components;
comment Procedure finds the connected components of a graph.
  The parameters are described as follows. v and e are the number
  of vertices and edges of the graph. edgelist[1:2×e] is the initial
  list of edges of the graph. components[1:3×e] is the list of edges
  for each component. The list of edges for each component is pre-
  ceded by an entry giving the number of edges of the compo-
  nent. cptr is a pointer to the last entry in components. The global
  variables are described as follows. head[v+1:v+2×e] and
  next[1:v+2×e] contain the structural representation of the
  graph. freenext is the last entry in the array next. The local
  variables are described as follows. number[1:v+1] is used for
  numbering the vertices during the depth first search. code con-
  tains the current highest vertex number. point is the current
  vertex being examined during the search. v2 is the next vertex
  to be examined during the search. oldptr contains the position
  in components to place the value of the next component. The
  global procedures are add2 and nextlink. A recursive depth-
  first search procedure is used to examine connected components
  of the graph;
begin
  integer array number [1:v+1];
  integer code, point, v2, oldptr, i;
  procedure connector (point, oldpt);
    value point, oldpt; integer point, oldpt;
    comment This recursive procedure finds a connected component
      using a depth-first search. The parameters are described as fol-
      lows. point is the startpoint of search. oldpt is the previous
      startpoint. Global variables are the same as for connect. The
      global procedures are add2;
    comment Examine each edge out of point;
    for i = i while next[ point] > 0 do
    begin
      comment v2 is head of edge. Delete edge from structural repre-
        sentation;
      v2 := head[next[point]];
      next[point] := next[next[point]];
      comment Has this edge been searched in the other direction?
        If so, look for another edge;
      if (number[v2] < number[point]) ∧ (v2≠oldpt) then
      begin
        comment Add edge to components;
        add2(point, v2, components, cptr);
        comment Determine if a new point has been found;
        if number[v2] = 0 then
        begin
          comment New point found. Number it;
          number[v2] := code := code + 1;
          comment Initiate a depth-first search from the new point;
          connector(v2, point)
        end
      end
    end;
  comment Construct the structural representation of the graph;
  freenext := v;
  for i := 1 step 1 until v do next[i] := 0;
  for i := 1 step 1 until e do
  begin
    comment Each edge occurs twice, once for each endpoint;
    nextlink(edgelist[2×i−1], edgelist[2×i]);
    nextlink(edgelist[2×i], edgelist[2×i−1])
  end;
  comment Initialize variables for search;
  cptr := 0; point := 1;
  for i := 1 step 1 until v + 1 do number[i] := 0;
  for i := i while point ≤ v do
  begin
    comment Each execution of connector searches a connected
      component. After each search, find an unnumbered vertex
      and search again. Repeat until all vertices are investigated;
    number[point] := code := 1;
    oldptr := cptr := cptr + 1;
    connector(point,0);
    comment Compute number of edges of components;
      components[oldptr] := (cptr-oldptr)÷2;
    for i := i while number[point] ≠ 0 do point := point + 1
  end
end;
procedure biconnect(v, e, bptr, edgelist, bicomponents);
  value v, e; integer v, e, bptr;
  integer array edgelist, bicomponents;
begin
  comment Procedure finds biconnected components of a graph.
    The parameters are described as follows. v and e are the num-
    ber of vertices and edges of the graph. edgelist[1:2×e] is the
    initial list of edges of the graph. bicomponents[1:3×e] is the list
    of edges for each component found. Each component is pre-
    ceded by an entry giving the number of edges of the com-
    ponent. bptr is a pointer to the last entry of bicomponents. The
    global variables are described as follows. head[v+1:v+2×e]
    and next[1:v+2×e] contain the structural representation of the
    graph. freenext is the last entry in the array next. The local
    variables are described as follows. number[1:v+1] is an array
    used for numbering the vertices during the depth-first search.
    code is the current highest vertex number. edgestack[1:2×e]
  is used for storage of edges examined during search. eptr is
  a pointer to last entry in edgestack. point is the current point
  being examined during search. v2 is the next point to be examined
  during search. newlowpt is the lowpoint for the biconnected
  part of graph above and including v2. oldptr is
  pointer to position in bicomponents to place a value of next
  component. The global procedures are min, add2, and nextlink.
  A recursive depth-first search procedure is used to divide
  the graph. The lowest point reachable from the current point
  without going through previously searched points is calculated.
  This information allows determination of the articulation
  points and division of the graph;
integer array number[1:v+1], edgestack[1:2×e];
integer code, eptr, point, v2, newlowpt, oldptr, i;
procedure biconnector (point, oldpt, lowpoint);
  integer point, oldpt, lowpoint;
comment Recursive procedure to search a connected component
  and find its biconnected components using depth-first search.
  The parameters are described as follows. point is the startpoint
  of the search. oldpt is the previous startpoint. lowpoint is the
  lowest point reachable on a path found during search. The
  global variables are the same as for biconnect. The global
  procedures are min and add2;
comment Examine each edge out of point;
for i := i while next[point] > 0 do
begin
  comment v2 is the head of the edge. Delete edge from structural
    representation;
  integer v2;
  v2 := head[next[point]];
  next[point] := next[next[point]];
  comment If the edge has been searched in the other direction,
    then look for another edge;
  if (number[v2]<number[point]) ∧ (v2≠oldpt) then
  begin
    comment Add edge to edgestack;
    add2 (point, v2, edgestack, eptr);
    if number[v2] = 0 then
    begin
      comment New point found. Number it;
      number[v2] := code := code + 1;
      comment Initiate a depth-first search from the new point;
      newlowpt := v + 1;
      biconnector (v2, point, newlowpt);
      comment Note that although the global variable v2 is
        changed, its value is restored upon exit from this procedure.
        Recalculate lowpoint;
      lowpoint := min(lowpoint, newlowpt);
      if newlowpt ≥ number[point] then
      begin
        comment point is an articulation point. Output edges
          of component from edgestack;
        oldptr := bptr := bptr + 1;
        for i := i while number[edgestack[eptr−1]] >
          number[point] do
        begin
          add2(edgestack[eptr−1],   edgestack[eptr],   bicomponents,
            bptr);
          eptr := eptr − 2
        end;
        comment Add last edge;
        add2(point, v2, bicomponents, bptr);
        eptr := eptr − 2;
        comment Compute number of edges of component;
        bicomponents[oldptr] := (bptr-oldptr) ÷ 2
      end
    end
    else
    begin
```

```
        comment New point not found. Recalculate lowpoint;
        lowpoint := min(lowpoint, number[v2])
      end
    end
  end;
  comment Construct the structural representation of the graph;
  freenext := v;
  for i := 1 step 1 until v do next [i] := 0;
  for i := 1 step 1 until e do
  begin
    comment Each edge occurs twice, once for each endpoint;
    nextlink(edgelist[2×i−1], edgelist[2×i]);
    nextlink(edgelist[2×i], edgelist[2×i−1])
  end;
  comment Initialize variables for search;
  eptr := 0; bptr := 0; point := 1; v2 := 0;
  for i := 1 step 1 until v + 1 do number[i] := 0;
  for i := i while point ≤ v do
  begin
    comment Each execution of biconnector searches a connected
      component of the graph. After each search, find an unnumbered
      vertex and search again. Repeat until all vertices are
      examined;
    number[point] := code := 1; newlowpt := v + 1;
    biconnector(point, v2, newlowpt);
    for i := i while number[point] ¬ ≠ 0 do point := point + 1
  end
end;
procedure pathfinder (startpoint, pathpt, codevalue, path);
  integer startpoint, pathpt, codevalue;
  integer array path;
begin
  comment Procedure finds disjoint paths with arbitrary starting
    points in a biconnected graph. The points of each path are
    listed in the array path. The following variables are assumed
    global.  next[1:v+2×e],  head[v+1:v+2×e]  and  link
    [v+1:v+2×e] define the graph using singly linked edge
    lists and a set of cross reference pointers. old[1:v] and mark
    [v+1:v+2×e] indicate used points and edges. pathcode[1:v]
    is the consecutive numbering of the points. lowpoint[1:v],
    forward[1:v] and back[1:v] give information saved from depth-
    first search, node[1:v] gives the next unsearched edge from each
    point;
  integer point, pastedge, edge, pastpoint, v2, i;
  path[1] := startpoint;
  comment Choose initial edge;
  edge := next[startpoint];
  for i := i while (if edge=0 then false else mark[edge])
    do edge := next[edge]:
  begin
    comment No unused edge and thus no path exists:
    next[start point] := 0; pathpt := 0:
    go to done
  end;
  next[startpoint] := next[edge]; path[2] := edge;
  point := head[edge]; pathpt := 2;
  if old[point] then go to pathfound;
  if forward[point]≠ 0 then
  begin
    comment Use previously found information to build a path.
      forward, back, lowpoint describe trees investigated using
      depth-first search;
    if pathcode[startpoint] > pathcode[point] then
      go to nextback;
nextmark:
    if pathcode[startpoint] > lowpoint[point] then
    begin
nextforward:
      edge := forward[point]; point := head[edge];
      pathpt := pathpt +1; path[pathpt]: = edge;
      if old[point] then go to pathfound;
      if pathcode[startpoint] > pathcode[point]
        then go to nextback;
      go to nextforward
    end;
    edge := back[point]; point := head[edge];
    pathpt := pathpt + 1; path[pathpt] := edge;
    if old[point] then go to pathfound else
      go to nextmark;
nextback:
    edge := back[point]; point := head[edge];
    pathpt := pathpt + 1; path[pathpt] := edge;
    if old[point] then go to pathfound else
      go to nextback
  end;
  comment Use depth-first search to find a path. Save information
    describing search tree;
nextpoint:
  codevalue := codevalue + 1; pathcode[point] := codevalue;
nextedge:
  edge := node[point];
  for i := i while edge = 0 do
  begin
    back[point]  := link[path[pathpt]];
    pastpoint := head[back[point]];
    if (forward[pastpoint] = 0) ∨
      (lowpoint[point] < lowpoint[pastpoint]) then
    begin
      forward[pastpoint] := path[pathpt];
      lowpoint[pastpoint] := lowpoint[point]
    end;
    point := pastpoint; pathpt := pathpt − 1; edge := node[point]
  end;
  node[point] := next[edge]; v2 := head[edge];
  if pathcode[v2] = 0 then
  begin
    point := v2; pathpt := pathpt + 1;
    path[pathpt] := edge; go to nextpoint
  end;
  if old[v2] ∧ (v2≠startpoint) then
  begin
    pathpt := pathpt + 1; path[pathpt] := edge;
    go to pathfound
  end;
  if (forward[point]=0) ∨ (pathcode[v2] < lowpoint[point]) then
  begin
    forward[point] := edge; lowpoint[point] := pathcode[v2]
  end;
  go to nextedge;
  comment Path found. Convert stack of edges to list of points in
    path. Mark all edges and points in path;
  pathfound:
  for i := 2 step 1 until pathpt do
  begin
    edge := path [i]; point := head[edge];
    forward[point] := back[point]: = 0; old[point] := true;
    mark[link[edge]] := mark [edge] := true;
    path [i] := point
  end;
done:
end
```

Algorithm 448

Number of Multiply-Restricted Partitions [A1]

Terry Beyer* and D.F. Swinehart† (Recd. 1 Jan. 1971 and 28 June 1971)

* Computer Science Department and Computing Center, University of Oregon, Eugene, OR 97403.

† Department of Chemistry, University of Oregon, Eugene, OR 97403.

Key Words and Phrases: partitions, enumeration, change making, energy-level degeneracies, molecular vibrational energy-levels

CR Categories: 3.13, 5.30

Language: Fortran

Description

Given a positive integer m and an ordered k-tuple $c = (c_1, \cdots, c_k)$ of not necessarily distinct positive integers, then any ordered k-tuple $s = (s_1, \cdots, s_k)$ of nonnegative integers such that $m = \sum_{i=1}^{k} s_i c_i$ is said to be a partition of m restricted to c. Let $P_c(m)$ denote the number of distinct partitions of m restricted to c. The subroutine *COUNT*, when given a k-tuple c and an integer n, computes an array of the values of $P_c(m)$ for $m = 1$ to n. Many combinatorial enumeration problems may be expressed in terms of the numbers $P_c(m)$. We mention two below.

Applications: Change making. Letting $c = (1,5,10)$ and $n = 100$, the subroutine computes the number of ways of making each amount of change from one cent to one dollar using pennies, nickels, and dimes. Letting $c = (1,5,5,10)$ corresponds to using two distinct types of nickels.

Applications: Chemistry. This algorithm is of some importance to problems in chemistry. In the theory of unimolecular reactions [2,6] a quantity appears, $\sum_{e_v=0}^{e} P(e_v)$, in which $P(e_v)$ is the number of ways a given amount of vibrational energy, e_v, may be distributed among the quantized vibrational modes of a polyatomic molecule, assuming all of these modes to be harmonic. Setting $m = e_v$ and $c = (c_1, \cdots, c_k)$, where c_i is the energy corresponding to the fundamental frequency of the ith vibrational mode, then s_i becomes the corresponding vibrational quantum number and we have $P(e_v) = P_c(m)$. The desired quantity $\sum_{e_v=0}^{e} P(e_v)$ may thus be readily obtained from the output of the subroutine *COUNT*. No algorithm previously available has been sufficiently efficient for calculating these sums directly. Various functions have been proposed as approximations for this calculation [5]. The present algorithm allows calculation of $\sum P(e_v)$ directly and efficiently.

Method. Input to *COUNT* is a positive integer N and an integer array C containing K entries. Output is the array P containing N integers where $P(M)$ is the number of partitions of M restricted to C for $M = 1$ to N. The following assumptions are made concerning the input; (1) K is positive: (2) C contains positive integers only; and (3) N is greater than the maximum value in C. Restriction 3 is not inherent in the problem but is a restriction required by *COUNT*. The algorithm operates by initializing P to contain the number of partitions of an integer restricted to an empty sequence. Each pass through the outer loop which follows, updates P to reflect an additional element of C by using the recursive relations

$$P_{(c_1,\cdots,c_i)}(m) = \begin{cases} P_{(c_1,\cdots,c_{i-1})}(m) & \text{if } m < c_i\,, \\ P_{(c_1,\cdots,c_{i-1})}(m) + 1 & \text{if } m = c_i\,, \\ P_{(c_1,\cdots,c_{i-1})}(m) + P_{(c_1,\cdots,c_i)}(m - c_i) & \text{if } m > c_i\,. \end{cases}$$

These equations are derived by counting additional partitions of m obtained by using c_i. Thus if $m < c_i$, no additional partitions are obtained. If $m = c_i$, the single additional partition consisting of c_i is obtained. If $m > c_i$, then any partition of m involving c_i comes from a partition of $(m - c_i)$ which involves one less occurrence of c_i. Readers may wish to refer to [3 and 4] which contain recurrence algorithms for more classical forms of the partition enumeration problem of which the problem presented here is a generalization.

Scaling. The time required by the algorithm is roughly proportional to $k \times n$. If the integers $c_1, \cdots, c_k$ have a common divisor d, the results may be obtained approximately d times as quickly by making use of the relations

$$P_c(m) = \begin{cases} 0 & \text{if } d \nmid m \\ P_{c/d}(m/d) & \text{if } d \mid m \end{cases}$$

where $c/d = (c_1/d, \cdots, c_k/d)$. The computation of $P_{c/d}(m/d)$ for $m/d = 1$ to n/d will require time proportional to $k \times (n/d)$ and an array of dimension n/d rather than n. *COUNT* does not automatically perform this scaling.

Accuracy. The algorithm itself is precise. However in typical applications to chemistry the numbers $P(M)$ generated may exceed the magnitude limitation for Fortran integers. In this case one may simulate multiple precision integer arithmetic and continue to obtain precise results, or one may switch to floating point. In the latter case, roundoff errors will be introduced into the calculation. The authors have not investigated the accumulation of roundoff errors under these conditions.

Test cases. The subroutine *COUNT* has been tested on the following compiler/computer combinations.

IBM FORTRAN IV(G)	IBM S/360 (Mod. 50)
University of Waterloo WATFOR	IBM S/360 (Mod. 50)

Results for several change counting problems were compared with results from hand calculations. Results for the special case of unrestricted partitions were compared to published table values [1].

Acknowledgment. The authors wish to thank the University of Oregon Computing Center for supporting the testing of this work.

References

1. Hall, M. Jr. *Combinatorial Theory*. Blaisdell, Waltham, Mass., 1967, pp. 29–35.
2. Marcus, R.A., and Rice, O.K. The Kinetics of the recombination of methyl radicals and iodine atoms. *J. Physical and Colloid Chem. 55* (June 1951), 894–908.
3. McKay, J.K.S. Algorithm 262, Number of restricted partitions of N. *Comm. ACM 8* (Aug. 1965), 493.
4. White, J.S. Algorithm 373. Number of doubly restricted partitions. *Comm. ACM 13* (Feb. 1970), 120.
5. Whitten, G.Z., and Rabinovitch, B.S. Accurate and facile approximation for vibrational energy-level sums. *J. Chem. Phys. 38* (15 May 1963), 2466–2473.
6. Wieder, G.M., and Marcus, R.A. Dissociation and isomerization of vibrationally excited species. II. Unimolecular

reaction rate theory and its application. *J. Chem. Phys. 37* (15 Oct. 1962), 1835–1852.

Algorithm

```
      SUBRØUTINE CØUNT(C, K, P, N)
      INTEGER C, P
      DIMENSIØN C(K), P(N)
C CØUNT COMPUTES THE NUMBER ØF PARTITIONS ØF AN INTEGER
C RESTRICTED TØ C FØR INTEGERS IN THE RANGE 1 TO N.
C INPUT:  K   -- A PØSITIVE INTEGER
C         C   -- AN ARRAY ØF K PØSITIVE INTEGERS
C         N   -- AN INTEGER LARGER THAN THE MAXIMUM VALUE IN C
C ØUTPUT: P   -- AN ARRAY ØF N INTEGERS, WHERE P(M) IS THE
C                NUMBER ØF PARTITIONS ØF M RESTRICTED TØ C
C INITIALIZE P
      DØ 10 I=1,N
        P(I) = 0
   10 CØNTINUE
C EACH PASS THRØUGH THE ØUTER LØØP BELØW TRANSFØRMS P FRØM
C PARTITIØNS RESTRICTED TØ C(1), ... , C(I-1) TØ
C PARTITIØNS RESTRICTED TØ C(1), ... , C(I).
      DØ 30 I=1,K
        J = C(I)
        JP1 = J + 1
        P(J) = P(J) + 1
        DØ 20 M=JP1,N
          MMJ = M - J
          P(M) = P(M) + P(MMJ)
   20   CØNTINUE
   30 CØNTINUE
      RETURN
      END
```

Algorithm 449 Solution of Linear Programming Problems in 0-1 Variables [H]

František Fiala [Recd. 5 Feb. 1971]
Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada*

Key Words and Phrases: linear programming, zero-one variable
CR Categories: 5.41
Language: Fortran

Description

This subroutine solves the linear zero-one programming problem of the following form.

Find the maximum and all maximizing points of the objective function

$$f = a_{11}x_1 + \cdots + a_{1n}x_n + a_{10} \tag{1}$$

subject to

$$a_{i1}x_1 + \cdots + a_{in}x_n \geq b_i \, , \; i = 2, \ldots, m, \tag{2.i}$$

where $x_j = 0$ or 1; a_{ij} , b_i are integer coefficients.

The algorithm follows the procedure described in [1, 2].

First of all we add a supplementary constraint

$$a_{11}x_1 + \cdots + a_{1n}x_n \geq b_1 \, , \tag{2.1}$$

where b_1 is equal either to the value of $f - a_{10}$ for a solution to the system of constraints, or to a lower bound of $f - a_{10}$. As soon as we find a feasible solution to the system of constraints, we replace b_1 by the corresponding value of $f - a_{10}$. Consequently, if a feasible solution is found, then the following procedure can lead only to solutions with the same or better value of the objective function. Using the formula $x = 1 - \bar{x}$, we bring (2.1) into the form

$$a'_{1j_1}\tilde{x}_{j_1} + \cdots + a'_{1j_n}\tilde{x}_{j_n} \geq b_1', \tag{2'.1}$$

$$\text{with } a'_{1j_1} \geq \cdots \geq a'_{1j_n} \geq 0, \tag{3}$$

where $\tilde{x}$ is either x or $\bar{x}$. If there are coefficients with the same absolute value in (2.1), then their order in (2'.1) corresponds with that in (2.1). The order of coefficients in (2'.1) indicates the order of branching points. Coefficients in (2'.1) are used in the accelerating test.

At every stage of the procedure we have a partial solution and the corresponding (current) problem derived from the original one. In the partial solution, some variables are assigned fixed values (0 or 1) and the others remain free. The partial solution corresponding to the original problem has all variables free. A partial solution is completed if all variables are fixed.

Given a partial solution we try to complete it. If there is a completion, we change the supplementary constraint and backtrack. If there is no completion, we backtrack. In both cases we go back to the last branching point and examine the new partial solution with the complementary value for the branching variable. We use the accelerating test if applicable. As a result we find either all maximizing points and maximum of f or the problem has no solutions.

Accelerating test. Suppose that at a certain step we have a partial solution with the fixed variables x_{j_h} , $h \in H \subseteq \{1, \ldots, n\}$, and we have to branch. We take the first variable $\tilde{x}_{j_0}$ still occurring in (2'.1)—branching variable—and put first $\tilde{x}_{j_0} = 1$ and then $\tilde{x}_{j_0} = 0$. We examine the new partial solution with $\tilde{x}_{j_0} = 1$. If there is a feasible completion of the partial solution and if

$$a'_{1j_0} > \sum_{k \epsilon K} a'_{1j_k} \, , \tag{4}$$

where K is the set of all indices $k \epsilon \{1, \ldots, n\} - H$ such that $\tilde{x}_{j_k} = 0$ in the completion, then the branch with $\tilde{x}_{j_0} = 0$ can be dropped out.

The subroutine *MAXL*01 is self-contained, and communication to it is through the argument list. The calling statement is of the following form

*CALL MAXL*01 (*MO, NO, NEST, M, N, AO, BO, A, B, B*1, *S*1, *C, X, S, BC, T, IND, INC, NESTEX, V, NOPT, OPTS, NI, NAT*)

The meaning of the parameters is described in the comments at the beginning of the subroutine. Here the meaning of only two output parameters is explained. *INC* = 0 or 1 means that the problem has feasible solutions or not, respectively. As we have to estimate the number *NEST* of all alternative optimal solutions in advance (as to define the array *OPTS*), *NESTEX* = 1 or 0 indicates whether the estimated number is exceeded or not, respectively. Consequently, after return from the subroutine we have to examine first the values of *INC* and *NESTEX* in order to give the proper answer.

Test results. The subroutine has been tested on an IBM 360/67. No breakdown of the method has occurred. Further details about the computational experience are given in [1].

Two examples.

(*i*) The objective function:

$f = 2x_1 + 5x_2 + 4x_3 + x_4 - 3x_5 - x_6 + 3.$

The constraints;

$$\begin{array}{l}
2x_1 - x_2 + 3x_3 \qquad\qquad + 5x_5 - 2x_6 \geq 3 \\
\qquad 4x_2 - 7x_3 + 3x_4 + x_5 - x_6 \geq -9 \\
x_1 + 8x_2 \qquad\qquad + 4x_4 + 2x_5 + 3x_6 \geq 7 \\
5x_1 - 2x_2 + 4x_3 + 3x_4 \qquad\qquad - 5x_6 \geq -5 \\
x_1 - x_2 \qquad\qquad + x_4 \qquad\qquad + x_6 \geq 0
\end{array}$$

Maximum: 15. Maximizing point: (1, 1, 1, 1, 0, 0).
Iterations: 5. Accelerating test: 3.

(*ii*) The objective function:

$f = 2x_1 - x_2 + 4x_3 + 7x_4 - 5x_5 + 12x_6 + 9x_7 - 4x_8 - x_9 + 2x_{10} + 5.$

The constraints:

$$\begin{array}{l}
3x_1 - x_2 + 2x_3 + 4x_6 - 3x_7 + 8x_8 + x_9 \geq 5 \\
4x_2 + 7x_3 + x_4 + 2x_5 - 5x_6 + 3x_9 + 9x_{10} \geq 1 \\
x_1 - x_2 + 3x_4 + 7x_5 + 8x_6 + 5x_7 - x_8 - 7x_9 + 4x_{10} \geq 12 \\
2x_1 + 4x_3 - x_4 + 4x_8 + 5x_9 + 3x_{10} \geq 2
\end{array}$$

Maximum: 41. Maximizing point: (1, 0, 1, 1, 0, 1, 1, 0, 0, 1).
Iterations: 9. Accelerating test: 7.

References

1. Fiala, F. Computational experience with a modification of an algorithm by Hammer and Rudeanu for linear 0-1 Programming. Proc. ACM 1971 Nat. Conf. ACM, New York, pp. 482–488.

2. Hammer, P.L., and Rudeanu, S. *Boolean Methods in Operations Research and Related Areas.* Springer-Verlag, New York, 1968.

* This research was done while the author was a post-doctorate fellow with the Department. Present address: Department of Mathematics, Carleton University, Colonel By Drive, Ottawa, Ontario, Canada.

Algorithm

```
      SUBRØUTINE MAXL01(MØ, NØ, NEST, M, N, AØ, BØ, A, B, B1,
     * S1, C, X, S, SØ, BC, T, IND, INC, NESTEX, V, NØPT, ØPTS,
     * NI, NAT)
      INTEGER AØ(MØ,NØ), A(MØ,NØ), BØ(MØ), B(MØ), B1(MØ),
     * S1(MØ), C(NØ), X(NØ), S(NØ), SØ(NØ), BC(NØ), T(NØ),
     * IND(MØ), V, VNEG, ØPTS(NEST,NØ)
C THIS SUBRØUTINE FINDS THE MAXIMUM AND ALL MAXIMIZING
C PØINTS TØ THE LINEAR ØBJECTIVE FUNCTIØN (1) SUBJECT TØ M-1
C LINEAR CØNSTRAINTS (2.I) WITH N(GREATER THAN 1) ZERØ-ØNE
C VARIABLES AND INTEGER CØEFFICIENTS.
C THE MEANING ØF THE INPUT PARAMETERS.
C MØ, NØ, NEST ARE THE ADJUSTABLE DIMENSIØNS SPECIFYING THE
C UPPER BØUNDS FØR THE NUMBER ØF ALL CØNSTRAINTS, VARIABLES
C AND ALTERNATIVE ØPTIMAL SØLUTIØNS, RESPECTIVELY.
C M IS THE NUMBER ØF CØNSTRAINTS INCLUDING THE SUPPLEMENTARY
C ØNE. N IS THE NUMBER ØF THE VARIABLES. AØ IS THE TWØ-
C DIMENSIØNAL ARRAY CØNTAINING IN THE FIRST M RØWS AND N
C CØLUMNS THE CØEFFICIENTS ØF ALL CØNSTRAINTS. THE FIRST RØW
C CØNTAINS THE CØEFFICIENTS ØF THE SUPPLEMENTARY CØNSTRAINT.
C THE ØNE-DIMENSIØNAL ARRAY BØ CØNTAINS THE RIGHT-HAND SIDE
C TERMS ØF THE CØNSTRAINTS. BØ(1) IS THE ABSØLUTE TERM ØF
C THE ØBJECTIVE FUNCTIØN. AØ, BØ REMAIN UNCHANGED DURING
C THE WHØLE PRØCEDURE.
C THE MEANING ØF THE AUXILIARY PARAMETERS.
C THE TWØ-DIMENSIØNAL ARRAY A ØR THE ØNE-DIMENSIØNAL ARRAY B
C CØNTAINS THE CØEFFICIENTS ØR THE RIGHT-HAND SIDE TERMS
C ØF THE CURRENT SYSTEM ØF CØNSTRAINTS, RESPECTIVELY.
C VNEG IS THE SUM ØF ALL NEGATIVE CØEFFICIENTS IN THE
C ØBJECTIVE FUNCTIØN MINUS 1.
C ITEST=1 ØR 0 INDICATES IF THE WHØLE SYSTEM ØF CØNSTRAINTS
C IS REDUNDANT ØR NØT, RESPECTIVELY. SIMILARLY, THE I-TH
C CØMPØNENT ØF THE ØNE-DIMENSIØNAL ARRAY IND INDICATES
C WHETHER THE I-TH CØNSTRAINT IS REDUNDANT ØR NØT.
C THE ØNE-DIMENSIØNAL ARRAY X CØNTAINS THE CURRENT PARTIAL
C SØLUTIØN. A FREE VARIABLE IS REPRESENTED BY A CØMPØNENT
C EQUAL TØ 2.
C THE ØNE-DIMENSIØNAL ARRAY S ØR BC ØR T INDICATES THE ØRDER
C AND MANNER IN WHICH THE FIXED VARIABLES WERE ASSIGNED
C THEIR VALUES ØR THE BRANCHING PØINTS ØR THE BRANCHING
C PØINTS IN WHICH THE ACCELERATING TEST CAN BE APPLIED,
C RESPECTIVELY. NS IS THE NUMBER ØF CØMPØNENTS IN S AND BC.
C THE ØNE-DIMENSIØNAL ARRAYS B1,S1,SØ AND C HAVE AN
C AUXILIARY CHARACTER.
C THE MEANING ØF THE ØUTPUT PARAMETERS.
C INC=0 ØR 1 MEANS THAT THE GIVEN PRØBLEM IS CØNSISTENT
C ØR INCØNSISTENT, RESPECTIVELY.
C NESTEX=0 ØR 1 INDICATES THAT THE ESTIMATED NUMBER ØF
C FEASIBLE SØLUTIØNS WAS NØT ØR WAS EXCEEDED, RESPECTIVELY.
C V IS THE MAXIMAL VALUE ØF THE ØBJECTIVE FUNCTIØN.
C NØPT IS THE NUMBER ØF ALL MAXIMIZING PØINTS.
C THE TWØ-DIMENSIØNAL ARRAY ØPTS CØNTAINS IN THE FIRST NØPT
C RØWS ALL MAXIMIZING PØINTS. A CØMPØNENT MAY BE EQUAL TØ 2
C WHICH INDICATES THAT THE VALUE ØF THE CØRRESPØNDING
C VARIABLE CAN BE ARBITRARY. NI ØR NAT INDICATE THE NUMBER
C ØF ITERATIØNS ØR  THE NUMBER ØF SUCCESSFUL APPLICATIØNS ØF
C THE ACCELERATING TEST, RESPECTIVELY.
C THE CALLING PRØGRAM SHØULD CØNTAIN THE FØLLØWING TYPE-
C STATEMENT
C INTEGER AØ(MØ,NØ),A(MØ,NØ),BØ(MØ),B(MØ),B1(MØ),S1(MØ),
C XC(NØ),X(NØ),S(NØ),SØ(NØ),BC(NØ),T(NØ),IND(MØ),V,
C XØPTS(NEST,NØ)
      INC = 0
      NESTEX = 0
      NØPT = 0
      NS = 0
      NI = 0
      NAT = 0
      DØ 10 J=1,N
        T(J) = 0
   10 CØNTINUE
C CØPY THE ARRAYS AØ, BØ.
      DØ 30 I=1,M
        B(I) = BØ(I)
        DØ 20 J=1,N
          A(I,J) = AØ(I,J)
   20   CØNTINUE
   30 CØNTINUE
C ADD THE SUPPLEMENTARY CØNSTRAINT, DETERMINE THE INITIAL
C PARTIAL SØLUTIØN.
      VNEG = -1
      DØ 40 J=1,N
        X(J) = 2
        IF (A(1,J).LT.0) VNEG = VNEG + A(1,J)
   40 CØNTINUE
      B(1) = VNEG
      V = VNEG
   50 DØ 60 I=1,M
        IND(I) = 0
   60 CØNTINUE
C EXAMINE THE CURRENT SYSTEM ØF CØNSTRAINTS.
   70 DØ 80 I=1,M
        B1(I) = B(I)
   80 CØNTINUE
      NI = NI + 1
      ITEST = 1
      DØ 110 I=1,M
        S1(I) = 0
        IF (IND(I).EQ.1) GØ TØ 110
        DØ 90 J=1,N
          IF (A(I,J).LT.0) B1(I) = B1(I) - A(I,J)
          S1(I) = S1(I) + IABS(A(I,J))
   90   CØNTINUE
        IF (B1(I).LE.0) GØ TØ 100
        ITEST = 0
        GØ TØ 110
  100   IND(I) = 1
  110 CØNTINUE
      IF (ITEST.EQ.1) GØ TØ 420
C THE SYSTEM  CØNTAINS AT LEAST ØNE IRREDUNDANT INEQUALITY.
      DØ 120 I=1,M
        IF (IND(I).EQ.1) GØ TØ 120
        IF (S1(I)-B1(I).LT.0) GØ TØ 560
  120 CØNTINUE
C THE SYSTEM DØES NØT CØNTAIN ANY INCØNSISTENT INEQUALITY.
C CØNSIDER EACH INEQUALITY SEPARATELY.
      I = 1
  130 IF (IND(I).EQ.1) GØ TØ 360
      IF (S1(I)-B1(I).GT.0) GØ TØ 200
C SØME ØF THE FREE VARIABLES ARE FØRCED TØ CERTAIN FIXED
C VALUES.
  140 DØ 190 J=1,N
        IF (A(I,J).EQ.0) GØ TØ 190
        NS = NS + 1
        BC(NS) = 1
        IF (A(I,J).LT.0) GØ TØ 160
        S(NS) = J
        X(J) = 1
        DØ 150 IJ=1,M
          B(IJ) = B(IJ) - A(IJ,J)
  150   CØNTINUE
        GØ TØ 170
  160   S(NS) = -J
        X(J) = 0
  170   DØ 180 IJ=1,M
          A(IJ,J) = 0
  180   CØNTINUE
  190 CØNTINUE
      GØ TØ 70
  200 DØ 210 J=1,N
        C(J) = IABS(A(I,J))
  210 CØNTINUE
      L1 = 1
  220 J = L1 + 1
  230 IF (C(L1).GE.C(J)) GØ TØ 240
      IP = C(L1)
      C(L1) = C(J)
      C(J) = IP
  240 J = J + 1
      IF (J.GT.N) GØ TØ 250
      GØ TØ 230
  250 L1 = L1 + 1
      IF (L1.LT.N) GØ TØ 220
  260 IF (C(L1).GT.0) GØ TØ 270
      L1 = L1 - 1
      GØ TØ 260
  270 IF (S1(I)-C(L1).LT.B1(I)) GØ TØ 140
      IF (S1(I)-C(1)-B1(I).GE.0) GØ TØ 360
C ØNE FREE VARIABLE IS FØRCED TØ A CERTAIN FIXED VALUE.
      NS = NS + 1
      BC(NS) = 1
  280 DØ 290 J=1,N
        IF (IABS(A(I,J)).EQ.C(1)) GØ TØ 300
  290 CØNTINUE
  300 IF (A(I,J).LT.0) GØ TØ 330
  310 S(NS) = J
      X(J) = 1
      DØ 320 IJ=1,M
        B(IJ) = B(IJ) - A(IJ,J)
  320 CØNTINUE
      GØ TØ 340
  330 S(NS) = -J
      X(J) = 0
  340 DØ 350 IJ=1,M
        A(IJ,J) = 0
  350 CØNTINUE
      GØ TØ 70
  360 I = I + 1
      IF (I.LE.M) GØ TØ 130
      IF (NS.EQ.N) GØ TØ 480
C FIND A NEW BRANCHING PØINT.
      DØ 370 J=1,N
        C(J) = IABS(A(1,J))
  370 CØNTINUE
      DØ 380 J=2,N
        IF (C(1).GE.C(J)) GØ TØ 380
        C(1) = C(J)
  380 CØNTINUE
      IF (C(1).EQ.0) GØ TØ 390
      NS = NS + 1
      BC(NS) = 0
      I = 1
      GØ TØ 280
  390 DØ 410 J=1,N
        DØ 400 J1=1,NS
          IF (J.EQ.IABS(S(J1))) GØ TØ 410
  400   CØNTINUE
        NS = NS + 1
        BC(NS) = 0
        GØ TØ 310
  410 CØNTINUE
C THE SYSTEM ØF CØNSTRAINTS IS REDUNDANT. SØLVE AN
C UNCØNSTRAINED PRØBLEM.
  420 DØ 470 J=1,N
        IF (NS.EQ.N) GØ TØ 480
        IF ((X(J).NE.2) .ØR. (A(1,J).EQ.0)) GØ TØ 470
        NS = NS + 1
        BC(NS) = 1
        IF (A(1,J).LT.0) GØ TØ 440
        S(NS) = J
        X(J) = 1
        DØ 430 I=1,M
          B(I) = B(I) - A(I,J)
  430   CØNTINUE
        GØ TØ 450
  440   S(NS) = -J
        X(J) = 0
  450   DØ 460 I=1,M
          A(I,J) = 0
  460   CØNTINUE
  470 CØNTINUE
C FIND THE NEW VALUE ØF THE ØBJECTIVE FUNCTIØN.
C ADJUST THE ACCELERATING TEST SEQUENCE T.
  48Q NEWV = 0
      DØ 490 J=1,N
        NEWV = NEWV + X(J)*AØ(1,J)
  490 CØNTINUE
      DØ 500 J=1,NS
        K = NS + 1 - J
        IF (BC(K).EQ.0) T(K) = 1
  500 CØNTINUE
      IF (NEWV.GT.V) GØ TØ 510
      NØPT = NØPT + 1
      IF (NØPT.LE.NEST) GØ TØ 540
C THE ESTIMATED FIRST DIMENSIØN ØF THE ARRAY ØPTS IS
C EXCEEDED.
      NESTEX = 1
      RETURN
C THE NEW SØLUTIØN FØUND GIVES A BETTER VALUE TØ THE
C ØBJECTIVE FUNCTIØN. CHANGE THE SUPPLEMENTARY CØNSTRAINT.
  510 NØPT = 1
      V = NEWV
      B(1) = V
      DØ 520 J=1,N
        IF (X(J).NE.1) GØ TØ 520
        B(1) = B(1) - AØ(1,J)
  520 CØNTINUE
      DØ 530 J=1,N
        SØ(J) = S(J)
  530 CØNTINUE
C MØDIFY THE SET ØPTS.
  540 DØ 550 J=1,N
        ØPTS(NØPT,J) = X(J)
  550 CØNTINUE
  560 IF (NS.EQ.0) GØ TØ 580
C QUESTIØN IF A BACKTRACKING IS PØSSIBLE.
      IS = 0
      DØ 570 J=1,NS
        IS = IS + BC(J)
  570 CØNTINUE
      IF (IS.LT.NS) GØ TØ 600
      IF (V.GT.VNEG) GØ TØ 590
C THE SYSTEM ØF CØNSTRAINTS IS INCØNSISTENT. NØ SØLUTIØNS.
  580 INC = 1
      RETURN
C THE GIVEN PRØBLEM HAS A SØLUTIØN. ALL SØLUTIØNS HAVE BEEN
C FØUND.
  590 V = V + BØ(1)
      RETURN
C THE BACKTRACKING IS PØSSIBLE.
  600 DØ 610 J1=1,NS
        K = NS + 1 - J1
        IF (BC(K).EQ.0) GØ TØ 620
  610 CØNTINUE
  620 IF (T(K).EQ.1) GØ TØ 750
C BACKTRACK.
  630 DØ 740 J1=K,NS
        DØ 640 J=1,N
          IF (J.EQ.IABS(S(J1))) GØ TØ 650
  640   CØNTINUE
  650   IF (K.EQ.J1) GØ TØ 700
        IF (X(J).EQ.1) GØ TØ 670
        DØ 660 I=1,M
          A(I,J) = AØ(I,J)
  660   CØNTINUE
        GØ TØ 690
  670   DØ 680 I=1,M
          A(I,J) = AØ(I,J)
          B(I) = B(I) + A(I,J)
  680   CØNTINUE
  690   X(J) = 2
        GØ TØ 740
  700   S(K) = -S(K)
        BC(K) = 1
        X(J) = 1 - X(J)
        IF (X(J).EQ.0) GØ TØ 720
        DØ 710 I=1,M
          B(I) = B(I) - AØ(I,J)
  710   CØNTINUE
        GØ TØ 740
  720   DØ 730 I=1,M
          B(I) = B(I) + AØ(I,J)
  730   CØNTINUE
  740 CØNTINUE
      NS = K
      GØ TØ 50
C THE ACCELERATING TEST.
  750 T(K) = 0
      IT1 = 0
      IT2 = 0
      DØ 790 J1=K,N
        DØ 760 J=1,N
          IF (J.EQ.IABS(SØ(J1))) GØ TØ 770
  760   CØNTINUE
  770   IF (K.EQ.J1) GØ TØ 780
        IF (((X(J).EQ.0) .AND. (AØ(1,J).GT.0)) .ØR.
     * ((X(J).EQ.1) .AND. (AØ(1,J).LT.0))) IT2 = IT2 +
     * IABS(AØ(1,J))
        GØ TØ 790
  780   IT1 = IABS(AØ(1,J))
  790 CØNTINUE
      IF (IT1.LE.IT2) GØ TØ 630
C THE APPLICATIØN ØF THE ACCELERATING TEST WAS SUCCESSFUL.
      BC(K) = 1
      NAT = NAT + 1
      GØ TØ 560
      END
```

Editor's note: *Algorithm 449 described here is available on magnetic tape from the Department of Computer Science, University of Colorado, Boulder, CO 80302. The cost for the tape is $16.00 (U.S. and Canada) or $18.00 (elsewhere). If the user sends a small tape (wt. less than 1 lb.) the algorithm will be copied on it and returned to him at a charge of $10.00 (U.S. only). All orders are to be prepaid with checks payable to ACM Algorithms. The algorithm is recorded as one file of BCD 80 character card images at 556 B.P.I., even parity, on seven track tape. We will supply algorithm at a density of 800 B.P.I. if requested. Cards for algorithms are sequenced starting at 10 and incremented by 10. The sequence number is right justified in column 80. Although we will make every attempt to insure that the algorithm conforms to the description printed here, we cannot guarantee it, nor can we guarantee that the algorithm is correct.—L.D.F. and A.K.C.*

Algorithm 450

Rosenbrock Function Minimization [E4]

Marek Machura* and Andrzej Mulawa†

[Recd. 22 March 1971]

* Institute of Automation and Measurements, Warsaw, Poland.
† Institute of Computing Machinery, Warsaw, Poland.

Key words and phrases: function minimization, Rosenbrock's method
CR Categories : 5.19
Language : Fortran

Description

Purpose. This subroutine finds the local minimum of a function of *n* variables for an unconstrained problem. It uses the method for direct search minimization as described by Rosenbrock [1].

Method. The local minimum of a function is sought by conducting cyclic searches parallel to each of the *n* orthogonal unit vectors, the coordinate directions, in turn. *n* such searches constitute one stage of the iteration process. For the next stage a new set of *n* orthogonal unit vectors is generated, such that the first vector of this set lies along the direction of greatest advance for the previous stage. The Gram-Schmidt orthogonalization procedure is used to calculate the new unit vectors.

Program. The communication to the subroutine *ROMIN* is solely through the argument list. The user must supply two additional subroutines *FUNCT* and *MONITOR*. The entrance to the subroutine is achieved by

CALL ROMIN (N, X, FUNCT, STEP, MONITOR)

The meaning of the parameters is as follows. *N* is the number of independent variables of the function to be minimized. *X(N)* is an estimate of the solution. On entry it is an initial estimate to be provided by the user; on exit it is the best estimate of the solution found. *FUNCT (N, X, F)* is a subroutine calculating the value *F* of the minimized function at any point *X*. *STEP* is an initial step length for all searches of the first stage. The subroutine *MONITOR (N, X, F, R, B, CON, NR)* supplies printouts of any parameter from the argument list and contains convergence criteria chosen by the user. (Different kinds of convergence criteria and their use are discussed in [1] and [4].) *R* is the actual number of function evaluations. *B* is the value of the Euclidean norm of the vector representing the total progress made since the axes were last rotated, i.e. the total progress in one stage. *CON* is a logical variable. At the start of the subroutine *ROMIN CON* is set *.FALSE.*. If the convergence criteria are satisfied *CON* must be set *.TRUE.* in the subroutine *MONITOR*, which transfers control back to the main program. *NR* is the *MONITOR* index used as described in [3]. The *CALL* statement of the subroutine *MONITOR* with *NR* equal to 1 occurs once per function evaluation and with *NR* equal to 2 once per stage of the iteration process.

Test results. As a test example, the parabolic valley function $f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$ was chosen. This function attains its minimum equal to 0 at the point (1, 1). Starting from the point (−1.2, 1.0) the best estimate of the solution after 200 function evaluations as found by the subroutine *ROMIN* was $0.29774 \cdot 10^{-4}$ at the point (0.99513, 0.99053). The initial step length *STEP* was set equal to 0.1 [2].

References

1. Rosenbrock, H.H. An automatic method for finding the greatest or least value of a function. *Computer J. 3* (1960), 175–184.
2. Rosenbrock, H.H., Storey, C. *Computational Techniques for Chemical Engineers.* Pergamon Press, New York, 1966.
3. Rutishauser, H. *Interference with an ALGOL Procedure, in Annual Review in Automatic Programming, Vol. 2.* R. Goodman (Ed.), Pergamon Press, New York, 1961.
4. Powell, M.J.D. An efficient method for finding the minimum of a function of several variables without calculating derivatives. *Computer J. 6* (1964), 155–162.

Algorithm

```
      SUBRØUTINE RØMIN(N, X, FUNCT, STEP, MØNITR)
      INTEGER N, IP
      REAL STEP
      DIMENSIØN X(N)
      LØGICAL CØN
      INTEGER I, J, K, L, P, R
      REAL FO, F1, B, BETY
      DIMENSIØN A(30), D(30), V(30,30), ALPHA(30,30), BETA(30),
     * E(30), AV(30)
C   THIS SUBRØUTINE MINIMIZES A FUNCTIØN ØF N VARIABLES
C   USING THE METHØD ØF RØSENBRØCK. THE PARAMETERS ARE
C   DESCRIBED AS FØLLØWS:
C     N IS THE NUMBER ØF INDEPENDENT VARIABLES
C     X(N) IS AN ESTIMATE ØF THE SØLUTIØN ( ØN ENTRY -
C         AN INITIAL ESTIMATE, ØN EXIT - THE BEST ESTIMATE
C         ØF THE SØLUTIØN FØUND )
C     FUNCT(N,X,F) IS A RØUTINE PRØVIDED BY THE USER TØ
C         CALCULATE THE VALUE F ØF THE MINIMIZED FUNCTIØN
C         AT ANY PØINT X
C     STEP IS AN INITIAL STEP LENGTH FØR ALL CØØRDINATE
C         DIRECTIØNS AT THE START ØF THE PRØCESS
C     MØNITR (N,X,F,R,B,CØN,NR) IS A RØUTINE PRØVIDED BY
C         THE USER FØR DIAGNØSTIC AND CØNVERGENCE PURPØSES
C     R IS THE ACTUAL NUMBER ØF FUNCTIØN EVALUATIØNS ( FØR
C         THE INITIAL ESTIMATE R=0 )
C     B IS THE VALUE ØF THE EUCLIDEAN NØRM ØF THE VECTØR
C         REPRESENTING THE TØTAL PRØGRESS MADE SINCE THE
C         AXES WERE LAST RØTATED
C     CØN IS A LØGICAL VARIABLE. AT THE START ØF THE
C         SUBRØUTINE RØMIN CØN=.FALSE. IF THE CØNVERGENCE
C         CRITERIA ØF THE RØUTINE MØNITØR ARE SATISFIED
C         CØN MUST BE SET .TRUE. TØ STØP THE PRØCESS
C     NR  IS THE MØNITØR INDEX
C   INITIALIZE CØN, E(I) AND R
C   E(I) IS A SET ØF STEPS TØ BE TAKEN IN THE CØRRESPØNDING
C   CØØRDINATE DIRECTIØNS
      CØN = .FALSE.
      DØ 10 I=1,N
        E(I) = STEP
   10 CØNTINUE
      R = 0
C   V(I,J) IS AN NXN MATRIX DEFINING A SET ØF N MUTUALLY
C   ØRTHØGØNAL 6ØØRDINATE DIRECTIØNS. V(I,J) IS THE UNIT
C   MATRIX AT THE START ØF THE PRØCESS
      DØ 30 I=1,N
        DØ 20 J=1,N
          V(I,J) = 0.0
          IF (I.EQ.J) V(I,J) = 1.0
   20   CØNTINUE
   30 CØNTINUE
      CALL FUNCT(N, X, FO)
C   START ØF THE ITERATIØN LØØP
   40 DØ 50 I=1,N
        A(I) = 2.0
        D(I) = 0.0
   50 CØNTINUE
C   EVALUATE F AT THE NEW PØINT X
   60 DØ 130 I=1,N
        DØ 70 J=1,N
          X(J) = X(J) + F(I)*V(I,J)
   70   CØNTINUE
        R = R + 1
        CALL FUNCT(N, X, F1)
        CALL MØNITR(N, X, F1, R, 0, CØN, 1)
        IF (CØN) GØ TØ 290
        IF (F1-FO) 80, 90, 90
```

```
C   THE NEW VALUE ØF THE FUNCTIØN IS LESS THAN THE ØLD ØNE
  80   D(I) = D(I) + E(I)
       E(I) = 3.0*F(I)
       FO = F1
       IF (A(I).GT.1.5) A(I) = 1.0
       GØ TØ 110
C   THE NEW VALUE ØF THE FUNCTIØN IS GREATER THAN ØK EQUAL
C   TØ THE ØLD ØNE
  90   DØ 100 J=1,N
         X(J) = X(J) - F(I)*V(I,J)
 100   CØNTINUE
       E(I) = -0.5*F(I)
       IF (A(I).LT.1.5) A(I) = 0.0
 110   DØ 120 J=1,N
         IF (A(J).GE.0.5) GØ TØ 130
 120   CØNTINUE
       GØ TØ 140
 130 CØNTINUE
     GØ TØ 60
C   GRAM-SCHMIDT ØRTHØGØNALIZATIØN PRØCESS
 140 DØ 160 K=1,N
       DØ 150 L=1,N
         ALPHA(K,L) = 0.0
 150   CØNTINUE
 160 CØNTINUE
     DØ 190 I=1,N
       DØ 180 J=1,N
         DØ 170 L=I,N
           ALPHA(I,J) = ALPHA(I,J) + D(L)*V(L,J)
 170     CØNTINUE
 180   CØNTINUE
 190 CØNTINUE
     B = 0.0
     DØ 200 J=1,N
       B = B + ALPHA(I,J)**2
 200 CØNTINUE
     B = SQRT(B)
C   CALCULATE THE NEW SET ØF ØRTHØNØRMAL CØØRDINATE
C   DIRECTIØNS ( THE NEW MATRIX V(I,J) )
     DØ 210 J=1,N
       V(1,J) = ALPHA(1,J)/B
 210 CØNTINUE
     DØ 280 P=2,N
       BETY = 0.0
       IP = P - 1
       DØ 220 M=1,N
         BETA(M) = 0.0
 220   CØNTINUE
       DØ 250 J=1,N
         DØ 240 K=1,IP
           AV(K) = 0.0
           DØ 230 L=1,N
             AV(K) = AV(K) + ALPHA(P,L)*V(K,L)
 230       CØNTINUE
           BETA(J) = BETA(J) - AV(K)*V(K,J)
 240     CØNTINUE
 250   CØNTINUE
       DØ 260 J=1,N
         BETA(J) = BETA(J) + ALPHA(P,J)
         BETY = BETY + RETA(J)**2
 260   CØNTINUE
       BETY = SQRT(BETY)
       DØ 270 J=1,N
         V(P,J) = BETA(J)/BETY
 270   CØNTINUE
 280 CØNTINUE
C   END ØF GRAM-SCHMIDT PRØCESS
     CALL MØNITR(N, X, FO, R, B, CØN, 2)
     IF (CØN) GØ TØ 290
C   GØ TØ THE NEXT ITERATIØN
     GØ TØ 40
 290 RETURN
     END
```

Remark on Algorithm 450 [E4]

Rosenbrock Function Minimization [Marek Machura and Andrzej Mulawa, *Comm. ACM 16* (Aug. 1973), 482–483]

Adhemar Bultheel [Recd. 10 Oct. 1973]
Katholieke Universiteit Leuven, Faculty of Applied Sciences, Applied Math Division, Celestijnenlaan 200 B, B-3030 Heverlee, Belgium

1. Some misprints were found in the listing of the algorithm.

(a) An *E* has to replace the *F* printed in the following statements:
The one preceding the statement labeled 70.
The one following the statement labeled 80.
The one preceding the statement labeled 100.
The one following the statement labeled 100.

(b) The digit 1 should replace the character *I* as the first index of *ALPHA* in the statement preceding the statement labeled 200.

(c) *RETA* should be read *BETA* in the statement preceding the statement labeled 260.

2. Some compilers detect an error in the calling sequence of *MONITR* in the third line following the statement labeled 70 because the fifth argument of *MONITR* is an *INTEGER*-type constant, and in the subroutine *MONITR* the fifth argument stands for the norm *B* of a vector which is obviously a *REAL*-type variable as is also assumed in the other calls of *MONITR*. One way to overcome this difficulty is to replace 0 by any *REAL* constant, say 0.

3. Since it is often useful to have the initial guess and the corresponding function value printed, an additional call to *MONITR* could be inserted just preceding the *COMMENT*

C START OF THE ITERATION LOOP

This statement could be

CALL MONITR (N, X, FO, R, 1.E 10, CON, 0).

The last argument is the monitoring index *NR*. The user of *Romin* should program *MONITR* to handle the initial guess when $NR = 0$ (printing or not, checking for convergence or not, . . .). The fifth argument is chosen to be a large constant because it stands for the norm *B* of a vector. The routine *MONITR* will contain a test to see if $B < \epsilon$ with ϵ "small" and chosen by the user. If one wants to check the initial guess for convergence, then the routine would stop when *B* equals 0..

4. With these corrections and changes the algorithm was successfully used under a WATFIV compiler on the IBM 370-155 computer of the Computing Centre of the University of Leuven. For the example of the parabolic valley function given by the authors of the algorithm and with the same starting point the following results were obtained: in a single-precision version 202 function evaluations were needed to reach $F = 0.299986.10^{-4}$, and in a double-precision version 194 function evaluations to reach $F = 0.297742.10^{-4}$ and 290 function evaluations gave $F = 0.489134.10^{-13}$.

Remark on Algorithm 450 [E4]

Rosenbrock Function Minimization [Marek Machura and Andrzej Mulawa, *Comm. ACM 16* (Aug. 1973), 482–483]

Jiří Klemeš and Jaroslav Klemsa (Recd. 14 Nov. 1973)
Applied Mathematics Department, Research Institute of Chemical Equipment, CHEPOS, Brno, Czechoslovakia

After correcting misprints [1] this algorithm runs successfully using an ODRA 1204 computer made by ELWRO, Poland. The results were the same as reported by authors. Some successful tests have been also made in optimization problems concerning the Williams-Otto chemical plant [2]. It can be seen from the solution of some application problems [3] that it is very useful to select different step lengths in different coordinate directions.

Therefore, we recommend replacement of the third and fourth line in the subroutine *ROMIN*:

```
REAL STEP
DIMENSION X(N)
```

by

```
DIMENSION X(N), STEP(N)
```

and the line before label 10

```
E(I) = STEP
```

by

```
E(I) = STEP(I)
```

In addition we recommend that the lines between labels 220 and 260 be replaced by the lines:

```
220  CONTINUE
     DO 240 K =1,IP
        AV(K) =0.0
        DO 230 L =1,N
           AV(K) = AV(K) + ALPHA(P,L) * V(K,L)
230     CONTINUE
240  CONTINUE
     DO 260 J =1,N
        DO 250 K =1,IP
           BETA(J) = BETA(J) - AV(K) * V(K,J)
250     CONTINUE
        BETA(J) = BETA(J) + ALPHA(P,J)
        BETY = BETY + BETA(J)**2
260  CONTINUE
```

Although, this change does not reduce the number of function evaluations, in each Gram-Schmidt step some computer time may be saved. This is caused by the difference between the number of executions of the statement $AV(K) = AV(K) + ALPHA(P, L) * V(K, L)$ in the original program and the suggested modification which may be estimated as $N(N - 1) \sum_{P=2}^{N} (P - 1)$, whereas for the statement $AV(K) = 0.0$, this difference is $(N - 1) \sum_{P=2}^{N} (P - 1)$. (Note that N is the number of independent variables.)

References

1. Bultheel, A. Remark on Algorithm 450. *Comm. ACM 17*, 8 (Aug. 1974), 470.
2. Williams, T.J., and Otto, R.E. A generalized chemical processing model for the investigation computer control. *A.I.E.E. Trans. 79*, P. 1, Communications and Electronics, (1960), 458–473
3. Klemeš, J., and Vasek, V. Methods for optimizing complex chemical processes. In Proc. 2nd Symp. on Use of Computers in Chemical Engineering, ČVTS, Ústínad Labem, Czechoslovakia, Sept. 1973, pp. O 84–O 102

ACM Transactions on Mathematical Software, Vol. 2, No. 3, September 1976. Pages 300–301

REMARK ON ALGORITHM 450

Rosenbrock Function Minimization [E4]
[M. Machur and A. Mulawa, *Comm. ACM. 16*, 8 (Aug. 1973), 482–483]

Alan M. Davies [Recd 20 June 1975 and 3 Dec. 1975]
Institute of Oceanographic Sciences, Bidston Observatory, Birkenhead, Cheshire, L43 7RA, England.

The algorithm, incorporating the corrections given in [1], was compiled using the Fortran H compiler OPT=2, and run on an IBM 370/165 computer in single precision. The test problem given by the authors gave a function value of $0.29923 \cdot 10^{-4}$ at (0.99512, 0.99051), after 200 function evaluations, and a minimum of $0.38379 \cdot 10^{-8}$ was obtained at (0.99994, 0.99988), after 240 evaluations.

In problems with a large number of variables, the Schmidt orthogonalization process can be affected by numerical errors, producing a set of vectors which are only approximately orthogonal, and this can increase the number of function evaluations required to reach a minimum.

The orthogonalization of the basis can be enhanced by using the improved Gram-Schmidt procedure, together with a few re-orthogonalizations. These changes are readily incorporated into *ROMIN* by replacing the coding following statement 210 through statement 280 with:

```
        DO 300 JCYC=1,NCYC
        DO 250 P=2,N
        IP=P-1
        DO 230 M=P,N
        BETY=0.0
        DO 220 K=1,N
220     BETY=BETY-ALPHA(M,K)*V(IP,K)
        DO 230 J=1,N
230     ALPHA(M,J)=ALPHA(M,J)+BETY*V(IP,J)
        BETY=0.0
        DO 240 K=1,N
240     BETY=BETY+ALPHA(P,K)**2
        BETY=SQRT(BETY)
        DO 250 K=1,N
250     V(P,K)=ALPHA(P,K)/BETY
        IF(JCYC.EQ.NCYC) GO TO 300
        DO 302 I=2,N
        DO 302 J=1,N
302     ALPHA(I,J)=V(I,J)
300     CONTINUE
```

Since the arrays *AV* and *BETA* are no longer required (a slight saving of core), the *DIMENSION* statement becomes

```
DIMENSION A(30), D(30), V(30,30), ALPHA(30,30), E(30)
```

with the variable *NCYC*, which determines the number of re-orthogonalizations, being incorporated into the argument list of *ROMIN*.

In problems with four or less variables, this coding did not improve the result. However, in an extension of Rosenbrock's problem [2],

$$f(x_1, x_2, \ldots, x_n) = \sum_{i=1,2}^{n-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2$$

(where $i=1,2$ indicates that i increases in increments of 2), for $N=6$ starting at (0.5, 1.5, 0.6, 1.4, 1.7, 0.3) using this improved Gram-Schmidt procedure, with two re-orthogonalizations, a function value of $0.11241 \cdot 10^{-3}$ at (1.00150, 1.00308, 1.00933, 1.01877, 0.99559, 0.99136) was obtained after 1000 function evaluations compared with $0.11296 \cdot 10^{-2}$ produced by the original program plus corrections [1].

For $N=12$, after 1300 evaluations the re-orthogonalized ($NCYC=3$) calculation gave $f=0.10871 \cdot 10^{-2}$ compared with $0.17160 \cdot 10^{-2}$ ($NCYC=1$, no re-orthogonalization), and after 2600 evaluations the results were $0.57029 \cdot 10^{-4}$ ($NCYC=1$) and $0.75086 \cdot 10^{-5}$ ($NCYC=3$). The original program gave $0.15628 \cdot 10^{-3}$ compared with the above value of $0.57029 \cdot 10^{-4}$ produced by using the improved Gram-Schmidt procedure alone. However, using the original program but incorporating just the code for re-orthogonalizing a value of $0.73922 \cdot 10^{-5}$ ($NCYC=3$) was obtained, illustrating the improvement to be gained by just re-orthogonalization.

The method was also tested on an extension of Box's problem [3] using 18 variables. The original program calculated a minimum of $0.40310 \cdot 10^{-3}$; however, by incorporating the changes given above a minimum of $0.48176 \cdot 10^{-4}$ was obtained.

REFERENCES

1. Bultheel, A. Remark on Algorithm 450. *Comm. ACM 17*, 8 (Aug. 1974), 470.
2. Rosenbrock, H.H. An automatic method for finding the greatest or least value of a function. *Computer J. 3* (1960), 175–184.
3. Box, M.J. A comparison of several current optimization methods and the use of transformations in constrained problems. *Computer J. 9* (1966), 67–77.

Algorithm 451

Chi-Square Quantiles [G1]

Richard B. Goldstein [Recd. 30 June 1971 and 20 March 1972]
Department of Mathematics, Providence College, Providence, R.I.

Key Words and Phrases: Chi-square statistic, asymptotic approximation, normal deviate, chi-square deviate, degrees of freedom

CR Categories: 5.12, 5.5

Language: Fortran

Description

The algorithm evaluates the quantile at the probability level P for the Chi-square distribution with N degrees of freedom. The quantile function is an inverse of the function

$$P(X \mid N) = (2^{N/2}\Gamma(N/2))^{-1} \int_{X(P)}^{\infty} Z^{\frac{1}{2}N-1}e^{-\frac{1}{2}Z}\,dZ \quad (x \geq 0, N \geq 1).$$

The function $GAUSSD(P)$ is assumed to return the normal deviate for the level P, e.g. -1.95996 for $P = .025$. The procedure found in Hastings [5] may be used, or for increased accuracy, the procedure found in Cunningham [3] may be used.

The Wilson-Hilferty cubic formula [7] which is

$$\chi^2 \sim N\{1 - 2/9N + X\,(2/9N)^{\frac{1}{2}}\}^3$$

where X is the normal deviate can be extended to the 19-term asymptotic approximation:

$$\begin{aligned}\chi^2 \sim N\{1 &- 2/9N + (4X^4+16X^2-28)/1215N^2 \\ &+ (8X^6+720X^4+3216X^2+2904)/229635N^3 + \cdots \\ &+ (2/N)^{\frac{1}{2}}[X/3 + (-X^3+3X)/162N \\ &- (3X^5+40X^3 + 45X)/5832N^2 \\ &+ (301X^7-1519X^5-32769X^3-79349X)/7873200N^3 + \cdots]\}^3\end{aligned}$$

where X is the normal deviate by taking the cube root of the polynomial expansion in Campbell [2]. For $N = 1$

$$\chi^2 = \{GAUSSD(\tfrac{1}{2}P)\}^2$$

and for $N = 2$

$$\chi^2 = -2 \ln (P).$$

For $2 < N < 2 + 4|X|$, χ^2 was fit with polynomials of the same form as the asymptotic approximation:

$$\begin{aligned}\chi^2 \cong N\{&(1.0000886-.2237368/N-.01513904/N^2) \\ &+ N^{-\frac{1}{2}}X(.4713941+.02607083/N-.008986007/N^2) \\ &+ N^{-1}X^2(.0001348028+.01128186/N+.02277679/N^2) \\ &+ N^{-3/2}X^3(-.008553069-.01153761/N-.01323293/N^2) \\ &+ N^{-2}X^4(.00312558+.005169654/N-.006950356/N^2) \\ &+ N^{-5/2}X^5(-.0008426812+.00253001/N+.001060438/N^2) \\ &+ N^{-3}X^6(.00009780499-.001450117/N+.001565326/N^2)\}^3\end{aligned}$$

from Abramowitz and Stegun [1] for $P = .0001, .0005, \ldots, .995$ and Hald and Sinkbaek [4] for $P = .999, .9995$. The deviates for $N = 3, 4, \ldots, 9$ were made accurate within 10^{-6} by using Algorithm 299 of Hill and Pike [6].

Fig. 1

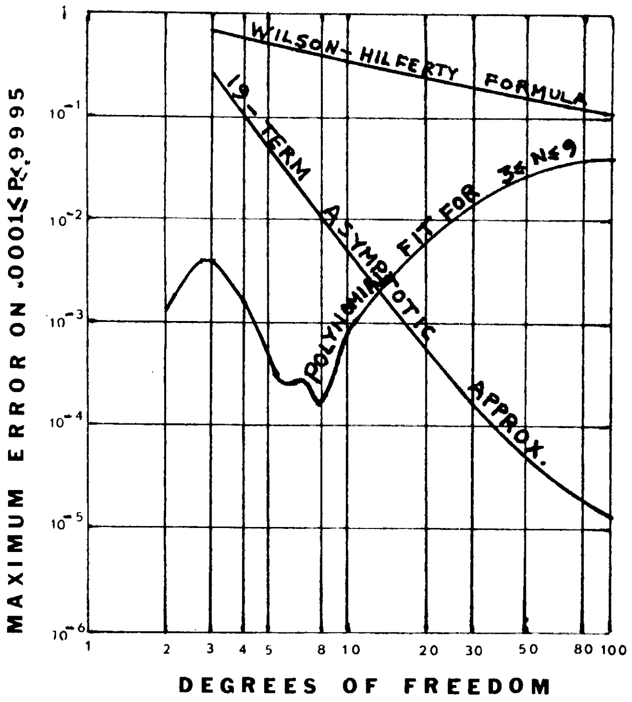
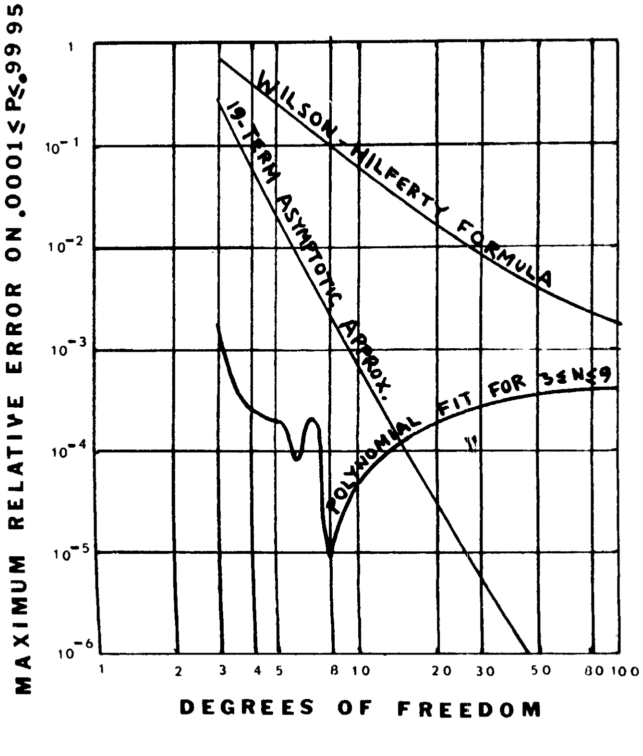


Fig. 2



For $N = 1$ and $N = 2$ the χ^2 deviate is as accurate as the *GAUSSD* and *ALOG* procedure of the system. For $.0001 \leq P \leq .9995$ and $N \geq 3$ the absolute error in χ^2 is less than .005 and the relative error is less than .0003. This is some 100 to 1000 times as accurate as the Wilson-Hilferty formula even for large N. Error curves for three approximations are shown in Figures 1 and 2.

The program was tested on an IBM/360 at Rhode Island College and resulted in the output of Table I.

Table I.
Table of Computed Values

Deg. Fr.	0.9995	0.9950	0.5000	0.0010	0.0001
1	0.000000	0.000039	0.454933	10.827576	15.135827
2	0.001000	0.010025	1.386293	13.815512	18.420670
3	0.015312	0.071641	2.365390	16.268982	21.106873
4	0.063955	0.206904	3.356400	18.467987	23.510040
5	0.158168	0.411690	4.351295	20.515503	25.744583
10	1.264941	2.155869	9.341794	29.589081	35.565170
15	3.107881	4.601008	14.338853	37.697662	44.267853
20	5.398208	7.433892	19.337418	45.314896	52.387360
50	23.460876	27.990784	49.334930	86.660767	95.969482
100	59.895508	67.327621	99.334122	149.449051	161.319733

References

1. Abramowitz, M., and Stegun, I. (Eds.) *Handbook of Mathematical Functions*, Appl. Math. Ser. Vol. 55. Nat. Bur. Stand., U.S. Govt. Printing Office, Washington, D.C., 1965, pp. 984–985.
2. Campbell, G.A., Probability curves showing Poisson's exponential summation. *Bell Syst. J. 2* (1923), 95–113.
3. Cunningham, S.W. From normal integral to deviate. In *Applied Statistics*. Vol. 18, Royal Statis. Soc., 1969, pp. 290–293.
4. Hald, O.O., and Sinkbaek, O.O. *Skandinavisk Akturarie-tidskrift* (1950), 168–175.
5. Hastings, C. Jr. *Approximations for Digital Computers.* Princeton U. Press, Princeton, N.J., 1958, p. 192.
6. Hill, I.D., and Pike, M.C. Algorithm 299, Chi-squared integral. *Comm. ACM, 10*, 4 (Apr., 1967), 243–244.
7. Hilferty, M.M., and Wilson, E.B. *Proc. Nat. Acad. Sci., 17* (1931), 684.
8. Riordan, J. Inversion formulas in normal variable mapping. *Annals of Math. Statist. 20* (1949), 417–425.

Algorithm

```
      FUNCTIØN CHISQD(P, N)
      DIMENSIØN C(21), A(19)
      DATA C(1)/1.565326E-3/, C(2)/1.060438E-3/,
     * C(3)/-6.950356E-3/, C(4)/-1.323293E-2/,
     * C(5)/2.277679E-2/, C(6)/-8.986007E-3/,
     * C(7)/-1.513904E-2/, C(8)/2.530010E-3/,
     * C(9)/-1.450117E-3/, C(10)/5.169654E-3/,
     * C(11)/-1.153761E-2/, C(12)/1.128186E-2/,
     * C(13)/2.607083E-2/, C(14)/-0.2237368/,
     * C(15)/9.780499E-5/, C(16)/-8.426812E-4/,
     * C(17)/3.125580E-3/, C(18)/-8.553069E-3/,
     * C(19)/1.348028E-4/, C(20)/0.4713941/, C(21)/1.0000886/
      DATA A(1)/1.264616E-2/, A(2)/-1.425296E-2/,
     * A(3)/1.400483E-2/, A(4)/-5.886090E-3/,
     * A(5)/-1.091214E-2/, A(6)/-2.304527E-2/,
     * A(7)/3.135411E-3/, A(8)/-2.728484E-4/,
     * A(9)/-9.699681E-3/, A(10)/1.316872E-2/,
     * A(11)/2.618914E-2/, A(12)/-0.2222222/,
     * A(13)/5.406674E-5/, A(14)/3.483789E-5/,
     * A(15)/-7.274761E-4/, A(16)/3.292181E-3/,
     * A(17)/-8.729713E-3/, A(18)/0.4714045/, A(19)/1./
      IF (N-2) 10, 20, 30
   10 CHISQD = GAUSSD(.5*P)
      CHISQD = CHISQD*CHISQD
      RETURN
   20 CHISQD = -2.*ALØG(P)
      RETURN
   30 F = N
      F1 = 1./F
      T = GAUSSD(1.-P)
      F2 = SQRT(F1)*T
      IF (N.GE.(2+INT(4.*ABS(T)))) GØ TØ 40
      CHISQD=(((((((C(1)*F2+C(2))*F2+C(3))*F2+C(4))*F2
     * +C(5))*F2+C(6))*F2+C(7))*F1+((((((C(8)+C(9)*F2)*F2
     * +C(10))*F2+C(11))*F2+C(12))*F2+C(13))*F2+C(14)))*F1 +
     * (((((C(15)*F2+C(16))*F2+C(17))*F2+C(18))*F2
     * +C(19))*F2+C(20))*F2+C(21)
      GØ TØ 50
   40 CHISQD=(((A(1)+A(2)*F2)*F1+(((A(3)+A(4)*F2)*F2
     * +A(5))*F2+A(6)))*F1+(((((A(7)+A(8)*F2)*F2+A(9))*F2
     * +A(10))*F2+A(11))*F2+A(12)))*F1 + (((((A(13)*F2
     * +A(14))*F2+A(15))*F2+A(16))*F2+A(17))*F2*F2
     * +A(18))*F2+A(19)
   50 CHISQD = CHISQD*CHISQD*CHISQD*F
      RETURN
      END
```

Certification of Algorithm 451 [G1]

Chi-Square Quantiles [Richard B. Goldstein, *Comm. ACM* (Aug. 1973), 483–484]

William Knight [Recd 26 Nov. 1973]
Department of Computer Science
University of New Brunswick*

The algorithm was tested for degrees of freedom, $N = 3$ (1) 5 (5) 25 (25) 100, and tail probabilities, P, of

.00010	.0010	.010	.10	.80	.980	.9980
.00015	.0015	.015	.15	.85	.985	.9985
.00020	.0020	.020	.20	.90	.990	.9990
.00030	.0030	.030	.30	.93	.993	.9993
.00050	.0050	.050	.50	.95	.995	.9995
.00070	.0070	.070	.70	.97	.997	

The descriptive text of the algorithm claimed absolute error no more than 0.005 and relative error no more than 0.0003 for $0.0001 \leq P \leq 0.9995$; the values of P listed above were chosen to cover this domain.

The largest absolute error found on the above grid was 0.0059 at $N = 3$, $P = 0.0003$; a finer scale search nearby uncovered an erroı of 0.0062 at $N = 3$, $P = 0.00031$. The largest relative error found on the grid was 0.0035 at $N = 3$, $P = 0.9985$; this being an order of magnitude more than the figure claimed, I conjecture a typographical error, especially as the table of computed values accompanying the algorithm lists 0.071641 for $N = 3$, $P = 0.9950$ which has a relative error exceeding 0.001.

The remainder of this note describes computational details.

Testing was done using the Watfiv compiler on an IBM 370/165 at the University of Toronto.

The following changes were made in the data statements. (1) Since the Watfiv compiler will not accept a representation of a number consisting of more than seven digits (including, it seems, leading zeros) as a short real constant, $C(14)$, $C(20)$, $C(21)$, $A(12)$ and $A(18)$ were rejected by the compiler. This was easily circumvented by changing these representations to "E" form. (2) The two long data statements were broken into several shorter data statements to simplify detection and correction of punching errors. (Moreover, some compilers will not accept nine continuation cards!)

For the inverse normal distribution function subroutine,

* On sabbatical leave to Mathematics Department, University of Toronto, 1973–74.

GAUSSD, formula 26.2.23 of Abramowitz and Stegun [1] was used, followed by two Newton-Raphson iterations in double precision, the normal distribution function needed being constructed from the *DERFC* (complimentary error function) which is included in Watfiv. This should give accuracy to single precision; spot checks against tables in Abramowitz and Stegun [1] bore this out.

The actual testing procedure was this: From a given *N* and *P*, *CHISQD* computed a chi-square value. To establish the correct value with which to compare this, it was refined by a single Newton Raphson iteration. A rather free Fortran translation of Algorithm 299 [2] was used to compute the chi-square integral. (Algorithm 299 should be accurate to the limit set by word length and the square root, exponential, logarithm and error function routines.) Where possible corrected chi-square values were checked against table 26.8 of Abramowitz and Stegun [1], agreement to at least three places after the decimal point or four significant figures, whichever was more stringent, being found in all cases.

References

1. Abramowitz, M., and Stegun, I. (Eds) *Handbook of Mathematical Functions.* Appl. Math. Ser. Vol. 55. Nat. Bur. Stand., U.S. Gov. Print. Off., Washington, D.C., 1965.
2. Hill, I.D., and Pike, M.C., Algorithm 299, Chi-squared integral. *Comm. ACM. 10*, 4 (Apr., 1967), 234–244.

Algorithm 452

Enumerating Combinations of *m* Out of *n* Objects [G 6]

C.N. Liu and D.T. Tang [Recd. 7 July 1971 and
1 May 1972]
IBM Thomas J. Watson Research Center, Yorktown
Heights, NY 10598

Key Words and Phrases: permutations, combinations
CR categories: 5.30
Language: Fortran

Description

NXCBN can be used to generate all combinations of m out of n objects. Let the binary n-vector of m1's and $(n - m)$ 0's representing a combination of m out of n objects be stored in an integer array, say $IC(n)$. If *NXCBN* (n, m, IC) is called, a binary vector representing a new combination is made available in the array $IC(n)$. If *NXCBN* (n, m, IC) is called $\binom{n}{m}$ times successively, then all combinations will be generated.

The algorithm has the following features; (a) each output binary n-vector differs from the input at exactly two positions—consequently each generated combination differs from the previous one by a single object: (b) the n-vectors generated by this subroutine form a closed loop of $\binom{n}{m}$ elements—therefore the initial combination may be specified arbitrarily, and the enumeration of any subset of $\binom{n}{m}$ combinations can be readily achieved. The second feature is not found in Chase's algorithm [1].

The algorithm underlying this procedure is based upon our study of properties of Gray codes. It can be shown that constant weight code vectors from a Gray code sequence are separated by a Hamming distance of 2. The mathematical analysis is contained in [2] and [3].

References

1. Chase, P.J. Algorithm 382, Combinations of m out of n objects. *Comm. ACM 13* (June 1970), 368.
2. Tang, D.T., and Liu, C.N. On enumerating m out of n combinations with minimal replacements. Proc. of Fifth Ann. Princeton Conf. on Info. Sci. and Sys., Mar. 1971.
3. Tang, D.T., and Liu, C.N. Distance-Two Cyclic Chaining of Constant-Weight Codes. *IEEETC.* C-*22*, 2 (Feb. 1973), 176–180.

Algorithm

```
      SUBRØUTINE NXCBN(N, M, IC)
C EXPLANATIØN ØF THE PARAMETERS IN THE CALLING SEQUENCE
C   N     THE TØTAL NUMBER ØF ØBJECTS
C   M     THE NUMBER ØF ØBJECTS TØ BE TAKEN FRØM N
C IF M=Q, ØR M>=N, EXIT WITH ARGUMENTS UNCHANGED
C   IC    AN INTEGER ARRAY.  IC CØNTAINS AN N-DIMEN-
C         SIØNAL BINARY VECTØR WITH M ELEMENTS SET TØ 1
C         REPRESENTING THE M ØBJECTS IN A CØMBINATIØN
C THIS ALGØRITHM IS PRØGRAMMED IN ANSI STANDARD FØRTRAN
      INTEGER IC(N)
C CHECK ENDING PATTERN ØF VECTØR
      IF (M.GE.N .ØR. M.EQ.0) GØ TØ 140
      N1 = N - 1
      DØ 10 J=1,N1
        NJ = N - J
        IF (IC(N).EQ.IC(NJ)) GØ TØ 10
        J1 = J
        GØ TØ 20
   10 CØNTINUE
   20 IF (MØD(M,2).EQ.1) GØ TØ 90
C FØR M EVEN
      IF (IC(N).EQ.1) GØ TØ 30
      K1 = N - J1
      K2 = K1 + 1
      GØ TØ 130
   30 IF (MØD(J1,2).EQ.1) GØ TØ 40
      GØ TØ 120
C SCAN FRØM RIGHT TØ LEFT
   40 JP = (N-J1) - 1
      DØ 50 I=1,JP
        I1 = JP + 2 - I
        IF (IC(I1).EQ.0) GØ TØ 50
        IF (IC(I1-1).EQ.1) GØ TØ 70
        GØ TØ 80
   50 CØNTINUE
   60 K1 = 1
      K2 = (N+1) - M
      GØ TØ 130
   70 K1 = I1 - 1
      K2 = N - J1
      GØ TØ 130
   80 K1 = I1 - 1
      K2 = (N+1) - J1
      GØ TØ 130
C FØR M ØDD
   90 IF (IC(N).EQ.1) GØ TØ 110
      K2 = (N-J1) - 1
      IF (K2.EQ.0) GØ TØ 60
      IF (IC(K2+1).EQ.1 .AND. IC(K2).EQ.1) GØ TØ 100
      K1 = K2 + 1
      GØ TØ 130
  100 K1 = N
      GØ TØ 130
  110 IF (MØD(J1,2).EQ.1) GØ TØ 120
      GØ TØ 40
  120 K1 = N - J1
      K2 = MIN0((K1+2),N)
C CØMPLEMENTING TWØ BITS TØ ØBTAIN THE NEXT CØMBINATIØN
  130 IC(K1) = 1 - IC(K1)
      IC(K2) = 1 - IC(K2)
  140 RETURN
      END
```

Algorithm 453

Gaussian Quadrature Formulas for Bromwich's Integral [D1]

Robert Piessens [Recd. 2 Aug. 1970 and 8 Feb. 1972]
Applied Mathematics Division, University of Leuven, Heverlee, Belgium

Key Words and Phrases: Gaussian quadrature, Bromwich's integral, complex integration, numerical inversion of the Laplace transform

CR Categories: 5.16, 5.13

Language: Fortran

Description

BROMIN calculates the abscissas $x_k^{(s)}$ and weights $w_k^{(s)}$ of the Gaussian quadrature formula

$$(1/2\pi j)\int_{c-j\infty}^{c+j\infty} e^x x^{-s} F(x)\,dx \simeq \sum_{k=1}^{N} w_k^{(s)} F(x_k^{(s)}) \tag{1}$$

where c is an arbitrary real positive number, s is a real nonnegative parameter, and $F(x)$ must be analytic in the right-half plane of the complex plane.

Abscissas $x_k^{(s)}$ and weights $w_k^{(s)}$ are to be determined so that (1) is exact whenever $F(x)$ is a polynomial in x^{-1}, of degree $\leq 2N - 1$.

The abscissas $x_k^{(s)}$ are the zeros of $P_{N,s}(x^{-1})$ where

$$P_{N,s}(u) = (-1)^N {}_2F_0(-N, N+s-1; \;-;\; u). \tag{2}$$

Properties of $P_{N,s}(u)$ are studied in [1].

The quadrature formulas of even order have no real abscissas; those of odd order have exactly one real abscissa. All the abscissas have positive real parts and occur in complex conjugate pairs.

The zeros of (2) are calculated using Newton-Raphson's method. Finding an approximate zero as starting value for the iteration process is based on a certain regularity in the distribution of the zeros (see [1] and [2]). The starting values, used by *BROMIN* were tested for $s = 0.1(0.1)4.0$ and $N = 4(1)12$. Each abscissa was found to at least eight significant figures in at most six iteration steps.

The weights are given by

$$A_k = (-1)^{N-1} \frac{(N-1)!}{\Gamma(N+s-1)Nx_k^2}\left[\frac{2N+s-2}{P_{N-1,s}(x_k^{-1})}\right]^2 \tag{3}$$

The polynomial (2) is evaluated by a three-term recurrence relation (see [1]). Due to roundoff errors, the accuracy of abscissas and weights decreases significantly for increasing N. In Table I we give for some values of s and N the moduli of the relative errors in the abscissas and weights, calculated by *BROMIN* (with $TOL = 0.1E - 10$) on an IBM 370 computer in double precision (approximately 16 significant figures). For comparison we used the $16 - S$ values given in [3].

Note that the relative errors in the weights are larger than in the abscissas.

The use of complex arithmetic is avoided in *BROMIN* in order to facilitate the conversion to a double precision subroutine.

Table I. Maximum Relative Errors in Abscissas and Weights

	Maximum error in abscissas		Maximum error in weights	
s	$N = 6$	$N = 12$	$N = 6$	$N = 12$
0.1	1.8×10^{-13}	1.7×10^{-9}	1.2×10^{-13}	2.3×10^{-8}
1.0	1.9×10^{-14}	5.3×10^{-11}	1.5×10^{-14}	6.4×10^{-10}
4.0	1.3×10^{-15}	2.3×10^{-12}	1.0×10^{-14}	4.3×10^{-11}

References

1. Piessens, R. Gaussian quadrature formulas for the numerical integration of Bromwich's integral and the inversion of the Laplace transform. *J. Eng. Math. 5* (Jan. 1971), 1-9.
2. Piessens, R. Some aspects of Gaussian quadrature formulas for the numerical inversion of the Laplace transform. *Comput. J. 14* (Nov. 1971), 433-435.
3. Piessens, R. Gaussian quadrature formulas for the numerical integration of Bromwich's integral and the inversion of the Laplace transform. Rep. TW1, Appl. Math. Div. U. of Leuven, 1969.

Algorithm

```
      SUBRØUTINE BRØMIN(N, S, TØL, XR, XI, WR, WI, EPS, IER)
      DØUBLE PRECISIØN AK, AN, ARG, CI, CR, D, D1, D2, E, EPS,
     * FAC, FACTI, FACTR, PI, PR, QI, QR, RI, RR, S, T1, T2,
     * TØL, U, V, WI, WR, XI, XR, YI, YR, Z
      INTEGER IER, J, K, L, N, N1, NUM, NUP, IGNAL
      DIMENSIØN XR(N), XI(N), WR(N), WI(N)
C   THIS SUBRØUTINE CALCULATES ABSCISSAS AND WEIGHTS ØF THE
C   GAUSSIAN QUADRATURE FØRMULA ØF ØRDER N FØR THE BRØMWICH
C   INTEGRAL.  ØNLY THE ABSCISSAS ØF THE FIRST QUADRANT ØF
C   THE CØMPLEX PLANE, THE REAL ABSCISSA (IF N IS ØDD) AND
C   THE CØRRESPØNDING WEIGHTS ARE CALCULATED.  THE ØTHER
C   ABSCISSAS AND WEIGHTS ARE CØMPLEX CØNJUGATES.
C   INPUT PARAMETERS
C        N   ØRDER ØF THE QUADRATURE FØRMULA.
C            N MUST BE GREATER THAN 2.
C        TØL REQUESTED RELATIVE ACCURACY ØF THE ABSCISSAS.
C        S   PARAMETER ØF THE WEIGHT FUNCTIØN.
C   ØUTPUT PARAMETERS
C        XR AND XI CØNTAIN THE REAL AND IMAGINARY PARTS ØF
C           THE ABSCISSAS.  IF N IS ØDD, THE REAL ABSCISSA
C           IS XR(1).
C        WR AND WI CØNTAIN THE REAL AND IMAGINARY PARTS ØF
C           THE CØRRESPØNDING WEIGHTS.
C        EPS IS A CRUDE ESTIMATIØN ØF THE ØBTAINED RELATIVE
C           ACCURACY ØF THE ABSCISSAS.
C        IER IS AN ERRØR CØDE.
C           IF IER=0    THE CØMPUTATIØN IS CARRIED ØUT TØ
C                       THE REQUESTED ACCURACY.
C           IF IER.GT.0 THE IER-TH ABSCISSA IS NØT FØUND.
C           IF IER=-1   THE CØMPUTATIØNS ARE CARRIED ØUT,
C                       BUT THE REQUESTED ACCURACY IS NØT
C                       ACHIEVED.
C           IF IER=-2   N IS LESS THAN 3.
C   FUNCTIØN PRØGRAMS REQUIRED
C        FUNCTIØN GAMMA(X) WHICH EVALUATES THE GAMMA
C            FUNCTIØN FØR PØSITIVE X.
      IER = -2
      IF (N.LT.3) RETURN
      N1 = (N+1)/2
      L = N - 1
      AN = N
      IER = 0
      EPS = TØL
      ARG = 0.034D0*(30.D0+AN+AN)/(AN-1.D0)
      FACTR = DCØS(ARG)
      FACTI = DSIN(ARG)
      FAC = 1.D0
      AK = 0.D0
      DØ 10 K=1,L
        AK = AK + 1.D0
        FAC = -FAC*AK
   10 CØNTINUE
      FAC = FAC*(AN+AN+S-2.D0)**2/(AN*DGAMMA(AN+S-1.D0))
C   CALCULATIØN ØF AN APPRØXIMATIØN ØF THE FIRST ABSCISSA
      YR = 1.333D0*AN + S - 1.5D0
      YI = 0.0D0
      IF (N-N1-N1) 30, 20, 20
   20 YI = YI + 1.6D0 + 0.07D0*S
C   START MAIN LØØP
   30 DØ 140 K=1,N1
        E = TØL
        IGNAL = 0
        NUM = 0
        NUP = 0
```

```
C   NEWTØN-RAPHSØN METHØD
          D = YR*YR + YI*YI
          YR = YR/D
          YI = -YI/D
          GØ TØ 50
     40   IGNAL = 1
     50   QR = S*YR - 1.D0
          QI = S*YI
          PR = (S+1.D0)*((S+2.D0)*(YR*YR-YI*YI)-2.D0*YR) + 1.D0
          PI = 2.D0*(S+1.D0)*YI*((S+2.D0)*YR-1.D0)
          Z = 2.D0
          DØ 60 J=3,N
            RR = QR
            RI = QI
            QR = PR
            QI = PI
            Z = Z + 1.D0
            U = Z + S - 2.D0
            V = U + Z
            D = (V*YR+(2.D0-S)/(V-2.D0))/U
            D1 = (Z-1.D0)*V/(U*(V-2.D0))
            D2 = V*YI/U
            PR = (V-1.D0)*(QR*D-QI*D2) + D1*RR
            PI = (V-1.D0)*(QI*D+QR*D2) + D1*RI
     60   CØNTINUE
          IF (IGNAL.EQ.1) GØ TØ 100
          D = (YR*YR+YI*YI)*V
          D1 = ((PR+QR)*YR+(PI+QI)*YI)/D + PR
          D2 = ((PI+QI)*YR-(PR+QR)*YI)/D + PI
          D = (D1*D1+D2*D2)*AN
          T1 = PR*YR - PI*YI
          T2 = PI*YR + PR*YI
          CR = (T1*D1+T2*D2)/D
          CI = (T2*D1-T1*D2)/D
          YR = YR - CR
          YI = YI - CI
          NUM = NUM + 1
C   TEST ØF CØNVERGENCE ØF ITERATIØN PRØCESS
          IF (CR*CR+CI*CI-E*E*(YR*YR+YI*YI)) 40, 40, 70
C   TEST ØF NUMBER ØF ITERATIØN STEPS
     70   IF (NUM-10) 50, 50, 80
     80   E = E*10.D0
          IER = -1
          NUP = NUP + 1
          IF (NUP-5) 50, 50, 90
     90   IER = K
          RETURN
C   CALCULATIØN ØF WEIGHTS
    100   IF (EPS.GE.E) GØ TØ 110
          EPS = E
    110   D = (QR*QR+QI*QI)**2
          D1 = YR*QR + YI*QI
          D2 = YI*QR - YR*QI
          WR(K) = FAC*(D1*D1-D2*D2)/D
          WI(K) = 2.D0*FAC*D2*D1/D
          D = YR*YR + YI*YI
          XR(K) = YR/D
          XI(K) = -YI/D
          IF (K+1-N1) 130, 120, 150
    120   FACTR = DCØS(1.5D0*ARG)
          FACTI = DSIN(1.5D0*ARG)
C   CALCULATIØN ØF AN APPRØXIMATIØN ØF THE (K+1)-TH ABSCISSA
    130   YR = (XR(K)+0.67D0*AN)*FACTR - XI(K)*FACTI - 0.67D0*AN
          YI = (XR(K)+0.67D0*AN)*FACTI + XI(K)*FACTR
    140 CØNTINUE
    150 RETURN
        END
```

Algorithm 454

The Complex Method for Constrained Optimization [E4]

Joel A. Richardson and J.L. Kuester* [Rec'd. Dec. 22, 1970 and May 5, 1971]
Arizona State University, Tempe, AZ 85281

Key Words and Phrases: optimization, constrained optimization, Box's algorithm
CR Categories: 5.41
Language: Fortran

Description

Purpose. This program finds the maximum of a multivariable, nonlinear function subject to constraints:

Maximize $F(X_1, X_2, \ldots, X_N)$
Subject to $G_k \leq X_k \leq H_k, \quad k = 1, 2, \ldots, M.$

The implicit variables $X_{N+1}, \ldots, X_M$ are dependent functions of the explicit independent variables $X_1, X_2, \ldots, X_N$. The upper and lower constraints H_k and G_k are either constants or functions of the independent variables.

Method. The program is based on the "complex" method of M.J. Box [2]. This method is a sequential search technique, which has proven effective in solving problems with nonlinear objective functions subject to nonlinear inequality constraints. No derivatives are required. The procedure should tend to find the global maximum because the initial set of points is randomly scattered throughout the feasible region. If linear constraints are present or equality constraints are involved, other methods should prove to be more efficient [1]. The algorithm proceeds as follows:

(1) An original "complex" of $K \geq N + 1$ points is generated consisting of a feasible starting point and $K - 1$ additional points generated from random numbers and constraints for each of the independent variables: $X_{i,j} = G_i + r_{i,j}(H_i - G_i)$, $i = 1, 2, \ldots, N$, and $j = 1, 2, \ldots, K - 1$, where $r_{i,j}$ are random numbers between 0 and 1.

(2) The selected points must satisfy both the explicit and implicit constraints. If at any time the explicit constraints are violated, the point is moved a small distance δ inside the violated limit. If an implicit constraint is violated, the point is moved one half of the distance to the centroid of the remaining points: $X_{i,j}(\text{new}) = (X_{i,j}(\text{old}) + \bar{X}_{i,c})/2$, $i = 1, 2, \ldots, N$, where the coordinates of the centroid of the remaining points, $\bar{X}_{i,c}$, are defined by

$$\bar{X}_{i,c} = \frac{1}{K-1}\left[\sum_{j=1}^{K} X_{i,j} - X_{i,j}(\text{old})\right], \quad i = 1, 2, \ldots, N.$$

This process is repeated as necessary until all the implicit constraints are satisfied.

(3) The objective function is evaluated at each point. The point having the lowest function value is replaced by a point which is located at a distance α times as far from the centroid of the remaining points as the distance of the rejected point on the line joining the rejected point and the centroid:

$$X_{i,j}(\text{new}) = \alpha(\bar{X}_{i,c} - X_{i,j}(\text{old})) + \bar{X}_{i,c}, \quad i = 2, \ldots, N.$$

Box [2] recommends a value of $\alpha = 1.3$.

(4) If a point repeats in giving the lowest function value on consecutive trials, it is moved one half the distance to the centroid of the remaining points.

(5) The new point is checked against the constraints and is adjusted as before if the constraints are violated.

(6) Convergence is assumed when the objective function values at each point are within β units for γ consecutive iterations.

Program. The program consists of three general subroutines (*JCONSX, JCEK*1, *JCENT*) and two user supplied subroutines (*JFUNC, JCNST*1). The use of the program and the meaning of the parameters are described in the comments at the beginning of subroutine *JCONSX*. All communication between the main program and subroutines is achieved in the subroutine argument lists. An iteration is defined as the calculations required to select a new point which satisfies the constraints and does not repeat in yielding the lowest function value.

Test results. Several functions were chosen to test the program. The calculations were performed on a CDC 6400 computer. Some examples:

1. Box Problem [2]
   Function: $F = (9 - (X_1 - 3)^2)X_2^3/27\sqrt{3}$
   Constraints: $0 \leq X_1 \leq 100$
   $0 \leq X_2 \leq X_1/\sqrt{3}$
   $0 \leq (X_3 = X_1 + \sqrt{3}X_2) \leq 6$
   Starting point: $X_1 = 1.0$, $X_2 = 0.5$
   Parameters: $K = 4$, $\alpha = 1.3$, $\beta = .001$, $\gamma = 5$, $\delta = .0001$

Computed results	Correct results:
$F = 1.0000$	$F = 1.0000$
$X_1 = 3.0000$	$X_1 = 3.0000$
$X_2 = 1.7320$	$X_2 = 1.7321$

   Number of iterations: 68
   Central processor time: 6 sec.

2. Post Office Problem [3]
   Function: $F = X_1X_2X_3$
   Constraints: $0 \leq X_i \leq 42, \quad i = 1, 2, 3$
   $0 \leq (X_4 = X_1 + 2X_2 + 2X_3) \leq 72$
   Starting point: $X_1 = 1.0$, $X_2 = 1.0$, $X_3 = 1.0$
   Parameters: $K = 6$, $\alpha = 1.3$, $\beta = .01$, $\gamma = 5$, $\delta = .0001$

Computed results:	Correct results:
$F = 3456$	$F = 3456$
$X_1 = 24.01$	$X_1 = 24.00$
$X_2 = 12.00$	$X_2 = 12.00$
$X_3 = 12.00$	$X_3 = 12.00$

   Number of iterations: 72
   Central processor time: 6 sec.

* The authors acknowledge financial support from a National Science Foundation summer fellowship and Arizona State University Grants Committee fellowship. Computer facilities were provided by the Arizona State University Computer Center and AiResearch Manufacturing Company.

3. Beveridge and Schechter Problem [1]

Function: $F = -(X_1 - 0.5)^2 - (X_2 - 1.0)^2$

Constraints: $-2 \leq X_1 \leq 2$

$-\sqrt{2} \leq X_2 \leq \sqrt{2}$

$-4 \leq (X_3 = X_1^2 + 2X_2^2 - 4) \leq 0$

Starting point: $X_1 = 0.$, $X_2 = 0.$

Parameters: $K = 4$, $\alpha = 1.3$, $\beta = .00001$, $\gamma = 5$, $\delta = .0001$

Computed results:	Correct results:
$F = .0000$	$F = .0000$
$X_1 = .5035$	$X_1 = .5000$
$X_2 = .9990$	$X_2 = 1.0000$

Number of iterations: 40

Central processor time = 5 sec.

References

1. Beveridge, G.S., and Schechter, R.S. *Optimization: Theory and Practice*. McGraw-Hill, New York, 1970.
2. Box, M.J. A new method of constrained optimization and a comparison with other methods. *Comp. J. 8* (1965), 42–52.
3. Rosenbrock, H.H. An automatic method for finding the greatest or least value of a function. *Comp. J. 3* (1960), 175–184.

Algorithm

```
      SUBRØUTINE JCØNSX(N, M, K, ITMAX, ALPHA, BETA, GAMMA,
     * DELTA, X, R, F, IT, IEV2, KØ, G, H, XC, L)
C PURPØSE
C   TØ FIND THE CØNSTRAINED MAXIMUM ØF A FUNCTIØN ØF
C   SEVERAL VARIABLES BY THE CØMPLEX METHØD ØF M. J. BØX.
C   THIS IS THE PRIMARY SUBRØUTINE AND CØØRDINATES THE
C   SPECIAL PURPØSE SUBRØUTINES (JCEK1, JCENT, JFUNC,
C   JCNST1).  INITIAL GUESSES ØF THE INDEPENDENT VARIABLES,
C   RANDØM NUMBERS, SØLUTIØN PARAMETERS, DIMENSIØN LIMITS
C   AND PRINTER CØDE DESIGNATIØN ARE ØBTAINED FRØM THE MAIN
C   PRØGRAM.  FINAL FUNCTIØN AND INDEPENDENT VARIABLE
C   VALUES ARE TRANSFERRED TØ THE MAIN PRØGRAM FØR
C   PRINTØUT.  INTERMEDIATE PRINTØUTS ARE PRØVIDED IN THIS
C   SUBRØUTINE. THE USER MUST PRØVIDE THE MAIN PRØGRAM AND
C   THE SUBRØUTINES THAT SPECIFY THE FUNCTIØN (JFUNC) AND
C   CØNSTRAINTS (JCNST1). FØRMAT CHANGES MAY BE REQUIRED
C   WITHIN THIS SUBRØUTINE DEPENDING ØN THE PARTICULAR
C   PRØBLEM UNDER CØNSIDERATIØN.
C USAGE
C   CALL JCØNSX(N,M,K,ITMAX,ALPHA,BETA,GAMMA,DELTA,X,R,F,
C   IT,IEV2,KØ,G,H,XC,L)
C SUBRØUTINES REQUIRED
C   JCEK1(N,M,K,X,G,H,I,KØDE,XC,DELTA,L,K1)
C     CHECKS ALL PØINTS AGAINST EXPLICIT AND IMPLICIT
C     CØNSTRAINTS AND APPLYS CØRRECTIØN IF VIØLATIØNS ARE
C     FØUND
C   JCENT(N,M,K,IEV1,I,XC,X,L,K1)
C     CALCULATES THE CENTRØID ØF PØINTS
C   JFUNC(N,M,K,X,F,I,L)
C     SPECIFIES ØBJECTIVE FUNCTIØN (USER SUPPLIED)
C   JCNST1(N,M,K,X,G,H,I,L)
C     SPECIFIES EXPLICIT AND IMPLICIT CØNSTRAINT LIMITS
C     (USER SUPPLIED).  ØRDER EXPLICIT CØNSTRAINTS FIRST
C DESCRIPTIØN ØF PARAMETERS
C   N      NUMBER ØF EXPLICIT INDEPENDENT VARIABLES - DEFINE
C          IN MAIN PRØGRAM
C   M      NUMBER ØF SETS ØF CØNSTRAINTS - DEFINE IN MAIN
C          PRØGRAM
C   K      NUMBER ØF PØINTS IN THE CØMPLEX - DEFINE IN MAIN
C          PRØGRAM
C   ITMAX MAXIMUM NUMBER ØF ITERATIØNS - DEFINE IN MAIN
C          PRØGRAM
C   ALPHA REFLECTIØN FACTØR - DEFINE IN MAIN PRØGRAM
C   BETA  CØNVERGENCE PARAMETER -  DEFINE IN MAIN PRØGRAM
C   GAMMA CØNVERGENCE PARAMETER - DEFINE IN MAIN PRØGRAM
C   DELTA EXPLICIT CØNSTRAINT VIØLATIØN CØRRECTIØN - DEFINE
C          IN MAIN PRØGRAM
C   X      INDEPENDENT VARIABLES - DEFINE INITIAL VALUES IN
C          MAIN PRØGRAM
C   R      RANDØM NUMBERS BETWEEN 0 AND 1 - DEFINE IN MAIN
C          PRØGRAM
C   F      ØBJECTIVE FUNCTIØN - DEFINE IN SUBRØUTINE JFUNC
C   IT     ITERATIØN INDEX - DEFINED IN SUBRØUTINE JCØNSX
C   IEV2   INDEX ØF PØINT WITH MAXIMUM FUNCTIØN VALUE -
C          DEFINED IN SUBRØUTINE JCØNSX
C   IEV1   INDEX ØF PØINT WITH MINIMUM FUNCTIØN VALUE -
C          DEFINED IN SUBRØUTINE JCØNSX AND JCEK1
C   KØ     PRINTER UNIT NUMBER - DEFINE IN MAIN PRØGRAM
C   G      LØWER CØNSTRAINT - DEFINE IN SUBRØUTINE JCNST1
C   H      UPPER CØNSTRAINT - DEFINE IN SUBRØUTINE JCNST1
C   XC     CENTRØID - DEFINED IN SUBRØUTINE JCENT
C   L      TØTAL NUMBER ØF INDEPENDENT VARIABLES (EXPLICIT +
C          IMPLICIT) - DEFINE IN MAIN PRØGRAM
C   I      PØINT INDEX - DEFINED IN SUBRØUTINE JCØNSX
C   KØDE   KEY USED TØ DETERMINE IF IMPLICIT CØNSTRAINTS ARE
C          PRØVIDED - DEFINED IN SUBRØUTINE JCØNSX AND JCEK1
C   K1     DØ LØØP LIMIT - DEFINED IN SUBRØUTINE JCØNSX
      DIMENSIØN X(K,L), R(K,N), F(K), G(M), H(M), XC(N)
      INTEGER GAMMA
      IT = 1
      WRITE (KØ,99995) IT
      KØDE = 0
      IF (M-N) 20, 20, 10
   10 KØDE = 1
   20 CØNTINUE
      DØ 40 II=2,K
        DØ 30 J=1,N
          X(II,J) = 0.
   30   CØNTINUE
   40 CØNTINUE
C CALCULATE CØMPLEX PØINTS AND CHECK AGAINST CØNSTRAINTS
      DØ 60 II=2,K
        DØ 50 J=1,N
          I = II
          CALL JCNST1(N, M, K, X, G, H, I, L)
          X(II,J) = G(J) + R(II,J)*(H(J)-G(J))
   50   CØNTINUE
        K1 = II
        CALL JCEK1(N, M, K, X, G, H, I, KØDE, XC, DELTA, L, K1)
        WRITE (KØ,99999) II, (X(II,J),J=1,N)
   60 CØNTINUE
      K1 = K
      DØ 70 I=1,K
        CALL JFUNC(N, M, K, X, F, I, L)
   70 CØNTINUE
      KØUNT = 1
      IA = 0
C FIND PØINT WITH LØWEST FUNCTIØN VALUE
      WRITE (KØ,99998) (F(I),I=1,K)
   80 IEV1 = 1
      DØ 100 ICM=2,K
        IF (F(IEV1)-F(ICM)) 100, 100, 90
   90   IEV1 = ICM
  100 CØNTINUE
C FIND PØINT WITH HIGHEST FUNCTIØN VALUE
      IEV2 = 1
      DØ 120 ICM=2,K
        IF (F(IEV2)-F(ICM)) 110, 110, 120
  110   IEV2 = ICM
  120 CØNTINUE
C CHECK CØNVERGENCE CRITERIA
      IF (F(IEV2)-(F(IEV1)+BETA)) 140, 130, 130
  130 KØUNT = 1
      GØ TØ 150
  140 KØUNT = KØUNT + 1
      IF (KØUNT-GAMMA) 150, 240, 240
C REPLACE PØINT WITH LØWEST FUNCTIØN VALUE
  150 CALL JCENT(N, M, K, IEV1, I, XC, X, L, K1)
      DØ 160 J=1,N
        X(IEV1,J) = (1.+ALPHA)*(XC(J)) - ALPHA*(X(IEV1,J))
  160 CØNTINUE
      I = IEV1
      CALL JCEK1(N, M, K, X, G, H, I, KØDE, XC, DELTA, L, K1)
      CALL JFUNC(N, M, K, X, F, I, L)
C REPLACE NEW PØINT IF IT REPEATS AS LØWEST FUNCTIØN VALUE
  170 IEV2 = 1
      DØ 190 ICM=2,K
        IF (F(IEV2)-F(ICM)) 190, 190, 180
  180   IEV2 = ICM
  190 CØNTINUE
      IF (IEV2-IEV1) 220, 200, 220
  200 DØ 210 JJ=1,N
        X(IEV1,JJ) = (X(IEV1,JJ)+XC(JJ))/2.
  210 CØNTINUE
      I = IEV1
      CALL JCEK1(N, M, K, X, G, H, I, KØDE, XC, DELTA, L, K1)
      CALL JFUNC(N, M, K, X, F, I, L)
      GØ TØ 170
  220 CØNTINUE
      WRITE (KØ,99997) (X(IEV1,JB),JB=1,N)
      WRITE (KØ,99998) (F(I),I=1,K)
      WRITE (KØ,99996) (XC(J),J=1,N)
      IT = IT + 1
      IF (IT-ITMAX) 230, 230, 240
  230 CØNTINUE
      WRITE (KØ,99995) IT
      GØ TØ 80
  240 RETURN
99999 FØRMAT(1H , 15X, 21H CØØRDINATES AT PØINT, I4/8(F8.4, 2X))
99998 FØRMAT(1H , 20X, 16H FUNCTIØN VALUES, /8(F10.4, 2X))
99997 FØRMAT(1H , 20X, 16H CØRRECTED PØINT, /8(F8.4, 2X))
99996 FØRMAT(1H , 21H CENTRØID CØØRDINATES, 2X, 8(F8.4, 2X))
99995 FØRMAT(1H , //10H ITERATIØN, 4X, I5)
      END

      SUBRØUTINE JCEK1(N, M, K, X, G, H, I, KØDE, XC, DELTA, L,
     * K1)
C PURPØSE
C   TØ CHECK ALL PØINTS AGAINST THE EXPLICIT AND IMPLICIT
C   CØNSTRAINTS AND TØ APPLY CØRRECTIØNS IF VIØLATIØNS ARE
C   FØUND
C USAGE
C   CALL JCEK1(N,M,K,X,G,H,I,KØDE,XC,DELTA,L,K1)
C SUBRØUTINES REQUIRED
C   JCENT(N,M,K,IEV1,I,XC,X,L,K1)
C   JCNST1(N,M,K,X,G,H,I,L)
C DESCRIPTIØN ØF PARAMETERS
C   PREVIØUSLY DEFINED IN SUBRØUTINE JCØNSX
      DIMENSIØN X(K,L), G(M), H(M), XC(N)
   10 KT = 0
      CALL JCNST1(N, M, K, X, G, H, I, L)
C CHECK AGAINST EXPLICIT CØNSTRAINTS
      DØ 50 J=1,N
        IF (X(I,J)-G(J)) 20, 20, 30
   20   X(I,J) = G(J) + DELTA
        GØ TØ 50
   30   IF (H(J)-X(I,J)) 40, 40, 50
   40   X(I,J) = H(J) - DELTA
   50 CØNTINUE
      IF (KØDE) 110, 110, 60
C CHECK AGAINST THE IMPLICIT CØNSTRAINTS
   60 CØNTINUE
      NN = N + 1
      DØ 100 J=NN,M
        CALL JCNST1(N, M, K, X, G, H, I, L)
        IF (X(I,J)-G(J)) 80, 70, 70
```

```
 70    IF (H(J)-X(I,J)) 80, 100, 100
 80    IEV1 = I
       KT = 1
       CALL JCENT(N, M, K, IEV1, I, XC, X, L, K1)
       DØ 90 JJ=1,N
         X(I,JJ) = (X(I,JJ)+XC(JJ))/2.
 90    CØNTINUE
100 CØNTINUE
    IF (KT) 110, 110, 10
110 RETURN
    END

    SUBRØUTINE JCENT(N, M, K, IEV1, I, XC, X, L, K1)
C PURPØSE
C   TØ CALCULATE THE CENTRØID ØF PØINTS
C USAGE
C   CALL JCENT(N,M,K,IEV1,I,XC,X,L,K1)
C SUBRØUTINES REQUIRED
C   NØNE
C DESCRIPTIØN ØF PARAMETERS
C   PREVIØUSLY DEFINED IN SUBRØUTINE JCØNSX
    DIMENSIØN X(K,L), XC(N)
    DØ 20 J=1,N
      XC(J) = 0.
      DØ 10 IL=1,K1
        XC(J) = XC(J) + X(IL,J)
 10   CØNTINUE
      RK = K1
      XC(J) = (XC(J)-X(IEV1,J))/(RK-1.)
 20 CØNTINUE
    RETURN
    END
```

Remark on Algorithm 454 [E4]

The Complex Method for Constrained Optimization [Joel A. Richardson and J.L. Kuester, *Comm. ACM 16* (Aug. 1973), 487–489]

Kenneth D. Shere [Recd. 8 Oct. 1973]
Mathematical Analysis Division, Naval Ordnance Laboratory, Silver Spring, MD 20910

This algorithm can result in an infinite loop. This happens whenever the "corrected point," the centroid of the remaining "complex" points, and every point on the line segment joining these two points all have functional values lower than the functional values at each of the remaining complex points. Two examples for which this algorithm fails are [1] and [2]:

1. maximize $f(x) = -100(x_2 - x_1^2)^2 - (1-x_1)^2$
   $-10 \leq x_1, x_2 \leq 10$, initial value $(x_1, x_2) = (-2.5, 5.0)$

and

2. maximize
   $f(\theta, \phi) = 0.2\ (sin\ (\theta_0)\ cos\ (\phi_0)\ sin\ (\theta)\ cos\ (\phi) + sin\ (\theta_0)\ sin\ (\phi_0)\ sin\ (\theta)\ sin\ (\phi) + cos\ (\theta_0)\ cos\ (\theta)) - 1.0\ (sin^2\ (\theta)\ cos^2\ (\theta) + cos^2\ (\phi)\ sin^2\ (\phi)\ sin^4\ (\theta))$
   $0 \leq \theta, \phi \leq \pi/2$, $(\theta_0, \phi_0) = (.8726, .0873)$,
   initial $(\theta, \phi) = (\pi/4, \pi/4)$

Also, there is no difference in usage between M and L.

A similar method is the "simplex method" [3]. A modification to the "complex method" which uses the ideas of [3] has been programmed. The modified *JCONSX* solves each of the above problems in under 5 CP sec on a CDC 6400. The modified routine is available to interested parties upon request.

It is also worth noting that the variable *IA*, which appears in the second statement after 70 *CONTINUE* is not used elsewhere.

References

1. Rosenbrock, H.H. An automatic method for finding the greatest or least value of a function. *Comput. J. 3* (1960), 175-184.
2. Ferguson, R.E. An electromagnetism problem. (Private communication.)
3. Parkinson, J.M., and Hutchinson, D. An investigation into the efficiency of variants on the simplex method. In *Numerical Methods for Nonlinear Optimization*, F.A. Lootsma, Ed., Academic Press, New York, 1972.

Algorithm 455

Analysis of Skew Representations of the Symmetric Group [Z]

D.B. Hunter* and Julia M. Williams† [Recd. 5 Feb. 1971]
* Department of Mathematics, University of Bradford, Yorkshire, England
† 12 Peel Close, Heslington, York, England

Key Words and Phrases: symmetric group, skew representation, partition, Young diagram, lattice permutation, binary model, outer product
CR Categories: 5.30
Language: Algol

Description

This algorithm analyzes the skew representation $[\lambda]$-$[\mu]$ of the symmetric group σ_n corresponding to a pair of partitions

$(\lambda) = (\lambda_1, \lambda_2, \ldots, \lambda_r)$ and $(\mu) = (\mu_1, \mu_2, \ldots, \mu_s)$ where

$$\left.\begin{array}{l} r \geq s \\ \lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_r \\ \mu_1 \geq \mu_2 \geq \cdots \geq \mu_s \\ \lambda_i \geq \mu_i \qquad (1 \leq i \leq s) \\ n = \sum_{i=1}^{r} \lambda_i - \sum_{i-1}^{s} \mu_i \end{array}\right\} \tag{1}$$

(see Robinson [4, sec. 2.5]). The analysis takes the form

$$[\lambda]\text{-}[\mu] = \sum_{(\nu)} c_{(\nu)} [\nu], \tag{2}$$

where the summation is over all partitions (ν) of n, the coefficients $c_{(\nu)}$ being nonnegative integers.

The method used may be described as follows: construct all possible diagrams which can be built up in accordance with the following two rules.

(a) Replace μ_s of the nodes in the Young diagram corresponding to (λ) by identical symbols α_s in such a way that: (i) the unchanged nodes form a regular Young diagram; and (ii) no two identical symbols α_s lie in the same column. Then replace μ_{s-1} further nodes by identical symbols α_{s-1} in accordance with the same rules, and so on, finally replacing μ_1 nodes by identical symbols α_1.

(b) In the final diagram the altered nodes should form a lattice permutation of $\alpha_1^{\mu_1}\alpha_2^{\mu_2} \cdots \alpha_s^{\mu_s}$ (Robinson [4, sec. 2.4]) when read from right-to-left through successive rows.

Then the pattern of unchanged nodes in each diagram so constructed defines a term $[\nu]$ in the analysis.

This method appears not to have been explicitly stated in the above form before, but is an immediate consequence of Littlewood's method for analyzing the outer product $[\lambda].[\mu]$ (see Littlewood [3, sec. 6.3, th. V], Robinson [4, sec. 3.3]), noting that $c_{(\nu)}$ is also the coefficient of $[\lambda]$ in the analysis of $[\mu].[\nu]$ (Littlewood [3, sec. 6.4, th. VIII]).

In the procedure, binary models of those partitions (ν) in (2) for which $c_{(\nu)} \neq 0$ are stored, in lexicographic order, in *nu*[1], *nu*[2], . . . , *nu*[*p*], the corresponding values $c_{(\nu)}$ being stored in *c*[1], *c*[2], . . . , *c*[*p*]. The binary model used is due to Comét [1], a partition $(\nu) = (\nu_1, \nu_2, \ldots, \nu_t)$ being represented by the number

$$2^{n-\nu_1} + 2^{n-\nu_1-\nu_2} + \cdots + 2^{\nu_t} + 1. \tag{3}$$

The techniques used are similar to those employed in [2]. In particular, two two-dimensional arrays *lam* and *sigma* are required. Corresponding to any particular diagram, *lam* [*i, j*] specifies the number of nodes in row *j* which are still unchanged when all the symbols $\alpha_s, \alpha_{s-1}, \ldots, \alpha_i$ have been inserted $(j = i, i+1, \ldots, r)$, and *sigma* [*i, j*] specifies the total number of symbols α_i inserted in rows $i, i+1, \ldots, j$. Thus the quantities *lam*[*i,j*] are generated by the equation

$$lam[i,j] = lam[i{+}1,j] - sigma[i,j] + sigma[i,j{-}1]. \tag{4}$$

The rules for constructing the diagrams impose the restrictions

$$sigma[i{-}1,j{-}1] \geq sigma[i{-}1,j] - lam[i,j] + lam[i,j{+}1] \tag{5}$$

and

$$sigma[i{-}1,j{-}1] \geq sigma[i,j]. \tag{6}$$

Each time array *lam* is completed, a term

$$(\nu) = (lam[1,1], lam[1,2], \ldots, lam[1,r]) \tag{7}$$

is added to the analysis.

Note 1. In view of the identity

$$[\lambda].[\mu] = [\lambda_1{+}\mu_1, \lambda_1{+}\mu_2, \ldots, \lambda_1{+}\mu_s, \lambda_1, \lambda_2, \ldots, \lambda_r] - [\lambda_1^s],$$

procedure *skew* may also be used to analyse the outer product $[\lambda].[\mu]$. It is, however, less convenient for this purpose than procedure *outer product* of Hunter [2].

Note 2. Value of p. It is difficult to predict the value of p in any example. Clearly, $p \leq p(n)$, where $p(n)$ denotes the number of partitions of n. On the other hand, for any value of n, there are partitions (λ) and (μ) for which $p = p(n)$, namely, $(\lambda) = (n, n-1, \ldots, 1)$, $(\mu) = (n-1, \ldots, 1)$.

References

1. Comét, S. Notations for partitions. *MTAC 9* (1955), 143–146.
2. Hunter, D.B. Outer product of symmetric group representations. *BIT 10* (1970), 106–114.
3. Littlewood, D.E. *Theory of Group Characters*, 2nd ed. Oxford U. Press, England, 1950.
4. Robinson, Gilbert B. *Representation Theory of the Symmetric Group*. U. of Toronto Press, Toronto, Ont., Canada, 1961.

Algorithm

```
procedure skew (r, s, lambda, mu, p, c, nu);
  value r, s; integer r, s, p; integer array lambda, mu, c, nu;
begin
comment Input parameters.
  r:       the number of parts in partition (λ).
  s:       the number of parts in partition (μ).
  lambda:  the part λ_i is stored in lambda[i], i = 1, 2, . . . , r.
  mu:      the part μ_i is stored in mu[i], i = 1, 2, . . . , s.
  Output parameters.
  p:       the number of terms on the right in (2) for which
           c_(ν) ≠ 0.
  nu:      Binary models (3) of the partitions (ν) in (2) for which
           c_(ν) ≠ 0 are placed in lexicographic order in nu[1],
           nu[2], . . . , nu[p].
  c:       c[i] contains the coefficient c_(ν) of the partition whose
           binary model is in nu[i];
```

```
   integer i, j, k, x, y;
   integer array lam[1:s+1,1:r], sigma[1:s+1,0:r];
   p := 0; for i := 1 step 1 until s do lam[i+1,i] := lambda [i];
   for j := s+1 step 1 until r do
   begin
     lam[s+1,j] := lambda[j]; sigma[s+1,j−1] := 0
   end;
   for i := 1 step 1 until s do sigma[i, r] := mu[i];
   k := mu[s] − lambda[r]; sigma[s, s−1] := 0;
   for j := r − 1 step −1 until s do
   begin
     sigma[s, j] := if k ≥ 0 then k else 0;
     k := sigma[s, j] − lambda[j] + lambda[j+1]
   end;
   i := s;
build:
   for i := i step −1 until 1 do
   begin
     for j := i step 1 until r do
     lam[i, j] := lam[i+1, j] − sigma[i, j] + sigma[i, j−1];
     if i ≠ 1 then
     begin
       k := mu[i−1] − lam[i, r]; sigma[i−1, i−2] := 0;
       for j := r step −1 until i do
       begin
         sigma[i−1, j−1] := if k ≥ sigma[i, j] then k
         else sigma[i, j];
         k := sigma[i−1, j−1] − lam[i, j−1] + lam[i, j]
       end
     end
   end;
   x := j := 1;
   for j := j + 1 while (if j>r then false else lam[i,j]>0)
     do x := x × 2 ↑ lam[1,j] + 1;
   if (if p = 0 then true else x>nu[p]) then
   begin
     p := p + 1; nu[p] := x; c[p] := 1
   end
   else
   if x = nu[p] then c[p] := c[p] + 1
   else
   begin
     j := 1; k := p;
search:
     y := (j+k) ÷ 2; if x = nu[y] then c[y] := c[y] + 1
     else
     if nu[y] < x ∧ x < nu[y+1] then
     begin
       for k := p step −1 until y + 1 do
       begin,
         c[k+1] := c[k]; nu[k+1] := nu[k]
       end;
       c[y+1] := 1; nu[y+1] := x; p := p + 1
     end
     else
     begin
       if x < nu[y] then k := y else j := y; go to search
     end
   end;
   for i := 1 step 1 until s do
   for y := i step 1 until r − 1 do
   if sigma[i,y] < sigma[i,y+1] then
   begin
     sigma[i,y] := sigma[i,y] + 1;
     for j := y step − 1 until i do
     begin
       k := sigma[i,j] − lam[i+1,j] + lam[i+1,j+1];
       sigma[i,j−1] := if k > sigma[i+1,j] then k
       else sigma[i+1,j];
       if sigma[i,j−1] = 0 then
       begin
         for x := j − 1 step −1 until i do sigma[i,x−1] := 0;
         go to build
       end
     end
   end
end skew
```

Algorithm 456
Routing Problem [H]

Zdeněk Fencl [Recd. 16 Nov. 1970, 4 Oct. 1971, and 28 Jan. 1972]
RCA, Computer Systems Division, 200 Forest Street, Marlborough, MA 01752

The algorithm was originally developed as a part of vector ordering procedures at the Design Automation Center, RCA, Marlborough, Massachusetts, and was extended to general use in the traveling salesman and nonsymmetric routing problem.

Key Words and Phrases: routing problem, shortest path, traveling salesman problem, Hamiltonian circuit
CR Categories: 5.40
Language: Fortran

Description

The algorithm finds the shortest serial (branchless) connection between n nodes of a net beginning in the start node sn and terminating in the end node en or terminating in any node. Also given is the $m \times m$ matrix d of distances (with zero diagonal and not necessarily symmetric) between all pairs of nodes, and the vector p containing n node numbers to be connected referring to appropriate entries in the matrix d. The algorithm is constructed so that for one net (given by the matrix d) various connections, not necessarily exhausting all of m nodes, may be created; hence $n \leq m$. The case $sn = en$ is also permitted, which actually yields a Hamiltonian circuit—traveling-salesman problem. If, in input, $en = 0$, the start-to-any connection is assumed. Also as an input is the number of runs r, which is discussed below. In the output, the original vector p is replaced by conjectured optimal sequence of n nodes, and l contains the connection length. The matrix d does not need to represent a Euclidean net nor be symmetric. Thus the algorithm may serve as a more general tool to solutions of related problems.

Since the method is heuristic, which implies it is approximate, guaranty of an optimal solution is based on empiric probability. The algorithm uses a tour-building method combined with tour-to-tour improvements.

In the first phase, the tour, or sequence of nodes, is built up by successively inserting not-yet-involved nodes into the tour. If, in the middle of tour building, the tour, for instance, consists of the nodes $p_1, p_2, \ldots, p_k$, the next node among the nodes $p_{k+1}, p_{k+2}, \ldots, p_n$, and the arc (to be split by the chosen inserted node) among the arcs $p_1p_2, p_2p_3, \ldots, p_kp_1$, are chosen so that the tour increment will be minimum; i.e. i ($1 \leq i \leq k$) and $j(k < j \leq n)$ are chosen in such a manner that $d_{(p_i, p_j)} + d_{(p_j, p_{i+1})} - d_{(p_i, p_{i+1})} = \min$. Tour building starts with the arc p_1p_1 and terminates when all n nodes have been included. The tour-building approach of this kind for the traveling-salesman problem was originated by Karg and Thompson [1] and further developed by Raymond [2]. This algorithm, however, handles an open connection—start-to-end or start-to-any node. The maintenance of this property is ensured in the algorithm by assigning to the end-to-start or each-to-start distance sufficiently large negative values ($-n \times \max_{ij}[d_{ij}]$) which, in some way, firmly attach the end or any of n nodes to the start node permitting a circuit to form. In fact, the algorithm works on a net as if it were a closed circuit and keeps the node configuration by modifying the distance matrix. In output, the distance matrix is returned to its original form.

A tour thus built is hardly optimal and for larger nets it is probably far from optimum. The second phase improves the tour (for $n \geq 3$) by the so-called 3-opt method proposed by Lin [3]. Improvements consist in exchanging three arcs, or links of the given connection by three other links. If there are no more 3 links to exchange for tour improvement, the tour is said to be 3-optimal. In general, λ-optimality can be considered. The implication of the 3-link exchange is essentially in reinsertions. Consecutive node chains of length k ($1 \leq k < n$) are successively tried to be reinserted (both as are and inverted) into remaining links for tour improvements, which actually represent 3-link exchanges (and also 2-link at the same time). A 3-opt tour shows a certain probability to be an optimal one in relation to n. Different 3-opt tours can be achieved if different initial nodes are chosen, which allows us to increase the probability of obtaining an optimal solution.

The algorithm can run r trials (as specified in input) with different initial nodes (p_1, set automatically), thus obtaining different solutions while the best is saved and replaced in the vector p in output. For runs $r > n$ ($r \leq 2n$) there is little chance for further improvement, because initial nodes repeat and the tour development can be affected only by previous contents of the vector p on which the tour is built. Probability that the 3-opt tour is optimal is somewhat higher in this algoiithm, than in the one Lin suggests. In contrast to finding a 3-opt solution from a given random sequence of nodes, the fast building of an appropriate tour in the first phase considerably reduces the number of reinsertions in the second phase. The algorithm generalization to the noncyclic and nonsymmetric problems, in comparison to the traveling-salesman problem, increases computational time.

A considerable number of test examples have been run by the algorithm including the three problem types mentioned and the non-Euclidean and nonsymmetric problems. To outline the capability and how the "cost-approximation" factor r should be set for various n's, a survey of tested problems is presented, most of which problems have been solved and published before. The algorithm in Fortran was run on the RCA's SPECTRA 70/45 (fixed-point add time equals 8.88 μsec), and is recommended for a high probability (over 95 percent) of obtaining an optimum if $r = 2$ to 5 for $n \leq 10$ and $r = 5$ to 15 for $n \leq 30$. For higher n's, unless cost is out of consideration and r can be set up to $2n$, the checking of successive results is advisable to see how improvements are developing (p and l_1 should be checked after the tour-length calculation). These checks can also serve for getting suboptimal solutions.

In the program, the distance matrix d is in fixed-point mode, which makes computation faster and does not seem to be a serious restriction. Decimal order range of distances is expected to be small enough to be represented in fixed point, and calculations (additions and subtractions) will, most likely, not face overflow problem.

The arrays ID and Q should have the maximum subscript set at least to n.

The algorithm is believed to be applicable also to problems in which all connections do not necessarily exist. In terms of graph theory a graph representing the net to be routed need not be complete; i.e. every pair of vertices may be connected only in one of the two possible directions. The graph, however, must be strongly connected; i.e. there must be a path joining any pair of arbitrary distinct vertices. Nonexisting arcs might be expressed by assigning

Survey of tested problems

Ref.	*n*	*sn*	*en*	Conjectured optimum		r_{opt}	t_1 [sec]
Karg and Thompson [1]	5	1	2		118	1	<1
		1	0	*en* = 5	108	1	
		1	1		148	1	
Raymond [2]	7	1	5		165	1	<1
		1	0	*en* = 4	140	1	
		1	1		179	1	
Barachet [4]	10	1	2		350	1	1.4
		1	0	*en* = 7	298	1	
		1	1		378	2	
	10*	1	2		308	1	
		1	0	*en* = 7	257	2	
		1	1		336	2	
Author	12	1	2		102	1	3.0
		1	0	*en* = 12	95	1	
		1	1		114	1	
Author	13	1	6		117	1	3.0
		1	0	*en* = 12	102	1	
		1	1		130	1	
Held and Karp [5]	25	1	25	**	1517	10	21.8
		1	0	** *en* = 25	1517	2	22.3
		1	1	**	1711	1	29.7
Karg and Thompson [1]	33	1	33	**	10655	2	53.6
		1	0	** *en* = 14	10585	10	53.4
		1	1	**	10861	6	53.7

* Nonsymmetric problem (two distances changed: (6, 5) = 1, and (8, 3) = 1).
** Results obtained from 10 runs.

to the appropriate distances d_{kl} sufficiently large positive values, for instance $n \times \max_{ij} [d_{ij}]$.

Symbol summary

n number of nodes to be connected ($2 \leq n \leq m$).

p vector containing *n* node numbers (in output, it contains node number sequence of conjectured shortest path).

sn start node number ($1 \leq sn \leq m$; no check is provided whether *sn* is contained in *p*).

en end node number ($1 \leq en \leq m$; if $en = 0$, start-to-any connection is assumed; $en = sn$ is allowed, which is traveling-salesman problem; no check is provided whether *sn* is contained in *p*)

m order of distance matrix *d* ($m \geq n \geq 2$).

d $m \times m$ matrix of distances of all node pairs (zero diagonal, not necessarily symmetric).

l length of conjectured shortest path (output).

r number of runs (trials; $r \leq 2n$).

r_{opt} serial run number during which optimum has been achieved.

t_1 average computational time of one run in seconds.

References

1. Karg, R.L., and Thompson, G.L. A heuristic approach to solving traveling salesman problem. *Mgmt. Sci. 10*, 2 (1964), 225–248.

2. Raymond, T.C. Heuristic algorithm for the traveling-salesman problem. *IBM J. Res. Develop. 13*, 4 (1969), 400–407.

3. Lin, S. Computer solutions of the traveling salesman problem. *Bell Syst. Tech. J. 44* (Dec. 1965), 2245–2269.

4. Barachet, L.L. Graphic solution of the traveling salesman problem. *Oper. Res. 5* (1957), 841–845.

5. Held, M., and Karp, R.M. A dynamic programming approach to sequencing problems. *J. Soc. Indust. Appl. Math. 10*, 1 (1962), 196–210.

6. Saksena, J.P., and Kumar S. The routing problem with '*k*' specified nodes. *Oper. Res. 14* (1969), 909–913.

7. Bellmore, M., and Nemhauser, G.L. The traveling salesman problem: A survey. *Oper. Res. 16* (1968), 538–558.

8. Berge, C. *The Theory of Graphs and Its Applications.* Wiley, New York, 1962.

9. Berge, C., and Ghouila-Houri, A. *Programming, Games and Transportation Networks.* Wiley, New York, 1965.

Algorithm

```
      SUBRØUTINE RØUTNG(N, P, SN, EN, M, D, L, R)
      INTEGER P(N), D(M,M), ID(60), Q(60), SN, EN, R
C N - NUMBER ØF NØDES TØ BE CØNNECTED
C P - NØDE NUMBER VECTØR (IN ØUTPUT, ØPTIMAL CØNNECTIØN)
C SN- START NØDE NUMBER
C EN- END NØDE NUMBER
C M - DISTANCE MATRIX ØRDER
C D - DISTANCE MATRIX
C L - SHØRTEST CØNNECTIØN LENGTH (ØUTPUT)
C R - NUMBER ØF RUNS
C GET LARGE NUMBER (= N X MAX D(I,J))
      LARGE = 0
      DØ 20 I=1,M
        DØ 10 J=1,M
          IF (D(I,J).GT.LARGE) LARGE = D(I,J)
   10   CØNTINUE
   20 CØNTINUE
      LARGE = LARGE*N
C DEFINE NØN-EXISTING ARCS BY ASSIGNING
C THEIR DISTANCES LARGE NEGATIVE VALUES
      IF (EN.NE.0) GØ TØ 40
      DØ 30 I=1,M
        ID(I) = D(I,SN)
        D(I,SN) = -LARGE
        D(SN,SN) = 0
   30 CØNTINUE
   40 IF (SN.EQ.EN .ØR. EN.EQ.0) GØ TØ 50
      ID(1) = D(EN,SN)
      D(EN,SN) = -LARGE
C RUN R TRIALS
   50 L = LARGE
      DØ 280 IRS=1,R
C BUILD TØUR BY SUCCESSIVE INSERTING
C NØT-YET-INVØLVED NØDES
C INITIATE TØUR IS CØNSIDERED AS
C ARC P(1) TØ P(1)
        DØ 90 JS=2,N
          MININC = LARGE
C TRACE ALL NØT-YET-INVØLVED NØDES
C TØ CHØØSE THE ØNE WITH MINIMUM INCREMENT
          DØ 70 J=JS,N
            JP = P(J)
            JE = JS - 1
C FØR EACH NØT-YET-INVØLVED NØDE TRACE ALREADY
C BUILT-UP TØUR TØ CHØØSE THE MINIMUM INCREMENT ARC
            DØ 60 I=1,JE
              IP = P(I)
              IP1 = P(I+1)
              IF (I.EQ.JE) IP1 = P(1)
              INC = D(IP,JP) + D(JP,IP1) - D(IP,IP1)
              IF (INC.GE.MININC) GØ TØ 60
              J1 = J
              I1 = I
              MININC = INC
   60       CØNTINUE
   70     CØNTINUE
C STRETCH TØUR BY INSERTING THE CHØSEN NØDE P(J1)
C BETWEEN THE NØDES P(I1) AND P(I1+1)
   80     J1 = J1 - 1
          IF (J1.EQ.I1) GØ TØ 90
          IP = P(J1)
          P(J1) = P(J1+1)
          P(J1+1) = IP
          GØ TØ 80
   90   CØNTINUE
C CØRRECT TØUR BY 3-ØPT METHØD
C VARY CØNSECUTIVE CHAIN LENGTH K
        N1 = N - 1
        IF (N.LT.3) GØ TØ 210
        DØ 200 K=1,N1
          ICØUNT = 0
C SHIFT CØ.,SECUTIVE CHAIN
C THRØUGHØUT SEQUENCE ØF N NØDES
  100     ICØR = 0
          DØ 190 J=1,N
C CALCULATE CHAIN LENGTH IN FØRWARD
C AND BACKWARD DIRECTIØN
            L1 = 0
            LR = 0
            IF (K.EQ.1) GØ TØ 120
            I = J
            K1 = 1
  110       IF (I.GT.N) I = I - N
            IP = P(I)
            IP1 = I + 1
            IF (IP1.GT.N) IP1 = 1
            IP1 = P(IP1)
            L1 = L1 + D(IP,IP1)
            LR = LR + D(IP1,IP)
            I = I + 1
            K1 = K1 + 1
            IF (K1.LT.K) GØ TØ 110
C FØR EACH PØSITIØNED CHAIN (AS IS AND INVERTED)
C CHECK ALL ARCS IF INSERTIØN IMPRØVES TØUR
  120       MININC = LARGE
            J1 = J + K - 1
            IF (J1.GT.N) J1 = J1 - N
            DØ 150 I=1,N
              IF (J.LE.J1 .AND. (I.GE.J .AND. I.LE.J1)) GØ TØ
```

```
 *              150
                IF (J.GT.J1 .AND. (I.LE.J1 .ØR. I.GE.J)) GØ TØ 150
                IP = P(I)
                JP = P(J)
                JP1 = P(J1)
                IP1 = I + 1
                IF (IP1.GT.N) IP1 = 1
                JE = IP1
                IF (IP1.EQ.J) IP1 = J1 + 1
                IF (IP1.GT.N) IP1 = 1
                IP1 = P(IP1)
                LN = L1
                IR = 0
 130            INC = D(IP,JP) + LN + D(JP1,IP1) - D(IP,IP1)
                IF (INC.GT.MININC .ØR. (INC.EQ.MININC .AND.
 *              (JE.NE.J .ØR. JE.EQ.J .AND. IR.EQ.1))) GØ TØ 140
                I1 = I
                IR1 = IR
                MININC = INC
 140            IF (IR.EQ.1) GØ TØ 150
                IR = 1
                LN = LR
                JS = JP
                JP = JP1
                JP1 = JS
                GØ TØ 130
 150          CØNTINUE
              I = I1 + 1
              IF (I.GT.N) I = 1
              IF (I.EQ.J .AND. IR1.EQ.0) GØ TØ 190
C REINSERT CHAIN ØF LENGTH K STARTING IN J
C BETWEEN NØDES P(I1) AND P(I1+1)
              ICØR = 1
              JS = J
              JE = 0
              IF (IR1.EQ.0) GØ TØ 160
              JS = J1
              JE = -1
 160          K1 = 0
 170          K1 = K1 + 1
              IF (K1.GT.K) GØ TØ 190
              I = JS
              JS = JS + JE
              IF (JS.LT.1) JS = N
 180          IP = I + 1
              IF (IP.GT.N) IP = 1
              JP = P(I)
              P(I) = P(IP)
              P(IP) = JP
              I = I + 1
              IF (I.GT.N) I = 1
              IF (IP-I1) 180, 170, 180
 190        CØNTINUE
            IF (ICØR.EQ.0) GØ TØ 200
            ICØUNT = ICØUNT + 1
            IF (ICØUNT.LT.N) GØ TØ 100
 200      CØNTINUE
C ØRIENT TØUR WITH SN IN P(1)
 210      DO 230 I=1,N
            IF (P(1).EQ.SN) GØ TØ 240
            JS = P(1)
            DØ 220 J=1,N1
              P(J) = P(J+1)
 220        CØNTINUE
            P(N) = JS
 230      CØNTINUE
C CALCULATE TØUR LENGTH
 240      L1 = 0
          DO 250 I=1,N1
            IP = P(I)
            IP1 = P(I+1)
            L1 = L1 + D(IP,IP1)
 250      CØNTINUE
          IP = P(1)
          IF (SN.EQ.EN) L1 = L1 + D(IP1,IP)
C SAVE SØLUTIØN, IF BETTER, AND SET NEW INITIATE NODE
          IF (L1.GE.L) GØ TO 270
          L = L1
          DO 260 I=1,N
            Q(I) = P(I)
 260      CONTINUE
 270      J = IRS + 1
          IF (J.GT.N) J = J - N
          JS = P(1)
          P(1) = P(J)
          P(J) = JS
 280 CONTINUE
C RESTORE P AND DUMMY DISTANCES
        DO 290 I=1,N
          P(I) = Q(I)
 290 CONTINUE
        IF (EN.NE.0) GO TO 310
        DO 300 I=1,N
          D(I,SN) = ID(I)
 300 CONTINUE
 310 IF (SN.EQ.EN .OR. EN.EQ.0) GØ TO 320
        D(EN,SN) = ID(1)
 320 RETURN
        END
```

Remark on Algorithm 456 [H]
Routing Problem
[Zdeněk Fencl, *Comm. ACM 16* (Sept. 1973), 572]

Gerhard Tesch [Recd. 15 Oct. 1973] VFW Vereinigte Flugtechnische Werke GMBH, 28 Bremen 1, Hunefeldstrasse 1–5, Germany and Zdeněk Fencl, M.I.T., Department of Urban Studies, R. 9–643, Cambridge, Mass.

Some confusion arose from the description of the algorithm capability. It should have been stated that the generated tour must pass through each of the n nodes once and only once, although this is the base for the definition of the traveling salesman problem. This algorithm solves an extended traveling salesman problem in which the end node does not have to be the start node. Such connections may be sought in the design automation of serial printed circuits as well as in transportation problems. The traveling salesman problem is discussed in [3, p. 232] and methods of solution are surveyed in [1].

The users who seek the shortest paths in electric networks (the shortest connection between the two specified nodes in a net without regard to the number of nodes to be connected) are referred to Ford's shortest path algorithm [2, p. 69] and Dantzig's shortest path algorithm [3, p. 175]. There is a set of three efficient Algol algorithms by J. Boothroyd [4] handling the shortest path problem as defined in [2, p. 69] and [3, p. 175]. These Algol algorithms can be modified so that even the number of nodes may be minimized or a restriction of some nodes may be imposed, etc.

Another type of shortest path algorithm is Lee's algorithm [5 and 6]. This algorithm is applicable for the orthogonal routing of printed circuit boards.

References

1. Bellmore, M., and Nemhauser, G.L. The traveling salesman problem: A survey. *Oper. Res. 16* (1968), 538–558.
2. Berge, C. *The Theory of Graphs and Its Applications.* Wiley, New York, 1962.
3. Berge, C., and Ghouila-Houri, A. *Programming, Games and Transportation Networks.* Wiley, New York, 1965.
4. Boothroyd, J. Algorithms 22, 23, 24. Shortest path. *Comp. J. 10* (1967), 306–308.
5. Lee, C.J. An algorithm for path connections and its applications. *IEEE Trans. Elect. Comput. EC-10* (Sept. 1961), 346–365.
6. Akers, S.B. A modification of Lee's path connection algorithm. *IEEE Trans. Elect. Comput.* (Feb. 1967), 97–98.

Algorithm 457

Finding All Cliques of an Undirected Graph [H]

Coen Bron* and Joep Kerbosch† [Recd. 27 April 1971 and 23 August 1971]
* Department of Mathematics † Department of Industrial Engineering, Technological University Eindhoven, P.O. Box 513, Eindhoven, The Netherlands

Present address of C. Bron: Department of Electrical Engineering, Twente University of Technology, P.O. Box 217, Enschade, The Netherlands.

Key Words and Phrases: cliques, maximal complete subgraphs, clusters, backtracking algorithm, branch and bound technique, recursion

CR Categories: 3.71, 5.32

Language: Algol

Description

Introduction. A maximal complete subgraph (clique) is a complete subgraph that is not contained in any other complete subgraph.

A recent paper [1] describes a number of techniques to find maximal complete subgraphs of a given undirected graph. In this paper, we present two backtracking algorithms, using a branch-and-bound technique [4] to cut off branches that cannot lead to a clique.

The first version is a straightforward implementation of the basic algorithm. It is mainly presented to illustrate the method used. This version generates cliques in alphabetic (lexicographic) order.

The second version is derived from the first and generates cliques in a rather unpredictable order in an attempt to minimize the number of branches to be traversed. This version tends to produce the larger cliques first and to generate sequentially cliques having a large common intersection. The detailed algorithm for version 2 is presented here.

Description of the algorithm—Version 1. Three sets play an important role in the algorithm. (1) The set *compsub* is the set to be extended by a new point or shrunk by one point on traveling along a branch of the backtracking tree. The points that are eligible to extend *compsub*, i.e. that are connected to all points in *compsub*, are collected recursively in the remaining two sets. (2) The set *candidates* is the set of all points that will in due time serve as an extension to the present configuration of *compsub*. (3) The set *not* is the set of all points that have at an earlier stage already served as an extension of the present configuration of *compsub* and are now explicitly excluded. The reason for maintaining this set *not* will soon be made clear.

The core of the algorithm consists of a recursively defined extension operator that will be applied to the three sets just described. It has the duty to generate all extensions of the given configuration of *compsub* that it can make with the given set of candidates and that do not contain any of the points in *not*. To put it differently: all extensions of *compsub* containing any point in *not* have already been generated. The basic mechanism now consists of the following five steps:

Step 1. Selection of a candidate.
Step 2. Adding the selected candidate to *compsub*.
Step 3. Creating new sets *candidates* and *not* from the old sets by removing all points not connected to the selected candidate (to remain consistent with the definition), keeping the old sets in tact.
Step 4. Calling the extension operator to operate on the sets just formed.
Step 5. Upon return, removal of the selected candidate from *compsub* and its addition to the old set *not*.

We will now motivate the extra labor involved in maintaining the sets *not*. A necessary condition for having created a clique is that the set *candidates* be empty; otherwise *compsub* could still be extended. This condition, however, is not sufficient, because if now *not* is nonempty, we know from the definition of *not* that the present configuration of *compsub* has already been contained in another configuration and is therefore not maximal. We may now state that *compsub* is a clique as soon as both *not* and *candidates* are empty.

If at some stage *not* contains a point connected to all points in *candidates*, we can predict that further extensions (further selection of candidates) will never lead to the removal (in Step 3) of that particular point from subsequent configurations of *not* and, therefore, not to a clique. This is the branch and bound method which enables us to detect in an early stage branches of the backtracking tree that do not lead to successful endpoints.

A few more remarks about the implementation of the algorithm seem in place. The set *compsub* behaves like a stack and can be maintained and updated in the form of a global array. The sets *candidates* and *not* are handed to the extensions operator as a parameter. The operator then declares a local array, in which the new sets are built up, that will be handed to the inner call. Both sets are stored in a single one-dimensional array with the following layout:

```
                | not    | candidates
index values: 1.....ne...............ce....
```

The following properties obviously hold:

1. $ne \leq ce$
2. $ne = ce$:empty (*candidates*)
3. $ne = 0$:empty (*not*)
4. $ce = 0$:empty (*not*) and empty (*candidates*)
   = clique found

If the selected candidate is in array position $ne + 1$, then the second part of Step 5 is implemented as $ne := ne + 1$.

In version 1 we use element $ne + 1$ as selected candidate. This strategy never gives rise to internal shuffling, and thus all cliques are generated in a lexicographic ordering according to the initial ordering of the candidates (all points) in the outer call.

For an implementation of version 1 we refer to [3].

Description of the algorithm—Version 2. This version does not select the candidate in position $ne + 1$, but a well-chosen candidate from position, say s. In order to be able to complete Step 5 as simply as described above, elements s and $ne + 1$ will be interchanged as soon as selection has taken place. This interchange does not affect the set *candidates* since there is not implicit ordering.

Fig. 1. Random graphs show the computing time per clique (in ms) versus dimension of the graph (in brackets: total number of cliques in the test sample).

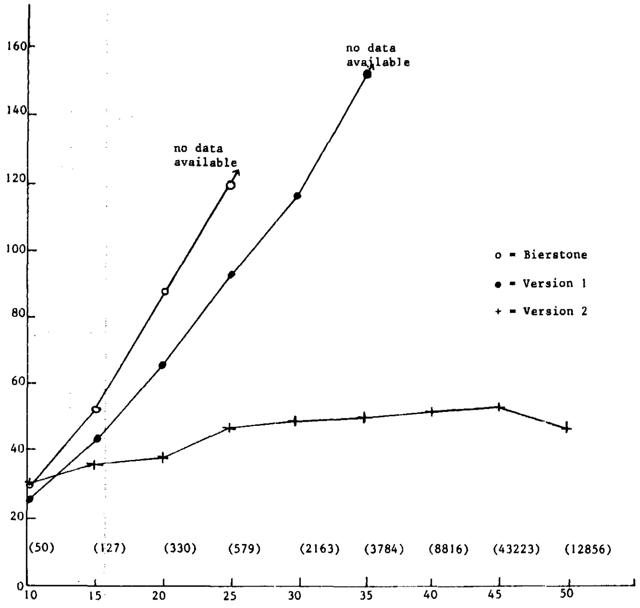
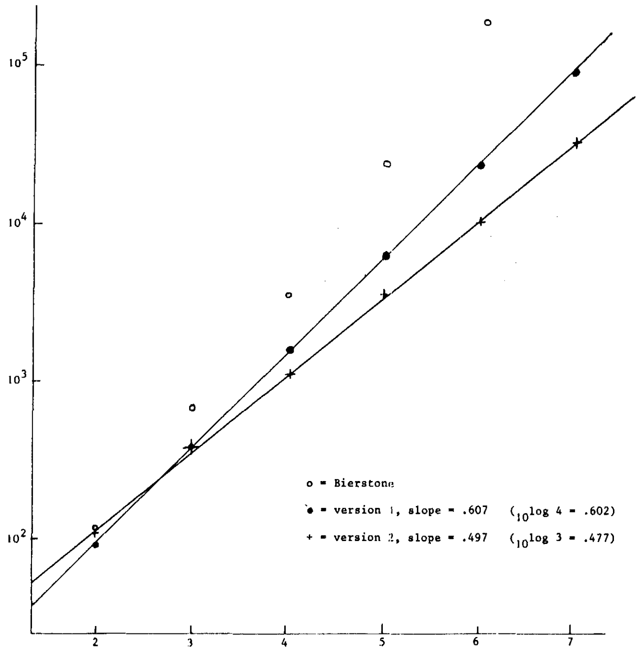


Fig. 2. Moon-Moser graphs show the computing time (in ms) versus k. Dimension of the graph = $3k$. Plotted on logarithmic scale.



The selection does affect, however, the order in which the cliques are eventually generated.

Now what do we mean by "well chosen"? The object we have in mind is to minimize the number of repetitions of Steps 1–5 inside the extension operator. The repetitions terminate as soon as the bound condition is reached. We recall that this condition is formulated as: there exists a point in *not* connected to all points in *candidates*. We would like the existence of such a point to come about at the earliest possible stage.

Let us assume that with every point in *not* is associated a counter, counting the number of candidates that this point is not connected to (*n*umber *o*f *d*isconnections). Moving a selected candidate into *not* (this occurs after extension) decreases by one all counters of the points in *not* to which it is disconnected and introduces a new counter of its own. Note that no counter is ever decreased by more than one at any one instant. Whenever a counter goes to zero the bound condition has been reached.

Now let us fix one particular point in *not*. If we keep selecting candidates disconnected to this fixed point, the counter of the fixed point will be decreased by one at every repetition. No other counter can go down more rapidly. If, to begin with, the fixed point has the lowest counter, no other counter can reach zero sooner, as long as the counters for points newly added to *not* cannot be smaller. We see to this requirement upon entry into the extension operator, where the fixed point is taken either from *not* or from the original *candidates*, whichever point yields the lowest counter value after the first addition to *not*. From that moment on we only keep track of this one counter, decreasing it for every next selection, since we will only select disconnected points.

The Algol 60 implementation of this version is given below.

Discussion of comparative tests. Augustson and Minker [1] have evaluated a number of clique finding techniques and report an algorithm by Bierstone [2] as being the most efficient one.

In order to evaluate the performance of the new algorithms, we implemented the Bierstone algorithm[1] and ran the three algorithms on two rather different testcases under the Algol system for the EL-X8.

For our first testcase we considered random graphs ranging in dimension from 10 to 50 nodes. For each dimension we generated a collection of graphs where the percentage of edges took on the following values: 10, 30, 50, 70, 90, 95. The cpu time per clique for each dimension was averaged over such a collection. The results are graphically represented in Figure 1.

The detailed figures [3] showed the Bierstone algorithm to be of slight advantage in the case of small graphs containing a small number of relatively large cliques. The most striking feature, however, appears to be that the time/clique for version 2 is hardly dependent on the size of the graph.

The difference between version 1 and "Bierstone" is not so striking and may be due to the particular Algol implementation. It should be borne in mind that the sets of nodes as they appear in the Bierstone algorithm were coded as one-word binary vectors, and that a sudden increase in processing time will take place when the input graph is too large for "one-word representation" of its subgraphs.

The second testcase was suggested by the referee and consisted of regular graphs of dimensions 3 × k. These graphs are constructed as the complement of k disjoint 3-cliques. Such graphs contain 3^k cliques and are proved by Moon and Moser [5] to contain the largest number of cliques per node.

In Figure 2 a logarithmic plot of computing time versus k is presented. We see that both version 1 and version 2 perform significantly better than Bierstone's algorithm. The processing time for version 1 is proportional to 4^k, and for version 2 it is proportional to $(3.14)^k$ where 3^k is the theoretical limit.

Another aspect to be taken into account when comparing algorithms is their storage requirements. The new algorithms presented in this paper will need at most $\frac{1}{2}M(M+3)$ storage locations to contain arrays of (small) integers where M is the size of largest connected component in the input graph. In practice this limit will only be approached if the input graph is an almost com-

[1] Bierstone's algorithm as reported in [1] contained an error. In our implementation the error was corrected. The error was independently found by Mulligan and Corneil at the University of Toronto, and reported in [6].

plete graph. The Bierstone algorithm requires a rather unpredictable amount of store, dependent on the number of cliques that will be generated. This number may be quite large, even for moderate dimensions, as the Moon-Moser graphs show.

Finally it should be pointed out that Bierstone's algorithm does not report isolated points as cliques, whereas the new algorithm does. Either algorithm can, however, be modified to produce results equivalent to the other. Suppression of 1-cliques in the new algorithm is the simplest adaption.

Acknowledgments. The authors are indebted to H.J. Schell for preparation of the test programs and collection of performance statistics. Acknowledgments are also due to the referees for their valuable suggestions.

References

1. Augustson, J.G., and Minker, J. An analysis of some graph theoretical cluster techniques, *J. ACM 17* (1970), 571–588.
2. Bierstone, E. Unpublished report. U of Toronto.
3. Bron, C., Kerbosch, J.A.G.M., and Schell, H.J. Finding cliques in an undirected graph. Tech. Rep. Technological U. of Eindhoven, The Netherlands.
4. Little, John D.C., et al. An algorithm for the traveling salesman problem. *Oper. Res. 11* (1963), 972–989.
5. Moon, J.W., and Moser, L. On cliques in graphs. *Israel J. Math.* 3 (1965), 23–28.
6. Mulligan, G.D., and Corneil, D.G. Corrections to Bierstone's algorithm for generating cliques. *J. ACM 19* (Apr. 1972), 244–247.

Algorithm

```
procedure output maximal complete subgraphs 2(connected, N);
  value N;  integer N;
  Boolean array connected;
comment The input graph is expected in the form of a symmetrical
  Boolean matrix connected. N is the number of nodes in the
  graph. The values of the diagonal elements should be true;
begin
  integer array ALL, compsub[1 : N];
  integer c;
  procedure extend version 2(old, ne, ce);
    value ne, ce;  integer ne, ce;
    integer array old;
  begin
    integer array new[1 : ce];
    integer nod, fixp;
    integer newne, newce, i, j, count, pos, p, s, sel, minnod;
    comment The latter set of integers is local in scope but need
      not be declared recursively;
    minnod := ce;  i := nod := 0;
DETERMINE EACH COUNTER VALUE AND LOOK FOR
MINIMUM:
    for i := i + 1 while i ≤ ce ∧ minnod ≠ 0 do
    begin
      p := old[i];  count := 0;  j := ne;
COUNT DISCONNECTIONS:
      for j := j + 1 while j ≤ ce ∧ count < minnod do
        if ¬ connected[p, old[j]] then
        begin
          count := count + 1;
SAVE POSITION OF POTENTIAL CANDIDATE:
          pos := j
        end;
TEST NEW MINIMUM:
      if count < minnod then
      begin
        fixp := p;  minnod := count;
        if i ≤ ne then s := pos
        else
        begin s := i;  PREINCR: nod := 1 end
      end NEW MINIMUM;
    end i;
    comment If fixed point initially chosen from candidates then
      number of disconnections will be preincreased by one;
BACKTRACKCYCLE:
    for nod := minnod + nod step −1 until 1 do
    begin
INTERCHANGE:
      p := old[s];  old[s] := old[ne + 1];
      sel := old[ne + 1] := p;
FILL NEW SET not:
      newne := i := 0;
      for i := i + 1 while i ≤ ne do
        if connected[sel, old[i]] then
        begin newne := newne + 1;  new[newne] := old[i] end;
FILL NEW SET cand:
      newce := newne;  i := ne + 1;
      for i := i + 1 while i ≤ ce do
        if connected[sel, old[i]] then
        begin newce := newce + 1;  new[newce] := old[i] end;
ADD TO compsub:
      c := c + 1;  compsub[c] := sel;
      if newce = 0 then
      begin
        integer loc;
        outstring(1, 'clique = ');
        for loc := 1 step 1 until c do
          outinteger(1, compsub[loc])
      end output of clique
      else
      if newne < newce then extend version 2(new, newne, newce);
REMOVE FROM compsub:
      c := c − 1;
ADD TO not:
      ne := ne + 1;
      if nod > 1 then
      begin
SELECT A CANDIDATE DISCONNECTED TO THE FIXED
POINT:
        s := ne;
LOOK: FOR CANDIDATE:
        s := s + 1;
        if connected[fixp, old[s]] then go to LOOK
      end selection
    end BACKTRACKCYCLE
  end extend version 2;
  for c := 1 step 1 until N do ALL[c] := c;
  c := 0;  extend version 2(ALL, 0, N)
end output maximal complete subgraphs 2;
```

Algorithm 458

Discrete Linear L1 Approximation by Interval Linear Programming [E2]

P.D. Robers* and S.S. Robers†

[Recd. 26 Feb. 1971 and 6 Oct. 1971]

* Ernst & Ernst, 1225 Connecticut Ave., NW, Washington, D.C. 26636.
† 2308 Riviera Drive, Vienna, VA 22180

Key Words and Phrases: discrete approximation, L, approximation
CR Categories: 5.13, 5.41
Language: Fortran

Description

Purpose. This subroutine finds the discrete linear L_1 approximation using the suboptimization method of interval linear programming.

Problem. The problem is stated as:

$$\text{minimize} \sum_{i=1}^{n} |\epsilon_i| \qquad (1)$$

subject to

$$Fx + \epsilon = t \qquad (2)$$

where the matrix $F = (f_{ij})$ and the vector $t = (t_i)$ are given; the vectors $\epsilon = (\epsilon_i)$ and $x = (x_j)$ are to be found ($i = 1, \dots, n$; $j = 1, \dots, m$).

Such problems arise, for instance, if a given set of data $\{(s_i, t_i): i = 1, \dots, n\}$ is to be approximated, in the sense of the L_1 norm, by a linear combination of given functions $\{g_j(s); j = 1, \dots, m\}$.

Work on this algorithm was done while P.D. Robers and S.S. Robers were employed by The Research Analysis Corporation, McLean, Va., and the Mitre Corporation, McLean, Va., respectively.

The problem is then:

$$\text{minimize} \sum_{i=1}^{n} |\epsilon_i|$$

subject to

$$\sum_{j=1}^{m} g_j(s_i)x_j + \epsilon_i = t_i, \; i = 1, \dots, n,$$

which has the form of problem ((1), (2)) if we let $f_{ij} = g_j(s_i)$ for all i and j.

Method. The algorithm works with the dual problem of ((1), (2)), which may be written:

$$\text{maximize} \quad t^T y$$

subject to

$$F^T y = 0, \quad -e \le y \le e,$$

where $e^T = (1, 1, \dots, 1)$. This problem could be solved by any linear programming algorithm. The suboptimization method of interval linear programming, however, is specially suited to solve the dual problem because of its structure. It is an iterative method which solves a subproblem at each stage.

The details of applying the suboptimization method to the L_1 approximation problem are contained in [1] and will not be presented here. A general discussion and development of the suboptimization method is contained in [2].

Program. Subroutine *APPROX* is completely self-contained and communication to it is solely through the argument list. It can be used in two modes: (1) to solve a problem from scratch; and (2) to solve a problem using an advanced start from a previous run. The advanced start mode is useful if the optimal value of the objective function is too large on a given problem (i.e. the approximation is too poor) and the problem is to be rerun after adding additional columns to the F matrix (i.e. increasing the order of the approximation). In some applications the user may wish to construct the calling program in such a way that the advanced start mode for *APPROX* is easily utilized. For example, the program might punch out information about an optimal problem solution, on request, which could automatically be read in at a later time for use as an advanced start if the problem was resolved. The main program might also contain a step-wise option which provides the capability for increasing the order of approximation iteratively until either the program runs out of data or a desired approximation accuracy is reached. The Fortran listing for a general purpose calling program which has both of the above features is available on request from the authors. Entrance to the subroutine *APPROX* is achieved by using the statement

CALL APPROX (MD, M, N, T, FT, INBASE, AINV, Y, XOPT, ZOPT, IER).

The meanings of the parameters in *APPROX* are as follows:

MD, the mode of operation indicator. Note that if $MD = 1$, the problem is to be solved from the beginning. If $MD = 2$, the problem is to be solved from an advanced start from a previous run.

M, the number of columns in the F matrix (if $MD = 2$, M must be the modified value).

N, the number of rows in the F matrix.

T, the right hand side vector for the problem (dimension N).

FT, the transpose of the F matrix (dimension $M \times N$).

INBASE, a vector which contains indices of basic columns in the optimal solution to the linear program when *APPROX* returns control (dimension N).

AINV, a matrix which contains the inverse of the matrix of optimal basic columns when *APPROX* returns control (dimension $N \times N$).

Y, a vector containing the optimal dual solution when *APPROX* returns control (dimension *N*). Note that no initial values are required for *INBASE, AINV*, and *Y* when *APPROX* is called with $MD = 1$. However, when $MD = 2$, these parameters should contain the saved values that were contained in the respective positions when *APPROX* returned control on the previous run which is now to be used as the advanced start.

XOPT, a vector containing the optimal *x*-values when *APPROX* returns control.

ZOPT, the optimal value of the objective function when *APPROX* returns control.

IER, error indicator. Note that $IER = 0$ at return is normal. If $IER = 1$ at return, a singular matrix was generated. If $IER = 2$, *APPROX* exceeded the iteration limit ($10 \times (m + 1)$). The latter two conditions are abnormal returns, and the contents of *INBASE, AINV, Y, XOPT*, and *ZOPT* are unpredictable.

As presently dimensioned, the size limitations for *APPROX* are $M \leq 15$ and $N \leq 50$. The dimension statements could clearly be changed to accommodate larger problem or ones with different proportions. Core storage and running time requirements for *APPROX* are modest. Since L_1 approximation will typically be "moderate" in size, the authors' experience indicates that *APPROX* should adequately solve all problems of practical interest, although specific tests directed at determining size limitations have not been performed. The ultimate size limitation will probably depend on the conditioning of the particular coefficient matrix, which is indeed an interesting area of study in itself.

Test Results. All tests have been performed on a CDC 6400 computer. No breakdown in the method has occurred, and in general very accurate results have been obtained.

Some examples:

(i) $t^T = (0.5, 1.0, 2.0, 3.0)$

$$F^T = \begin{Bmatrix} 1.0 & 1.0 & 1.0 & 1.0 \\ 0.0 & 1.0 & 2.0 & 3.0 \\ 0.0 & 1.0 & 4.0 & 9.0 \end{Bmatrix}$$

The optimal solution found by *APPROX* in three iterations is

$x^T = (0.5000000, 0.6666667, -0.1666667)$,

and the minimum value of (1) is

$z^* = 0.8333333$.

(ii) $t^T = (1.52, 1.025, 0.475, 0.0100, -0.475, -1.005)$

$$F^T = \begin{Bmatrix} 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 \end{Bmatrix}$$

The optimal solution found by *APPROX* in two iterations is

$x^T = (1.520000, -0.5033333)$,

and the minimum value of the objective function is

$z^* = 0.07333333$.

(iii) $t^T = (0.0, 1.5, 4.0, 3.0, 4.5, 5.0, 3.0, 7.0, 10.0)$

$$F^T = \begin{Bmatrix} 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\ 0.0 & 1.0 & 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 & 8.0 \\ 0.0 & 1.0 & 4.0 & 9.0 & 16.0 & 25.0 & 36.0 & 49.0 & 64.0 \\ 0.0 & 1.0 & 8.0 & 27.0 & 64.0 & 125.0 & 216.0 & 343.0 & 512.0 \end{Bmatrix}$$

The optimal solution found by *APPROX* after eight iterations is

$$x^T = (.7771561 \times 10^{-14}, .3333333 \times 10^{1}, -.8437500, .7291667 \times 10^{-1})$$

and the minimum objective function value is

$z^* = 5.250000$.

The above set of three problems was solved on the CDC 6400 using *APPROX* in less than four seconds of central processor time. This estimate is the complete running time including Fortran compilation time of a main program and *APPROX*.

References

1. Robers, P.D., and Ben-Israel, A. An interval programming algorithm for discrete linear L_1 approximation problems. *J. Approximation Theory*, *2*(1969), 323–336.

2. Robers, P.D., and Ben-Israel, A. A suboptimization method for interval linear programming: A new method for linear programming. *Linear Algebra and Its Applications*, *3* (1970), 383–405.

Algorithm

```
      SUBRØUTINE APPRØX (MD,M,N,T,FT,INBASE,AINV,Y,XØPT,
     * ZØPT,IER)
 C
C
C     THIS SUBRØUTINE SØLVES THE DISCRETE LINEAR L1
C APPRØXIMATIØN PRØBLEM USING THE SUBØPTIMIZATIØN METHØD ØF
C INTERVAL LINEAR PRØGRAMMING.  THE PRØBLEM TØ BE SØLVED IS
C
C           MINIMIZE Z = ABS(E(1)) + ... + ABS(E(N))
C         SUBJECT TØ
C           FX + E = T
C
C WHERE F IS A GIVEN N BY M MATRIX, T IS A GIVEN N VECTØR,
C X AND E ARE VECTØRS ØF VARIABLES HAVING DIMENSIØN M AND N
C RESPECTIVELY.
 C
C     SUBRØUTINE APPRØX IS DESIGNED TØ BE USED IN TWØ MØDES-
C (1) TØ SØLVE A PRØBLEM FRØM SCRATCH, AND
C (2) TØ SØLVE A PRØBLEM USING AN ADVANCED START FRØM A
C     PREVIØUS RUN.
C THE ADVANCED START MØDE IS USEFUL IF THE ØPTIMAL VALUE ØF
C Z IS TØØ LARGE ØN A GIVEN PRØBLEM (I.E. THE APPRØXIMATIØN
C IS TØØ PØØR) AND THE PRØBLEM IS TØ .BE RERUN AFTER ADDING
C ADDITIØNAL CØLUMNS TØ THE F MATRIX (I.E. INCREASING
C THE ØRDER ØF THE APPRØXIMATIØN).
C
C     SUBRØUTINE APPRØX IS CØMPLETELY SELF-CØNTAINED AND
C CØMMUNICATIØN IS ACHIEVED SØLELY THRØUGH THE ARGUMENT
C LIST.  THE MEANING ØF THE PARAMETERS ARE AS FØLLØWS-
C MD = THE MØDE ØF ØPERATIØN INDICATØR.
C     IF MD = 1, THE PRØBLEM IS SØLVED FRØM THE BEGINNING.
C     IF MD = 2, THE PRØBLEM IS TØ BE SØLVED USING AN
C     ADVANCED START FRØM A PREVIØUS RUN.
C M = THE NUMBER ØF CØLUMNS IN THE F MATRIX (IF MD = 2,
C     M MUST BE THE MØDIFIED VALUE.)
C N = THE NUMBER ØF RØWS IN MATRIX F.
C T = THE RIGHT HAND SIDE VECTØR FØR THE PRØBLEM.
C FT = THE TRANSPØSE ØF MATRIX F.
C INBASE = A VECTØR WHICH CØNTAINS INDICES ØF BASIC CØLUMNS
C     IN THE ØPTIMAL SØLUTIØN TØ THE LINEAR PRØGRAM WHEN
C     APPRØX RETURNS CØNTRØL.
C AINV = A MATRIX WHICH CØNTAINS THE INVERSE ØF THE MATRIX
C     ØF BASIC CØLUMNS WHEN APPRØX RETURNS CØNTRØL.
C Y = A VECTØR CØNTAINING THE ØPTIMAL DUAL SØLUTIØN WHEN
C     APPRØX RETURNS CØNTRØL.
C XØPT = A VECTØR CØNTAINING THE ØPTIMAL X-VALUES WHEN
C     APPRØX RETURNS CØNTRØL.
C ZØPT = THE ØPTIMAL VALUE ØF THE ØBJECTIVE FUNCTIØN
C     WHEN APPRØX RETURNS CØNTRØL.
C IER = ERRØR INDICATØR WHEN APPRØX RETURNS CØNTRØL.  IER=0
C     INDICATES NØRMAL RETURN.
C
C     NØ INITIAL VALUES ARE REQUIRED FØR INBASE, AINV, AND
C Y WHEN APPRØX IS CALLED WITH MD = 1.  WHEN MD = 2, AN
C ADVANCED START IS INDICATED.  THESE VARIABLES MUST THEN
C CØNTAIN THEIR FINAL VALUES FRØM THE PREVIØUS RUN.  THE
C USER WILL THUS WANT TØ MAKE PRØVISIØUS FØR SAVING THESE
C VALUES IN THE CALLING PRØGRAM SØ THAT THEY CAN BE REUSED
C IF NEEDED.
C
C
C     THE CALLING PRØGRAM AND APPRØX SHØULD CØNTAIN THE
C FØLLØWING DIMENSIØN STATEMENT-
C DIMENSIØN T(N),FT(M,N),INBASE(N),AINV(N,N),Y(N),XØPT(M)
C
C     APPRØX MUST ALSØ CØNTAIN THE FØLLØWING DIMENSIØN
C STATEMENT-
 CDIMENSIØN BP(N),BM(N),AR(N),ARAINV(N),Q(N),GAMMA(N),DEL(N)
C ,TEMP(N)
     DIMENSIØN T(50),FT(15,50),INBASE(50),AINV(50,50),Y(50)
     *,XØPT(15)
      DIMENSIØN BP(50),BM(50),AR(50),ARAINV(50),Q(50),
     * GAMMA(50),DEL(50),TEMP(50)
      INTEGER ENT,QQ,ADBASE,Q,P
      EQUIVALENCE (GAMMA,DEL)
C EPSI IS THE SINGULAR MATRIX ERRØR MESSAGE CRITERIØN.
C THE VALUE ØF EPSI CAN BE REDUCED FØR ILL CØNDITIØNED
C PRØBLEMS.
      EPSI = .0000001
      IF (MD.EQ.2) GØ TØ 70
C PRØBLEM TØ BE SØLVED FRØM THE BEGINNING.
C DEFINE INITIAL SUBPRØBLEM.
      IT = 1
      ADBASE = N+1
      DØ 20 I=1,N
        BP(I) = 1.0
        BM(I) = -1.0
        AR(I) = FT(1,I)
        INBASE(I) = I
C INITIALIZE AINV AS THE IDENTITY MATRIX.
        DØ 10 J=1,N
   10     AINV(I,J) = 0.0
   20   AINV(I,I) = 1.0
C FIND THE INITIAL Y VECTØR.
      DØ 60 I=1,N
        IF(T(I)) 30,40,50
   30   Y(I) = -1.0
        GØ TØ 60
  40   Y(I) = 0.
```

```
      GØ TØ 60
  50   Y(I) = 1.0
  60   CØNTINUE
     GØ TØ 100
CPRØBLEM TØ BE SØLVED FRØM AN ADVANCED START.
  70 ADBASE = M+N
     DØ 90 I=1,N
       IF (INBASE(I).LE.N) GØ TØ 80
       BP(I) = 0.0
       BM(I) = 0.0
       GØ TØ 90
  80   BP(I) = 1.0
       BM(I) =-1.0
  90   AR(I) = FT(M,I)
     IT = IT+1
 100 BRM = 0.0
     BRP = 0.0
C BEGIN GENERAL ITERATIØN.
C DETERMINE DEL (THE AMØUNT ØF INFEASIBILITY IN THE BØTTØM
C CØNSTRAINT ØF THE CURRENT SUBPRØBLEM).
 110 CØNTINUE
     S = 0.0
     DØ 120 I=1,N
 120   S = S + AR(I)*Y(I)
     D = S-BRP
     IF (D.GT.0.) GØ TØ 130
     D = S - BRM
     IF (D.GE.0.) GØ TØ 430
 130 CØNTINUE
     DØ 140 I=1,N
       ARAINV(I) = 0.0
       DØ 140 J=1,N
 140     ARAINV(I) = ARAINV(I)+AR(J)*AINV(J,I)
C CALCULATE GAMMA VECTØR (THE VECTØR ØF MARGINAL CØSTS FØR
C MØVING TØWARD FEASIBILITY).
     DØ 170 I=1,N
      TEMP(I) = 0.0
       DØ 150 J=1,N
 150     TEMP(I) = TEMP(I)+T(J)*AINV(J,I)
       IF (ARAINV(I).NE.0.0)GØ TØ 160
       GAMMA(I) = -1.0
       GØ TØ 170
 160   GAMMA(I) = TEMP(I)/ARAINV(I)
       IF (D.LT.0.0) GAMMA(I)=-GAMMA(I)
 170   CØNTINUE
C FIND Q VECTØR (THE VECTØR ØF INDICES WHICH INDICATE THE
C VARIABLES WHICH CAN BE CHANGED TØ MØVE TØWARD FEASIBILITY).
     QQ=0
     DØ 210 L=1,N
       DØ 180 I=1,N
         IF (GAMMA(I).LT.0.0) GØ TØ 180
         S = GAMMA(I)
         J=I
         GØ TØ 190
 180     CØNTINUE
       GØ TØ 215
 190   DØ 200 I=1,N
         IF (GAMMA(I).LT.0. .ØR. GAMMA(I).GE.S)GØ TØ 200
         S = GAMMA(I)
         J=I
 200     CØNTINUE
       QQ=QQ+1
       Q(L) = J
 210   GAMMA(J) = -1.0
C CALCULATE DELTA VECTØR (THE VECTØR INDICATING THE MAXIMUM
C PERMISSABLE CHANGES IN THE VARIABLES).
 215 DØ 260 I=1,QQ
       K=Q(I)
       S = 0.0
       II = INBASE(K)
       IF (II.LE.N) GØ TØ 230
       L = II-N
       DØ 220 J=1,N
 220     S = S + FT(L,J)*Y(J)
       GØ TØ 240
 230   S = Y(II)
 240   IF(D*ARAINV(K).LE.0.) GØ TØ 250
       DEL(K) = BM(K)-S
       GØ TØ 260
 250   DEL(K) = BP(K)-S
 260   CØNTINUE
C DETERMINE P (THE NUMBER ØF VARIABLES CHANGED
C THIS ITERATIØN).
     DØ 280 I=1,QQ
       P = I
       S = 0.0
       DØ 270 J=1,I
         K = Q(J)
 270     S = S+DEL(K)*ARAINV(K)
       IF (ABS(S).GE.ABS(D)) GØ TØ 290
 280   CØNTINUE
C CALCULATE THETA (THE AMØUNT WHICH THE PTH VECTØR IS
C CHANGED).
 290 L = P-1
     S=0.0
     IF (L .LT. 1) GØ TØ 310
     DØ 300 J=1,L
       K = Q(J)
 300   S = S+DEL(K)*ARAINV(K)
 310 K=Q(P)
     THETA = -(D+S)/ARAINV(K)
C UPDATE Y VECTØR (THE ØPTIMAL SØLUTIØN TØ THE CURRENT
C SUBPRØBLEM).
     DØ 320 I=1,N
 320   TEMP(I) = 0.0
     IF (L .LT. 1) GØ TØ 340
     DØ 330 I=1,L
       K = Q(I)
 330   TEMP(K) = DEL(K)
 340 K = Q(P)
     TEMP(K) = THETA
     DØ 360 I=1,N
       S = 0.0
       DØ 350 J=1,N
 350     S = S+AINV(I,J)*TEMP(J)
 360   Y(I) = Y(I)+S
     K = Q(P)
     BP(K) = BRP
     BM(K) = BRM
     INBASE(K) = ADBASE
C CALCULATE NEW AINV MATRIX.
     DØ 370 I=1,N
       TEMP(I) = 0.0
       DØ 370 L=1,N
 370     TEMP(I) = TEMP(I)+AR(L)*AINV(L,I)
     IF (ABS(TEMP(K)).GT. EPSI ) GØ TØ 380
C SINGULAR MATRIX INDICATED. SET ERRØR TAG AND TERMINATE.
     IER = 1
     RETURN
 380 DØ 390 I=1,N
 390   AINV(I,K) = AINV(I,K)/TEMP(K)
     DØ 420 J=1,N
       IF(J.EQ.K) GØ TØ 410
       DØ 400 I=1,N
         AINV(I,J) = AINV(I,J)-AINV(I,K)*TEMP(J)
 400     CØNTINUE
 410   CØNTINUE
 420   CØNTINUE
C FIND S (THE LARGEST INFEASIBILITY), AND ENT(THE INDEX ØF
C THE CØRRESPØNDING CØNSTRAINT).
 430 TEMP(1) = 0.
     L = M+N
     DØ 510 I=1,L
       DØ 440 J=1,N
         IF(INBASE(J).EQ.I) GØ TØ 510
 440     CØNTINUE
       IF(I.LE.N) GØ TØ 470
       S = 0.
       II = I-N
       DØ 450 J=1,N
 450     S = S + FT(II,J)*Y(J)
       IF(S.EQ.0.) GØ TØ 510
 460   IF(ABS(S) .LE. TEMP(1)) GØ TØ 510
       TEMP(1) = ABS(S)
       ENT = I
       GØ TØ 510
 470   S = Y(I)
       IF (S-1.) 490,490,480
 480   S = S-1.
       GØ TØ 460
 490   IF (S+1.) 500,510,510
 500   S = S+1.
       GØ TØ 460
 510   CØNTINUE
     S = TEMP(1)
     IF (S.EQ.0.) GØ TØ 560
C PRESENT SØLUTIØN INFEASIBLE. START THE NEXT ITERATIØN.
     IT = IT+1
C DEFINE THE NEXT SUBPRØBLEM.
     IF (ENT.LE.N) GØ TØ 530
     BRP = 0.0
     BRM = 0.0
     L = ENT-N
     DØ 520 J=1,N
 520   AR(J) = FT(L,J)
     GØ TØ 550
 530 BRP = 1.0
     BRM = -1.0
     DØ 540 J=1,N
 540   AR(J) = 0.0
     AR(ENT) = 1.0
 550 ADBASE = ENT
     IF (IT.LE.10*(M+1)) GØ TØ 110
C ITERATIØN LIMIT EXCEEDED. SET ERRØR TAG AND TERMINATE.
     IER = 2
     RETURN
C ØPTIMAL DUAL SØLUTIØN FØUND. CALCULATE PRIMAL SØLUTIØN.
 560 DØ 600 J=1,M
       L = J+N
       DØ 570 I=1,N
         IF(INBASE(I).EQ.L) GØ TØ 580
 570     CØNTINUE
       XØPT(J) = 0.
       GØ TØ 600
 580   TEMP(I) = 0.
       DØ 590 L=1,N
 590     TEMP(I) = TEMP(I)+T(L)*AINV(L,I)
       XØPT(J) = TEMP(I)
 600   CØNTINUE
     ZØPT = 0.
     DØ 610 I=1,N
 610   ZØPT = ZØPT + Y(I)*T(I)
     RETURN
     END
```

Algorithm 459

The Elementary Circuits of a Graph [H]

Maciej M. Syslo [Recd. 30 Apr. 1971 and 15 Aug. 1972]
Department of Numerical Methods, University of Wroclaw, Wroclaw, pl. Grunwaldzki 2/4, Poland

Key Words and Phrases: algorithm, graph theory, circuit search algorithm, path search algorithm, searching
CR Categories: 3.74, 4.22, 5.32
Language: Algol

Description

This algorithm investigates the existence of elementary circuits of a directed graph G.

Data: n is the number of vertices; *arc*(i,j) is the Boolean procedure with two parameters i, j of type integer, which is equal to **true** if $(i, j) \in G$, and **false** otherwise.

Results:

(a) If the graph has no circuits, then the following sequence of symbols will be printed:

Graph without elementary circuits.
Ordered numeration of vertices $i_1\ i_2\ i_3\ \cdots\ i_n$
where $(i_1, i_2, \ldots, i_n)$ is the permutation of numbers $(1, 2, \ldots, n)$, and a new numeration of vertices such that if $(j, i) \in G$, then $j < i$.

(b) In the other case the following sequence of symbols will be printed:

Graph contains the circuits:
Circuit $i_1\ i_2\ \cdots\ i_r\ i_1$
Circuit $j_1\ j_2\ \cdots\ j_s\ j_1$
. . .

Every elementary circuit will be printed once and only once.

Method. This Algol program is based on the well-known method used while searching for cycles (circuits) in oriented graphs ([1, 2]). However, before the beginning of this method, vertices which do not belong to any circuits are labeled ($s[i] = n2$). The process uses only two arrays: *nodes*$[1:n]$, which contains either the ordered numeration of vertices or the vertices of the elementary path of the move; and $s[1:n]$, the ith element of which denotes the investigation phase of vertex i.

If the incidence matrix is stored one bit per entry, the process needs $n[n/w] + 2n$ machine words, where w is the number of bits in a machine word.

The program has been run on the ODRA-1204 computer and numerous examples were tested, including complete graphs.

References

1. Tiernan, J.C. An efficient search algorithm to find the elementary circuits of a graph. *Comm. ACM 13* (Dec. 1970), 722–726.
2. Vantrusov, Ju.I. *About the Analysis of Finite Graph, in Mathematical Programming* (in Russian). Moscow, 1966, pp. 68–77.

Algorithm

```
begin
  integer n;
  ininteger (2, n);
  begin
    integer array s, nodes [1: n];
    integer i, j, k, k1, k2, k3, k4, n1, n2, sj;
    Boolean f;
    comment The body of procedure arc and all other declara-
      tions connected with it should be inserted here;
    n1 := -n - 1;    n2 := -n - 2;
    f := true;
    for i := 1 step 1 until n do s[i] := 0;
    for k := 1, k + 1 while k ≤ n ∧ f do
    begin
      for i := 1 step 1 until n do
      if s[i] = 0 then
      begin
        for j := 1 step 1 until n do
        if s[j] = 0 then
        begin
          if arc(j, i) then go to nexti;
        end s[j] = 0, j;
        nodes[k] := i; s[i] := n2;
        go to nextk;
nexti:
      end s[i] = 0, i;
      f := false;
nextk:
    end k;
    if f then
    begin
      outstring (1, 'Graph without elementary circuits.');
      outstring (1, 'Ordered numeration of vertices');
      outarray (1, nodes)
    end f
    else
    begin
rep:
      for j := 1 step 1 until n do
      if s[j] = 0 then
      begin
        for i := 1 step 1 until n do
        if s[i] = 0 then
        begin
          if arc(j, i) then go to nextj
        end s[i] = 0, i;
        s[j] := n2;
        go to rep;
nextj:
      end s[j] = 0, j;
      outstring(1, 'Graph contains the circuits:');
      k2 := 1;
scan:
      for k3 := s[k2] while (k3 = n2 ∨ k3 = n1) ∧ k2 < n do
      begin
        if k3 = n1 then s[k2] := n2;
        k2 := k2 + 1
      end k3;
      for k := k2 + 1 step 1 until n do
      if s[k] = n1 then s[k] := n2;
```

```
        if k3 = 0 then
        begin
          i := 1; k1 := nodes[1] := k2;
cd:
          i := i + 1;
cd1:
          for j := abs(s[k2]) + 1 step 1 until n do
          begin
            sj := s[j];
            if sj ≠ n2 then
            begin
              if arc(j, k2) ∧ (k3 = 0 ∨ sj = n1 ∨ sj ≥ 0) then
              begin
                s[k2] := if k3 = 0 then j else −j;
                if sj = n1 then
                begin
                  if k3 = 0 then k3 := k2;
                  s[j] := 0
                end sj = n1;
                if s[j] > 0 then
                begin
                  outstring(1, 'Circuit');
                  k4 := 0; k := i;
                  outinteger(1, j);
                  for k := k − 1 while k4 ≠ j do
                  begin
                    k4 := nodes[k];
                    outinteger(1, k4)
                  end k;
                  go to cd1
                end s[j] > 0
                else
                begin
                  k2 := nodes[i] := j;
                  go to cd
                end sj ≤ 0
              end arc(j, k2)···
            end sj ≠ n2
          end j;
          s[k2] := n1;
          if k2 ≠ k1 then
          begin
          i := i − 1;
          if k3 = i − 1 then k3 := 0;
          k2 := nodes[i − 1];
          go to cd1
          end k2 ≠ k1;
          go to scan
        end k3 = 0
      end ¬ f
    end
end
```

Remark on Algorithm 459 [H]

The Elementary Circuits of a Graph [Maciej M. Syslo, *Com. ACM 16* (Oct. 1973), 632–633]

Maciej M. Syslo (Recd 11 Feb. 1974) Department of Numerical Methods, University of Wroclaw, pl. Grunwaldzki 2/4, 50384 Wroclaw, Poland

Corrections are needed in the algorithm:

(i) Insert:

*k*3 := *s*[*k*2];

after the statement **end** *k*3;

(ii) The 9th line from the end of the algorithm

if *k*3 = *i* − 1 **then** *k*3 := 0;

and insert the line

if *k*3 = *k*2 **then** *k*3 := 0;

before the statement **go to** *cd*1.

Algorithm 460

Calculation of Optimum Parameters for Alternating Direction Implicit Procedures [D3]

Paul E. Saylor
Department of Computer Science, University of Illinois,
Urbana, IL 61801
and
James D. Sebastian
Boeing Computer Services, Seattle, WA 98124

[Recd. 26 May 1971 and 12 Nov. 1971]

Key Words and Phrases: elliptic difference equations, Peaceman-Rachford, Douglas-Rachford, W. B. Jordan, optimum parameters, alternating-direction-implicit, ADI
CR Categories: 5.14, 5.17
Language: Fortran

Description

Purpose. Let $Gz = s$ be a system of simultaneous equations, where G is a positive-definite matrix, s is a known vector, and z an unknown vector. Such systems arise, for example, as the result of the discretization of an elliptic boundary value problem. Beginning with an initial approximation z_0, one version of the Alternating Direction Implicit (*ADI*) method [2] determines successive approximations to the true solution, z, from two iterative formulas,

$$z_{k+1/2} = (H + \omega_{kH}I)^{-1}s - (H + \omega_{kH}I)^{-1} (V - \omega_{kH}I)z_k$$

and

$$z_{k+1} = (V + \omega_{kV}I)^{-1}s - (V + \omega_{kV}I)^{-1} (H - \omega_{kV}I)z_{k+1/2},$$

$k = 0, 1, \dots, m - 1$, where H and V are symmetric matrices such that $G = H + V$, I is the identity matrix, ω_{kH} and ω_{kV} are parameters chosen to accelerate convergence, and m is the number of iterations. When H and V commute, i.e. $HV = VH$, the parameters that yield fastest convergence for fixed m, the optimum parameters, are the solution to a min-max problem that has been completely solved by W.B. Jordan using techniques of elliptic function theory [1, App. and 2].

An algorithm for computing optimum parameters based on Jordan's solution does not appear to be generally available, and it is our aim to provide one here.

Method. The formulas used in the subroutines are taken from the solution of Jordan as presented in [2]. We refer to [2] for their derivation, and observe here only that, given either m or μ_m, but not both, where μ_m is the spectral norm of the m-step error propagation matrix

$$T_m = \prod_{k=0}^{m-1} (V + \omega_{kV}I)^{-1}(H - \omega_{kV}I)(H + \omega_{kH}I)^{-1}(V - \omega_{kH}I),$$

the subroutine computes the parameters ω_{kV} and ω_{kH} that minimize the value of m or μ_m, which is not given, while satisfying the inequality

$$\| z - z_m \|_2 \leq \mu_m \| z - z_0 \|_2 . \tag{1}$$

This makes the Jordan algorithm more flexible than alternative methods of computing parameters, due to Peaceman and Rachford [4] and Douglas and Rachford [5]. These methods compute an integer m and a satisfactory but not optimal sequence of m parameters such that (1) is satisfied, given μ_m. Unlike the Jordan algorithm, it is impractical to specify the number m of iterations then compute a sequence of m parameters and an estimate of μ_m.

For $m = 2^k$, where k is a nonnegative integer, another algorithm for computing an optimum sequence of m parameters and an estimate of μ_m is due to Wachspress [2]. Again, the greater flexibility of the Jordan algorithm is apparent. However, it employs truncated infinite series, whereas, for $m = 2^k$, the Wachspress algorithm only requires the approximation of square roots.

Program. The number of iterations, m, and the spectral radius, μ_m, are represented in the argument list by *ITNS* and *DMU* respectively. Iteration parameters ω_{kH} and ω_{kV} are the kth entries of the arrays *OMEH* and *OMEV* respectively. The dimension of each array is the value of N. The program parameter *IOPT*, specified on entry, determines one of two options:

(i) If *IOPT* has the value 1, then *ITNS* must be specified on entry. Optimum parameters *OMEH*(1), *OMEV*(1), ..., *OMEH*(*ITNS*), *OMEV*(*ITNS*) are computed together with the value of *DMU*.

(ii) If *IOPT* has the value 2, then *DMU* must be specified on entry. A value of *ITNS* is then computed with optimum parameters *OMEH*(1), *OMEV*(1), ..., *OMEH*(ITNS), *OMEV*(*ITNS*) such that *ITNS* is the minimum number of iterations for which μ_m is less than or equal to the value of *DMU*.

In option (ii), if *ITNS.GT.N* is satisfied, then *ITNS* is set equal o N, corresponding optimum parameters are computed, and the error flag *IER* is set to 2. Other possible values of *IER* are 0 and 1. These indicate that computation was normal or that some input parameter was improper.

Estimates of the minimum and maximum eigenvalues of H are assigned on entry to parameters A and B respectively. Estimates of the minimum and maximum eigenvalues of V are assigned to C and D. Gerschgorin's theorem yields satisfactory estimates of B and D, whereas estimates of A and C may be computed from an algorithm suggested by Wachspress [3].

Machine dependent constants. The constants −90, −10, 10, and 30 in the three *IF* statements following the last comment card in the program are machine dependent. At the point where this comment occurs, *DMU* is to be computed from the formula

$$DMU = (2.D0*DEXP(TEMP)/$$
$$(1.D0 + 2.D0*DEXP(TEMP)**4))**2$$

but for greater efficiency and to avoid underflow, overflow, or argument out of range conditions on the IBM 360, the formula actually used to compute *DMU* is chosen according to the value of *TEMP*. These constants are used as follows: If $TEMP \leq -90$ or $TEMP \geq 30$, then $DMU < 10**-77$, as may be verified from the above formula, and the program simply sets $DMU = 0$. Let $dfl(X)$ denote the IBM 360 Fortran internal double precision floating point representation of X. It is easily verified that if $TEMP \leq -10$ then

$$dfl(1.D0 + 2.D0*DEXP(TEMP)**4) = dfl(1.D0),$$

and if $TEMP \geq 10$, then

$$dfl(2.D0 + DEXP(TEMP)**-4) = dfl(2.D0).$$

Thus *DMU* is computed to full machine precision from $DMU = 4.D0*DEXP(2.D0*TEMP)$ when $TEMP \leq -10$, and from $DMU = DEXP(-6.D0*TEMP)$ when $TEMP \geq 10$. Finally, *DMU* is computed from the formula given at the beginning of this section when $-10 < TEMP < 10$.

Tests. The program has been tested on the 360/75 by applying the parameters to the solution by *ADI* of $Gz = s$, with $z = s = 0$. In each test, G is a 900 by 900 matrix obtained from discretizing $\alpha\partial^2/\partial x^2 + \beta\partial^2/\partial y^2$, α and β constants. Therefore, $G = \alpha H + \beta V$, where H and V are discrete analogs of $\partial^2/\partial x^2$ and $\partial^2/\partial y^2$ respectively. The initial vector, z_0, was chosen to have a nonzero component in the direction of each of the eigenvectors of H or V.

To test option (i), two pairs of values of α and β were used. For $\alpha = \frac{1}{2}$, $\beta = 2$, called the model problem *ITNS* was assigned the values $ITNS = 1, 2, \ldots, 20$. For $\alpha = \frac{1}{2}$, $\beta = 200$, called the generalized model problem, and considered a more difficult problem for *ADI*, *ITNS* was assigned $ITNS = 15, \ldots, 20$. In each case $z_1, \ldots, z_{ITNS}$ were computed and the validity of *E.LT.DMU* was tested where E is the l_2 relative error defined by

$$E = \| z - z_{ITNS} \|_2 / \| z - z_0 \|_2 . \qquad (2)$$

With $\alpha = \frac{1}{2}$, $\beta = 2$, the comparison *E.LT.DMU* was satisfied for $ITNS = 1, \ldots, 29$, whereas for $ITNS = 30$, it failed. Performance of the program may nevertheless be considered satisfactory since for $ITNS = 30$, *DMU* was less than .9 $D-17$, a value beyond practical interest and sufficiently small that one may expect to observe roundoff. In the second case for $\alpha = \frac{1}{2}$, $\beta = 200$, *E.LT.DMU* was satisfied for $ITNS = 15, \ldots, 18$, whereas for 19 and 20 the comparison fails. For each failure, *DMU* was less than .2 $D-25$.

To test option (ii), parameter *DMU* was assigned the values $DMU = 10^{-i}$ for $i = -1, -3, \ldots, -15$, then *ITNS* and the optimum parameters for *ITNS* iterations were computed and the validity of *E.LT.DMU* checked. For each value of *ITNS*, *E.LT.DMU* was satisfied for both problems.

Observe that tests of this kind are not in fact objective and do not test whether the iteration parameters are optimal; they verify that values of *DMU* or *ITNS*, depending on the option, are consistent with the results obtained by solving actual problems with the computed iteration parameters. To evaluate the accuracy of the program more objectively, we compared *ADIP* as follows to a FORTRAN version of the Wachspress algorithm for computing exact parameters when the number of iterations is a power of 2. Values of the optimum parameters and the spectral radius of the iteration matrix were computed from each program for 2, 4, 8, 16, and 32 iterations, with other input data taken from the model problem and generalized model problem described above. In addition each set of optimum parameters was applied to the solution of the model problem and generalized model problem.

Comparisons between the output of each program were made by computing the relative difference of the spectral radii and each pair of optimal parameters. (The relative difference of a_j and a_w is defined to be $| a_j - a_w | / a_j$ where a_j is computed from *ADIP* and a_w is computed from the Wachspress algorithm.)

For the model problem the relative difference of each quantity was bounded by 10^{-5} for 2, 4, 8, and 16 iterations. For each number of iterations, the l_2 relative errors (2) of each pair of computed solutions were in agreement to five significant digits. In each case the relative error of the computed solution as computed from *ADIP* parameters was larger (in the sixth decimal place or higher) than the relative error computed from the Wachspress exact parameter program. This confirms the expectation that the Wachspress algorithm is more accurate, although the difference is slight, since this is a comparison of *relative* error.

For 32 iterations in the solution of the model problem, the differences between the two algorithms were somewhat greater, but with the performance of *ADIP* superior. The l_2 relative error in the solutions as computed by *ADIP* and the Wachspress program were respectively .71 $D-18$ and .75 $D-18$. The difference in these values is not significant. For, 32 is an unrealistic number of iterations. Also, any difference in relative error does not imply the same difference in accuracy of the computed solutions. Here, each approximation agrees with the exact solution to 18 significant figures in the l_2 sense. Rather than this, the significant feature of the comparison is that *ADIP* is more reliable when input parameters are nontypical. This is also evident in testing with the generalized model problem.

In runs of the generalized model problem for 2, 4, and 8 iterations, the differences between corresponding l_2 relative errors of the approximate solution were greater than for the model problem but still insignificant. For 2 and 4 iterations, the l_2 relative error obtained by *ADIP* parameters was smaller than that obtained by parameters from the Wachspress exact parameter program. Parameters from the Wachspress program yielded more accurate results only for 8 iterations. For 16 and 32 iterations, the response of the Wachspress exact parameter program was bizarre. Certain parameter values returned were negative whereas exact parameters are positive. In each case, the spectral radius was assigned the value zero. Of course the conditions of the runs are extreme. They represent an attempt to reduce the l_2 relative error to unrealistically small values. The results again indicate that *ADIP* performs more reliably under adverse conditions. In fact, *ADIP* is self-consistent for 16 iterations in reducing the relative error to less than the computed value of the spectral radius, although for 32 iterations, the self consistency test fails.

In conclusion, these tests indicate that the Wachspress exact parameter program yields more accurate values under ideal conditions, but that the difference is of no practical significance. When the requirements of the problem are severe or fanciful, *ADIP* is more reliable than the Wachspress exact parameter program.

References

1. Wachspress, E.L. Extended application of alternating direction implicit model problem theory. *SIAM J. Appl. Math. 11* (1963).

2. Wachspress, E.L. *Iterative Solution of Elliptic Systems.* Prentice-Hall, Englewood Cliffs, N. J., 1966.

3. Wachspress, E.L. Numerical solution of neutron diffusion problems. In *Numerical Solution of Field Problems in Continuum Physics.* SIAM-AMS Proc. Vol. 2, AMS, Providence, R. I., 1970.

4. Peaceman, D.W., and Rachford, H.H. The numerical solution of parabolic and elliptic differential equations. *J. Soc. Ind. Appl. Math. 3* (1955), 28–41.

5. Douglas, J., and Rachford, H.H. On the numerical solution of heat conduction problems in two and three space variables. *Trans. A M S 82* (1956), 421–439.

Algorithm

```
      SUBRØUTINE ADIP(A, B, C, D, IØPT, N, ITNS, DMU, ØMEH,
     * ØMEV, IER)
      DØUBLE PRECISIØN A, ALFA, B, BETA, BMD, BPD, C, CMA, CPA,
     * D, DEL, DEXP, DKPR, DLØG, DM, DMU, DRJ, DSQRT, ØJ,
     * ØMEH(N), ØMEV(N), PISQ, TEMP, TEMPA, TEMPB, TEMPC
      DATA PISQ/9.869604401089359D0/
C GIVEN A MATRIX EQUATIØN GZ=S, WHERE G IS A REAL PØSITIVE
C DEFINITE MATRIX, S IS A KNØWN, AND Z THE UNKNØWN, VECTØR.
C LET H AND V BE SYMMETRIC CØMMUTING MATRICES SUCH THAT
C G=H+V. BEGINNING WITH AN INITIAL APPRØXIMATIØN Z(0), LET
C SUCCESSIVE APPRØXIMATIØNS TØ Z BE GENERATED FRØM
C Z(K+1/2)=(H+ØMEH(K)*I)**(-1)*(S-(V-ØMEH(K)*I)*Z(K))
C Z(K+1)  =(V+ØMEV(K)*I)**(-1)*(S-(H-ØMEV(K)*I)*Z(K+1/2)),
C WHERE I IS THE IDENTITY MATRIX. FINALLY, LET ØNE ØF ITNS
C AND DMU BE GIVEN. THEN THIS SUBRØUTINE CØMPUTES THE PAR-
C AMETERS ØMEH(K), ØMEV(K) THAT MINIMIZE THE VALUE ØF
C DMU AND ITNS WHICH IS NØT GIVEN WHILE SATISFYING THE
C INEQUALITY /Z-Z(ITNS)/.LE.DMU*/Z-Z(0)/, WHERE // DENØTES
C THE EUCLIDEAN NØRM.
C THE SUBRØUTINE ARGUMENTS HAVE THE FØLLØWING MEANING.
C A AND B ARE LØWER AND UPPER BØUNDS, RESPECTIVELY, ØN THE
C EIGENVALUES ØF H. C AND D ARE LØWER AND UPPER BØUNDS,
C RESPECTIVELY, ØN THE EIGENVALUES ØF V. THE VALUES ØF A,B,
C C, AND D MUST SATISFY THE INEQUALITIES 0.LT.A.LE.B, AND
C 0.LT.C.LE.D.
C IØPT DENØTES THE INPUT ØPTIØN. IF IØPT=1 THEN THE VALUE ØF
C ITNS MUST BE SPECIFIED ØN ENTRY AND DMU WILL BE CØMPUTED.
C IF IØPT=2 THEN THE VALUE ØF DMU MUST BE SPECIFIED ØN ENTRY
C AND ITNS WILL BE CØMPUTED.
C IF IØPT=1 THEN THE INEQUALITY 1.LE.ITNS.LE.N MUST BE
C SATISFIED, WHILE IF IØPT=2 THEN THE INEQUALITIES N.GE.1
C AND DMU.GT.0 MUST BE SATISFIED.
C N IS THE DIMENSIØN ØF THE ARRAYS ØMEV AND ØMEH.
C ITNS IS THE NUMBER ØF ITERATIØNS TØ BE PERFØRMED.
C DMU IS A BØUND ØN THE SPECTRAL NØRM ØF THE ITERATIØN
C MATRIX TØ THE ITNS PØWER. IF IØPT=2 A VALUE FØR DMU MUST
C BE SPECIFIED ØN ENTRY, AND THIS VALUE MAY BE CHANGED BY
C ADIP (SEE IER, BELØW).
C THE VALUES ØF THE REQUIRED PARAMETERS ARE CØNTAINED IN THE
C LØCATIØNS ØMEV(K), ØMEH(K), K=1,...,ITNS ØN EXIT FRØM
C ADIP.
C IER IS A VARIABLE WHØSE VALUE ØN EXIT FRØM ADIP HAS THE
C FØLLØWING MEANING
C IER=0 SIGNIFIES CØMPUTATIØN ØF THE PARAMETERS HAS BEEN
C PERFØRMED WITH NØ CHANGE ØF THE VALUES SPECIFIED ØN ENTRY.
C IER=1 SIGNIFIES THAT SØME INPUT VALUE VIØLATES THE
C CØNSTRAINTS GIVEN ABØVE, AND HENCE THE PARAMETERS HAVE NØT
C BEEN CØMPUTED.
C IER=2 (PØSSIBLE ØNLY IF IØPT=2) SIGNIFIES THAT FØR THE
C GIVEN VALUE ØF DMU, THE CØMPUTED VALUE ØF ITNS WØULD BE
C GREATER THAN N, SØ THAT ITNS HAS BEEN SET EQUAL TØ N AND
C DMU HAS BEEN RECØMPUTED AS FØR IØPT=1.
C TEST INPUT VALUES FØR RANGE
      IER = 1
      IF ( .NØT. (A.GT.0.D0 .AND. A.LE.B .AND. C.GT.0.D0 .AND.
     * C.LE.D)) GØ TØ 90
      IF ( .NØT. (IØPT.EQ.1 .ØR. IØPT.EQ.2)) GØ TØ 90
      IF (IØPT.EQ.2) GØ TØ 10
      IF ( .NØT. (ITNS.GE.1 .AND. ITNS.LE.N)) GØ TØ 90
      GØ TØ 20
   10 IF ( .NØT. (N.GE.1 .AND. DMU.GT.0.D0)) GØ TØ 90
C STAGE 1 - PRELIMINARY CØMPUTATIØNS CØMMØN TØ BØTH ØPTIØNS
   20 IER = 0
      BPD = B + D
      BMD = B - D
      CPA = C + A
      CMA = C - A
      DM = 2.D0*((D-C)*(B-A))/(BPD*CPA)
      DKPR = 1.D0/(1.D0+DM+DSQRT(DM*(DM+2.D0)))
      DEL = 0.D0
      IF (BMD.EQ.0.D0 .AND. CMA.EQ.0.D0) GØ TØ 30
      TEMP = BPD*DKPR
      DEL = 2.D0*(TEMP-CPA)/(CPA*BMD+TEMP*CMA)
   30 ALFA = DKPR*(CMA+2.D0*DEL*A*C)/CPA
      BETA = (2.D0+DEL*BMD)/BPD
      TEMP = DKPR/4.D0
C END ØF STAGE 1 - CØMPUTE ITNS FØR ØPTIØN 2
      IF (IØPT.EQ.1) GØ TØ 40
      ITNS = (DLØG(DMU/4.D0)*DLØG(TEMP))/PISQ + 1.D0
      IF (ITNS.LE.N) GØ TØ 40
      ITNS = N
      IER = 2
C STAGE 2 - CØMPUTATIØN ØF THE ØPTIMAL PARAMETERS
   40 TEMPA = 2*ITNS
      TEMPB = TEMP*TEMP
      DØ 50 J=1,ITNS
        DRJ = 2*J - 1
        DRJ = DRJ/TEMPA
        TEMPC = TEMP**DRJ
        ØJ = 2.D0*(TEMPC+TEMPB/TEMPC)/(1.D0+TEMPC*TEMPC)
        TEMPC = DEL*ØJ
        ØMEV(J) = (ØJ-ALFA)/(BETA-TEMPC)
        ØMEH(J) = (ØJ+ALFA)/(BETA+TEMPC)
   50 CØNTINUE
      IF (IØPT.EQ.2 .AND. IER.EQ.0) GØ TØ 90
C END ØF STAGE 2 - CØMPUTE DMU FØR ØPTIØN 1
      TEMPA = ITNS
      TEMP = PISQ*TEMPA/DLØG(TEMPB*(1.D0+8.D0*TEMPB))
C CHØØSE PRØPER FØRMULA TØ AVØID UNDERFLØW ØR ØVERFLØW
      IF (TEMP.LE.-90.D0 .ØR. TEMP.GE.30.D0) GØ TØ 60
      IF (TEMP.LE.-10.D0) GØ TØ 70
      IF (TEMP.LT.10.D0) GØ TØ 80
      DMU = DEXP(-6.D0*TEMP)
      GØ TØ 90
   60 DMU = 0.D0
      GØ TØ 90
   70 DMU = 4.D0*DEXP(2.D0*TEMP)
      GØ TØ 90
   80 TEMP = DEXP(TEMP)
      DMU = ((2.D0*TEMP)/(1.D0+2.D0*TEMP**4))**2
   90 RETURN
      END
```

Algorithm 461 Cubic Spline Solutions to a Class of Functional Differential Equations [D2]

F.J. Burkowski and W.D. Hoskins [Recd. 3 June 1971 and 27 Apr. 1972]

Department of Computer Science, University of Manitoba, Winnepeg, 19, Manitoba, Canada

Key Words and Phrases: differential equation, spline approximation

CR Categories: 5.17

Language: Fortran

Description

Purpose. The subroutine *SPNBVP* calculates a piecewise continuous approximation to the solution of the boundary value problem

$$X''(t) = P(t)X(t) + Q(t)X(G(t)) + R(t) \qquad (1)$$

on the interval $[A, B]$. The existence of such a solution has been demonstrated by Grimm and Schmitt [5], and it should be noted that the boundary values take the form of two continuous functions $U(t)$ and $V(t)$ specified on the two intervals $[\alpha, A]$ and $[B, \Omega]$ respectively where

$$\alpha = \min_{t \epsilon [A, B]} \{G(t), A\} \quad \text{and} \quad \Omega = \max_{t \epsilon [A, B]} \{G(t), B\}.$$

Boundary value problems of this type can arise in the study of variational problems in control theory where the problem is complicated by the effect of time delays in signal transmission. For example, one may wish to determine extrema of the functional

$$\int_a^b F(t, x(t), x(g(t)))\, dt$$

under the conditions

$$x(t) = \psi(t),\ t \leq a,\ x(b) = B.$$

Under suitable hypotheses on F, this problem leads to a boundary value problem of the above type. Such problems have been treated by El'sgol'ts [3]. Other related works are the survey papers [6, 7, 8, 9].

Method. SPNBVP utilizes an iterative scheme where each iterate is a cubic spline [4, p. 1] serving as an approximation to the true solution. Burkowski and Cowan [2] have demonstrated that such an iterative procedure will converge to an approximation of the solution if the condition

$$\max_{A \leq t \leq B} \{| P(t) | + \bar{g}(t) | Q(t) |\} \leq 8/((B - A)^2 + 6H^2)$$

is satisfied where H is defined below and

$$\bar{g}(t) = 1 \quad \text{if } G(t) \in [A, B],$$
$$= 0 \quad \text{if } G(t) \notin [A, B].$$

The interval $[A, B]$ is partitioned into N equal subintervals of length $H = (B - A)/N$. That is we have a sequence of "knots"

$$A = t_0 < t_1 < \cdots < t_N = B$$

such that $t_j - t_{j-1} = H$ for $j = 1, 2, \ldots, N$. For our purpose the equation of the cubic polynomial in the interval $[t_{j-1}, t_j]$ may be written as

$$\begin{aligned} S(t) = {} & x''_{j-1}((t_j - t)^3/6H) + x_j''((t - t_{j-1})^3/6H) \\ & + (x_{j-1} - (H^2/6)x''_{j-1})((t_j - t)/H) \\ & + (x_j - (H^2/6)x_j'')((t - t_{j-1})/H) \end{aligned} \qquad (2)$$

where $x_i = X(t_i)$ and $x_i'' = X''(t_i)$.

In order to ensure that the spline has the necessary continuity conditions at the knots, the x_i and x_i'' values are subject to the following "continuity equations"

$$x_{j+1} - 2x_j + x_{j-1} = (H^2/6)[x''_{j+1} + 4x_j'' + x''_{j-1}] \qquad (3)$$

valid for $j = 1, 2, 3, \ldots, N - 1$. Using the central difference operator δ this can be rewritten as

$$\delta^2 x_j = H^2((\delta^2/6) + 1)x_j'' \qquad j = 1, 2, 3, \ldots, N - 1 \qquad (4)$$

In [2], it is also demonstrated that the accuracy of the spline approximation is proportional to H^2.

The difficulty in constructing solutions to such equations as (1) arises in having to evaluate terms such as $X(G(t_i))$ in order to calculate the value of $X''(t)$ at a point t_i. By using splines a continuous rather than discrete approximation to the solution is generated and hence a value for $X(G(t_i))$ can be determined even if $G(t_i)$ does not correspond to a value t_j for some j. Since a cubic spline is used, the method is superior to any algorithm which simply evaluates $X(G(t_i))$ by using a linear interpolation.

The basic strategy used in *SPNBVP* is to calculate a sequence of successive splines or essentially a sequence of vectors each containing the values x_i, $i = 1, 2, \ldots, N - 1$. Once a set of x_i values is calculated, we may use the continuity equations and boundary values to evaluate the x_i'' values and hence determine the corresponding spline.

The x_i, $i = 1, 2, \ldots, N - 1$ are treated as unknowns in the system of equations

$$\delta^2 x_j = H^2((\delta^2/6) + 1)\{P(t_j)x_j + Q(t_j)X(G(t_j)) + R(t_j)\} \qquad (5)$$
$$j = 1, 2, \ldots, N - 1$$

derived from (1) and (4). The solution of (5) is obtained by setting up the matrix equation

$$(MAT)(X) = (VM) \qquad (6)$$

where the vector (X) contains the unknowns x_j, $j = 1, 2, \cdots, N - 1$ and the matrix (MAT) contains the coefficients of the x_j unknowns. The vector (VM) contains values arising from the function $R(t)$ and also other quantities discussed below. In the calculation of a spline, the iterative character of the algorithm arises from the fact that the values $X(G(t_j))$ are calculated from the previous spline or from the current spline depending upon the nature of $G(t_j)$. More precisely, if for a certain t_j we have $G(t_j) \notin [A, B]$, then

$$\begin{aligned} X(G(t_j)) &= U(G(t_j)) && \text{if } G(t_j) \leq A \\ &= V(G(t_j)) && \text{if } G(t_j) \geqq B. \end{aligned}$$

Since the value of this term is independent of any x_i, an appropriate entry is made in the vector (VM). If $G(t_j) = t_k$ for some t_k, then $X(G(t_j)) = x_k$, and in this case (MAT) is accordingly modified. If neither of these last two conditions prevails, we set $X(G(t_j)) = S(G(t_j))$ in eq. (2), and hence $X(G(t_j))$ is expressed in terms of two unknowns x_k and x_{k-1} (for some k) and also in terms of x_k'' and $x/''_{k-1}$, two values which are taken from the previous spline. Thus we use only the x_j'' values of any spline when we calculate the next successive spline. To start the iteration we assume an initial spline with $x_j'' = 0$, $j = 0, 1, 2, \ldots, N$.

Program Call. Parameters in the call statement for *SPNBVP* include the following:

A, *B* are the endpoints of the inverval under consideration.

NP is the number of knots in [*A*, *B*], and hence $NP = N + 1$.

NK is the number of interior knots in [*A*, *B*], and so $NK = N - 1$.

X will contain the values of x_j, $j = 1, 2, 3, \ldots, N - 1$ on return to the calling program.

XDP will contain the values $(H^2/6)x_j''$, $j = 0, 1, 2, \ldots N$.

EP *SPNBVP* returns to the calling program when convergence has progressed so far that

$$\sum_{i=1}^{N-1} |x_i - \bar{x}_i| \le EP \sum_{i=1}^{N-1} |x_i|.$$

Thus, if *EP* is set to the value $10^{-(m+1)}$, convergence of the iteration to the approximation has been attained if the x_i have *m* persistent figures in successive iterates. Hence this may be considered as a machine dependent constant. The term $\bar{x}_i$ denotes the value of x_i in the previous spline.

The remaining eight variable names have been included in the parameter list in order to achieve execution-time dimensioning of arrays. The user need only concern himself with the dimension and type of each of these arrays as explained in the comment cards.

SPNBVP requires six function subprograms defining the functions $U(t)$, $V(t)$, $P(t)$, $Q(t)$, $R(t)$, and $G(t)$ as defined above. Four other subroutines are required. *GAGB* is used when x_0 and x_N are calculated. These quantities require rather special treatment since the continuity equations apply only to the internal knots t_j, $j = 1, 2, \ldots, N - 1$. *SOLVE* is simply a special routine which when given x_j values quickly calculates x_j'', $j = 1, 2, \ldots, N - 1$ by using the continuity equations. Finally, the user is responsible for the provision of routines which compute the solution of the matrix system (6). In this case the routine *LUDCMP* replaces (*MAT*) by its *LU* decomposition. The routine *LUSUB* uses this new matrix and the vector (*VM*) to compute the next iterate (*X*). The description and analysis of such routines are given in [1, pp. 93–110].

References

1. Bowdler, H.J., Martin, R.S., Peters, G., and Wilkinson, J.H. Solution of real and complex systems of linear equations. In *Handbook for Automatic Computation*, Vol. II, Springer Verlag,
2. Burkowski, F.J., and Cowan, D.D. The numerical solution of a boundary value problem involving differential-difference equations. *SIAM J. Numer Anal. 10* (1973), 489–495.
3. El'sgol'ts, L.E. *Qualitative Methods in Mathematical Analysis*. Trans. Math. Mono. 12, AMS, Providence, R.I., 1964.
4. Greville, L.J. *Theory and Application of Spline Functions*. Academic Press, New York, 1969.
5. Grimm, L.J., and Schmitt, K. Boundary value problems for differential equations with deviating arguments. *Aequationes Mathematicae 3* (1969), 24–38.
6. Kamenskii, G.A., Norkin, S.B., and El'sgol'ts, L.E. Some directions of investigation in the theory of differential equations with deviating arguments, Trudy Sem. Teor. Differential. Uravnenii s Otklon. *Argumenton Univ. Druzhby Narodov Patrisa Lumumba 6* (1968), 3–36.
7. Kamenskii, G.A. On existence and uniqueness of solutions of differential equations with deviating arguments. *Ibid. 5* (1967), 107–108.
8. Myshkis, A.D., and El'sgol'ts, L.E. Some results and problems in the theory of differential equations. *Uspehi Mat. Nauk, 22* (1967), 21–57.
9. Zverkin, A.M., Kamenskii, G.A., Norkin, S.B., and El'sgol'ts, L.E. Differential equations with a perturbed argument. *Ibid. 17* (1962), 77–164.

Algorithm

```
      SUBRØUTINE SPNBVP(A, B, NP, NK, X, XDP, EP, GT, KG, VP,
     * VQ, VR, VG, MAT, VM)
C THIS ALGØRITHM CØMPUTES ITERATIVELY A CUBIC SPLINE
C APPRØXIMATIØN TØ THE SØLUTIØN ØF THE DIFFERENTIAL EQUATIØN
C X**(T)=P(T)X(T)+Q(T)X(G(T))+R(T) ØN THE INTERVAL (A,B)
C WITH BØUNDARY CØNDITIØNS GIVEN BY U(T) IF T.LE.A AND
C V(T) IF T.GE.B.
C A AND B ARE TWØ REAL VARIABLES DEFINED AS ABØVE.
C NP   AN INTEGER VARIABLE SPECIFYING THE NUMBER ØF KNØTS
C      ØN THE INTERVAL (A,B).
C NK   AN INTEGER VARIABLE SPECIFYING THE NUMBER ØF INTERIØR
C      KNØTS. THUS NK=NP-2. IT IS USED TØ ESTABLISH THE
C      DIMENSIØN ØF CERTAIN ARRAYS MENTIØNED BELØW.
C X    ØN RETURN TØ THE CALLING PRØGRAM X WILL CØNTAIN THE
C      VALUES ØF THE APPRØXIMATIØN TØ THE SØLUTIØN AT THE
C      NK INTERIØR KNØTS. THIS IS AN ARRAY ØF DIMENSIØN
C      NK AND TYPE REAL.
C XDP  ØN RETURN, XDP CØNTAINS THE QUANTITIES H*H/6.0
C      MULTIPLIED BY THE SECØND DERIVATIVE VALUES AT ALL THE
C      KNØTS. XDP IS A REAL ARRAY ØF DIMENSIØN NP.
C EP   THIS REAL VARIABLE IS SET TØ THE VALUE 1.0E-M IF WE
C      REQUIRE M-1 IDENTICAL FIGURES IN SUCCESSIVE ITERATES.
C GT   AN INTEGER ARRAY ØF LENGTH NP WHICH ASSIGNS TØ EACH
C      KNØT T SUB J AN INTEGER VALUE BETWEEN 1 AND 6. THIS
C      VALUE DESIGNATES RESPECTIVELY THE CASES WHEN
C      G(T SUB J) IS  1) .LE. A,  2) .GE.B,  3) WITHIN EP ØF
C      SØME KNØT VALUE,  4) IN THE FIRST SUBINTERVAL,
C      5) IN THE LAST SUBINTERVAL, AND 6) IN ANY ØTHER
C      SUBINTERVAL. GT(I+1) CØRRESPØNDS TØ KNØT T SUB I.
C KG   AN INTEGER ARRAY ØF LENGTH NP WHICH ASSIGNS TØ EACH
C      KNØT AN INTEGER BETWEEN 2 AND NP-1. IF GT(I+1)=3
C      THEN KG(I+1) CØNTAINS THE SUBSCRIPT ØF THE KNØT
C      AT THE PØINT G(T SUB I). IF GT(I+1)=6 THEN KG(I+1)
C      CØNTAINS THE SUBSCRIPT ØF THE KNØT AT THE RIGHT HAND
C      ENDPØINT ØF THE SUBINTERVAL IN WHICH G(T SUB I) LIES.
C VP, VQ, VR, AND VG ARE ALL REAL ARRAYS ØF DIMENSIØN NP AND
C      CØNTAIN THE VALUES ØF THE FUNCTIØNS P, Q, R AND G
C      RESPECTIVELY EACH EVALUATED AT THE NP KNØTS.
C MAT  IS A REAL NK BY NK ARRAY USED IN THE MATRIX EQUATIØN
C      (MAT)(X)=(VM) SET UP TØ SØLVE FØR THE X SUB J VALUES
C      STØRED IN ARRAY X.
C VM   AN ARRAY ØF LENGTH NK AND TYPE REAL USED AS
C      DESCRIBED ABØVE.
C THE USER MUST SUPPLY REAL FUNCTIØN SUBPRØGRAMS TØ CØMPUTE
C THE FUNCTIØNS U(T),V(T),P(T),Q(T),R(T) AND G(T) DEFINED AS
C ABØVE. HE MUST ALSØ SUPPLY SUBPRØGRAMS WHICH SØLVE THE
C SYSTEM (MAT)(X)=(VM). THE RØUTINE LUDCMP(MAT,NK) IS TØ
C REPLACE MAT BY ITS LU DECØMPØSITIØN. THE RØUTINE
C LUSUB(VM,MAT,X,NK) IS TØ CØMPUTE X WHEN VM AND THE LU
C FØRM ØF MAT IS GIVEN.
      INTEGER GTYP, GTE, GT(NP), KG(NP), GT1, GTNP
      REAL XDP(NP), VP(NP), VQ(NP), VR(NP), VG(NP)
      REAL MAT(NK,NK), VM(NK), X(NK)
C KPR IS PRINTER DEVICE NUMBER
      DATA KPR/6/
      C(T) = T*(T*T-1.)
C INITIALIZATIØN
      N = NP - 1
      RN = N
      NK = N - 1
      DØ 20 K=1,NK
        DØ 10 J=1,NK
          MAT(K,J) = 0.0
   10   CØNTINUE
   20 CØNTINUE
      XA = U(A)
      XB = V(B)
C INITIALIZE XDP TØ ZERØ (INITIAL SPLINE)
      DØ 30 K=1,NP
        XDP(K) = 0.0
   30 CØNTINUE
C SET UP P,Q,R,G VECTØRS
      H = (B-A)/RN
      HS = H*H/6.0
      HR = 1./H
      DØ 40 K=1,NP
        RK = K - 1
        TM = A + RK*H
        VP(K) = P(TM)
        VQ(K) = Q(TM)
        VR(K) = R(TM)
        VG(K) = G(TM)
   40 CØNTINUE
C SET UP *TYPE ØF G VALUE* ARRAY AND KG ARRAY
      APLSE = A + EP*ABS(A)
      BMINE = B - EP*ABS(B)
      DØ 70 K=1,NP
        GTE = 6
        VGE = VG(K)
        IF (VGE.LT.A+H) GTE = 4
        IF (VGE.GT.B-H) GTE = 5
        IF (VGE.LE.APLSE) GTE = 1
        IF (VGE.GE.BMINE) GTE = 2
        VDH = (VGE-A)/H
        KNØT = VDH + EP
        RKNØT = KNØT
        IF ((KNØT.LT.1) .ØR. (KNØT.GT.NK)) GØ TØ 50
        IF (ABS(VDH-RKNØT).GT.EP) GØ TØ 50
        GTE = 3
        KG(K) = KNØT
        GØ TØ 60
   50   KG(K) = KNØT + 1
   60   GT(K) = GTE
   70 CØNTINUE
C PUT XSUBJ CØEFFICIENTS INTØ (MAT) AND INITIALIZE X TØ ZERØ
      DØ 90 J=1,NK
        X(J) = 0.0
        IF (J.EQ.1) GØ TØ 80
        MAT(J,J-1) = 1. - HS*VP(J)
   80   MAT(J,J) = -2.*(1.+2.*HS*VP(J+1))
        IF (J.EQ.NK) GØ TØ 90
        MAT(J,J+1) = 1. - HS*VP(J+2)
   90 CØNTINUE
C ADD INTØ (MAT)  X SUB G SUB T CØEFFICIENTS
      DØ 150 J=1,NK
        DØ 140 JJ=1,3
          JZ = JJ - 1
          JJZ = J + JZ
          CØEF = HS*VQ(JJZ)
          IF (JZ.EQ.1) CØEF = CØEF*4.
          GTYP = GT(JJZ)
          GØ TØ (140,140,100,110,120,130), GTYP
  100     KNØT = KG(JJZ)
          MAT(J,KNØT) = MAT(J,KNØT) - CØEF
          GØ TØ 140
  110     MAT(J,1) = MAT(J,1) - CØEF*(VG(JJZ)-A)*HR
          GØ TØ 140
  120     MAT(J,NK) = MAT(J,NK) - CØEF*(B-VG(JJZ))*HR
          GØ TØ 140
  130     KNØT = KG(JJZ)
          RKNØT = KNØT
          CCC = RKNØT + (A-VG(JJZ))*HR
          MAT(J,KNØT-1) = MAT(J,KNØT-1) - CØEF*CCC
          CCC = (VG(JJZ)-A)*HR - RKNØT + 1.
          MAT(J,KNØT) = MAT(J,KNØT) - CØEF*CCC
  140   CØNTINUE
  150 CØNTINUE
C REPLACE (MAT) BY ITS LU DECØMPØSITIØN.
      CALL LUDCMP(MAT, NK)
C A SEQUENCE ØF SPLINES (UP TØ 20) IS NØW GENERATED
      VPA = VP(1)
      VPB = VP(NP)
      DØ 250 NNN=1,20
C VECTØR VM IS NØW SET UP
        DØ 200 J=1,NK
          VM(J) = (VR(J)+4.*VR(J+1)+VR(J+2))*HS
          DØ 190 JJ=1,3
            JZ = JJ - 1
            JJZ = J + JZ
            GTYP = GT(JJZ)
            CØEF = HS*VQ(JJZ)
            IF (JZ.EQ.1) CØEF = CØEF*4.
            IF (GTYP.EQ.1) VM(J) = VM(J) + CØEF*U(VG(JJZ))
            IF (GTYP.EQ.2) VM(J) = VM(J) + CØEF*V(VG(JJZ))
            GØ TØ (190,190,190,160,170,180), GTYP
  160       TM = (VG(JJZ)-A)*HR
            TJ = 1. - TM
            CCC = TJ*XA + C(TM)*XDP(2) + C(TJ)*XDP(1)
            VM(J) = VM(J) + CØEF*CCC
            GØ TØ 190
  170       TJ = (B-VG(JJZ))*HR
            TM = 1. - TJ
            CCC = TM*XB + C(TM)*XDP(NK+2) + C(TJ)*XDP(NK+1)
            VM(J) = VM(J) + CØEF*CCC
            GØ TØ 190
  180       KNØT = KG(JJZ)
            RKNØT = KNØT
            TJ = (A-VG(JJZ))*HR + RKNØT
            TM = 1. - TJ
            CCC = C(TM)*XDP(KNØT+1) + C(TJ)*XDP(KNØT)
            VM(J) = VM(J) + CØEF*CCC
  190     CØNTINUE
  200   CØNTINUE
        VM(1) = VM(1) - (1.-HS*VPA)*U(A)
        VM(NK) = VM(NK) - (1.-HS*VPB)*V(B)
C THE NEW X ARRAY IS NØW CØMPUTED.
C THE ARRAY VP SERVES AS A WØRK AREA.
        DØ 210 JF=1,NK
          VP(JF) = X(JF)
  210   CØNTINUE
        CALL LUSUB(VM, MAT, X, NK)
        TSTVL1 = 0.0
        TSTVL2 = 0.0
        DØ 220 JF=1,NK
          TSTVL1 = TSTVL1 + ABS(VP(JF)-X(JF))
          TSTVL2 = TSTVL2 + ABS(X(JF))
  220   CØNTINUE
C CALCULATIØN ØF XDP AT A AND B
        GT1 = GT(1)
        GTNP = GT(NP)
        IF (GT1.EQ.1) XGAA = U(VG(1))
        IF (GT1.EQ.2) XGAA = V(VG(1))
        IF (GTNP.EQ.1) XGAB = U(VG(NP))
        IF (GTNP.EQ.2) XGAB = V(VG(NP))
        CALL GAGB(GT1, XGAA, KG(1), VG(1), X, XDP, A, B, NP, NK)
        CALL GAGB(GTNP, XGAB, KG(NP), VG(NP), X, XDP, A, B, NP,
     *  NK)
        XDPA = (VPA*XA+VQ(1)*XGAA+VR(1))*HS
        XDPB = (VPB*XB+VQ(NP)*XGAB+VR(NP))*HS
C SØLVE FØR XDP VALUES ØF CURRENT SPLINE USING CØNTINUITY
C EQUATIØNS. VM AND VP ARE USED AS WØRKING AREAS.
        VM(1) = XA + X(2) - 2.*X(1) - XDPA
        NK1 = NK - 1
        VM(NK) = XB + X(NK1) - 2.*X(NK) - XDPB
        DØ 230 J=2,NK1
          VM(J) = X(J-1) + X(J+1) - 2.*X(J)
  230   CØNTINUE
        CALL SØLVE(VM, NK, VP, NP)
        XDP(1) = XDPA
        XDP(NP) = XDPB
        DØ 240 J=1,NK
          XDP(J+1) = VM(J)
  240   CØNTINUE
        IF (TSTVL1.LE.TSTVL2*EP) RETURN
        IF(NNN.EQ.20) WRITE(KPR,1000)
 1000   FØRMAT(32H NØ CØNVERGENCE IN 20 ITERATIØNS   )
  250 CØNTINUE
      RETURN
      END

      SUBRØUTINE GAGB(GTYP, ANS, K, GV, X, XDP, A, B, NP, NK)
      REAL X(NK), XDP(NP)
      INTEGER GTYP
      C(T) = T*(T*T-1.)
      RNKD = NK + 1
      RK = K
      XA = U(A)
      XB = V(B)
      H = (B-A)/RNKD
      GØ TØ (10,20,30,40,50,60), GTYP
   10 RETURN
   20 RETURN
   30 ANS = X(K)
      RETURN
```

```
   40 TM = (GV-A)/H
      TJ = 1. - TM
      ANS = TM*X(1) + TJ*XA + C(TM)*XDP(2) + C(TJ)*XDP(1)
      RETURN
   50 TJ = (B-GV)/H
      TM = 1. - TJ
      ANS = TM*XB + TJ*X(NK) + C(TM)*XDP(NK+2) + C(TJ)*XDP(NK+1)
      RETURN
   60 TJ = (A-GV)/H + RK
      TM = 1. - TJ
      ANS = TM*X(K) + TJ*X(K-1) + C(TM)*XDP(K+1) + C(TJ)*XDP(K)
      RETURN
      END

      SUBRØUTINE SØLVE(D, NK, M, NP)
      REAL D(NK), M(NP)
      NK1 = NK - 1
      M(NK) = .25
      DØ 10 I=1,NK1
        J = NK - I
        M(J) = 1./(4.-M(J+1))
        D(J) = D(J) - D(J+1)*M(J+1)
   10 CØNTINUE
      D(1)=D(1)*M(1)
      DØ 20 I=2,NK
        D(I) = (D(I)-D(I-1))*M(I)
   20 CØNTINUE
      RETURN
      END
```

Algorithm 462
Bivariate Normal Distribution [S15]

Thomas G. Donnelly [Recd. 9 July 1971]
Department of Biostatistics and Center for Urban and Regional Studies, University of North Carolina at Chapel Hill, Chapel Hill, NC 27514

Key Words and Phrases: bivariate, normal Gaussian, frequency distribution
CR Categories: 5.5
Language: Fortran

Description

Purpose. Tables of the bivariate normal distribution are available [1] for $H, K = 0(.1)4$ and $R = \pm.0(.05)0.95(.01)1$ to six decimal places for positive R and to seven decimal places for negative R. A valuable section in the preface to [1] by D. B. Owen describes a wide variety of problem areas in which the tables can be applied.

The advantages of being able to access these data in a computer are many. Frequently the values of (H, K, R) in which one is interested will have been produced through computer calculations, and it is much more convenient if the user can produce the corresponding probability immediately and continue his calculations. Secondly, use of tables ordinarily involves the user in three-dimensional hand calculated interpolation, and the risk of errors here can be eliminated by use of a functional subprogram. Finally, a functional subprogram is a starting point for additional refinements, such as confidence regions and tetrachoric correlations.

Method. The methods employed in the program were basically those described in [2, eqs. 3.5, 3.8, 3.9], and comments in the program have been reduced to a minimum because the relations between the program and the equations should be self-evident. Because the expression used [2, eq. 3.9] in evaluating $T(h, a)$ is an alternating convergent series, it was possible to provide controlled precision in the algorithm. As written, it provides accuracy to 15 decimal places, but the parameter controlling this, *IDIG*, may be adjusted to suit the computer environment in which the algorithm is to be used. Of course, the value selected must conform to the precision obtainable from the univariate error function used, such as Algorithm 304, [3] and the other standard subroutines used, as well as to the computer characteristics.

The lower-left tail values of the distribution, if desired, are obtained by reversing the signs of H and K.

References

1. National Bureau of Standards, *Tables of the Bivariate Normal Distribution and Related Functions*, N.B.S., Applied Math. Series, No. 50, 1959.
2. Owen, D.B. Tables for computing bivariate normal probabilities. *Ann. Math. Stat. 27* (1956), 1075–1090.
3. Hill, I.D., and Joyce, S.A. Algorithm 304. Normal curve Integral. *Comm. ACM 10* (June 1967), 374.

This work was supported in part by NASA Grant No. NGR 34-003-040 and by OWRR Project No. B-012-NC, Matching Grant Agreement No. 14-01-0001-1935.

Algorithm

```
      DØUBLE PRECISIØN FUNCTIØN BIVNØR(AH, AK, R)
C  BIVNØR IS A CØNTRØLLED PRECISIØN
C FØRTRAN FUNCTIØN TØ CALCULATE THE
C BIVARIATE NØRMAL UPPER RIGHT AREA, VIZ.
C THE PRØBABILITY FØR TWØ NØRMAL
C VARIATES X AND Y WHØSE CØRRELATIØN
C IS R,  THAT X .GT. AH AND Y .GT. AK.
      DØUBLE PRECISIØN TWØPI, B, AH, AK, R, GH, GK, RR, GAUSS,
     * DERF, H2, A2, H4, DEXP, EX, W2, AP, S2, SP, S1, SN, SQR,
     * DSQRT, CØN, DATAN, WH, WK, GW, SGN, T, DABS, G2, CØNEX,
     * CN
      GAUSS(T) = (1.0D0+DERF(T/DSQRT(2.0D0)))/2.0D0
C GAUSS IS A UNIVARIATE LØWER NØRMAL
C TAIL AREA CALCULATED HERE FRØM THE
C CENTRAL ERRØR FUNCTIØN DERF.
C IT MAY BE REPLACED BY THE ALGØRITHM IN
C HILL,I.D. AND JØYCE,S.A. ALGØRITHM 304,
C NØRMAL CURVE INTEGRAL(S15), CØMM.A.C.M.(10)
C (JUNE,1967),P.374.
C SØURCE: ØWEN, D.B. ANN.MATH.STAT.
C VØL. 27(1956), P.1075.
C TWØPI = 2. * PI
      TWØPI = 6.283185307179587D0
      B = 0.0D0
      IDIG = 15
C THE PARAMETER 'IDIG' GIVES THE
C NUMBER ØF SIGNIFICANT DIGITS
C TØ THE RIGHT ØF THE DECIMAL PØINT
C DESIRED IN THE ANSWER, IF
C IT IS WITHIN THE CØMPUTER'S
C CAPACITY ØF CØURSE.
      GH = GAUSS(-AH)/2.0D0
      GK = GAUSS(-AK)/2.0D0
      IF (R) 10, 30, 10
   10 RR = 1.0D0 - R*R
      IF (RR) 20, 40, 100
   20 WRITE (3,99999) R
C ERRØR EXIT FØR ABS(R) .GT. 1.0D0
99999 FØRMAT(12H BIVNØR R IS, D26.16)
      STØP
   30 B = 4.0D0*GH*GK
      GØ TØ 350
   40 IF (R) 50, 70, 70
   50 IF (AH+AK) 60, 350, 350
   60 B = 2.0D0*(GH+GK) - 1.0D0
      GØ TØ 350
   70 IF (AH-AK) 80, 90, 90
   80 B = 2.0D0*GK
      GØ TØ 350
   90 B = 2.0D0*GH
      GØ TØ 350
  100 SQR = DSQRT(RR)
      IF (IDIG-15) 120, 110, 120
  110 CØN = TWØPI*1.D-15/2.0D0
      GØ TØ 140
  120 CØN = TWØPI/2.0D0
      DØ 130 I=1,IDIG
        CØN = CØN/10.0D0
  130 CØNTINUE
  140 IF (AH) 170, 150, 170
  150 IF (AK) 190, 160, 190
  160 B = DATAN(R/SQR)/TWØPI + 0.25D0
      GØ TØ 350
  170 B = GH

      IF (AH*AK) 180, 200, 190
  180 B = B - 0.5D0
  190 B = B + GK
      IF (AH) 200, 340, 200
  200 WH = -AH
      WK = (AK/AH-R)/SQR
      GW = 2.0D0*GH
      IS = -1
  210 SGN = -1.0D0
      T = 0.0D0
      IF (WK) 220, 320, 220
  220 IF (DABS(WK)-1.0D0) 270, 230, 240
  230 T = WK*GW*(1.0D0-GW)/2.0D0
      GØ TØ 310
  240 SGN = -SGN
      WH = WH*WK
      G2 = GAUSS(WH)
      WK = 1.0D0/WK
      IF (WK) 250, 260, 260
  250 B = B + 0.5D0
  260 B = B - (GW+G2)/2.0D0 + GW*G2
  270 H2 = WH*WH
      A2 = WK*WK
      H4 = H2/2.0D0
      EX = DEXP(-H4)
      W2 = H4*EX
      AP = 1.0D0
      S2 = AP - EX
      SP = AP
      S1 = 0.0D0
      SN = S1
      CØNEX = DABS(CØN/WK)
      GØ TØ 290
  280 SN = SP
      SP = SP + 1.0D0
      S2 = S2 - W2
      W2 = W2*H4/SP
      AP = -AP*A2
  290 CN = AP*S2/(SN+SP)
      S1 = S1 + CN
      IF (DABS(CN)-CØNEX) 300, 300, 280
  300 T = (DATAN(WK)-WK*S1)/TWØPI
  310 B = B + SGN*T
  320 IF (IS) 330, 350, 350
  330 IF (AK) 340, 350, 340
  340 WH = -AK
      WK = (AH/AK-R)/SQR
      GW = 2.0D0*GK
      IS = 1
      GØ TØ 210
  350 IF (B) 360, 370, 370
  360 B = 0.0D0
  370 IF (B-1.0D0) 390, 390, 380
  380 B = 1.0D0
  390 BIVNØR = B
      RETURN
      END
```

Algorithm 463

Algorithms SCALE1, SCALE2, and SCALE3 for Determination of Scales on Computer Generated Plots [J6]

C.R. Lewart (Recd. 6 Aug. 1971 and 28 Jan. 1972)
Bell Telephone Laboratories, Incorporated, Holmdel, NJ 07733

Key Words and Phrases: plotting, scaling for plotting
CR Categories: 4.41
Language: Fortran

Description

Introduction. It is often desirable to plot computer generated output or obtain discrete distribution functions such as histograms automatically. In general the raw data does not lend itself directly to an easily readable presentation. The three related algorithms as presented here obtain readable linear or logarithmic scales with uniform interval sizes for users of various plot routines.

Readability. A readable linear scale is defined here as a scale with interval size a product of an integer power of 10 and 1, 2 or 5, and scale values integer multiples of the interval size.

A readable logarithmic scale on a display with uniform plotting intervals is defined here such that the ratio of adjoining scale values $DIST = 10^{(1/L+K)}$, where K and L are integers, with $1 \leq L \leq 10$; scale values are equal to $DIST^M$, where M is a set of successive integers.

The definition of readability used for *SCALE* 1 and *SCALE* 2 permits scale values such as:

−0.5, 0.0, 0.5, 1.0, . . .
1.24, 1.26, 1.28, . . .
100.0, 200.0, 300.0, . . . , etc.

It prohibits the following examples:

−1.0, 4.0, 9.0, . . .
1.2, 1.31, 1.42, . . .
0.0, 4.0, 8.0, 12.0, . . . , etc.

The definition of readability for logarithmic plots would permit scale values of 1, $\sqrt[3]{10}$, $(\sqrt[3]{10})^2$, 10, . . . , but disallow 1, $\sqrt{5}$, 5, $5\sqrt{5}$, 25, ...

Usage. A call of the form

*CALL SCALE*1 (*XMIN*, *XMAX*, *N*, *XMINP*, *XMAXP*, *DIST*)

where *XMIN* and *XMAX* are the minimum and maximum, respectively, of a given array and *N* a requested number of grid intervals will return a new minimum and maximum *XMINP* and *XMAXP* such that the range [*XMINP*, *XMAXP*] is the smallest range which will embrace the range [*XMIN*, *XMAX*] and simultaneously result in approximately *N* grid intervals, each of the length *DIST*. Interval *DIST* is selected by *SCALE*1 as the product of an integer power of 10 and 1, 2, or 5. *XMINP* and *XMAXP* are integer multiples of *DIST*.

In certain cases the number of plot intervals *N* has to be fixed. In particular, for plots generated by devices with relatively large pen increments, e.g. line printers or teletypewriters, *N* is restricted. For such cases *SCALE*2 for linear plots and *SCALE*3 for logarithmic plots have to be used.

*SCALE*2 with the same arguments as *SCALE*1 differs from *SCALE*1 in that *XMINP* and *XMAXP* are determined such that exactly *N* grid intervals will result; as a consequence the range [*XMINP*, *XMAXP*] will in general be less economical than that obtained by *SCALE*1. Parameters *DIST*, *XMINP*, and *XMAXP* will still satisfy requirements specified for *SCALE*1, namely *DIST* will be an integer power of 10 times 1, 2, or 5; and *XMINP* and *XMAXP* will be integer multiples of *DIST*.

*SCALE*3 with the same arguments as *SCALE*1 will set *XMINP* and *XMAXP* such that *N* logarithmic uniformly spaced grid intervals will cover the range [*XMIN*, *XMAX*]. *DIST* will be the ratio of adjacent grid line values.

*SCALE*3 selects *DIST* as $10^{(1/L+K)}$, where K and L are integers and $1 \leq L \leq 10$. *XMINP* and *XMAXP* are selected so that $XMINP = DIST^j$, and $XMAXP = DIST^l$ where j and l are integers.

Calling *SCALE*1, *SCALE*2, or *SCALE*3 will approximately center the range [*XMIN*, *XMAX*] between *XMINP* and *XMAXP*. *SCALE*1, having determined *DIST*, selects the most economical limits, i.e. $(XMIN - DIST) < XMINP \leq XMIN$ and $XMAX \leq XMAXP < (XMAX + DIST)$. *SCALE*2 and *SCALE*3 select limits to minimize $(XMAXP - XMAX)$ and $(XMIN - XMINP)$ without necessarily satisfying the previous inequalities, but subject to the constraints of a fixed number of intervals.

The actual number of intervals N_a, determined from the outputs returned by *SCALE*1 is as follows:

$$N_a = (XMAXP - XMINP)/DIST.$$

N_a may be slightly larger or smaller than N as shown by the following inequality:

$$(N/\sqrt{2.5}) < N_a < (N \times \sqrt{2.5} + 2).$$

N_a will always equal N if *SCALE*2 or *SCALE*3 is called.

Round-off considerations. The three algorithms compensate for the computer round-off to assure that *XMIN* and *XMAX* are within the range [*XMINP*, *XMAXP*]. A normalized parameter *DEL* is introduced to serve as a narrow gate around the minimum *XMIN* and the maximum *XMAX* to avoid an unnecessarily large range [*XMINP*, *XMAXP*] caused by computer round-off. For example, if $DEL = 0.0001$, $N = 3$ and *SCALE*1 or *SCALE*2 is called, *XMINP* of 1.0 and *XMAXP* of 4.0 will result for $0.9999 < XMIN \leq 1.0001$ and $3.9999 \leq XMAX < 4.0001$. *DEL* is normalized to the interval size and should satisfy the following inequality:

$$A < DEL < (B \times N)/C,$$

where A is the round-off expected from a division and float operation, B is the minimum increment of the plotting device in inches, N is the number of intervals on the plot, and C is the plot size in inches. For example, using single precision *REAL**4 variables (IBM 360): $A \sim 0.0000002$; for a precision flat bed plotter: $B = 0.002$, $C = 50.0$. Assuming $N = 10$ the following inequality is obtained:

$$0.0000002 < DEL < 0.0004.$$

It is obvious from this inequality that in practical cases the range of permissible values of *DEL* is so large that *DEL* is quite insensitive to the type of plotter and the type of computer used.

Examples

*SCALE*1

XMIN	*XMAX*	*N*	*XMINP*	*XMAXP*	*DIST*	*Actual No. of Intervals*
−3.1	11.1	5	−4.0	12.0	2.0	8
5.2	10.1	5	5.0	11.0	1.0	6
−12000	−100	9	−12000	0	1000	12

*SCALE*2

XMIN	*XMAX*	*N*	*XMINP*	*XMAXP*	*DIST*	*Actual No. of Intervals*
−3.1	11.1	5	−5.0	20.0	5.0	5
5.2	10.1	5	4.0	14.0	2.0	5
−12000	−100	9	−14000	4000	2000	9

SCALE 3

XMIN	*XMAX*	*N*	*XMINP*	*XMAXP*	*DIST*	*Actual No. of Intervals*
1.8	125.0	10	1.58	158.49	1.58 $(=\sqrt[5]{10})$	10
0.1	10.0	2	0.1	10.0	10.0	2
0.1	1500.0	4	0.077	2154.4	12.92 $(=10^{(1+1/9)})$	4

Algorithm

```
      SUBRØUTINE SCALE1(XMIN, XMAX, N, XMINP, XMAXP, DIST)
C ANSI FØRTRAN
C GIVEN XMIN,XMAX AND N SCALE1 FINDS A NEW RANGE XMINP AND
C XMAXP DIVISIBLE INTØ APPROXIMATELY N LINEAR INTERVALS
C ØF SIZE DIST
C VINT IS AN ARRAY ØF ACCEPTABLE VALUES FØR DIST (TIMES
C AN INTEGER PØWER ØF 10)
C SQR IS AN ARRAY ØF GEØMETRIC MEANS ØF ADJACENT VALUES
C ØF VINT, IT IS USED AS BREAK PØINTS TØ DETERMINE
C WHICH VINT VALUE TØ ASSIGN TØ DIST
      DIMENSION VINT(4), SQR(3)
      DATA VINT(1), VINT(2), VINT(3), VINT(4)/1., 2., 5., 10./
      DATA SQR(1), SQR(2), SQR(3)/1.414214, 3.162278, 7.071068/
C CHECK WHETHER PRØPER INPUT VALUES WERE SUPPLIED
      IF (XMIN.LT.XMAX .AND. N.GT.0) GØ TØ 10
      WRITE (6,99999)
99999 FØRMAT(34H IMPRØPER INPUT SUPPLIED TØ SCALE1)
      RETURN
C DEL ACCØUNTS FØR CØMPUTER RØUND-ØFF
C DEL SHØULD BE GREATER THAN THE RØUND-ØFF EXPECTED FRØM
C A DIVISIØN AND FLØAT ØPERATIØN, IT SHØULD BE LESS THAN
C THE MINIMUM INCREMENT ØF THE PLØTTING DEVICE USED BY
C THE MAIN PRØGRAM (IN.) DIVIDED BY THE PLØT SIZE (IN.)
C TIMES NUMBER ØF INTERVALS N
   10 DEL = .00002
      FN = N
C FIND APPRØXIMATE INTERVAL SIZE A
      A = (XMAX-XMIN)/FN
      AL = ALØG10(A)
      NAL = AL
      IF (A.LT.1.) NAL = NAL - 1
C A IS SCALED INTØ VARIABLE NAMED B BETWEEN 1 AND 10
      B = A/10.**NAL
C THE CLØSEST PERMISSIBLE VALUE FØR B IS FØUND
      DO 20 I=1,3
        IF (B.LT.SQR(I)) GØ TØ 30
   20 CØNTINUE
      I = 4
C THE INTERVAL SIZE IS CØMPUTED
   30 DIST = VINT(I)*10.**NAL
      FM1 = XMIN/DIST
      M1 = FM1
      IF (FM1.LT.0.) M1 = M1 - 1
      IF (ABS(FLØAT(M1)+1.-FM1).LT.DEL) M1 = M1 + 1
C THE NEW MINIMUM AND MAXIMUM LIMITS ARE FØUND
      XMINP = DIST*FLØAT(M1)
      FM2 = XMAX/DIST
      M2 = FM2 + 1.
      IF (FM2.LT.(-1.)) M2 = M2 - 1
      IF (ABS(FM2+1.-FLØAT(M2)).LT.DEL) M2 = M2 - 1
      XMAXP = DIST*FLØAT(M2)
C ADJUST LIMITS TO ACCOUNT FØR RØUND-ØFF IF NECESSARY
      IF (XMINP.GT.XMIN) XMINP = XMIN
      IF (XMAXP.LT.XMAX) XMAXP = XMAX
      RETURN
      END

      SUBRØUTINE SCALE2(XMIN, XMAX, N, XMINP, XMAXP, DIST)
C ANSI FØRTRAN
C GIVEN XMIN,XMAX AND N SCALE2 FINDS A NEW RANGE XMINP AND
C XMAXP DIVISIBLE INTO EXACTLY N LINEAR INTERVALS ØF SIZE
C DIST, WHERE N IS GREATER THAN 1
      DIMENSIØN VINT(5)
      DATA VINT(1), VINT(2), VINT(3), VINT(4), VINT(5)/1., 2.,
     * 5., 10., 20./
C CHECK WHETHER PRØPER INPUT VALUES WERE SUPPLIED
      IF (XMIN.LT.XMAX .AND. N.GT.1).GØ TO 10
      WRITE (6,99999)
99999 FØRMAT(34H IMPRØPER INPUT SUPPLIED TO SCALE2)
      RETURN
   10 DEL = .00002
      FN = N
C FIND APPRØXIMATE INTERVAL SIZE A
      A = (XMAX-XMIN)/FN
      AL = ALØG10(A)
      NAL = AL
      IF (A.LT.1.) NAL = NAL - 1
C A IS SCALED INTØ VARIABLE NAMED B BETWEEN 1 AND 10
      B = A/10.**NAL
C THE CLØSEST PERMISSIBLE VALUE FØR B IS FØUND
      DØ 20 I=1,3
        IF (B.LT.(VINT(I)+DEL)) GØ TØ 30
   20 CØNTINUE
      I = 4
C THE INTERVAL SIZE IS CØMPUTED
   30 DIST = VINT(I)*10.**NAL
      FM1 = XMIN/DIST
      M1 = FM1
      IF (FM1.LT.0.) M1 = M1 - 1
      IF (ABS(FLØAT(M1)+1.-FM1).LT.DEL) M1 = M1 + 1
C THE NEW MINIMUM AND MAXIMUM LIMITS ARE FØUND
      XMINP = DIST*FLØAT(M1)
      FM2 = XMAX/DIST
      M2 = FM2 + 1.
      IF (FM2.LT.(-1.)) M2 = M2 - 1
      IF (ABS(FM2+1.-FLØAT(M2)).LT.DEL) M2 = M2 - 1
      XMAXP = DIST*FLØAT(M2)
C CHECK WHETHER A SECØND PASS IS REQUIRED
      NP = M2 - M1
      IF (NP.LE.N) GØ TØ 40
      I = I + 1
      GØ TØ 30
   40 NX = (N-NP)/2
      XMINP = XMINP - FLØAT(NX)*DIST
      XMAXP = XMINP + FLØAT(N)*DIST
C ADJUST LIMITS TØ ACCØUNT FØR RØUND-ØFF IF NECESSARY
      IF (XMINP.GT.XMIN) XMINP = XMIN
      IF (XMAXP.LT.XMAX) XMAXP = XMAX
      RETURN
      END

      SUBRØUTINE SCALE3(XMIN, XMAX, N, XMINP, XMAXP, DIST)
C ANSI FØRTRAN
C GIVEN XMIN,XMAX AND N, WHERE N IS GREATER THAN 1, SCALE3
C FINDS A NEW RANGE XMINP AND XMAXP DIVISIBLE INTØ EXACTLY
C N LØGARITHMIC INTERVALS, WHERE THE RATIØ ØF ADJACENT
C UNIFØRMLY SPACED SCALE VALUES IS DIST
      DIMENSIØN VINT(11)
      DATA VINT(1), VINT(2), VINT(3), VINT(4), VINT(5), VINT(6),
     * VINT(7), VINT(8), VINT(9), VINT(10), VINT(11)/10., 9.,
     * 8., 7., 6., 5., 4., 3., 2., 1., .5/
C CHECK WHETHER PRØPER INPUT VALUES WERE SUPPLIED
      IF (XMIN.LT.XMAX .AND. N.GT.1 .AND. XMIN.GT.0.) GØ TØ 10
      WRITE (6,99999)
99999 FØRMAT(34H IMPRØPER INPUT SUPPLIED TØ SCALE3)
      RETURN
   10 DEL = .00002
C VALUES ARE TRANSLATED FRØM THE LINEAR INTØ LØGARITHMIC
C REGIØN
      XMINL = ALØG10(XMIN)
      XMAXL = ALØG10(XMAX)
      FN = N
C FIND APPRØXIMATE INTERVAL SIZE A
      A = (XMAXL-XMINL)/FN
      AL = ALØG10(A)
      NAL = AL
      IF (A.LT.1.) NAL = NAL - 1
C A IS SCALED INTØ VARIABLE NAMED B BETWEEN 1 AND 10
      B = A/10.**NAL
C THE CLØSEST PERMISSIBLE VALUE FØR B IS FØUND
      DØ 20 I=1,9
        IF (B.LT.(10./VINT(I)+DEL)) GØ TØ 30
   20 CØNTINUE
      I = 10
C THE INTERVAL SIZE IS COMPUTED
   30 DISTL = 10.**(NAL+1)/VINT(I)
      FM1 = XMINL/DISTL
      M1 = FM1
      IF (FM1.LT.0.) M1 = M1 - 1
      IF (ABS(FLØAT(M1)+1.-FM1).LT.DEL) M1 = M1 + 1
C THE NEW MINIMUM AND MAXIMUM LIMITS ARE FØUND
      XMINP = DISTL*FLØAT(M1)
      FM2 = XMAXL/DISTL
      M2 = FM2 + 1.
      IF (FM2.LT.(-1.)) M2 = M2 - 1
      IF (ABS(FM2+1.-FLØAT(M2)).LT.DEL) M2 = M2 - 1
      XMAXP = DISTL*FLØAT(M2)
      NP = M2 - M1
C CHECK WHETHER ANØTHER PASS IS NECESSARY
      IF (NP.LE.N) GØ TØ 40
      I = I + 1
      GØ TØ 30
   40 NX = (N-NP)/2
      XMINP = XMINP - FLØAT(NX)*DISTL
      XMAXP = XMINP + FLØAT(N)*DISTL
C VALUES ARE TRANSLATED FRØM THE LØGARITHMIC INTØ THE LINEAR
C REGIØN
      DIST = 10.**DISTL
      XMINP = 10.**XMINP
      XMAXP = 10.**XMAXP
C ADJUST LIMITS TØ ACCØUNT FØR RØUND-ØFF IF NECESSARY
      IF (XMINP.GT.XMIN) XMINP = XMIN
      IF (XMAXP.LT.XMAX) XMAXP = XMAX
      RETURN
      END
```

Algorithm 464

Eigenvalues of a Real, Symmetric, Tridiagonal Matrix [F2]

Christian H. Reinsch [Recd. 11 Mar. 1971]
Mathematisches Institut der Technischen Universität,
8000 München 2, Arcisstra 21, Germany

Key Words and Phrases: eigenvalues, QR Algorithm
CR Categories: 5.14
Language: Algol

Description

This algorithm uses a rational variant of the QR transformation with explicit shift for the computation of all of the eigenvalues of a real, symmetric, and tridiagonal matrix. Details are described in [1]. Procedures *tred1* or *tred3* published in [2] may be used to reduce any real, symmetric matrix to tridiagonal form. Turn the matrix end-for-end if necessary to bring very large entries to the bottom right-hand corner.

References

1. Reinsch, C.H. A stable, rational QR algorithm for the computation of the eigenvalues of an Hermitian, tridiagonal matrix. *Math. Comp. 25* (1971), 591-597.
2. Martin, R.S., Reinsch, C.H., Wilkinson, J. H. Householder's tridiagonalization of a symmetric matrix. *Numer. Math. 11* (1968), 181-195.

Algorithm

```
procedure tqlrat (n,macheps) trans: (d,e2);
  value n, macheps;
  integer n; real macheps; array d, e2;
comment
             Input:
  n          order of the matrix,
  macheps    the machine precision, i.e. minimum of all x such that
             1 + x > 1 on the computer,
  d[1:n]     represents the diagonal of the matrix,
  e2[1:n]    represents the squares of the sub-diagonal entries,
             (e2[1] is arbitrary).
             Output:
  d[1:n]     the computed eigenvalues are stored in this array in
             ascending sequence,
  e2[1:n]    is used as working storage and the original informa-
             tion stored in this array is lost;
begin
  integer i, k, m; real b, b2, f, g, h, p2, r2, s2;
  for i := 2 step 1 until n do e2[i-1] := e2[i];
  e2[n] := b := b2 := f := 0.0;
  for k := 1 step 1 until n do
  begin
    h := macheps × macheps × (d[k]↑2 + e2[k]);
    if b2 < h then
    begin b := sqrt(h); b2 := h end;
    comment Test for splitting;
    for m := k step 1 until n do
      if e2[m] ≤ b2 then go to cont1;
cont1:
    if m = k then go to root;
    comment Form the shift from leading 2 × 2 block;
nextit:
    g := d[k]; p2 := sqrt(e2[k]);
    h := (d[k+1]-g)/(2.0×p2); r2 := sqrt(h×h+1.0);
    d[k] := h := p2/(if h<0.0 then h-r2 else h+r2);
    h := g - h; f := f + h;
    for i := k + 1 step 1 until n do d[i] := d[i] - h;
    comment Rational QL transformation, rows k through m;
    g := d[m]; if g = 0.0 then g := b;
    h := g; s2 := 0.0;
    for i := m - 1 step -1 until k do
    begin
      p2 := g × h; r2 := p2 + e2[i];
      e2[i+1] := s2 × r2; s2 := e2[i]/r2;
      d[i+1] := h + s2 × (h+d[i]);
      g := d[i] - e2[i]/g; if g = 0.0 then g := b;
      h := g × p2/r2
    end i;
    e2[k] := s2 × g × h; d[k] := h;
    if e2[k] > b2 then go to nextit;
root:
    h := d[k] + f;
    comment One eigenvalue found, sort eigenvalues;
    for i := k step -1 until 2 do
      if h < d[i-1] then d[i] := d[i-1] else go to cont2;
    i := 1;
cont2:
    d[i] := h
  end k
end tqlrat;
```

Algorithm 465
Student's *t* Frequency [S14]

G.W. Hill [Recd. 24 Aug. 1971, 23 Feb. 1972, 10 July 1972]
C.S.I.R.O., Division of Mathematical Statistics, Glen Osmond, South Australia

Key Words and Phrases: Student's t statistic, density function, series approximation
CR Categories: 5.12, 5.5
Language: Algol

Description

The frequency function for Student's t distribution,

$$f(t \mid n) = \frac{\Gamma(\frac{1}{2}n + \frac{1}{2})}{(\pi n)^{\frac{1}{2}}\Gamma(\frac{1}{2}n)} (1 + t^2/n)^{-(\frac{1}{2}n+\frac{1}{2})},$$

is evaluated for real t and real $n > 0$ to a precision near that of the processor, even for large values of n.

The factor involving t is evaluated as $exp(-\frac{1}{2}b)$ where b is computed as $(n + 1)ln(1 + t^2/n)$ if $t^2/n = c$ is large ($>cmax$, say) or, to avoid loss of precision for smaller c, by summing the series for $b = (t^2 + c)(1 - c/2 + c^2/3 - c^3/4 + \cdots)$ until negligible terms occur, i.e. $c^r/(r + 1) < \epsilon$, where ϵ is the relative magnitude of processor round-off. The relative error up to $\epsilon/cmax$ in evaluating $ln(1 + c)$ and the accumulated round-off error of order $\epsilon\sqrt{R}$ in summing a maximum of R terms of the series can be limited to about the same low level by choosing $cmax = R^{-\frac{1}{2}}$ where $R^{-\frac{1}{2}R}/R \approx \epsilon$. Thus for $R = 12, 16, 23$, or 32, values of $cmax \approx 0.2887, 0.25, 0.2085$, or 0.1762, respectively, correspond to processor precision where $\epsilon = 2^{-24}, 2^{-36}, 2^{-56}$, or 2^{-84}, respectively.

Evaluation of the ratio of gamma functions by exponentiating the difference of almost equal values of their logarithms would involve considerable loss of precision for large n. This is avoided by use of the asymptotic series obtained by differencing the Stirling approximations, changing the variable to $a = n - \frac{1}{2}$, and exponentiating the result (see also [1]):

$$\frac{\Gamma(\frac{1}{2}n + \frac{1}{2})}{\Gamma(\frac{1}{2}n)} = (\tfrac{1}{2}a)^{\frac{1}{2}} \sum_{r=0} C_r(4a)^{-2r},$$

where $C_0 = C_1 = 1$, $C_2 = -19/2$, $C_3 = 631/2$, $C_4 = -174317/8$, $C_5 = 204\ 91783/8$, $C_6 = -73348\ 01895/16$, $C_7 = 185\ 85901\ 54455/16$, $C_8 = -5\ 06774\ 10817\ 68765/128$, $C_9 = 2236\ 25929\ 81667\ 88235/128$, $C_{10} = -24\ 80926\ 53157\ 85763\ 70237/256$.

The relative error of the sum of the first s terms is negligible for $n > nmin$ where $| C_s | \times [4\ (nmin - \frac{1}{2})]^{-2s} \approx \epsilon$, e.g. for $s = 5$ and $\epsilon = 2^{-24}$ or 2^{-36}, $nmin \approx 6.271$ or 13.76, respectively, and for $s = 10$ and $\epsilon = 2^{-56}$ or 2^{-84}, $nmin \approx 15.5$ or 40.89, respectively. For smaller n the ratio of gamma functions is obtained from the ratio for some $N \geq nmin$ by the relation:

$$\frac{\Gamma(\frac{1}{2}n + \frac{1}{2})}{\Gamma(\frac{1}{2}n)} = \frac{n}{(n+1)}\frac{(n+2)}{(n+3)}\cdots\frac{(N-2)}{(N-1)}\frac{\Gamma(\frac{1}{2}N + \frac{1}{2})}{\Gamma(\frac{1}{2}N)}.$$

For large n, processor underflow at line 21 is avoided by use of the normal approximation, which is adequate for values of $n > 1/\epsilon$, whose representation is unaffected by subtraction of 0.5. Protection against negative or zero n is provided by returning the distinctive value, -1.0, which may be supplemented by an error diagnostic process, if required.

For double precision calculations speed is improved by evaluating higher order terms of the gamma ratio series using single precision operations. Comparison of double precision ($\epsilon = 2^{-84}$) results with single precision results ($\epsilon = 2^{-36}$, $nmin = 13.76$, $cmax = 0.25$) for a Control Data 3200 indicated achievement generally of about ten significant decimal digits, dropping to about eight significant decimals for arguments beyond the 10^{-20} probability level.

Valuable comments from the referee are gratefully acknowledged.

Reference

1. Fields, J.L. A note on the asymptotic expansion of a ratio of Gamma functions. *Proc. Edinburgh Math. Soc. Ser. 2 15* (1966), 43–45.

Algorithm

```
real procedure t frequency (t, n);
  value t, n; real t, n;
if n ≤ 0.0 then t frequency := −1.0
else
begin
  real a, b, c, d, e, nmin, cmax;
  comment for 36-bit precision processor;
  nmin := 13.76; cmax := 0.25;
  b := t × t; c := b/n; a := d := b + c;
  if c > cmax then b := (n+1.0) × ln(1.0+c)
  else
  for e := 2.0, e + 1.0 while b ≠ d do
  begin a := −a × c; b := d; d := a/e + d end;
  a := n; c := 0.3989422804;
  comment 1/sqrt(2π) = 0.398942280401432677939946l . . . ;
  for e := a while e < nmin do
  begin c := c × a/(a+1.0); a := a + 2.0 end;
  a := a − 0.5;
  if a ≠ n then
  begin
    c := sqrt(a/n) × c; a := 0.25/a; a := a × a;
    c := ((((−21789.625×a+315.5)×a−9.5)×a+1.0)×a+1.0)
      × c
  end;
  t frequency := exp(−0.5×b) × c
end Student's t-frequency
```

Algorithm 466

Four Combinatorial Algorithms [G6]

Gideon Ehrlich [Recd. 25 Aug. 1971, 4 Jan. 1972, and 12 Dec. 1972]
Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel

Key Words and Phrases: permutations and combinations
CR Categories: 5.39
Language: PL/I

Description

Each of the following algorithms produce, by successive calls, a sequence of all combinatorial configurations, belonging to the appropriate type.

PERMU Permutations of $N \geq 3$ objects: $X(1), X(2), \ldots, X(N)$.

COMBI Combinations of M natural numbers out of the first N.

COMPOMIN Compositions of an integer P to $M + 1$ ordered terms, $INDEX(k)$, each of which is not less than a given minimum $MIN(k)$.

COMPOMAX The same as *COMPOMIN* but each term has its own maximum $MAX\ (k)$.

The four algorithms have in common the important property that they use neither loops nor recursion; thus the time needed for producing a new configuration is unaffected by the "size" (N, N and M, P and M respectively) of that configuration.

Each algorithm.uses a single simple operation for producing a new configuration from the old one, that is:

PERMU A single transposition of two adjacent elements.

COMBI Replacing a single element x by a y having the property that there is no element between x and y belonging to the combination.

COMPOMIN(*MAX*) Changing the values of two adjacent terms (usually only by 1).

The algorithms are written in *PL*1(*F*).

Special instructions for the user and notes.

PERMU (1) The mean work-time is actually a decreasing function of N since, on $(N - 1)/N$ of the calls, it returns by the first *RETURN*. (2) The procedure operates directly on any object vector $x[1{:}N]$. (3) For the first permutation one must call *FIRSTPER*; for other permutations *PERMU* must be used. (4) Together with the last permutation, which is the original one, we will get *DONE* = '1'*B*. If we continue to call *PERMU*, the entire sequence will repeat indefinitely. If at any stage we set *DONE* = '0'*B*, then at the end of the appropriate sequence it will become '1'*B*. (5) The entire resulting sequence is the same as that of Johnson [1] and Trotter [2].

COMBI Every combination is represented in two forms: (1) As a bit array of M '1's and $N - M$ '0's which is identical to $A(1), A(2), \ldots, A(N)$. (2) As an array C of M different integers not greater than N. The M elements are ordered according to their magnitude. If the second representation is not needed one can omit Z, H and C together with the last line of the procedure. For the first combination we can use the following initialization (for other initializations see [3]):

```
DECLARE A(0:N) BIT (1), (X, Y, T(N), F(0:N),
    I, L, Z, H(N), C(M)) FIXED;
DO K = 0 TO N − M; A(K) = '0'B; END;
DO K = N − M + 1 TO N; A(K) = '1'B; END;
DO K = 1 TO M; C(K) = N − M + K; H(N − M + K) = K;
    END;
T(N − M) = −1; T(1) = 0; F(N) = N − M + 1; I = N − M;
    L = N;
```

(The initialization was not done in the body of the procedure *COMBI* only in order to simplify the procedures *COMPOMIN-MAX*:.)

Instead of using such a large number of parameters it is possible to retain only A, I, L as parameters of the procedure and declare and initialize the other present parameters in the body of the procedure (as is done in *PERMU*). In such a case N, T, F, L, H must be declared as *STATIC* or *CONTROLLED* ('own' in *ALGOL*).

COMPOMIN Each of the $M + 1$ $MIN(k)$, as well as P, can be any integer (positive, negative, or zero), but the sum S of all those minima cannot be greater than P.

For the first composition set $INDEX(1) = P - S + MIN(1)$ $INDEX(k) = MIN(k)$, for $k > 1$.

Set $N = P - S + M$, and declare and initialize all variables that also appear in *COMBI* in the same way as was done for *COMBI*.

Together with the last composition, we will get $I = 0$ as a signal to halt.

COMPOMAX The instructions for *COMPOMIN* are valid for *COMPOMAX* provided: (1) *MIN* is replaced by *MAX* ($S \geq P$); and (2) N is initialized to $N = S - P + M$.

The vector C (but not H!) has no use in *COMPOMIN*(*MAX*), so one can omit all statements in which it appears. A justification for the four algorithms and for some others can be found in [3].

Acknowledgment. I would like to thank Professor Shimon Even for guidance and encouragement.

References

1. Johnson, S.N. Generation of permutations by adjacent transformations. *Math. Comp. 17* (1963), 282–285.
2. Trotter, H.F. Algorithm 115, Perm. *Comm ACM 5* (Aug. 1962), pp. 434–435.
3. Ehrlich, G., Loopless algorithms for generation permutations combinations and other combinatorial configurations. *J. ACM 20* (July 1973), 500–513.

Algorithm

```
FIRSTPER:  PROCEDURE  (X,DONE);
DECLARE  (X(*),  (XN,XX) STATIC) DECIMAL, DONE BIT(1)
(N,S,V,M,L,I,DI,IPI) BINARY STATIC,
(P(0:N),IP(N-1),D(N-1),T(N)) BINARY CONTROLLED;
N=DIM(X,1);
IF ALLOCATION (P) THEN FREE P,IP,D,T;  ALLOCATE P,IP,D,T;
DO M=1  TO  N-1;  P(M),IP(M)=M; D(M)=-1;  END;
XN=X(N); V=-1; S,P(0),P(N)=N; M,L=1;
T(N)=N-1; T(N-1)=-2; T(2)=2;
DONE='0'B;
PERMU:  ENTRY  (X,DONE);
IF S¬=M THEN DO; X(S)=X(S+V); S=S+V; X(S)=XN; RETURN; END;
```

```
I=T(N);                    DI=D(I);
IP(I),IPI=IP(I)+DI;        M=P(IPI);               IP(M)=IPI-DI;
P(IPI-DI)=M;               P(IPI)=I;               M=IPI+L;
XX=X(M);                   X(M)=X(M-DI);           X(M-DI)=XX;
L=1-L;                     V=-V;                   M=N+1-S;
IF P(IPI+DI) < I   THEN
DO;   IF I=N-1 THEN RETURN;
      T(N)=N-1; T(N-1) = -I; RETURN;
END;
D(I)=-DI;
IF T(I) < 0   THEN
DO; IF T(I)¬=1-I  THEN T(I-1)=T(I); T(I)=I-1; END;
IF I ¬ =N-1 THEN DO; T(N)=N-1; T(N-1)=-I-1; END;
T(I+1)=T(I):
IF I=2 & P(2)=2 THEN DONE='1'B;
END;
COMBI  PROCEDURE  (A,N,X,Y,T,F,I,L,Z,H,C);
DECLARE  A(*)BIT(1),  (N,X,Y,T(*),F(*),I,L,Z,H(*),C(*))  FIXED;
IF  T(I) < 0  THEN
DO;  IF -T(I)¬ =I-1  THEN  T(I-1)=T(I);  T(I)=I-1; END;
IF ¬ A(I)  THEN
DO;  X=I;  Y=F(L);
     IF  A(I-1)  THEN  F(I)=F(I-1);  ELSE F(I)=I; IF F(L)=L THEN
     DO; L=I; I=T(I); GOTO CHANGE; END;
     IF L=N THEN
     DO; T(F(N))=-I-1; T(I+1)=T(I); I=F(N);
         F(N)=F(N)+1; GOTO CHANGE;
     END;
     T(L)=-I-1; T(I+1)=T(I);
     F(L)=F(L)+1; I=L; GOTO CHANGE
END;
Y=I;
IF I¬=L THEN
DO;
     F(L),X=F(L)-1; F(I-1)=F(I);
     IF L=N THEN
     DO; IF I=F(N) -1 THEN DO; I=T(I); GOTO CHANGE; END;
         T(F(N)-1)=-I-1; T(I+1)=T(I);
         I=F(N)-1; GOTO CHANGE;
     END;
    T(L)=-I-1; T(I+1)=T(I); I=L; GOTO CHANGE;
END;
X=N; F(L-1)=F(L); F(N)=N; L=N;
IF I=N-1 THEN DO; I=T(N-1); GOTO CHANGE; END;
T(N-1)=-I-1; T(I+1)=T(I); I=N-1;
CHANGE;
A(X)='1'B; A(Y)='0'B;
H(X),Z=H(Y); C(Z)=X;
END COMBI;
COMPOMIN:  PROCEDURE (INDEX,A,N,X,Y,T,F,I,L,Z,H,C);
DECLARE A(*)  BIT(1),
       (INDEX(*),N,X,Y,T (*),F(*),I,L,Z,H(*),C(*))   FIXED;
CALL COMBI   (A,N,X,Y,T,F,I,L,Z,H,C);
INDEX(Z)=INDEX(Z)+X-Y;      INDEX(Z+1)=INDEX(Z+1)+Y-X:
END COMPOMIN;
COMPOMAX: PROCEDURE (INDEX,A,N,X,Y,T,F,I,L,Z,H,C);
DECLARE A(*)  BIT(1),
       (INDEX(*),N,X,Y,T(*),F(*),I,L,Z,H(*),C(*))  FIXED;
CALL COMBI (A,N,X,Y,T,F,I,L,Z,H,C);
INDEX(Z)=INDEX(Z)-X+Y;      INDEX(Z+1)=INDEX(Z+1)-Y+X;
END COMPOMAX;
```

Algorithm 467

Matrix Transposition in Place [F1]

Norman Brenner [Recd. 14 Feb. 1972, 2 Aug. 1972]
M.I.T., Department of Earth and Planetary Sciences,
Cambridge, MA 02139

Key Words and Phrases: transposition, matrix operations, permutations, primitive roots, number theory

CR Categories: 3.15, 5.14, 5.39

Language: Fortran

Description

Introduction. Since the problem of transposing a rectangular matrix in place was first proposed by Windley in 1959 [1], several algorithms have been used for its solution [2, 3, 7]. A significantly faster algorithm, based on a number theoretical analysis, is described and compared experimentally with existing algorithms.

Theory. A matrix a, of n_1 rows and n_2 columns, may be stored in a vector v in one of two ways. Element a_{ij} (0-origin subscripts) may be placed rowwise at v_k, $k = in_2 + j$, or columnwise at $v_{k'}$, $k' = i + jn_1$. Clearly, letting $n = n_1$ and $m = n_1n_2 - 1$,

$$k' \equiv nk \text{ (mod } m). \tag{1}$$

Transposition of the matrix is its conversion from one mode oi storage to the other, by performing the permutation (1). This permutation may be done with a minimum of working storage in a minimum number of exchanges by breaking it into its subcycles. For example, for a 4 × 9 matrix, one subcycle representation is

(0) (1 4 16 29 11 9) (34 31 19 6 24 26)
(22 18 2 8 32 23) (13 17 33 27 3 12)
(5 20 10) (30 15 25) (7 28) (14 21) (35).

The notation for the sixth subcycle, for example, means that $v_5 \leftarrow v_{20} \leftarrow v_{10} \leftarrow v_5$.

For a subcycle starting with element s, the elements of the subcycle are sn^r (mod m), for $r = 0, 1, \ldots$. The following theorems are easily established.

Theorem 1. *All the elements of the subcycle beginning with s are divisible by $d = (s, m)$, the largest common factor of both s and m. They are divisible by no larger divisor of m.*

Proof. Both m and s are divisible by d, and therefore so is any subcycle element sn^r (mod m). But n and m have no common factors (since $m = nn_2 - 1$), so no divisor of m larger than d can divide sn^r.□

Theorem 2. *For every subcycle beginning with s, there is another (possibly the same) subcycle beginning with $m - s$.*

Proof. The elements of the second subcycle are just $-sn^r$ (mod m). It is the same subcycle if for some r, $n^r \equiv -1$ (mod m'), for $m' = m/(s, m)$.□

The next theorem gives the group representation of the integers modulo m.

Theorem 3. *Factor m into powers of primes, $m = p_1^{\alpha_1} \cdots p_l^{\alpha_l}$. Let r_i be a primitive root of p_i; that is, the powers $r_i{}^k$ (mod p_i) for $k = 0, 1, \ldots, p - 2$, comprise every positive integer less than p_i. Define the generator $g_i = 1 + Rm/p_i^{\alpha_i}$, where $R \equiv (r_i - 1)$ $(m/p_i^{\alpha_i})^{-1}$ (mod $p_i^{\alpha_i}$). Define the Euler totient function $\phi(1) = 1$; otherwise $\phi(k)$ = the number of integers less than k having no common factor with it. Then, for any integer x less than m, there exist unique indices j_i for which $0 \leq j_i < \phi(p_i^{\alpha_i}/(x, p_i^{\alpha_i}))$ and $x \equiv (x, m)g_1^{j_1} \cdots g_l^{j_l}$(mod m).*

Proof. In [4]; if any $p_i = 2$, replace $g_i^{j_i}$ by $\pm 5^{j_i}$, where $0 \leq j_i < \phi(2^{\alpha_i - 2}/(x, 2^{\alpha_1 - 2}))$. □

For example, for $m = 35$, as in our example above, $x \equiv 22^{j_1}31^{j_2}$ (mod 35) for $(x,35) = 1$ and for $0 \leq j_1 < 4$ and $0 \leq j_2 < 6$.

Index notation is analogous to logarithmic notation in that multiplication modulo m becomes merely addition of indices.

The following theorem solves the problem of the subcycle starting points. It is similar to the algorithm in [6].

Theorem 4. *Let n and m be defined as for (1). Then, for any integer x less than m, upper bounds J_i may be found so that unique indices j_i exist in the range $0 \leq j_i < J_i$ and $x \equiv \pm(x, m)$ $n^{j_0}g_1^{j_1} \cdots g_l^{j_l}$ (mod m).*

Proof. Express n and -1 in index notation. Then, compute from the indices of n the smallest e such that $n^e \equiv 1$ (mod m). Initially, set each $J_i = \phi(p_i^{\alpha_i}/(x, p_i^{\alpha_i}))$. Next, doing only index arithmetic, examine each power $\pm n^j$ for nontrivial relations of the form $g_i^{j_i} \equiv \pm n^j g_1^{j_1} \cdots g_l^{j_l}$ (mod $m/(x, m)$) where $0 \leq j_k < J_k$ for each k. Then set $J_i = j_i$. Stop when the product of the J_i and e equals $\phi(m/(x, m))$, which is the number of integers in subcycles divisible only by (x, m).□

Notice that the choice of J_i by this method is not unique. For example, continuing from above, for $(x, m) = 7$, $n = 4$, $x \equiv 7 \cdot 4^{j_0}22^{j_1}$ (mod 35), for $0 \leq j_0 < 2$ and $0 \leq j_1 < 2$. The relations found were $(-1)^1 \equiv 4^1$ (mod 5), $22^2 \equiv 4^1$ (mod 5) and $31^1 \equiv 4^0$ (mod 5).

Theorem 4 is more important in theory than in practice. The tremendous labor in finding primitive roots for large primes (since a table of roots is very bulky) and in finding the index representation of n is not compensated for by time savings afterward; see the timing tests below. The same practical objection holds against the algorithm in [6].

Algorithm. An efficient program breaks naturally into two parts. First determine starting points for the subcycles and then move the data. In each part, the program below is significantly faster than Algorithm 380 in [3].

For each divisor d of m, the subcycles beginning with d and with $m - d$ are done. If the number of data moved is still less than $\phi(m/d)$, further subcycle starting points of the form sd are tried, for $s = 2, 3, \ldots$. The most general test is that sd is acceptable if no element in its subcycle is less than sd or greater than $m - sd$. Since this test requires much time-consuming computation, it is much faster to look for sd in a table where marks are made to indicate that an element has been moved. In some applications, a bit within each datum may be used. For example, if the data are all biased positive, the sign bit may be used; or, for normalized, nonzero, binary floating point data, the high bit of the fraction is always one and so may be used. In general, a special table of length *NWORK* is used. As in [3], *NWORK* = $(n_1 + n_2)/2$ was found to be sufficient for most cases. However, when m has many divisors, Algorithm 380 must perform the time-consuming general test for many possible starting points when the new algorithm need not.

The inner loop of the algorithm computes (1), moves data, marks in the table, and checks for loop closure. Since the major part of the time of the inner loop is calculating (1), time is saved over Algorithm 380 by moving elements v_k and v_{m-k} simultaneously.

Timing Tests

n_1	n_2	m	(all times in msec)							
			Alg.302	Alg.380	Alg.380	*XPOS*	*XPOS*			
				IWRK=0	*IWRK*=	*NWORK*=0	*NWORK*=			
					$(n_1+n_2)/2$		$(n_1+n_2)/2$			
			T_1	T_2	T_3	T_4	T_5	T_1/T_4	T_2/T_4	T_3/T_5
45	50	13·173	350	317	167	133	67	2,62	2,38	2,50
45	60	2699	558	123	117	90	100	6,20	1,37	1,17
46	50	$11^2\cdot 19$	367	339	217	106	83	3,46	3,21	2,60
46	60	31·89	425	350	250	133	83	3,19	2,63	3,00
47	50	$3^4\cdot 29$	383	378	267	72	67	5,18	5,23	4,00
47	60	2819	483	127	133	90	100	5,36	1,41	1,33
45	180	7·13·89	1200	1050	816	517	300	2,25	2,03	2,72
45	200	8999	1767	408	416	283	300	6,25	1,44	1,39
46	180	17·487	1816	1233	583	267	267	6,41	4,63	2,19
46	200	9199	1700	508	417	383	317	4,44	1,33	1,32
47	180	11·769	1450	1133	667	383	267	3,78	2,96	2,50
47	200	3·13·241	983	1150	1067	550	467	1,69	2,09	2,29

In special cases, further savings may be made. For example, m is divisible by 2 only when both n_1 and n_2 are odd. Then the subcycles beginning at $m/2 - s$ and $m/2 + s$ may be done simultaneously with the subcycles from s and $m - s$, thus reducing the number of times (1) is computed.

Timing tests. A set of test matrices were transposed on the 360/65 with all programs written in Fortran H, OPT = 2. The new algorithm was always faster than both Algorithm 380 [3] and Algorithm 302 [2] when $NWORK = (n_1 + n_2)/2$. When $NWORK = 0$, it was slower than Algorithm 380 (for $IWRK = 0$) and Algorithm 302 only for a few cases when $n_1 n_2 < 100$. It was especially faster than Algorithm 380 when $m = n_1 n_2 - 1$ had many factors and there were hence many subcycles.

An experiment was made for cases when m was prime. A known primitive root of m was then taken from a table [5] and was used to generate subcycle starting points. Since no time was wasted in finding the primitive root or in finding subcycle starting points, this test showed the maximum time savable by implementing Theorem 4. For $NWORK = (n_1 + n_2)/2$ and $m > 200$, no improvement was found over the normal algorithm. For $NWORK = 0$, the gain in speed was never more than 25 percent.

References

1. Windley, P.F. Transposing matrices in a digital computer. *Comp. J. 2* (Apr. 1959), 47–48.
2. Boothroyd, J. Algorithm 302, Transpose vector stored array. *Comm. ACM 10* (May 1967), 292–293.
3. Laflin, S., and Brebner, M.A. Algorithm 380: In-situ transposition of a rectangular matrix. *Comm. ACM 13* (May 1970), 324–326.
4. Bolker, E. *An Introduction to Number Theory: An Algebraic Approach.* Benjamin, New York, 1970.
5. Abramowitz, M., and Stegun, I. *Handbook of Mathematical Functions,* Table 24.8. Nat. Bur. of Standards, Washington, D.C., 1964.
6. Pall, G., and Seiden, E. A problem in Abelian Groups, with application to the transposition of a matrix on an electronic computer. *Math. Comp. 14* (1960), 189–192.
7. Knuth, D., *The Art of Computer Programming, Vol. I.* Addison-Wesley, Reading, Mass., 1967, p. 180, prob. 12, and p. 517, solution to prob. 12.

Algorithm

```
      SUBRØUTINE XPØSE(A, N1, N2, N12, MØVED, NWØRK)
C TRANSPØSITIØN ØF A RECTANGULAR MATRIX IN SITU.
C BY NØRMAN BRENNER, MIT, 1/72.  CF. ALG. 380, CACM, 5/70.
C TRANSPØSITIØN ØF THE N1 BY N2 MATRIX A AMØUNTS TØ
C REPLACING THE ELEMENT AT VECTØR PØSITIØN I (0-ØRIGIN)
C WITH THE ELEMENT AT PØSITIØN N1*I (MØD N1*N2-1).
C EACH SUBCYCLE ØF THIS PERMUTATIØN IS CØMPLETED IN ØRDER.
C MØVED IS A LØGICAL WØRK ARRAY ØF LENGTH NWØRK.
      LØGICAL MØVED
      DIMENSIØN A(N12), MØVED(NWØRK)
C REALLY A(N1,N2), BUT N12 = N1*N2
      DIMENSIØN IFACT(8), IPØWER(8), NEXP(8), IEXP(8)
      IF (N1.LT.2 .ØR. N2.LT.2) RETURN
      N = N1
      M = N1*N2 - 1
      IF (N1.NE.N2) GØ TØ 30
C SQUARE MATRICES ARE DØNE SEPARATELY FØR SPEED
      I1MIN = 2
      DØ 20 I1MAX=N,M,N
        I2 = I1MIN + N - 1
        DØ 10 I1=I1MIN,I1MAX
          ATEMP = A(I1)
          A(I1) = A(I2)
          A(I2) = ATEMP
          I2 = I2 + N
   10   CØNTINUE
        I1MIN = I1MIN + N + 1
   20 CØNTINUE
      RETURN
C MØDULUS M IS FACTØRED INTØ PRIME PØWERS.  EIGHT FACTØRS
C SUFFICE UP TØ M = 2*3*5*7*11*13*17*19 = 9,767,520.
   30 CALL FACTØR(M, IFACT, IPØWER, NEXP, NPØWER)
      DØ 40 IP=1,NPØWER
        IEXP(IP) = 0
   40 CØNTINUE
C GENERATE EVERY DIVISØR ØF M LESS THAN M/2
      IDIV = 1
   50 IF (IDIV.GE.M/2) GØ TØ 190
C THE NUMBER ØF ELEMENTS WHØSE INDEX IS DIVISIBLE BY IDIV
C AND BY NØ ØTHER DIVISØR ØF M IS THE EULER TØTIENT
C FUNCTIØN, PHI(M/IDIV).
      NCØUNT = M/IDIV
      DØ 60 IP=1,NPØWER
        IF (IEXP(IP).EQ.NEXP(IP)) GØ TØ 60
        NCØUNT = (NCØUNT/IFACT(IP))*(IFACT(IP)-1)
   60 CØNTINUE
      DØ 70 I=1,NWØRK
        MØVED(I) = .FALSE.
   70 CØNTINUE
C THE STARTING PØINT ØF A SUBCYCLE IS DIVISIBLE ØNLY BY IDIV
C AND MUST NØT APPEAR IN ANY ØTHER SUBCYCLE.
      ISTART = IDIV
   80 MMIST = M - ISTART
      IF (ISTART.EQ.IDIV) GØ TØ 120
      IF (ISTART.GT.NWØRK) GØ TØ 90
      IF (MØVED(ISTART)) GØ TØ 160
   90 ISØID = ISTART/IDIV
      DØ 100 IP=1,NPØWER
        IF (IEXP(IP).EQ.NEXP(IP)) GØ TØ 100
        IF (MØD(ISØID,IFACT(IP)).EQ.0) GØ TØ 160
  100 CØNTINUE
      IF (ISTART.LE.NWØRK) GØ TØ 120
      ITEST = ISTART
  110 ITEST = MØD(N*ITEST,M)
      IF (ITEST.LT.ISTART .ØR. ITEST.GT.MMIST) GØ TØ 160
      IF (ITEST.GT.ISTART .AND. ITEST.LT.MMIST) GØ TØ 110
  120 ATEMP = A(ISTART+1)
      BTEMP = A(MMIST+1)
      IA1 = ISTART
  130 IA2 = MØD(N*IA1,M)
      MMIA1 = M - IA1
      MMIA2 = M - IA2
      IF (IA1.LE.NWØRK) MØVED(IA1) = .TRUE.
      IF (MMIA1.LE.NWØRK) MØVED(MMIA1) = .TRUE.
      NCØUNT = NCØUNT - 2
```

```
C MØVE TWØ ELEMENTS, THE SECØND FRØM THE NEGATIVE
C SUBCYCLE.  CHECK FIRST FØR SUBCYCLE CLØSURE.
      IF (IA2.EQ.ISTART) GØ TØ 140
      IF (MMIA2.EQ.ISTART) GØ TØ 150
      A(IA1+1) = A(IA2+1)
      A(MMIA1+1) = A(MMIA2+1)
      IA1 = IA2
      GØ TØ 130
  140 A(IA1+1) = ATEMP
      A(MMIA1+1) = BTEMP
      GØ TØ 160
  150 A(IA1+1) = BTEMP
      A(MMIA1+1) = ATEMP
  160 ISTART = ISTART + IDIV
      IF (NCØUNT.GT.0) GØ TØ 80
      DØ 180 IP=1,NPØWER
        IF (IEXP(IP).EQ.NEXP(IP)) GØ TØ 170
        IEXP(IP) = IEXP(IP) + 1
        IDIV = IDIV*IFACT(IP)
        GØ TØ 50
  170   IEXP(IP) = 0
        IDIV = IDIV/IPØWER(IP)
  180 CØNTINUE
  190 RETURN
      END

      SUBRØUTINE FACTØR(N, IFACT, IPØWER, NEXP, NPØWER)
C FACTØR N INTØ ITS PRIME PØWERS, NPØWER IN NUMBER.
C E.G., FØR N=1960=2**3 *5 *7**2, NPØWER=3, IFACT=3,5,7,
C IPØWER=8,5,49, AND NEXP=3,1,2.
      DIMENSIØN IFACT(8), IPØWER(8), NEXP(8)
      IP = 0
      IFCUR = 0
      NPART = N
      IDIV = 2
   10 IQUØT = NPART/IDIV
      IF (NPART-IDIV*IQUØT) 60, 20, 60
   20 IF (IDIV-IFCUR) 40, 40, 30
   30 IP = IP + 1
      IFACT(IP) = IDIV
      IPØWER(IP) = IDIV
      IFCUR = IDIV
      NEXP(IP) = 1
      GØ TØ 50
   40 IPØWER(IP) = IDIV*IPØWER(IP)
      NEXP(IP) = NEXP(IP) + 1
   50 NPART = IQUØT
      GØ TØ 10
   60 IF (IQUØT-IDIV) 100, 100, 70
   70 IF (IDIV-2) 80, 80, 90
   80 IDIV = 3
      GØ TØ 10
   90 IDIV = IDIV + 2
      GØ TØ 10
  100 IF (NPART-1) 140, 140, 110
  110 IF (NPART-IFCUR) 130, 130, 120
  120 IP = IP + 1
      IFACT(IP) = NPART
      IPØWER(IP) = NPART
      NEXP(IP) = 1
      GØ TØ 140
  130 IPØWER(IP) = NPART*IPØWER(IP)
      NEXP(IP) = NEXP(IP) + 1
  140 NPØWER = IP
      RETURN
      END
```

Algorithm 468

Algorithm for Automatic Numerical Integration Over a Finite Interval [D1]

T.N.L. Patterson [Recd. 20 Jan. 1971, 27 Nov. 1972, 12 Dec. 1972, 26 Mar. 1973]
Department of Applied Mathematics and Theoretical Physics, The Queen's University of Belfast, Belfast BT7 1NN Northern Ireland

Key Words and Phrases: automatic integration, numerical integration, automatic quadrature, numerical quadrature
CR Categories: 5.16
Language: Fortran

Editor's note: *Algorithm 468 described here is available on magnetic tape from the Department of Computer Science, University of Colorado, Boulder, CO 80302. The cost for the tape is $16.00 (U.S. and Canada) or $18.00 (elsewhere). If the user sends a small tape (wt. less than 1 lb.) the algorithm will be copied on it and returned to him at a charge of $10.00 (U.S. only). All orders are to be prepaid with checks payable to ACM Algorithms. The algorithm is recorded as one file of BCD 80 character card images at 556 B.P.I., even parity, on seven track tape. We will supply algorithm at a density of 800 B.P.I. if requested. Cards for algorithms are sequenced starting at 10 and incremented by 10. The sequence number is right justified in column 80. Although we will make every attempt to insure that the algorithm conforms to the description printed here, we cannot guarantee it, nor can we guarantee that the algorithm is correct.—L.D.F. and A.K.C.*

Description

Purpose. The algorithm attempts to calculate automatically the integral of $F(x)$ over the finite interval $[A, B]$ with relative error not exceeding a specified value ϵ.

Method. The method uses a basic integration algorithm applied under the control of algorithms which invoke, if necessary, adaptive or nonadaptive subdivision of the range of integration. The basic algorithm is sufficiently powerful that the subdivision processes will normally only be required on very difficult integrals and might be regarded as a rescue operation.

The Basic Algorithm. The basic algorithm, *QUAD*, uses a family of interlacing whole-interval, common-point, quadrature formulas. The construction of the family is described in detail in [1]. Beginning with the 3-point Gauss rule, a new 7-point rule is derived, with three of the abscissae coinciding with the original Gauss abscissae; the remaining four are chosen so as to give the greatest possible increase in polynomial integrating degree; the resulting 7-point rule has degree 11. The procedure is repeated, adding eight new abscissae to the 7-point rule to produce a 15-point rule of degree 23. Continuing, rules using 31, 63, 127, and 255 points of respective degree 47, 95, 191, and 383 are derived. The 255-point rule has not previously been published. In addition, a 1-point rule (abscissa at the mid-point of the interval of integration) is included in the family to make eight members in all. The 3-point Gauss rule is in fact formally the extension of this 1-point rule. The successive application of these rules, until the two most recent results differ relatively by ϵ or better, is the basis of the method. Due to their interlacing form, no integral evaluations need to be wasted in passing from one rule to the next.

The algorithm has been used for some time on practical problems and has been found to generally perform reliably and efficiently. Its domain of applicability generally coincides with that of the Gauss formula, which is much wider than commonly supposed [2]. It will perform best on "smooth" functions, but the degree of deterioration of performance when applied to functions with various types of eccentricities depends more on the harshness of these eccentricities than on their presence as such. Integrands with large peaks or even singularities at the ends of the interval of integration are handled reasonably well. It may be noted that none of the rules actually uses the end points of the interval as abscissae. Peaks in the integrand at the center of the interval and discontinuities in the integrand are less easily dealt with. Although it is recommended that the algorithm be applied using the control algorithms described later, if desired it can be used directly as follows.

The algorithm is entered by the statement:

CALL QUAD (A, B, RESULT, K, EPSIL, NPTS, ICHECK, F)

The user supplies:

A lower limit of integration.
B upper limit of integration.
EPSIL required relative error.
F $F(X)$ is a user written function to calculate the integrand.

The algorithm returns:

RESULT an array whose successive elements *RESULT*(1), *RESULT*(2), etc., contain the results of applying the successive members of the family of rules. The number of rules actually applied depends on *EPSIL*. The array should be declared by the calling program to have at least eight elements.

K element, *RESULT*(*K*), of array *RESULT* contains the value of the integral to the required relative accuracy. *K* is determined from the convergence criterion:

$$| RESULT\,(K) - RESULT\,(K-1)\,| \leq EPSIL^{*}\,|\,RESULT\,(K)\,|$$

NPTS number of integrand evaluations.

ICHECK this flag will normally be 0 on exiting from the subroutine. However, if the convergence criterion above is not satisfied after exhausting all members of the family of rules, then the flag is set to 1.

The control algorithms. Two control algorithms are provided, *QSUBA* and *QSUB*, which if necessary invoke subdivision respectively in either an adaptive or a nonadaptive manner. *QSUBA* is generally more efficient than *QSUB*, but since there are reasons for believing [2] that adaptive subdivision is intrinsically less reliable than the nonadaptive form, an alternative is provided.

Table I. Test Integrals and Their Values

1. $\int_0^1 \sqrt{x}\, dx = \frac{2}{3}$

2. $\int_{-1}^1 [0.92 \cosh (x) - \cos (x)]\, dx \doteq 0.4794282267$

3. $\int_{-1}^1 dx/(x^4 + x^2 + 0.9) \doteq 1.582232964$

4. $\int_0^1 x^{\frac{3}{2}}\, dx = \frac{2}{5}$

5. $\int_0^1 dx/(1 + x^4) \doteq 0.8669729873$

6. $\int_0^1 dx/(1 + 0.5 \sin (31.4159x)) \doteq 1.154700669$

7. $\int_0^1 x\, dx/(e^x - 1) \doteq 0.7775046341$

8. $\int_{0.1}^1 \sin (314.159x)/(3.14159x)\, dx \doteq 0.009098645256$

9. $\int_0^{10} 50\, dx/(2500x^2 + 1)/3.14159 \doteq 0.4993638029$

10. $\int_0^{3.1415927} \cos (\cos (x) + 3 \sin (x) + 2 \cos (2x) + 3 \cos (3x) + 3 \sin (2x))\, dx \doteq 0.8386763234$

11. $\int_0^1 \ln (x)\, dx = -1.0$

12. $\int_0^1 4\pi^2 x \sin (20\pi x) \cos (2\pi x)\, dx \doteq -0.6346651825$

13. $\int_0^1 dx/(1 + (230x - 30)^2) \doteq 0.0013492485650$

The adaptive algorithm QSUBA. QUAD is first applied to the whole interval. If a converged result is not obtained (that is, the convergence criterion is not satisfied), the following adaptive subdivision strategy is invoked. At each stage of the process an interval is presented for subdivision (initially the whole interval (*A*, *B*)). The interval is halved, and *QUAD* applied to each subinterval. If *QUAD* fails to converge on the first subinterval, the subinterval is stacked for future subdivision and the second subinterval immediately examined. If *QUAD* fails to converge on the second subinterval, it is immediately subdivided and the whole process repeated. Each time a converged result is obtained it is accumulated as the partial value of the integral. When *QUAD* converges on both subintervals the interval last stacked is chosen next for subdivision and the process repeated. A subinterval is not examined again once a converged result is obtained for it, so that a spurious convergence is more likely to slip through than for the nonadaptive algorithm *QSUB*.

The convergence criterion is slightly relaxed in that a panel is deemed to have been successfully integrated if either *QUAD* converges or the estimated absolute error committed on this panel does not exceed ϵ times the estimated absolute value of the integral over (*A*, *B*). This relaxation is to try to take account of a common situation where one particular panel causes special difficulty, perhaps due to a singularity of some type. In this case, *QUAD* could obtain nearly exact answers on all other panels, and so the relative error for the total integration would be almost entirely due to the delinquent panel. Without this condition the computation might continue despite the requested relative error being achieved. The risk of underestimating the relative error is increased by this procedure and a warning is provided when it is used.

The algorithm is written as a function with value that of the integral. The call takes the form:

QSUBA(*A*, *B*, *EPSIL*, *NPTS*, *ICHECK*, *RELERR*, *F*)

and causes $F(x)$ to be integrated over (*A*, *B*) with relative error hopefully not exceeding *EPSIL*. *RELERR* gives a crude estimate of the actual relative error obtained by summing the absolute values of the errors produced by *QUAD* on each panel (estimated as the differences of the last two iterates of *QUAD*) and dividing by the calculated value of the integral. The reliability of the algorithm will decrease for large *EPSIL*. It is recommended that *EPSIL* should generally be less than about 0.001. *F* should be declared *EXTERNAL* in the calling program. *NPTS* is the number of integrand evaluations used. The outcome of the integration is indicated by *ICHECK*:

ICHECK = 0. Convergence obtained without invoking subdivision. This corresponds to the direct use of *QUAD*.

ICHECK = 1. Subdivision invoked and a converged result obtained.

ICHECK = 2. Subdivision invoked and a converged result obtained but at some point the relaxed convergence criterion was used. If confidence in the result needs bolstering, *EPSIL* and *RELERR* may be checked for a serious discrepancy.

ICHECK negative. If during the subdivision process the stack of delinquent intervals becomes full a result is obtained, which may be unreliable, by continuing the integration and ignoring convergence failures of *QUAD* which cannot be accommodated on the stack. This occurrence is noted by returning *ICHECK* with negative sign.

The nonadaptive algorithm QSUB. QUAD is first applied to the whole interval. If a converged result is not obtained the following nonadaptive subdivision strategy is invoked.

Let the interval (*A*, *B*) be divided into 2^N panels at step *N* of the subdivision process. *QUAD* is first applied to the subdivided interval on which it last failed to converge, and if convergence is now achieved, the remaining panels are integrated. Should a convergence failure occur on any panel, the integration at that point is terminated and the procedure repeated with *N* increased by one. The strategy insures that possibly delinquent intervals are examined before work, which later might have to be discarded, is invested on well behaved panels. The process is complete when no convergence failure occurs on any panel, and the sum of the results obtained by *QUAD* on each panel is taken as the value of the integral.

The process is very cautious in that the subdivision of the interval (*A*, *B*) is uniform the fineness of which is controlled by the success of *QUAD*. In this way it is much more difficult for a spurious convergence to slip through than for *QSUBA*. The convergence criterion is relaxed as described for *QSUBA*.

The algorithm is used in the same way as *QSUBA* and is called with the same arguments as *QSUBA*. One of the possible values of *ICHECK* has a different interpretation:

ICHECK negative. If during the subdivision process the upper limit on the number of panels which may be generated is reached, a result is obtained, which may be unreliable, by continuing the integration ignoring convergence failures of *QUAD*. This occurrence is noted by returning *ICHECK* with negative sign.

Tests. The algorithms have been found to perform reliably on a large number of practical problems. To give a feeling for the performance, results for a number of contrived examples are given using the adaptive control algorithm, *QSUBA*. It would be difficult to justify these examples as acid tests of any method, but they have the advantage of having being quoted at various times in the literature.

For comparison a number of automatic procedures were used, which include *SQUANK* [3] (adaptive Simpson), as well as the

Table II. Relative Error Requested, 10^{-3}

Integral	N_{CADRE}	N_{QSUBA}	T_{CADRE}/T_{QSUBA}
1	17	15	1.8
2	17	7	2.9
3	33	15	4.4
4	9	7	1.9
5	9	7	2.2
6	175	127	3.2
7	9	7	1.8
8	1137	255	8.5
9	97	127	2.4
10	107	63	2.2
11	137	31	9.9
12	252	63	6.3
13	129	787	.52

N and *T* with appropriate subscripts give respectively the number of integrand evaluations and the time taken for the computation.

Table III. Relative Error Requested, 10^{-6}

1	33	63	.75
2	33	15	2.6
3	49	31	3.0
4	129	31	5.0
5	17	15	2.0
6	401	255	2.9
7	9	7	1.8
8	2633	255	18.
9	281	255	2.4
10	193	63	3.8
11	233	795	.74
12	532	127	6.4
13	305	1001	.90

Table IV. Relative Error Requested, 10^{-8}

1	65	255	.36
2	33	15	2.7
3	97	31	4.9
4	545	31	20.
5	65	31	3.6
6	569	255	3.8
7	17	15	1.6
8	4001	255	24.
9	337	255	2.8
10	305	127	2.8
11	297	2415	.28
12	932	127	10.
13	481	1017	1.1

modified Havie integrator [4] and *CADRE* [5] (both based on the Romberg scheme). The latter algorithm, which attempts to detect certain types of singularities using the Romberg table, was found, on the examples tried, to be the best overall competitor to *QSUBA*, and only this comparison is quoted. The Havie algorithm was particularly poor and had the disturbing feature of converging spuriously on periodic integrands. Thacher [6] has described the shortcomings of Romberg integration, and Algorithm 400 appears to exhibit them. *SQUANK* was found to be quite good when used at low accuracy, but the performance deteriorated as the demand for accuracy increased. It also gave trouble on some of the more awkward integrals such as 8 and 11. *SQUANK* also computes the integral in the context of absolute error, and since this is meaningless unless an estimate of the order of magnitude of the integral is known, the algorithm can hardly be described as automatic. *CADRE* allows a choice of absolute or relative error. A criticism sometimes levied at relative error is that should the integral turn out to be zero a difficulty will arise. The only advice that can be offered in this respect is that, should a user suspect that this is likely to happen, a constant should be added to the integrand reflecting some appropriate quantity such as the maximum of the integrand. The constant which will be integrated exactly can be removed after the algorithm has done its work.

The test integrals are listed in Table I, and the results obtained for various required relative accuracies in Tables II, III, and IV. Generally *QSUBA* is superior by a substantial margin. The methods are compared in terms of the number of integrand evaluations needed to obtain the required accuracy and also in terms of the times required. For simple integrands the bookkeeping time of some methods can be significant, and *QUAD* can obtain a considerable advantage by its relative simplicity. Integrals 11 and 13 are interesting examples of this. The number of integrand evaluations exceeding 255 indicates that *QSUBA* invoked subdivision to obtain the result. In Tables III and IV *QSUBA* returned *ICHECK* = 2 on integral 11, but the requested tolerance was achieved.

Integral 8 caused special difficulty to *CADRE*, and for Tables III and IV a converged result could be obtained only after a relatively large investment of computer time. The feature of *CADRE* to detect certain singularities should show up in integrals 1 and 11, but the gain does not emerge until high accuracy is requested as in Table IV. For harsher singularities the gain would likely become apparent earlier.

References

1. Patterson, T.N.L. The optimum addition of points to quadrature formulae. *Math. Comp. 22* (1968), 847–856.

2. Cranley, R., and Patterson, T.N.L. On the automatic numerical evaluation of definite integrals. *Comp. J., 14* (1971), 189–198.

3. Lyness, J.N. Algorithm 379, SQUANK. *Comm. ACM 13* (Apr. 1970), 260–263.

4. Wallick, G.C. Algorithm 400, Modified Havie integration. *Comm. ACM 13* (Oct. 1970), 622–624.

5. de Boor, Carl. CADRE: An algorithm for numerical quadrature. *Mathematical Software.* J.R. Rice (Ed.) Academic Press, New York, 1971, pp. 417–449.

6. Thacher, H.C. Jr. Remark on Algorithm 60, *Comm. ACM* (July, 1964), 420–421.

Algorithm

```
      SUBRØUTINE QUAD(A, B, RESULT, K, EPSIL, NPTS, ICHECK, F)
      DIMENSIØN FUNCT(127), P(381), RESULT(8)
C THIS SUBRØUTINE ATTEMPTS TØ CALCULATE THE INTEGRAL ØF F(X)
C ØVER THE INTERVAL *A* TØ *B* WITH RELATIVE ERRØR NØT
C EXCEEDING *EPSIL*.
C THE RESULT IS ØBTAINED USING A SEQUENCE ØF 1,3,7,15,31,63,
C 127, AND 255 PØINT INTERLACING FØRMULAE(NØ INTEGRAND
C EVALUATIØNS ARE WASTED) ØF RESPECTIVE DEGREE 1,5,11,23,
C 47,95,191 AND 383. THE FØRMULAE ARE BASED ØN THE ØPTIMAL
C EXTENSIØN ØF THE 3-PØINT GAUSS FØRMULA.  DETAILS ØF
C THE FØRMULAE ARE GIVEN IN 'THE ØPTIMUM ADDITIØN ØF PØINTS
C TØ QUADRATURE FØRMULAE' BY T.N.L. PATTERSØN,MATHS.CØMP.
C VØL 22,847-856,1968.
C                         *** INPUT ***
C A        LØWER LIMIT ØF INTEGRATIØN.
C B        UPPER LIMIT ØF INTEGRATIØN.
C EPSIL    RELATIVE ACCURACY REQUIRED. WHEN THE RELATIVE
C          DIFFERENCE ØF TWØ SUCCESSIVE FØRMULAE DØES NØT
C          EXCEED *EPSIL* THE LAST FØRMULA CØMPUTED IS TAKEN
C          AS THE RESULT.
C F        F(X) IS THE INTEGRAND.
C                         *** ØUTPUT ***
C RESULT   THIS ARRAY,WHICH SHØULD BE DECLARED TØ HAVE AT
C          LEAST 8 ELEMENTS, HØLDS THE RESULTS ØBTAINED BY
C          THE 1,3,7, ETC., PØINT FØRMULAE. THE NUMBER ØF
C          FØRMULAE CØMPUTED DEPENDS ØN *EPSIL*.
C K        RESULT(K) HØLDS THE VALUE ØF THE INTEGRAL TØ THE
C          SPECIFIED RELATIVE ACCURACY.
C NPTS     NUMBER INTEGRAND EVALUATIØNS.
C ICHECK   ØN EXIT NØRMALLY ICHECK=0. HØWEVER IF CØNVERGENCE
C          TØ THE ACCURACY REQUESTED IS NØT ACHIEVED ICHECK=1
C          ØN EXIT.
C ABSCISSAE AND WEIGHTS ØF QUADRATURE RULES ARE STACKED IN
C ARRAY *P* IN THE ØRDER IN WHICH THEY ARE NEEDED.
      DATA
     * P( 1),P( 2),P( 3),P( 4),P( 5),P( 6),P( 7),
     * P( 8),P( 9),P(10),P(11),P(12),P(13),P(14),
     * P(15),P(16),P(17),P(18),P(19),P(20),P(21),
     * P(22),P(23),P(24),P(25),P(26),P(27),P(28)/
     *  0.77459666924148337704E 00,0.55555555555555555556E 00,
     *  0.88888888888888888889E 00,0.26848808986833344073E 00,
     *  0.96049126870802028342E 00,0.10465622602646726519E 00,
     *  0.43424374934680255800E 00,0.40139741477596222291E 00,
*  0.45091653865847414235E 00,0.13441525524378422036E 00,
*  0.51603282997079739697E-01,0.20062852937698902103E 00,
*  0.99383196321275502221E 00,0.17001719629940260339E-01,
*  0.88845923287225699889E 00,0.92927195315124537686E-01,
*  0.62110294673722640294E 00,0.17151190913639138079E 00,
*  0.22338668642896688163E 00,0.21915685840158749640E 00,
*  0.22551049979820668739E 00,0.67207754295990703540E-01,
*  0.25807598096176653565E-01,0.10031427861179557877E 00,
*  0.84345657393211062463E-02,0.46462893261757986541E-01,
*  0.85755920049990351154E-01,0.10957842105592463824E 00/
 DATA
* P(29),P(30),P(31),P(32),P(33),P(34),P(35),
* P(36),P(37),P(38),P(39),P(40),P(41),P(42),
* P(43),P(44),P(45),P(46),P(47),P(48),P(49),
* P(50),P(51),P(52),P(53),P(54),P(55),P(56)/
*  0.99909812496766759766E 00,0.25447807915618744154E-02,
*  0.98153114955374010687E 00,0.16446049854387810934E-01,
*  0.92965485742974005667E 00,0.35957103307129322097E-01,
*  0.83672593816886873550E 00,0.56979509494123357412E-01,
*  0.70249620649152707861E 00,0.76879620499003531043E-01,
*  0.53131974364437562397E 00,0.93627109981264473617E-01,
*  0.33113539325797683309E 00,0.10566989358023480974E 00,
*  0.11248894313318662575E 00,0.11195687302095345688E 00,
*  0.11275525672076869161E 00,0.33603877148207730542E-01,
*  0.12903800100351265626E-01,0.50157139305899537414E-01,
*  0.42176304415588548391E-02,0.23231446639910269443E-01,
*  0.42877960025007734493E-01,0.54789210527962865032E-01,
*  0.12651565562300680114E-02,0.82230079572359296693E-02,
*  0.17978551568128270333E-01,0.28489754745833548613E-01/
 DATA
* P(57),P(58),P(59),P(60),P(61),P(62),P(63),
* P(64),P(65),P(66),P(67),P(68),P(69),P(70),
* P(71),P(72),P(73),P(74),P(75),P(76),P(77),
* P(78),P(79),P(80),P(81),P(82),P(83),P(84)/
*  0.38439810249455532039E-01,0.46813554990628012403E-01,
*  0.52834946790116519862E-01,0.55978436510476319408E-01,
*  0.99987288812035761194E 00,0.36322148184553065969E-03,
*  0.99720625937222195908E 00,0.25790497946856882724E-02,
*  0.98868475754742947994E 00,0.61155068221172463397E-02,
*  0.97218287474858179658E 00,0.10498246909621321898E-01,
*  0.94634285837340290515E 00,0.15406750466559497802E-01,
*  0.91037115695700429250E 00,0.20594233915912711149E-01,
*  0.86390793819369047715E 00,0.25869679327214746911E-01,
*  0.80694053195021761186E 00,0.31073551111687964880E-01,
*  0.73975604435269475868E 00,0.36064432780782572640E-01,
*  0.66290966002478059546E 00,0.40715510116944318934E-01,
*  0.57719571005204581484E 00,0.44914531653632197414E-01,
*  0.48361802694584102756E 00,0.48564330406673198716E-01/
 DATA
* P( 85),P( 86),P( 87),P( 88),P( 89),P( 90),P( 91),
* P( 92),P( 93),P( 94),P( 95),P( 96),P( 97),P( 98),
* P( 99),P(100),P(101),P(102),P(103),P(104),P(105),
* P(106),P(107),P(108),P(109),P(110),P(111),P(112)/
*  0.38335932419873034692E 00,0.51583253952048458777E-01,
*  0.27774982202182431507E 00,0.53905499335266063927E-01,
*  0.16823525155220746498E 00,0.55481404356559363988E-01,
*  0.56344313046592789972E-01,0.56277699831254301273E-01,
*  0.56377628360384717388E-01,0.16801938574103865271E-01,
*  0.64519000501757369228E-02,0.25078569652949768707E-01,
*  0.21088152457266328793E-02,0.11615723319955134727E-01,
*  0.21438980012503867246E-01,0.27394605263981432516E-01,
*  0.63260731936263354422E-03,0.41115039786546930472E-02,
*  0.89892757840641357233E-02,0.14244877372916774306E-01,
*  0.19219905124727766019E-01,0.23406777495314006201E-01,
*  0.26417473395058259931E-01,0.27989218255238159704E-01,
*  0.18073956444538835782E-03,0.12895240826104173921E-02,
*  0.30577534101755311361E-02,0.52491234548088591251E-02/
 DATA
* P(113),P(114),P(115),P(116),P(117),P(118),P(119),
* P(120),P(121),P(122),P(123),P(124),P(125),P(126),
* P(127),P(128),P(129),P(130),P(131),P(132),P(133),
* P(134),P(135),P(136),P(137),P(138),P(139),P(140)/
*  0.77033752332797418482E-02;0.10297116957956355524E-01,
*  0.12934839663607373455E-01,0.15536775555843982440E-01,
*  0.18032216390391286320E-01,0.20357755058472159467E-01,
*  0.22457265826816098707E-01,0.24282165203336599358E-01,
*  0.25791626976024229388E-01,0.26952749667633031963E-01,
*  0.27740702178279681994E-01,0.28138849915627150636E-01,
*  0.99998243035489159858E 00,0.50536095207862517625E-04,
*  0.99959879967191068325E 00,0.37774664632698466027E-03,
*  0.99831663531840739253E 00,0.93836984854238150079E-03,
*  0.99572410469840718851E 00,0.16811428654214699063E-02,
*  0.99149572117810613240E 00,0.25687649437940203731E-02,
*  0.98537149959852037111E 00,0.35728927835172996494E-02,
*  0.97714151463970571416E 00,0.46710503721143217474E-02,
*  0.96663785155841656709E 00,0.58434498758356395076E-02/
 DATA
* P(141),P(142),P(143),P(144),P(145),P(146),P(147),
* P(148),P(149),P(150),P(151),P(152),P(153),P(154),
* P(155),P(156),P(157),P(158),P(159),P(160),P(161),
* P(162),P(163),P(164),P(165),P(166),P(167),P(168)/
*  0.95373000642576113641E 00,0.70724899954335554680E-02,
*  0.93832039777959288365E 00,0.83428387539681577056E-02,
*  0.92034002547001242073E 00,0.96411777297025366953E-02,
*  0.89974489977694003664E 00,0.10955733387837901648E-01,
*  0.87651341448470526974E 00,0.12275830560082770087E-01,
*  0.85064449476835027976E 00,0.13591571009765546790E-01,
*  0.82215625436498040737E 00,0.14893641664815182035E-01,
*  0.79108493379984836143E 00,0.16173218729577719942E-01,
*  0.75748396638051363793E 00,0.17421930159464173747E-01,
*  0.72142308537009891548E 00,0.18631848256138790186E-01,
*  0.68298743109107922809E 00,0.19795495048097499488E-01,
*  0.64227664250975951377E 00,0.20905851445812023852E-01,
*  0.59940393024224289297E 00,0.21956366305317824939E-01,
*  0.55449513263193254887E 00,0.22940964229387748761E-01/
```

```
 DATA
* P(169),P(170),P(171),P(172),P(173),P(174),P(175),
* P(176),P(177),P(178),P(179),P(180),P(181),P(182),
* P(183),P(184),P(185),P(186),P(187),P(188),P(189),
* P(190),P(191),P(192),P(193),P(194),P(195),P(196)/
*  0.50768775753371660215E 00,0.23854052106038540080E-01,
*  0.45913001198983233287E 00,0.24690524744487676909E-01,
*  0.40897982122988867241E 00,0.25445769965464765813E-01,
*  0.35740383783153215238E 00,0.26115673376706097680E-01,
*  0.30457644155671404334E 00,0.26696622927450359906E-01,
*  0.25067873030348317661E 00,0.27185513229624791819E-01,
*  0.19589750271110015392E 00,0.27579749566481873035E-01,
*  0.14042423315256017459E 00,0.27877251476613701609E-01,
*  0.84454040083710883710E-01,0.28076455793817246607E-01,
*  0.28184648949745694339E-01,0.28176319033016602131E-01,
*  0.28188814180192358694E-01,0.84009692870519326354E-02,
*  0.32259500250878684614E-02,0.12539284826474884353E-01,
*  0.10544076228633167722E-02,0.58078616599775673635E-02,
*  0.10719490006251933623E-01,0.13697302631990716258E-01/
 DATA
* P(197),P(198),P(199),P(200),P(201),P(202),P(203),
* P(204),P(205),P(206),P(207),P(208),P(209),P(210),
* P(211),P(212),P(213),P(214),P(215),P(216),P(217),
* P(218),P(219),P(220),P(221),P(222),P(223),P(224)/
*  0.31630366082226447689E-03,0.20557519893273465236E-02,
*  0.44946378920320678616E-02,0.71224386864583871532E-02,
*  0.96099525623638830097E-02,0.11703388747657003101E-01,
*  0.13208736697529129966E-01,0.13994609127619079852E-01,
*  0.90372734658751149261E-04,0.64476204130572477933E-03,
*  0.15288767050877655684E-02,0.26245617274044295626E-02,
*  0.38516876166398709241E-02,0.51485584789781777618E-02,
*  0.64674198318036867274E-02,0.77683877779219912200E-02,
*  0.90161081951956431600E-02,0.10178877529236079733E-01,
*  0.11228632913408049354E-01,0.12141082601668299679E-01,
*  0.12895813488012114694E-01,0.13476374833816515982E-01,
*  0.13870351089139840997E-01,0.14069424957813575318E-01,
*  0.25157870384280661489E-04,0.18887326450650491366E-03,
*  0.46918492424785040975E-03,0.84057143271072246365E-03/
 DATA
* P(225),P(226),P(227),P(228),P(229),P(230),P(231),
* P(232),P(233),P(234),P(235),P(236),P(237),P(238),
* P(239),P(240),P(241),P(242),P(243),P(244),P(245),
* P(246),P(247),P(248),P(249),P(250),P(251),P(252)/
*  0.12843824718970101768E-02,0.17864463917586498247E-02,
*  0.23355251860571608737E-02,0.29217249379178197538E-02,
*  0.35362449977167777340E-02,0.41714193769840788528E-02,
*  0.48205888648512683476E-02,0.54778666939189508240E-02,
*  0.61379152800413850435E-02,0.67957855048827733948E-02,
*  0.74468208324075910174E-02,0.80866093647888599710E-02,
*  0.87109650797320868736E-02,0.93159241280693950932E-02,
*  0.98977475240487497440E-02,0.10452925722906011926E-01,
*  0.10978183152658912470E-01,0.11470482114693874380E-01,
*  0.11927026053019270040E-01,0.12345262372243838455E-01,
*  0.12722884982732382906E-01,0.13057836688353048840E-01,
*  0.13348311463725179953E-01,0.13592756614812395910E-01,
*  0.13789874783240936517E-01,0.13938625738306850804E-01,
*  0.14038227896908623303E-01,0.14088159516508301065E-01/
 DATA
* P(253),P(254),P(255),P(256),P(257),P(258),P(259),
* P(260),P(261),P(262),P(263),P(264),P(265),P(266),
* P(267),P(268),P(269),P(270),P(271),P(272),P(273),
* P(274),P(275),P(276),P(277),P(278),P(279),P(280)/
*  0.99999759637974846462E 00,0.69379364324108267170E-05,
*  0.99994399620705437576E 00,0.53275293669780613125E-04,
*  0.99976049092443204733E 00,0.13575491094922871973E-03,
*  0.99938033802502358193E 00,0.24921240048299729402E-03,
*  0.99874561446809511470E 00,0.38974528447328229322E-03,
*  0.99780535449595727456E 00,0.55429531493037471492E-03,
*  0.99651414591489027385E 00,0.74028280424450333046E-03,
*  0.99483150280062100052E 00,0.94536151685852538246E-03,
*  0.99272134428278861533E 00,0.11674841174299594077E-02,
*  0.99015137040077015918E 00,0.14049079956551446427E-02,
*  0.98709252795403406719E 00,0.16561127281544526052E-02,
*  0.98351865757863272876E 00,0.19197129710138724125E-02,
*  0.97940628167086268381E 00,0.21944069253638388388E-02,
*  0.97473445975240266776E 00,0.24789582266575679307E-02/
 DATA
* P(281),P(282),P(283),P(284),P(285),P(286),P(287),
* P(288),P(289),P(290),P(291),P(292),P(293),P(294),
* P(295),P(296),P(297),P(298),P(299),P(300),P(301),
* P(302),P(303),P(304),P(305),P(306),P(307),P(308)/
*  0.96948465950245923177E 00,0.27721957645934509940E-02,
*  0.96364062156981213252E 00,0.30730184347025783234E-02,
*  0.95718821610986096274E 00,0.33803979910869203823E-02,
*  0.95011529752129487656E 00,0.36933779170256508183E-02,
*  0.94241156519108305981E 00,0.40110687240750233989E-02,
*  0.93406843615772578800E 00,0.43326409680929828545E-02,
*  0.92507893290707565236E 00,0.46573172997568547773E-02,
*  0.91543758715576504064E 00,0.49843645647655386012E-02,
*  0.90514035881326159519E 00,0.53130866051870565663E-02,
*  0.89418456833555902286E 00,0.56428181013844441585E-02,
*  0.88256884024734190684E 00,0.59729195655081658049E-02,
*  0.87029305554811390585E 00,0.63027734490857587172E-02,
*  0.85735831088623215653E 00,0.66317812429018878941E-02,
*  0.84376688267270860104E 00,0.69593614093904229394E-02/
 DATA
* P(309),P(310),P(311),P(312),P(313),P(314),P(315),
* P(316),P(317),P(318),P(319),P(320),P(321),P(322),
* P(323),P(324),P(325),P(326),P(327),P(328),P(329),
* P(330),P(331),P(332),P(333),P(334),P(335),P(336)/
*  0.82952219463740140018E 00,0.72849479805538070639E-02,
*  0.81462878765513741344E 00,0.76079896657190565832E-02,
*  0.79909229096084140180E 00,0.79279493342948491103E-02,
*  0.78291939411828301639E 00,0.82443037630328680306E-02,
*  0.76611781930376009072E 00,0.85565435613076896192E-02,
     *  0.7486962936169366028 2E 00,0.88641732094824942641E-02,
     *  0.73066452124218126133E 00,0.91667111635607884067E-02,
     *  0.71203315536225203459E 00,0.94636899938300652943E-02,
     *  0.69281376977911470289E 00,0.97546565363174114611E-02,
     *  0.67301883023041847920E 00,0.10039172044056840798E-01,
     *  0.65266166541001749610E 00,0.10316812330947621682E-01,
     *  0.63175643771119423041E 00,0.10587167904885197931E-01,
     *  0.61031811371518640016E 00,0.10849844089337314099E-01,
     *  0.58836243444766254143E 00,0.11104461134006926537E-01/
      DATA
     * P(337),P(338),P(339),P(340),P(341),P(342),P(343),
     * P(344),P(345),P(346),P(347),P(348),P(349),P(350),
     * P(351),P(352),P(353),P(354),P(355),P(356),P(357),
     * P(358),P(359),P(360),P(361),P(362),P(363),P(364)/
     *  0.56590588542365442262E 00,0.11350654315980596602E-01,
     *  0.54296566649831149049E 00,0.11588074033043952568E-01,
     *  0.51955966153745702199E 00,0.11816385890830235763E-01,
     *  0.49570640791876146017E 00,0.12035270785279562630E-01,
     *  0.47142506587165887693E 00,0.12244424981611985899E-01,
     *  0.44673538766202847374E 00,0.12443560190714035263E-01,
     *  0.42165768662616330006E 00,0.12632403643542078765E-01,
     *  0.39621280605761593918E 00,0.12810698163877361967E-01,
     *  0.37042208795007823014E 00,0.12978202239537399286E-01,
     *  0.34430734159943802278E 00,0.13134690091960152836E-01,
     *  0.31789081206847668318E 00,0.13279951743930530650E-01,
     *  0.29119514851824668196E 00,0.13413793085110098513E-01,
     *  0.26424337241092676194E 00,0.13536035934956213614E-01,
     *  0.23705884558982972721E 00,0.13646518102571291428E-01/
      DATA
     * P(365),P(366),P(367),P(368),P(369),P(370),P(371),
     * P(372),P(373),P(374),P(375),P(376),P(377),P(378),
     * P(379),P(380),P(381)/
     *  0.20966523824318119477E 00,0.13745093443001896632E-01,
     *  0.18208649675925219825E 00,0.13831631909506428676E-01,
     *  0.15434681148137810869E 00,0.13906019601325461264E-01,
     *  0.12647058437230196685E 00,0.13968158806516938516E-01,
     *  0.98482396598119202090E-01,0.14017968039456608810E-01,
     *  0.70406976042855179063E-01,0.14055382072649964277E-01,
     *  0.42269164765363603212E-01,0.14080351962553661325E-01,
     *  0.14093886410782462614E-01,0.14092845069160408355E-01,
     *  0.14094407090096179347E-01/
      ICHECK = 0
C CHECK FØR TRIVIAL CASE.
      IF (A.EQ.B) GØ TØ 70
C SCALE FACTØRS.
      SUM = (B+A)/2.0
      DIFF = (B-A)/2.0
C 1-PØINT GAUSS
      FZERØ = F(SUM)
      RESULT(1) = 2.0*FZERØ*DIFF
      I = 0
      IØLD = 0
      INEW = 1
      K = 2
      ACUM = 0.0
      GØ TØ 30
   10 IF (K.EQ.8) GØ TØ 50
      K = K + 1
      ACUM = 0.0
C CØNTRIBUTIØN FRØM FUNCTIØN VALUES ALREADY CØMPUTED.
      DØ 20 J=1,IØLD
        I = I + 1
        ACUM = ACUM + P(I)*FUNCT(J)
   20 CØNTINUE
C CØNTRIBUTIØN FRØM NEW FUNCTIØN VALUES.
   30 IØLD = IØLD + INEW
      DØ 40 J=INEW,IØLD
        I = I + 1
        X = P(I)*DIFF
        FUNCT(J) = F(SUM+X) + F(SUM-X)
        I = I + 1
        ACUM = ACUM + P(I)*FUNCT(J)
   40 CØNTINUE
      INEW = IØLD + 1
      I = I + 1
      RESULT(K) = (ACUM+P(I)*FZERØ)*DIFF
C CHECK FØR CØNVERGENCE.
      IF (ABS(RESULT(K)-RESULT(K-1))-EPSIL*ABS(RESULT(K))) 60,
     * 60, 10
C CØNVERGENCE NØT ACHIEVED.
   50 ICHECK = 1
C NØRMAL TERMINATIØN.
   60 NPTS = INEW + IØLD
      RETURN
C TRIVIAL CASE
   70 K = 2
      RESULT(1) = 0.0
      RESULT(2) = 0.0
      NPTS = 0
      RETURN
      END

      FUNCTIØN QSUB(A, B, EPSIL, NPTS, ICHECK, RELERR, F)
C   THIS FUNCTIØN RØUTINE PERFØRMS AUTØMATIC INTEGRATIØN
C ØVER A FINITE INTERVAL USING THE BASIC INTEGRATIØN
C ALGØRITHM QUAD, TØGETHER WITH, IF NECESSARY, A NØN-
C ADAPTIVE SUBDIVISIØN PRØCESS.
C   THE CALL TAKES THE FØRM
C           QSUB(A,B,EPSIL,NPTS,ICHECK,RELERR,F)
C AND CAUSES F(X) TØ BE INTEGRATED ØVER (A,B) WITH RELATIVE
C ERRØR HØPEFULLY NØT EXCEEDING EPSIL.  SHØULD QUAD CØNVERGE
C (ICHECK=0) THEN QSUB WILL RETURN THE VALUE ØBTAINED BY IT
C ØTHERWISE SUBDIVISIØN WILL BE INVØKED AS A RESCUE
C ØPERATIØN IN A NØN-ADAPTIVE MANNER. THE ARGUMENT RELERR
C GIVES A CRUDE ESTIMATE ØF THE ACTUAL RELATIVE ERRØR
C ØBTAINED.
C   THE SUBDIVISIØN STRATEGY IS AS FØLLØWS
C LET THE INTERVAL (A,B) BE DIVIDED INTØ 2**N PANELS ,
C N ØF THE SUBDIVISIØN PRØCESS.  QUAD IS APPLIED FIRS1
C THE SUBDIVIDED INTERVAL ØN WHICH QUAD LAST FAILED TØ
C CØNVERGE AND IF CØNVERGENCE IS NØW ACHIEVED THE REMA.
C PANELS ARE INTEGRATED.  SHØULD A CØNVERGENCE FAILURE
C ØN ANY PANEL THE INTEGRATIØN AT THAT PØINT IS TERMINATED
C AND THE PRØCEDURE REPEATED WITH N INCREASED BY 1. THE
C STRATEGY INSURES THAT PØSSIBLY DELINQUENT INTERVALS ARE
C EXAMINED BEFØRE WØRK, WHICH LATER MIGHT HAVE TØ BE
C DISCARDED, IS INVESTED ØN WELL BEHAVED PANELS. THE
C PRØCESS IS CØMPLETE WHEN NØ CØNVERGENCE FAILURE ØCCURS ØN
C ANY PANEL AND THE SUM ØF THE RESULTS ØBTAINED BY QUAD ØN
C EACH PANEL IS TAKEN AS THE VALUE ØF THE INTEGRAL.
C   THE PRØCESS IS VERY CAUTIØUS IN THAT THE SUBDIVISIØN ØF
C THE INTERVAL (A,B) IS UNIFØRM, THE FINENESS ØF WHICH IS
C CØNTRØLLED BY THE SUCCESS ØF QUAD.  IN THIS WAY IT IS
C RATHER DIFFICULT FØR A SPURIØUS CØNVERGENCE TØ SLIP
C THRØUGH.
C   THE CØNVERGENCE CRITERIØN ØF QUAD IS SLIGHTLY RELAXED
C IN THAT A PANEL IS DEEMED TØ HAVE BEEN SUCCESSFULLY
C INTEGRATED IF EITHER QUAD CØNVERGES ØR THE ESTIMATED
C ABSØLUTE ERRØR CØMMITTED ØN THIS PANEL DØES NØT EXCEED
C EPSIL TIMES THE ESTIMATED ABSØLUTE VALUE ØF THE INTEGRAL
C ØVER (A,B).  THIS RELAXATIØN IS TØ TRY TØ TAKE ACCØUNT ØF
C A CØMMØN SITUATIØN WHERE ØNE PARTICULAR PANEL CAUSES
C SPECIAL DIFFICULTY, PERHAPS DUE TØ A SINGULARITY ØF SØME
C TYPE.  IN THIS CASE QUAD CØULD ØBTAIN NEARLY EXACT
C ANSWERS ØN ALL ØTHER PANELS AND SØ THE RELATIVE ERRØR FØR
C THE TØTAL INTEGRATIØN WØULD BE ALMØST ENTIRELY DUE TØ THE
C DELINQUENT PANEL.  WITHØUT THIS CØNDITIØN THE CØMPUTATIØN
C MIGHT CØNTINUE DESPITE THE REQUESTED RELATIVE ERRØR BEING
C ACHIEVED.
C   THE ØUTCØME ØF THE INTEGRATIØN IS INDICATED BY ICHECK.
C   ICHECK=0  -  CØNVERGENCE ØBTAINED WITHØUT INVØKING
C                SUBDIVISIØN.  THIS CØRRESPØNDS TØ THE
C                DIRECT USE ØF QUAD.
C   ICHECK=1  -  RESULT ØBTAINED AFTER INVØKING SUBDIVISIØN.
C   ICHECK=2  -  AS FØR ICHECK=1 BUT AT SØME PØINT THE
C                RELAXED CØNVERGENCE CRITERIØN WAS USED.
C                THE RISK ØF UNDERESTIMATING THE RELATIVE
C                ERRØR WILL BE INCREASED.  IF NECESSARY,
C                CØNFIDENCE MAY BE RESTØRED BY CHECKING
C                EPSIL AND RELERR FØR A SERIØUS DISCREPANCY.
C   ICHECK NEGATIVE
C                IF DURING THE SUBDIVISIØN PRØCESS THE
C                ALLØWED UPPER LIMIT ØN THE NUMBER ØF PANELS
C                THAT MAY BE GENERATED (PRESENTLY 4096) IS
C                REACHED A RESULT IS ØBTAINED WHICH MAY BE
C                UNRELIABLE BY CØNTINUING THE INTEGRATIØN
C                WITHØUT FURTHER SUBDIVISIØN IGNØRING
C                CØNVERGENCE FAILURES. THIS ØCCURRENCE IS
C                FLAGGED BY RETURNING ICHECK WITH NEGATIVE
C                SIGN.
C  THE RELIABILITY ØF THE ALGØRITHM WILL DECREASE FØR LARGE
C  VALUES ØF EPSIL.  IT IS RECØMMENDED THAT EPSIL SHØULD
C  GENERALLY BE LESS THAN ABØUT 0.001.
      DIMENSIØN RESULT(8)
      INTEGER BAD, ØUT
      LØGICAL RHS
      EXTERNAL F
      DATA NMAX/4096/
      CALL QUAD(A, B, RESULT, K, EPSIL, NPTS, ICHECK, F)
      QSUB = RESULT(K)
      RELERR = 0.0
      IF (QSUB.NE.0.0) RELERR =
     * ABS((RESULT(K)-RESULT(K-1))/QSUB)
C CHECK IF SUBDIVISIØN IS NEEDED.
      IF (ICHECK.EQ.0) RETURN
C SUBDIVIDE
      ESTIM = ABS(QSUB*EPSIL)
      IC = 1
      RHS = .FALSE.
      N = 1
      H = B - A
      BAD = 1
   10 QSUB = 0.0
      RELERR = 0.0
      H = H*0.5
      N = N + N
C INTERVAL (A,B) DIVIDED INTØ N EQUAL SUBINTERVALS.
C INTEGRATE ØVER SUBINTERVALS BAD TØ (BAD+1) WHERE TRØUBLE
C HAS ØCCURRED.
      M1 = BAD
      M2 = BAD + 1
      ØUT = 1
      GØ TØ 50
C INTEGRATE ØVER SUBINTERVALS 1 TØ (BAD-1)
   20 M1 = 1
      M2 = BAD - 1
      RHS = .FALSE.
      ØUT = 2
      GØ TØ 50
C INTEGRATE ØVER SUBINTERVALS (BAD+2) TØ N.
   30 M1 = BAD + 2
      M2 = N
      ØUT = 3
      GØ TØ 50
C SUBDIVISIØN RESULT
   40 ICHECK = IC
      RELERR = RELERR/ABS(QSUB)
      RETURN
C INTEGRATE ØVER SUBINTERVALS M1 TØ M2.
   50 IF (M1.GT.M2) GØ TØ 90
      DØ 80 JJ=M1,M2
        J = JJ
     *  0.45091653865847414235E 00,0.13441525524378422036E 00,
     *  0.51603282997079739697E-01,0.20062852937698902103E 00,
     *  0.99383196321275502221E 00,0.17001719629940260339E-01,
     *  0.88845923287225699889E 00,0.92927195315124537686E-01,
     *  0.62110294673722640294E 00,0.17151190913639138079E 00,
     *  0.22338668642896688163E 00,0.21915685840158749640E 00,
     *  0.22551049979820668739E 00,0.67207754295990703540E-01,
     *  0.25807598096176653565E-01,0.10031427861179557877E 00,
     *  0.84345657393211062463E-02,0.46462893261757986541E-01,
     *  0.85755920049990351154E-01,0.10957842105592463824E 00/
      DATA
     * P(29),P(30),P(31),P(32),P(33),P(34),P(35),
     * P(36),P(37),P(38),P(39),P(40),P(41),P(42),
     * P(43),P(44),P(45),P(46),P(47),P(48),P(49),
     * P(50),P(51),P(52),P(53),P(54),P(55),P(56)/
     *  0.99909812496766759766E 00,0.25447807915618744154E-02,
     *  0.98153114955374010687E 00,0.16446049854387810934E-01,
     *  0.92965485742974005667E 00,0.35957103307129322097E-01,
     *  0.83672593816886873550E 00,0.56979509494123357412E-01,
     *  0.70249620649152707861E 00,0.76879620499003531043E-01,
     *  0.53131974364437562397E 00,0.93627109981264473617E-01,
     *  0.33113539325797683309E 00,0.10566989358023480974E 00,
     *  0.11248894313318662575E 00,0.11195687302095345688E 00,
     *  0.11275525672076869161E 00,0.33603877148207730542E-01,
     *  0.12903800100351265626E-01,0.50157139305899537414E-01,
     *  0.42176304415588548391E-02,0.23231446639910269443E-01,
     *  0.42877960025007734493E-01,0.54789210527962865032E-01,
     *  0.12651565562300680114E-02,0.82230079572359296693E-02,
     *  0.17978551568128270333E-01,0.28489754745833548613E-01/
      DATA
     * P(57),P(58),P(59),P(60),P(61),P(62),P(63),
     * P(64),P(65),P(66),P(67),P(68),P(69),P(70),
     * P(71),P(72),P(73),P(74),P(75),P(76),P(77),
     * P(78),P(79),P(80),P(81),P(82),P(83),P(84)/
     *  0.38439810249455532039E-01,0.46813554990628012403E-01,
     *  0.52834946790116519862E-01,0.55978436510476319408E-01,
     *  0.99987288812035761194E 00,0.36322148184553065969E-03,
     *  0.99720625937222195908E 00,0.25790497946856882724E-02,
     *  0.98868475754742947994E 00,0.61155068221172463397E-02,
     *  0.97218287474858179658E 00,0.10498246909621321898E-01,
     *  0.94634285837340290515E 00,0.15406750466559497802E-01,
     *  0.91037115695700429250E 00,0.20594233915912711149E-01,
     *  0.86390793819369047715E 00,0.25869679327214746911E-01,
     *  0.80694053195021761186E 00,0.31073551111687964880E-01,
     *  0.73975604435269475868E 00,0.36064432780782572640E-01,
     *  0.66290966002478059546E 00,0.40715510116944318934E-01,
     *  0.57719571005204581484E 00,0.44914531653632197414E-01,
     *  0.48361802694584102756E 00,0.48564330406673198716E-01/
      DATA
     * P( 85),P( 86),P( 87),P( 88),P( 89),P( 90),P( 91),
     * P( 92),P( 93),P( 94),P( 95),P( 96),P( 97),P( 98),
     * P( 99),P(100),P(101),P(102),P(103),P(104),P(105),
     * P(106),P(107),P(108),P(109),P(110),P(111),P(112)/
     *  0.38335932419873034692E 00,0.51583253952048458777E-01,
     *  0.27774982202182431507E 00,0.53905499335266063927E-01,
     *  0.16823525155220746498E 00,0.55481404356559363988E-01,
     *  0.56344313046592789972E-01,0.56277699831254301273E-01,
     *  0.56377628360384717388E-01,0.16801938574103865271E-01,
     *  0.64519000501757369228E-02,0.25078569652949768707E-01,
     *  0.21088152457266328793E-02,0.11615723319955134727E-01,
     *  0.21438980012503867246E-01,0.27394605263981432516E-01,
     *  0.63260731936263354422E-03,0.41115039786546930472E-02,
     *  0.89892757840641357233E-02,0.14244877372916774306E-01,
     *  0.19219905124727766019E-01,0.23406777495314006201E-01,
     *  0.26417473395058259931E-01,0.27989218255238159704E-01,
     *  0.18073956444538835782E-03,0.12895240826104173921E-02,
     *  0.30577534101755311361E-02,0.52491234548088591251E-02/
      DATA
     * P(113),P(114),P(115),P(116),P(117),P(118),P(119),
     * P(120),P(121),P(122),P(123),P(124),P(125),P(126),
     * P(127),P(128),P(129),P(130),P(131),P(132),P(133),
     * P(134),P(135),P(136),P(137),P(138),P(139),P(140)/
     *  0.77033752332797418482E-02,0.10297116957956355524E-01,
     *  0.12934839663607373455E-01,0.15536775555843982440E-01,
     *  0.18032216390391286320E-01,0.20357755058472159467E-01,
     *  0.22457265826816098707E-01,0.24282165203336599358E-01,
     *  0.25791626976024229388E-01,0.26952749667633031963E-01,
     *  0.27740702178279681994E-01,0.28138849915627150636E-01,
     *  0.99998243035489159858E 00,0.50536095207862517625E-04,
     *  0.99959879967191068325E 00,0.37774664632698466027E-03,
     *  0.99831663531840739253E 00,0.93836984854238150079E-03,
     *  0.99572410469840718851E 00,0.16811428654214699063E-02,
     *  0.99149572117810613240E 00,0.25687649437940203731E-02,
     *  0.98537149959852037111E 00,0.35728927835172996494E-02,
     *  0.97714151463970571416E 00,0.46710503721143217474E-02,
     *  0.96663785155841656709E 00,0.58434498758356395076E-02/
      DATA
     * P(141),P(142),P(143),P(144),P(145),P(146),P(147),
     * P(148),P(149),P(150),P(151),P(152),P(153),P(154),
     * P(155),P(156),P(157),P(158),P(159),P(160),P(161),
     * P(162),P(163),P(164),P(165),P(166),P(167),P(168)/
     *  0.95373000642576113641E 00,0.70724899954335554680E-02,
     *  0.93832039777959288365E 00,0.83428387539681577056E-02,
     *  0.92034002547001242073E 00,0.96411777297025366953E-02,
     *  0.89974489977694003664E 00,0.10955733387837901648E-01,
     *  0.87651341448470526974E 00,0.12275830560082770087E-01,
     *  0.85064449476835027976E 00,0.13591571009765546790E-01,
     *  0.82215625436498040737E 00,0.14893641664815182035E-01,
     *  0.79108493379984836143E 00,0.16173218729577719942E-01,
     *  0.75748396638051363793E 00,0.17421930159464173747E-01,
     *  0.72142308537009891548E 00,0.18631848256138790186E-01,
     *  0.68298743109107922809E 00,0.19795495048097499488E-01,
     *  0.64227664250975951377E 00,0.20905851445812023852E-01,
     *  0.59940393024224289297E 00,0.21956366305317824939E-01,
     *  0.55449513263193254887E 00,0.22940964229387748761E-01/
      DATA
     * P(169),P(170),P(171),P(172),P(173),P(174),P(175),
     * P(176),P(177),P(178),P(179),P(180),P(181),P(182),
     * P(183),P(184),P(185),P(186),P(187),P(188),P(189),
     * P(190),P(191),P(192),P(193),P(194),P(195),P(196)/
     *  0.50768775753371660215E 00,0.23854052106038540080E-01,
     *  0.45913001198983233287E 00,0.24690524744487676909E-01,
     *  0.40897982122988867241E 00,0.25445769965464765813E-01,
     *  0.35740383783153215238E 00,0.26115673376706097680E-01,
     *  0.30457644155671404334E 00,0.26696622927450359906E-01,
     *  0.25067873030348317661E 00,0.27185513229624791819E-01,
     *  0.19589750271110015392E 00,0.27579749566481873035E-01,
     *  0.14042423315256017459E 00,0.27877251476613701609E-01,
     *  0.84454040083710883710E-01,0.28076455793817246607E-01,
     *  0.28184648949745694339E-01,0.28176319033016602131E-01,
     *  0.28188814180192358694E-01,0.84009692870519326354E-02,
     *  0.32259500250878684614E-02,0.12539284826474884353E-01,
     *  0.10544076228633167722E-02,0.58078616599775673635E-02,
     *  0.10719490006251933623E-01,0.13697302631990716258E-01/
      DATA
     * P(197),P(198),P(199),P(200),P(201),P(202),P(203),
     * P(204),P(205),P(206),P(207),P(208),P(209),P(210),
     * P(211),P(212),P(213),P(214),P(215),P(216),P(217),
     * P(218),P(219),P(220),P(221),P(222),P(223),P(224)/
     *  0.31630366082226447689E-03,0.20557519893273465236E-02,
     *  0.44946378920320678616E-02,0.71224386864583871532E-02,
     *  0.96099525623638830097E-02,0.11703388747657003101E-01,
     *  0.13208736697529129966E-01,0.13994609127619079852E-01,
     *  0.90372734658751149261E-04,0.64476204130572477933E-03,
     *  0.15288767050877655684E-02,0.26245617274044295626E-02,
     *  0.38516876166398709241E-02,0.51485584789781777618E-02,
     *  0.64674198318036867274E-02,0.77683877779219912200E-02,
     *  0.90161081951956431600E-02,0.10178877529236079733E-01,
     *  0.11228632913408049354E-01,0.12141082601668299679E-01,
     *  0.12895813488012114694E-01,0.13476374833816515982E-01,
     *  0.13870351089139840997E-01,0.14069424957813575318E-01,
     *  0.25157870384280661489E-04,0.18887326450650491366E-03,
     *  0.46918492424785040975E-03,0.84057143271072246365E-03/
      DATA
     * P(225),P(226),P(227),P(228),P(229),P(230),P(231),
     * P(232),P(233),P(234),P(235),P(236),P(237),P(238),
     * P(239),P(240),P(241),P(242),P(243),P(244),P(245),
     * P(246),P(247),P(248),P(249),P(250),P(251),P(252)/
     *  0.12843824718970101768E-02,0.17864463917586498247E-02,
     *  0.23355251860571608737E-02,0.29217249379178197538E-02,
     *  0.35362449977167777340E-02,0.41714193769840788528E-02,
     *  0.48205888648512683476E-02,0.54778666939189508240E-02,
     *  0.61379152800413850435E-02,0.67957855048827733948E-02,
     *  0.74468208324075910174E-02,0.80866093647888599710E-02,
     *  0.87109650797320868736E-02,0.93159241280693950932E-02,
     *  0.98977475240487497440E-02,0.10452925722906011926E-01,
     *  0.10978183152658912470E-01,0.11470482114693874380E-01,
     *  0.11927026053019270040E-01,0.12345262372243838455E-01,
     *  0.12722884982732382906E-01,0.13057836688353048840E-01,
     *  0.13348311463725179953E-01,0.13592756614812395910E-01,
     *  0.13789874783240936517E-01,0.13938625738306850804E-01,
     *  0.14038227896908623303E-01,0.14088159516508301065E-01/
      DATA
     * P(253),P(254),P(255),P(256),P(257),P(258),P(259),
     * P(260),P(261),P(262),P(263),P(264),P(265),P(266),
     * P(267),P(268),P(269),P(270),P(271),P(272),P(273),
     * P(274),P(275),P(276),P(277),P(278),P(279),P(280)/
     *  0.99999759637974846462E 00,0.69379364324108267170E-05,
     *  0.99994399620705437576E 00,0.53275293669780613125E-04,
     *  0.99976049092443204733E 00,0.13575491094922871973E-03,
     *  0.99938033802502358193E 00,0.24921240048299729402E-03,
     *  0.99874561446809511470E 00,0.38974528447328229322E-03,
     *  0.99780535449595727456E 00,0.55429531493037471492E-03,
     *  0.99651414591489027385E 00,0.74028280424450333046E-03,
     *  0.99483150280062100052E 00,0.94536151685852538246E-03,
     *  0.99272134428278861533E 00,0.11674841174299594077E-02,
     *  0.99015137040077015918E 00,0.14049079956551446427E-02,
     *  0.98709252795403406719E 00,0.16561127281544526052E-02,
     *  0.98351865757863272876E 00,0.19197129710138724125E-02,
     *  0.97940628167086268381E 00,0.21944069253638388388E-02,
     *  0.97473445975240266776E 00,0.24789582266575679307E-02/
      DATA
     * P(281),P(282),P(283),P(284),P(285),P(286),P(287),
     * P(288),P(289),P(290),P(291),P(292),P(293),P(294),
     * P(295),P(296),P(297),P(298),P(299),P(300),P(301),
     * P(302),P(303),P(304),P(305),P(306),P(307),P(308)/
     *  0.96948465950245923177E 00,0.27721957645934509940E-02,
     *  0.96364062156981213252E 00,0.30730184347025783234E-02,
     *  0.95718821610986096274E 00,0.33803979910869203823E-02,
     *  0.95011529752129487656E 00,0.36933779170256508183E-02,
     *  0.94241156519108305981E 00,0.40110687240750233989E-02,
     *  0.93406843615772578800E 00,0.43326409680929828545E-02,
     *  0.92507893290707565236E 00,0.46573172997568547773E-02,
     *  0.91543758715576504064E 00,0.49843645647655386012E-02,
     *  0.90514035881326159519E 00,0.53130866051870565663E-02,
     *  0.89418456833555902286E 00,0.56428181013844441585E-02,
     *  0.88256884024734190684E 00,0.59729195655081658049E-02,
     *  0.87029305554811390585E 00,0.63027734490857587172E-02,
     *  0.85735831088623215653E 00,0.66317812429018878941E-02,
     *  0.84376688267270860104E 00,0.69593614093904229394E-02/
      DATA
     * P(309),P(310),P(311),P(312),P(313),P(314),P(315),
     * P(316),P(317),P(318),P(319),P(320),P(321),P(322),
     * P(323),P(324),P(325),P(326),P(327),P(328),P(329),
     * P(330),P(331),P(332),P(333),P(334),P(335),P(336)/
     *  0.82952219463740140018E 00,0.72849479805538070639E-02,
     *  0.81462878765513741344E 00,0.76079896657190565832E-02,
     *  0.79909229096084140180E 00,0.79279493342948491103E-02,
     *  0.78291939411828301639E 00,0.82443037630328680306E-02,
     *  0.76611781930376009072E 00,0.85565435613076896192E-02,
     *  0.748696293616936602 82E 00,0.88641732094824942641E-02,
     *  0.73066452124218126133E 00,0.91667111635607884067E-02,
     *  0.71203315536225203459E 00,0.94636899938300652943E-02,
     *  0.69281376977911470289E 00,0.97546565363174114611E-02,
     *  0.67301883023041847920E 00,0.10039172044056840798E-01,
     *  0.65266166541001749610E 00,0.10316812330947621682E-01,
     *  0.63175643771119423041E 00,0.10587167904885197931E-01,
     *  0.61031811371518640016E 00,0.10849844089337314099E-01,
     *  0.58836243444766254143E 00,0.11104461134006926537E-01/
      DATA
     * P(337),P(338),P(339),P(340),P(341),P(342),P(343),
     * P(344),P(345),P(346),P(347),P(348),P(349),P(350),
     * P(351),P(352),P(353),P(354),P(355),P(356),P(357),
     * P(358),P(359),P(360),P(361),P(362),P(363),P(364)/
     *  0.56590588542365442262E 00,0.11350654315980596602E-01,
     *  0.54296566649831149049E 00,0.11588074033043952568E-01,
     *  0.51955966153745702199E 00,0.11816385890830235763E-01,
     *  0.49570640791876146017E 00,0.12035270785279562630E-01,
     *  0.47142506587165887693E 00,0.12244424981611985899E-01,
     *  0.44673538766202847374E 00,0.12443560190714035263E-01,
     *  0.42165768662616330006E 00,0.12632403643542078765E-01,
     *  0.39621280605761593918E 00,0.12810698163877361967E-01,
     *  0.37042208795007823014E 00,0.12978202239537399286E-01,
     *  0.34430734159943802278E 00,0.13134690091960152836E-01,
     *  0.31789081206847668318E 00,0.13279951743930530650E-01,
     *  0.29119514851824668196E 00,0.13413793085110098513E-01,
     *  0.26424337241092676194E 00,0.13536035934956213614E-01,
     *  0.23705884558982972721E 00,0.13646518102571291428E-01/
      DATA
     * P(365),P(366),P(367),P(368),P(369),P(370),P(371),
     * P(372),P(373),P(374),P(375),P(376),P(377),P(378),
     * P(379),P(380),P(381)/
     *  0.20966523824318119477E 00,0.13745093443001896632E-01,
     *  0.18208649675925219825E 00,0.13831631909506428676E-01,
     *  0.15434681148137810869E 00,0.13906019601325461264E-01,
     *  0.12647058437230196685E 00,0.13968158806516938516E-01,
     *  0.98482396598119202090E-01,0.14017968039456608810E-01,
     *  0.70406976042855179063E-01,0.14055382072649964277E-01,
     *  0.42269164765363603212E-01,0.14080351962553661325E-01,
     *  0.14093886410782462614E-01,0.14092845069160408355E-01,
     *  0.14094407090096179347E-01/
      ICHECK = 0
C CHECK FØR TRIVIAL CASE.
      IF (A.EQ.B) GØ TØ 70
C SCALE FACTØRS.
      SUM = (B+A)/2.0
      DIFF = (B-A)/2.0
C 1-PØINT GAUSS
      FZERØ = F(SUM)
      RESULT(1) = 2.0*FZERØ*DIFF
      I = 0
      IØLD = 0
      INEW = 1
      K = 2
      ACUM = 0.0
      GØ TØ 30
   10 IF (K.EQ.8) GØ TØ 50
      K = K + 1
      ACUM = 0.0
C CØNTRIBUTIØN FRØM FUNCTIØN VALUES ALREADY CØMPUTED.
      DØ 20 J=1,IØLD
        I = I + 1
        ACUM = ACUM + P(I)*FUNCT(J)
   20 CØNTINUE
C CØNTRIBUTIØN FRØM NEW FUNCTIØN VALUES.
   30 IØLD = IØLD + INEW
      DØ 40 J=INEW,IØLD
        I = I + 1
        X = P(I)*DIFF
        FUNCT(J) = F(SUM+X) + F(SUM-X)
        I = I + 1
        ACUM = ACUM + P(I)*FUNCT(J)
   40 CØNTINUE
      INEW = IØLD + 1
      I = I + 1
      RESULT(K) = (ACUM+P(I)*FZERØ)*DIFF
C CHECK FØR CØNVERGENCE.
      IF (ABS(RESULT(K)-RESULT(K-1))-EPSIL*ABS(RESULT(K))) 60,
     * 60, 10
C CØNVERGENCE NØT ACHIEVED.
   50 ICHECK = 1
C NØRMAL TERMINATIØN.
   60 NPTS = INEW + IØLD
      RETURN
C TRIVIAL CASE
   70 K = 2
      RESULT(1) = 0.0
      RESULT(2) = 0.0
      NPTS = 0
      RETURN
      END

      FUNCTIØN QSUB(A, B, EPSIL, NPTS, ICHECK, RELERR, F)
C   THIS FUNCTIØN RØUTINE PERFØRMS AUTØMATIC INTEGRATIØN
C ØVER A FINITE INTERVAL USING THE BASIC INTEGRATIØN
C ALGØRITHM QUAD, TØGETHER WITH, IF NECESSARY, A NØN-
C ADAPTIVE SUBDIVISIØN PRØCESS.
C   THE CALL TAKES THE FØRM
C            QSUB(A,B,EPSIL,NPTS,ICHECK,RELERR,F)
C AND CAUSES F(X) TØ BE INTEGRATED ØVER (A,B) WITH RELATIVE
C ERRØR HØPEFULLY NØT EXCEEDING EPSIL.  SHØULD QUAD CØNVERGE
C (ICHECK=0) THEN QSUB WILL RETURN THE VALUE ØBTAINED BY IT
C ØTHERWISE SUBDIVISIØN WILL BE INVØKED AS A RESCUE
C ØPERATIØN IN A NØN-ADAPTIVE MANNER. THE ARGUMENT RELERR
C GIVES A CRUDE ESTIMATE ØF THE ACTUAL RELATIVE ERRØR
C ØBTAINED.
C   THE SUBDIVISIØN STRATEGY IS AS FØLLØWS
C LET THE INTERVAL (A,B) BE DIVIDED INTØ 2**N PANELS AT STEP
C N ØF THE SUBDIVISIØN PRØCESS.  QUAD IS APPLIED FIRST TØ
C THE SUBDIVIDED INTERVAL ØN WHICH QUAD LAST FAILED TØ
C CØNVERGE AND IF CØNVERGENCE IS NØW ACHIEVED THE REMAINING
C PANELS ARE INTEGRATED.  SHØULD A CØNVERGENCE FAILURE ØCCUR
C ØN ANY PANEL THE INTEGRATIØN AT THAT PØINT IS TERMINATED
C AND THE PRØCEDURE REPEATED WITH N INCREASED BY 1. THE
C STRATEGY INSURES THAT PØSSIBLY DELINQUENT INTERVALS ARE
C EXAMINED BEFØRE WØRK, WHICH LATER MIGHT HAVE TØ BE
C DISCARDED, IS INVESTED ØN WELL BEHAVED PANELS. THE
C PRØCESS IS CØMPLETE WHEN NØ CØNVERGENCE FAILURE ØCCURS ØN
C ANY PANEL AND THE SUM ØF THE RESULTS ØBTAINED BY QUAD ØN
C EACH PANEL IS TAKEN AS THE VALUE ØF THE INTEGRAL.
C   THE PRØCESS IS VERY CAUTIØUS IN THAT THE SUBDIVISIØN ØF
C THE INTERVAL (A,B) IS UNIFØRM, THE FINENESS ØF WHICH IS
C CØNTRØLLED BY THE SUCCESS ØF QUAD.  IN THIS WAY IT IS
C RATHER DIFFICULT FØR A SPURIØUS CØNVERGENCE TØ SLIP
C THRØUGH.
C   THE CØNVERGENCE CRITERIØN ØF QUAD IS SLIGHTLY RELAXED
C IN THAT A PANEL IS DEEMED TØ HAVE BEEN SUCCESSFULLY
C INTEGRATED IF EITHER QUAD CØNVERGES ØR THE ESTIMATED
C ABSØLUTE ERRØR CØMMITTED ØN THIS PANEL DØES NØT EXCEED
C EPSIL TIMES THE ESTIMATED ABSØLUTE VALUE ØF THE INTEGRAL
C ØVER (A,B).  THIS RELAXATIØN IS TØ TRY TØ TAKE ACCØUNT ØF
C A CØMMØN SITUATIØN WHERE ØNE PARTICULAR PANEL CAUSES
C SPECIAL DIFFICULTY, PERHAPS DUE TØ A SINGULARITY ØF SØME
C TYPE.  IN THIS CASE QUAD CØULD ØBTAIN NEARLY EXACT
C ANSWERS ØN ALL ØTHER PANELS AND SØ THE RELATIVE ERRØR FØR
C THE TØTAL INTEGRATIØN WØULD BE ALMØST ENTIRELY DUE TØ THE
C DELINQUENT PANEL.  WITHØUT THIS CØNDITIØN THE CØMPUTATIØN
C MIGHT CØNTINUE DESPITE THE REQUESTED RELATIVE ERRØR BEING
C ACHIEVED.
C   THE ØUTCØME ØF THE INTEGRATIØN IS INDICATED BY ICHECK.
C   ICHECK=0  -  CØNVERGENCE ØBTAINED WITHØUT INVØKING
C                SUBDIVISIØN.  THIS CØRRESPØNDS TØ THE
C                DIRECT USE ØF QUAD.
C   ICHECK=1  -  RESULT ØBTAINED AFTER INVØKING SUBDIVISIØN.
C   ICHECK=2  -  AS FØR ICHECK=1 BUT AT SØME PØINT THE
C                RELAXED CØNVERGENCE CRITERIØN WAS USED.
C                THE RISK ØF UNDERESTIMATING THE RELATIVE
C                ERRØR WILL BE INCREASED.  IF NECESSARY,
C                CØNFIDENCE MAY BE RESTØRED BY CHECKING
C                EPSIL AND RELERR FØR A SERIØUS DISCREPANCY.
C   ICHECK NEGATIVE
C                IF DURING THE SUBDIVISIØN PRØCESS THE
C                ALLØWED UPPER LIMIT ØN THE NUMBER ØF PANELS
C                THAT MAY BE GENERATED (PRESENTLY 4096) IS
C                REACHED A RESULT IS ØBTAINED WHICH MAY BE
C                UNRELIABLE BY CØNTINUING THE INTEGRATIØN
C                WITHØUT FURTHER SUBDIVISIØN IGNØRING
C                CØNVERGENCE FAILURES. THIS ØCCURRENCE IS
C                FLAGGED BY RETURNING ICHECK WITH NEGATIVE
C                SIGN.
C  THE RELIABILITY ØF THE ALGØRITHM WILL DECREASE FØR LARGE
C  VALUES ØF EPSIL.  IT IS RECØMMENDED THAT EPSIL SHØULD
C  GENERALLY BE LESS THAN ABØUT 0.001.
      DIMENSIØN RESULT(8)
      INTEGER BAD, ØUT
      LØGICAL RHS
      EXTERNAL F
      DATA NMAX/4096/
      CALL QUAD(A, B, RESULT, K, EPSIL, NPTS, ICHECK, F)
      QSUB = RESULT(K)
      RELERR = 0.0
      IF (QSUB.NE.0.0) RELERR =
     * ABS((RESULT(K)-RESULT(K-1))/QSUB)
C CHECK IF SUBDIVISIØN IS NEEDED.
      IF (ICHECK.EQ.0) RETURN
C SUBDIVIDE
      ESTIM = ABS(QSUB*EPSIL)
      IC = 1
      RHS = .FALSE.
      N = 1
      H = B - A
      BAD = 1
   10 QSUB = 0.0
      RELERR = 0.0
      H = H*0.5
      N = N + N
C INTERVAL (A,B) DIVIDED INTØ N EQUAL SUBINTERVALS.
C INTEGRATE ØVER SUBINTERVALS BAD TØ (BAD+1) WHERE TRØUBLE
C HAS ØCCURRED.
      M1 = BAD
      M2 = BAD + 1
      ØUT = 1
      GØ TØ 50
C INTEGRATE ØVER SUBINTERVALS 1 TØ (BAD-1)
   20 M1 = 1
      M2 = BAD - 1
      RHS = .FALSE.
      ØUT = 2
      GØ TØ 50
C INTEGRATE ØVER SUBINTERVALS (BAD+2) TØ N.
   30 M1 = BAD + 2
      M2 = N
      ØUT = 3
      GØ TØ 50
C SUBDIVISIØN RESULT
   40 ICHECK = IC
      RELERR = RELERR/ABS(QSUB)
      RETURN
C INTEGRATE ØVER SUBINTERVALS M1 TØ M2.
   50 IF (M1.GT.M2) GØ TØ 90
      DØ 80 JJ=M1,M2
        J = JJ
```

Algorithm 469
Arithmetic Over a Finite Field [A1]

C. Lam* and J. McKay† [Recd. 8 Oct. 1971]
* Department of Mathematics, Caltech University, Pasadena, CA 91101 † School of Computer Science, McGill University, P.O. Box 6070, Montreal 101, P.Q. Canada

Key Words and Phrases: algebra; CR Categories: 5.19
Language: Algol

Description

The rational operations of arithmetic over the finite field F_q, of $q = p^n (n \geq 1)$ elements, may be performed with this algorithm.

On entry $a[i]$ contains $a_i \in F_p$ with $0 \leq a_i < p$, $i = 0, \ldots, n-1$, and $x \in F_q$ satisfies the primitive irreducible polynomial $P(x) = x^n + \sum_{k=0}^{n-1} a_k x^k$. fq produces e_i in $e[i]$, $i = -1, \ldots, q-2$, where $1 + x^i = x^{e_i}$ with the convention that -1 represents $*$ and $x* = 0$. During execution the range of the a_i is altered to $-p < a_i \leq 0$, $i = 0 \ldots n-1$. The storage used is $2q + n + 6$ locations including the final array e.

With appropriate conventions for $*$, multiplication and division are trivial, and addition and subtraction are given by $x^a + x^b = x^a(1 + x^{b-a})$ for $a \leq b$ and $x^a - x^b = x^a + x^{\frac{1}{2}(q-1)}\ x^b$ when $p \neq 2$. For small values of q, it is suggested that addition and multiplication tables be generated by this algorithm. A description of the method and its generalization to a multi-step process when n is composite is in [2]. A list of primitive irreducible polynomials is given in [1]. Further useful information (especially for $p = 2$) is to be found in [3].

References

1. Alanen, A.J., and Knuth, D.E. Tables of finite fields. *Sankhyā-(A) 26* (1964), 305-328.
2. Cannon, J.J. Ph.D. Th., 1967 U. of Sydney, Sydney, Australia.
3. Conway, J.H., and Guy, M.J.T. Information on finite fields. In *Computers in Mathematical Research.* North-Holland Pub. Co., Amsterdam, 1967.

Algorithm

```
procedure fq(p, n, a, e);
  integer p, n; integer array a, e
begin
  integer array c[0:n−1], f[0:p ↑ n−1]; integer q, i, j, d, s, w;
    q := p ↑ n;
  for i := 0 step 1 until n − 1 do if a[i] ≠ 0 then a[i] := a[i] − p;
  for i := 1 step 1 until n − 1 do c[i] := 0;
    c[0] := 1;  f[1] := 0;  f[0] := −1:
  for i := 1 step 1 until q − 2 do
    begin
    d := e[n − 1];  s := 0;
    for j := n − 1 step −1 until 1 do
      begin
      w := c[i−1] − d × a[j];  w := w − w ÷ p × p;
      c[j] := w;  s := p × s + w
      end;
    w := −d × a[0];  w := w − w ÷ p × p;  c[0] := w;
      f[p × s + w] := i
    end;
  for i := q step −p until p do
    begin
    e[f[i−1]] := f[i−p];
  for j := i − p step 1 until i − 2 do e[f[j]] := f[j+1]
    end
  end
```

```
C EXAMINE FIRST THE LEFT ØR RIGHT HALF ØF THE SUBDIVIDED
C TRØUBLESØME INTERVAL DEPENDING ØN THE ØBSERVED TREND.
        IF (RHS) J = M2 + M1 - JJ
        ALPHA = A + H*(J-1)
        BETA = ALPHA + H
        CALL QUAD(ALPHA, BETA, RESULT, M, EPSIL, NF, ICHECK, F)
        CØMP = ABS(RESULT(M)-RESULT(M-1))
        NPTS = NPTS + NF
        IF (ICHECK.NE.1) GØ TØ 70
        IF (CØMP.LE.ESTIM) GØ TØ 100
C SUBINTERVAL J HAS CAUSED TRØUBLE.
C CHECK IF FURTHER SUBDIVISIØN SHØULD BE CARRIED ØUT.
        IF (N.EQ.NMAX) GØ TØ 60
        BAD = 2*J - 1
        RHS = .FALSE.
        IF ((J-2*(J/2)).EQ.0) RHS = .TRUE.
        GØ TØ 10
   60   IC = -IABS(IC)
   70   QSUB = QSUB + RESULT(M)
   80 CØNTINUE
      RELERR = RELERR + CØMP
   90 GØ TØ (20,30,40), ØUT
C RELAXED CØNVERGENCE
  100 IC = ISIGN(2,IC)
      GØ TØ 70
      END

      FUNCTIØN QSUBA(A, B, EPSIL, NPTS, ICHECK, RELERR, F)
C   THIS FUNCTIØN RØUTINE PERFØRMS AUTØMATIC INTEGRATIØN
C ØVER A FINITE INTERVAL USING THE BASIC INTEGRATIØN
C ALGØRITHM QUAD TØGETHER WITH, IF NECESSARY AN ADAPTIVE
C SUBDIVISIØN PRØCESS.  IT IS GENERALLY MØRE EFFICIENT THAN
C THE NØN-ADAPTIVE ALGØRITHM QSUB BUT IS LIKELY TØ BE LESS
C RELIABLE(SEE CØMP.J.,14,189,1971).
C   THE CALL TAKES THE FØRM
C           QSUBA(A,B,EPSIL,NPTS,ICHECK,RELERR,F)
C AND CAUSES F(X) TØ BE INTEGRATED ØVER (A,B) WITH RELATIVE
C ERRØR HØPEFULLY NØT EXCEEDING EPSIL.  SHØULD QUAD CØNVERGE
C (ICHECK=0) THEN QSUBA WILL RETURN THE VALUE ØBTAINED BY IT
C ØTHERWISE SUBDIVISIØN WILL BE INVØKED AS A RESCUE
C ØPERATIØN IN AN ADAPTIVE MANNER. THE ARGUMENT RELERR GIVES
C A CRUDE ESTIMATE ØF THE ACTUAL RELATIVE ERRØR ØBTAINED.
C   THE SUBDIVISIØN STRATEGY IS AS FØLLØWS
C AT EACH STAGE ØF THE PRØCESS AN INTERVAL IS PRESENTED FØR
C SUBDIVISIØN (INITIALLY THIS WILL BE THE WHØLE INTERVAL
C (A,B)).  THE INTERVAL IS HALVED AND QUAD APPLIED TØ EACH
C SUBINTERVAL.  SHØULD QUAD FAIL ØN THE FIRST SUBINTERVAL
C THE SUBINTERVAL IS STACKED FØR FUTURE SUBDIVISIØN AND THE
C SECØND SUBINTERVAL IMMEDIATELY EXAMINED.  SHØULD QUAD FAIL
C ØN THE SECØND SUBINTERVAL THE SUBINTERVAL IS
C IMMEDIATELY SUBDIVIDED AND THE WHØLE PRØCESS REPEATED.
C EACH TIME A CØNVERGED RESULT IS ØBTAINED IT IS
C ACCUMULATED AS THE PARTIAL VALUE ØF THE INTEGRAL. WHEN
C QUAD CØNVERGES ØN BØTH SUBINTERVALS THE INTERVAL LAST
C STACKED IS CHØSEN NEXT FØR SUBDIVISIØN AND THE PRØCESS
C REPEATED.  A SUBINTERVAL IS NØT EXAMINED AGAIN ØNCE A
C CØNVERGED RESULT IS ØBTAINED FØR IT SØ THAT A SPURIØUS
C CØNVERGENCE IS MØRE LIKELY TØ SLIP THRØUGH THAN FØR THE
C NØN-ADAPTIVE ALGØRITHM QSUB.
C   THE CØNVERGENCE CRITERIØN ØF QUAD IS SLIGHTLY RELAXED
C IN THAT A PANEL IS DEEMED TØ HAVE BEEN SUCCESSFULLY
C INTEGRATED IF EITHER QUAD CØNVERGES ØR THE ESTIMATED
C ABSØLUTE ERRØR CØMMITTED ØN THIS PANEL DØES NØT EXCEED
C EPSIL TIMES THE ESTIMATED ABSØLUTE VALUE ØF THE INTEGRAL
C ØVER (A,B).  THIS RELAXATIØN IS TØ TRY TØ TAKE ACCØUNT ØF
C A CØMMØN SITUATIØN WHERE ØNE PARTICULAR PANEL CAUSES
C SPECIAL DIFFICULTY, PERHAPS DUE TØ A SINGULARITY ØF SØME
C TYPE.  IN THIS CASE QUAD CØULD ØBTAIN NEARLY EXACT
C ANSWERS ØN ALL ØTHER PANELS AND SØ THE RELATIVE ERRØR FØR
C THE TØTAL INTEGRATIØN WØULD BE ALMØST ENTIRELY DUE TØ THE
C DELINQUENT PANEL.  WITHØUT THIS CØNDITIØN THE CØMPUTATIØN
C MIGHT CØNTINUE DESPITE THE REQUESTED RELATIVE ERRØR BEING
C ACHIEVED.
C   THE ØUTCØME ØF THE INTEGRATIØN IS INDICATED BY ICHECK.
C   ICHECK=0  -  CØNVERGENCE ØBTAINED WITHØUT INVØKING SUB-
C                DIVISIØN.  THIS WØULD CØRRESPØND TØ THE
C                DIRECT USE ØF QUAD.
C   ICHECK=1  -  RESULT ØBTAINED AFTER INVØKING SUBDIVISIØN.
C   ICHECK=2  -  AS FØR ICHECK81 BUT AT SØME PØINT THE
C                RELAXED CØNVERGENCE CRITERIØN WAS USED.
C                THE RISK ØF UNDERESTIMATING THE RELATIVE
C                ERRØR WILL BE INCREASED.  IF NECESSARY,
C                CØNFIDENCE MAY BE RESTØRED BY CHECKING
C                EPSIL AND RELERR FØR A SERIØUS DISCREPANCY.
C   ICHECK NEGATIVE
C                IF DURING THE SUBDIVISIØN PRØCESS THE STACK
C                ØF DELINQUENT INTERVALS BECØMES FULL (IT IS
C                PRESENTLY SET TØ HØLD AT MØST 100 NUMBERS)
C                A RESULT IS ØBTAINED BY CØNTINUING THE
C                INTEGRATIØN IGNØRING CØNVERGENCE FAILURES
C                WHICH CANNØT BE ACCØMMØDATED ØN THE STACK.
C                THIS ØCCURRENCE IS FLAGGED BY RETURNING
C                ICHECK WITH NEGATIVE SIGN.
C  THE RELIABILITY ØF THE ALGØRITHM WILL DECREASE FØR LARGE
C  VALUES ØF EPSIL.  IT IS RECØMMENDED THAT EPSIL SHØULD
C  GENERALLY BE LESS THAN ABØUT 0.001.
      DIMENSIØN RESULT(8), STACK(100)
      EXTERNAL F
      DATA ISMAX/100/
      CALL QUAD(A, B, RESULT, K, EPSIL, NPTS, ICHECK, F)
      QSUBA = RESULT(K)
      RELERR = 0.0
      IF (QSUBA.NE.0.0)
     * RELERR = ABS((RESULT(K)-RESULT(K-1))/QSUBA)
C CHECK IF SUBDIVISIØN IS NEEDED
      IF (ICHECK.EQ.0) RETURN
C SUBDIVIDE
      ESTIM = ABS(QSUBA*EPSIL)
      RELERR = 0.0
      QSUBA = 0.0
      IS = 1
      IC = 1
      SUB1 = A
      SUB3 = B
   10 SUB2 = (SUB1+SUB3)*0.5
      CALL QUAD(SUB1, SUB2, RESULT, K, EPSIL, NF, ICHECK, F)
      NPTS = NPTS + NF
      CØMP = ABS(RESULT(K)-RESULT(K-1))
      IF (ICHECK.EQ.0) GØ TØ 30
      IF (CØMP.LE.ESTIM) GØ TØ 70
      IF (IS.GE.ISMAX) GØ TØ 20
C STACK SUBINTERVAL (SUB1,SUB2) FØR FUTURE EXAMINATIØN
      STACK(IS) = SUB1
      IS = IS + 1
      STACK(IS) = SUB2
      IS = IS + 1
      GØ TØ 40
   20 IC = -IABS(IC)
   30 QSUBA = QSUBA + RESULT(K)
      RELERR = RELERR + CØMP
   40 CALL QUAD(SUB2, SUB3, RESULT, K, EPSIL, NF, ICHECK, F)
      NPTS = NPTS + NF
      CØMP = ABS(RESULT(K)-RESULT(K-1))
      IF (ICHECK.EQ.0) GØ TØ 50
      IF (CØMP.LE.ESTIM) GØ TØ 80
C SUBDIVIDE INTERVAL (SUB2,SUB3)
      SUB1 = SUB2
      GØ TØ 10
   50 QSUBA = QSUBA + RESULT(K)
      RELERR = RELERR + CØMP
      IF (IS.EQ.1) GØ TØ 60
C SUBDIVIDE THE DELINQUENT INTERVAL LAST STACKED
      IS = IS - 1
      SUB3 = STACK(IS)
      IS = IS - 1
      SUB1 = STACK(IS)
      GØ TØ 10
C SUBDIVISIØN RESULT
   60 ICHECK = IC
      RELERR = RELERR/ABS(QSUBA)
      RETURN
C RELAXED CØNVERGENCE
   70 IC = ISIGN(2,IC)
      GØ TØ 30
   80 IC = ISIGN(2,IC)
      GØ TØ 50
      END
```

Algorithm 470

Linear Systems with Almost Tridiagonal Matrix [F4]

Milan Kubicek [Recd. 6 Dec. 1971, 8 May 1972, 12 Oct. 1972, 12 Dec. 1972]
Department of Chemical Engineering, Technical University, Technicka' 1905, Praha 6, Dejvice, Czechoslovakia

Key Words and Phrases: system of linear equations, almost tridiagonal matrix, sparse matrix
CR Categories: 5 14
Language: Fortran

Description

The program *FAKUB* is based on the method of modified matrices. In fact, *FAKUB* solves $\tilde{T}x = b$ where $\tilde{T} = T + R$, T is tridiagonal ($n \times n$) and R is a matrix of low rank. Let us write $R = R_1 R_2^T$ where R_1, R_2 are $n \times m$ matrices. R_1 contains columns $j_1, j_2, \dots, j_m$ of $\tilde{T} - T$, and R_2 is matrix of unit vectors $e_{j_1}, e_{j_2}, \dots, e_{j_m}$.

Subroutine *FAKUB* performs the following steps:

Step 1. Determine n by m matrix V and vector y satisfying $TV = R_1$ and $Ty = b$. (The Thomas algorithm [1] is used to split $T = LU$ and V and y are obtained by back solving $m + 1$ times. This algorithm is in principle the standard LU factorization of a tridiagonal matrix, see e.g. [2]. Note that we normalize L, while in [2] U is normalized.)
Step 2. Form m by m matrix $A = I + R_2^T V$ and vector $w = R_2^T y$.
Step 3. Solve $Az = w$ for z.
Step 4. Calculate the solution $x = y - Vz$.

The method described here will be particularly useful if $m \ll n$, however, it can be used advantageously also if $m < n$.

Let us now define the matrix B, $n \times (m+1)$, in the following way: (1) the first column of the matrix B is the vector b; (2) $(k+1)$-st column of the matrix B is equal to the kth column of the matrix R_1, i.e. to the j_kth column of the matrix $\tilde{T} - T$. This holds for $k = 1, 2, \dots, m$.

The description of the formal parameters of the subroutine *FAKUB* is given in the comments at the beginning. In accordance with the symbols used above we have

$N \sim n$,	$M \sim m + 1$,
$S(I) \sim t_{i,i-1}$,	$i = 2, 3, \dots, n$
$D(I) \sim t_{i,i}$,	$i = 1, 2, \dots, n$
$H(I) \sim t_{i,i+1}$,	$i = 1, 2, \dots, n - 1$
$B(I, J) \sim b_{i,j}$,	$JPROM\,(K) \sim j_k$,

where $T = \{t_{i,j}\}$ and $B = \{b_{i,j}\}$.

Two parameters deserve to be discussed in detail. The parameter *EPS* tests zero on the diagonal in the course of the Thomas algorithm. If $|\, D(I)\,| < EPS$, then the value of *ALFA* is added to $D(I)$ and the RHS of B is modified so that the solution of the system remains the same; at the same time the statement in the form

FAKUB INFORMATION ON ZERO ON LINE I

is printed. During this modification the matrix B can be expanded in one column, which has to be considered when declaring *MM*. If during the modification the space assigned for array B is exceeded, the statement

FAKUB MANY REARRANGEMENTS, END OF FAKUB

is printed, and after return the value of M is equal to -1. For practical problems this occurrence is a very rare event. The dimension specifications $A(20, 20)$, $PS(20)$ can be changed if 20 is low; however, we must have $M \leq 20$, and M can always increase by one during the above mentioned modifications. If the dimension specification was low (see statement number 49) the statement

FAKUB LOW DIMENSION, END OF FAKUB

is printed, and after return $M = -2$. This can be corrected, e.g. by increasing the parameter *ALFA*.

If the matrix $\tilde{T}$ is singular (see the comment under statement label 5 in subroutine *GAUSD*, which has to be modified specifically with respect to the type of computer) the statement

FAKUB SINGULAR MATRIX OF SYSTEM, END OF FAKUB

is printed, and after return $M = 0$.

After regular return ($M > 0$), the results are in the first column of the matrix B.

If $m = 0$, the given algorithm is equivalent to the Thomas algorithm with the exception that it insures against zeros occurring on the diagonal. Subroutine *GAUSD* plays the role of a standard linear equation solver. Any other standard routine can be used, e.g. see [2].

The program was successfully run for calculations of distillation columns ($n = 100$, $m = 3$). It can also be applied in linear multipoint boundary value problems.

Acknowledgment. The author would like to thank to Dr. Fred Gustavson of IBM Thomas J. Watson Research Center for his very valuable comment.

References

1. Thomas, L. H. Dept. of Watson Scientific computing Laboratory, New York, 1949.
2. Forsythe, G. E., and Moler, C. B. *Computer Solution of Linear Algebraic Systems*; Prentice Hall, Englewood Cliffs, N.J., pp. 115 and 68.

Algorithm

```
      SUBRØUTINE FAKUB(N, S, D, H, B, M, NN, MM, JPRØM, ALFA, EPS)
      DIMENSIØN S(N), D(N), H(N), B(NN,MM), JPRØM(20), A(20,20),
     * W(20)
C SØLUTIØN ØF SYSTEM ØF LINEAR EQUATIØNS WITH MATRIX ØF SPECIAL
C (ALMØST TRIDIAGØNAL) TYPE
C N=NUMBER ØF EQUATIØNS
C S(2),S(3),...=LØWER DIAGØNAL ELEMENTS
C D(1),D(2),...=MAIN DIAGØNAL ELEMENTS
C H(1),H(2),...=UPPER DIAGØNAL ELEMENTS
C B(1,1),B(2,1),...=RIGHT HAND SIDES
C JPRØM(1),JPRØM(2),....,JPRØM(M-1)=INDICES ØF UNKNØWNS FØR WHICH
C NØN-ZERØ NØNDIAGØNAL CØEFFICIENTS EXIST
C (B(I,J+1),I=1,N)=CØLUMN ØF CØEFFICIENTS
C (WITHØUT DIAGØNAL ELEMENTS), WHICH CØRRESPØNDS
C TØ UNKØWN WITH INDEX JPRØM(J)
C M-1=NUMBER ØF TRANSFERRED UNKNØWNS
C ALFA=NØN ZERØ PARAMETER USED FØR REARRANGEMENTS
C EPS=SCALE ØF ZERØ DIAGØNAL ELEMENT,DEPENDENT ØN THE CØMPUTER
C  TYPE
C M.EQ.0 AFTER RETURN: MATRIX WAS SINGULAR
C M.EQ.-1 AFTER RETURN: MANY REARRANGEMENTS,SMALL VALUE ØF MM
C M.EQ.-2 AFTER RETURN: LØW DIMENSIØN SPECIFICATIØN IN FAKUB
C WE WISH TØ SØLVE G*X=C WHERE G IS A N BY N MATRIX AND C IS A
C VECTØR.
C G = T + R. R = R1*R2T. R1 AND R2 ARE N BY M1 MATRICES ØF RANK
C  M1.
C (R2T---R2 TRANSPØSE) THE METHØD ØF MØDIFIED MATRICES IS USED.
```

```
C T IS A TRIDIAGØNAL MATRIX GIVEN BY INPUT VECTØRS S, D AND H.
C B = (C,R1) IS A N BY M MATRIX. R1 IS A SET ØF M1 CØLUMNS ØF G -
C  T.
C R2 IS A SET ØF M1 UNIT VECTØRS SPECIFIED BY JPRØM.
C FØR EFFICIENCY RANK M1 IS MUCH LESS THAN N.
C KPR IS PRINTER DEVICE NUMBER
      DATA KPR/6/
99999 FØRMAT(//45H FAKUB SINGULAR MATRIX ØF SYSTEM,END ØF FAKUB//)
99998 FØRMAT(//34H FAKUB INFØRMATIØN ØN ZERØ ØN LINE,I5//)
99997 FØRMAT(//39H FAKUB MANY REARRANGEMENTS,END ØF FAKUB//)
99996 FØRMAT(//33H FAKUB LØW DIMENSIØN,END ØF FAKUB//)
      N1 = N - 1
      M1 = M - 1
      JUMP = 1
C FØRM L,U AND L**(-1)*B, NØTE L*U = T.
      I = 1
   10 P = D(I)
      IF (ABS(P).LE.EPS) GØ TØ 40
   20 H(I) = H(I)/P
      P1 = S(I+1)
      DØ 30 J=1,M
        IF (B(I,J).EQ.0.0) GØ TØ 30
        B(I,J) = B(I,J)/P
        B(I+1,J) = B(I+1,J) - P1*B(I,J)
   30 CØNTINUE
      D(I+1) = D(I+1) - P1*H(I)
      I = I + 1
      IF (I.LE.N1) GØ TØ 10
C MATRICES L,U AND L**(-1)*B ARE DETERMINED HERE
      GØ TØ 100
   40 WRITE (KPR,99998) I
C PIVØT D(I) NEARLY ZERØ. ADJUST MATRICES T AND R1 SØ THAT
C G REMAINS EQUAL TØ T + R. NEW T HAS PIVØT D(I) NEAR TØ ALFA.
      IF (M1.EQ.0) GØ TØ 60
      DØ 50 J=1,M1
        IF (JPRØM(J).EQ.I) GØ TØ 80
   50 CØNTINUE
   60 M = M + 1
      M1 = M1 + 1
      IF (M.GT.MM) GØ TØ 200
      DØ 70 J=1,N
        B(J,M) = 0.0
   70 CØNTINUE
      B(I,M) = -ALFA
      JPRØM(M1) = I
      GØ TØ 90
   80 B(I,J+1) = B(I,J+1) - ALFA
   90 D(I) = D(I) + ALFA
      P = D(I)
      GØ TØ (20,110), JUMP
  100 IF (ABS(D(N)).GT.EPS) GØ TØ 110
      I = N
      JUMP = 2
      GØ TØ 40
  110 DØ 120 J=1,M
        B(N,J) = B(N,J)/D(N)
  120 CØNTINUE
C FØRM U**(-1)*L**(-1)*B = T**(-1)*B. T**(-1)*B = (Y,V)
      DØ 140 I1=1,N1
        I = N - I1
        DØ 130 J=1,M
          B(I,J) = B(I,J) - H(I)*B(I+1,J)
  130   CØNTINUE
  140 CØNTINUE
      IF (M1.EQ.0) RETURN
C THE NEXT STATEMENT NECESSARY AS A AND W HAVE DIMENSIØN ØF 20.
      IF (M1.GT.20) GØ TØ 210
C FØRM M1 BY M1 MATRIX A = I + R2T*V AND M1 VECTØR W = R2T*Y.
      DØ 160 I=1,M1
        I1 = JPRØM(I)
        DØ 150 J=1,M1
          A(I,J) = B(I1,J+1)
  150   CØNTINUE
        W(I) = B(I1,1)
        A(I,I) = A(I,I) + 1.0
  160 CØNTINUE
C SØLVE A*Z = W FØR Z USING SUBRØUTINE GAUSD.
      CALL GAUSD(M1, A, W, M2, 20)
      IF (M2.EQ.0) GØ TØ 190
C FØRM SØLUTIØN VECTØR X = Y - V*Z.
      DØ 180 I=1,N
        DØ 170 J=2,M
          B(I,1) = B(I,1) - B(I,J)*W(J-1)
  170   CØNTINUE
  180 CØNTINUE
      RETURN
  190 WRITE (KPR,99999)
      M = 0
      RETURN
  200 WRITE (KPR,99997)
      M = -1
      RETURN
  210 WRITE (KPR,99996)
      M = -2
      RETURN
      END

      SUBRØUTINE GAUSD(N, A, B, M, NN)
      DIMENSIØN A(NN,NN), B(NN), IRR(20), X(20)
C SØLUTIØN ØF SYSTEM ØF LINEAR EQUATIØNS
C N=NUMBER ØF EQUATIØNS (N.LE.20)
C A=MATRIX ØF SYSTEM
C B=RIGHT HAND SIDES
C M=IF M.EQ.0 AFTER RETURN,THEN MATRIX A WAS SINGULAR
      M = 1
      ID = 1
      DØ 10 I=1,N
        IRR(I) = 0
   10 CØNTINUE
```

```
   20 IR = 1
      IS = 1
      AMAX = 0.0
      DØ 60 I=1,N
        IF (IRR(I)) 60, 30, 60
   30   DØ 50 J=1,N
          P = ABS(A(I,J))
          IF (P-AMAX) 50, 50, 40
   40     IR = I
          IS = J
          AMAX = P
   50   CØNTINUE
   60 CØNTINUE
      IF (AMAX.NE.0.0) GØ TØ 70
C THIS CØNDITIØN MUST BE SPECIFIED MØRE EXACTLY
C WITH RESPECT TØ CØMPUTER ACTUALLY USED
      M = 0
      GØ TØ 120
   70 IRR(IR) = IS
      DØ 90 I=1,N
        IF (I.EQ.IR .ØR. A(I,IS).EQ.0.0) GØ TØ 90
        P = A(I,IS)/A(IR,IS)
        DØ 80 J=1,N
          IF (A(IR,J).NE.0.0) A(I,J) = A(I,J) - P*A(IR,J)
   80   CØNTINUE
        A(I,IS) = 0.0
        B(I) = B(I) - P*B(IR)
   90 CØNTINUE
      ID = ID + 1
      IF (ID.LE.N) GØ TØ 20
      DØ 100 I=1,N
        IR = IRR(I)
        X(IR) = B(I)/A(I,IR)
  100 CØNTINUE
      DØ 110 I=1,N
        B(I) = X(I)
  110 CØNTINUE
  120 RETURN
      END
```

Algorithm 471 Exponential Integrals [S13]

Walter Gautschi [Recd. 21 Jan. 1972 and 9 Oct. 1972]
Department of Computer Sciences, Purdue University, Lafayette, IN 47907

Key Words and Phrases: exponential integral, recurrence relations, recursive computation, continued fractions
CR Categories: 5.12
Language:Algol

Work supported in part by a Fulbright research grant, and in part by the U.S. Atomic Energy Commission.

Description

1. Introduction. The functions

$$E_n(x) = \int_1^\infty e^{-xt}t^{-n}dt, \quad x > 0, \quad n \text{ an integer},$$

are referred to as exponential integrals. The special case $n = 0$ gives $E_0(x) = e^{-x}/x$, and for n negative we have

$$E_n(x) = (-1)^n(d/dx)^{|n|}E_0(x), \quad n < 0,$$

for which an algorithm was published previously [3]. Our concern here is with the case of positive integers n. We present an algorithm which evaluates

$$f_n(x) = e^xE_n(x), \quad x > 0, \quad n = 1, 2, \ldots, N$$

to an accuracy of d significant decimal digits.

2. Method of Calculation. The basic tool of computation is the well-known recurrence relation

$$f_{n+1}(x) = (1 - xf_n(x))/n. \tag{2.1}$$

We use it in two different ways, depending on whether $0 < x \leq 1$ or $x > 1$.

On the first interval, we apply (2.1) for $n = 1, 2, \ldots, N - 1$, assuming a real procedure $f1$ to supply the starting value $f_1(x)$. The real procedure $f1$ furnished below obtains $f_1(x)$ accurately to d significant digits. It is based on the power series expansion

$$f_1(x) = e^x\left(\sum_{k=1}^{\infty} \frac{(-1)^{k-1}x^k}{k \times k!} - \gamma - ln\,(x)\right), \tag{2.2}$$

where $\gamma = .5772156649\ldots$ is Euler's constant. Since the terms in the infinite series of (2.2) are alternating in sign and strictly decreasing in modulus (if $0 < x \leq 1$), the partial sums of even order, s_{2k}, converge monotonically increasing to the limit value s_∞, while those of odd order, s_{2k+1}, converge monotonically decreasing to s_∞. Consequently, if $\bar{s}_k = (s_{2k} + s_{2k+1})/2$, we have $|\bar{s}_k - s_\infty| \leq \frac{1}{2}\epsilon|\bar{s}_k|$ as soon as $s_{2k+1} - s_{2k} \leq \epsilon\bar{s}_k$. The last inequality, with $\epsilon = 10^{-d}$, is used as a termination criterion for the summation of the infinite series in (2.2). In order to prevent infinite loops in cases where d is specified unreasonably large for a particular computer, we use Rutishauser's device [8, §36.3] of terminating the summation process also if the machine representations of s_{2k}, or s_{2k+1}, cease to exhibit monotonic behavior.

The subtraction of $\gamma + ln(x)$ from the infinite series in (2.2) does not cause any appreciable loss of significance if x is restricted to the interval $0 < x \leq 1$. This consideration was partly responsible for choosing $x = 1$ as the transition point.

On the remaining interval, $x > 1$, we let $n1 = \langle x\rangle$, the integer closest 1o x, and compute $f_n(x)$ by backward recurrence for $1 \leq n < n1$, and by forward recurrence for $n1 < n \leq N$ (if $N > n1$), thereby maintaining optimal error propagation characteristics [2, Ex. 5.4]. The starting value $f_{n_1}(x)$ for both recursions is obtained from Legendre's continued fraction [7, p. 103]

$$e^xE_n(x) = \frac{1}{x+}\,\frac{n}{1+}\,\frac{1}{x+}\,\frac{n+1}{1+}\,\frac{2}{x+}\,\frac{n+2}{1+}\,\frac{3}{x+}\cdots. \tag{2.3}$$

Noting that the partial numerators and denominators are all positive, it follows that the convergents of even and odd order approach the common limit value monotonically increasing and decreasing, respectively. Therefore, devices similar to those described above for $f_1(x)$ can be used to terminate the continued fraction evaluation. The convergents of even order are obtained as the successive convergents of the even contraction

$$e^xE_n(x) = \frac{a_1}{b_1-}\,\frac{a_2}{b_2-}\,\frac{a_3}{b_3-}\cdots, \tag{2.3e}$$

where

$$a_1 = 1, \quad b_1 = x + n,$$

$$\left.\begin{aligned} a_k &= (k-1)(n+k-2) \\ b_k &= x + n + 2k - 2 \end{aligned}\right\} k = 2, 3, 4, \ldots,$$

while those of odd order are obtained as the successive convergents of the odd contraction

$$e^xE_n(x) = \frac{1}{x}\left(1 - \frac{a_1}{b_1-}\,\frac{a_2}{b_2-}\,\frac{a_3}{b_3-}\cdots\right), \tag{2.3o}$$

where

$$a_1 = n, \; b_1 = x + n + 1,$$

$$\left.\begin{aligned} a_k &= (k-1)(n+k-1) \\ b_k &= x + n + 2k - 1 \end{aligned}\right\} k = 2, 3, 4, \ldots.$$

In either case, the successive convergents are evaluated directly by the third method described in [1, p. 29]. Overflow problems associated with the more common method based on the three-term recurrence relation for the numerators and denominators are thus avoided.

The number of convergents required in (2.3e) and (2.3o), to meet a particular accuracy requirement, was observed to be a non-increasing function of x on $x \geq 1$, if we take $n = \langle x\rangle$. In contrast, the number of terms required in the infinite series of (2.2) increases with x. Some relevant information is collected in Table I. For values of x between 0 and 1, the numbers listed represent the number of even (and odd) partial sums required in (2.2) to obtain $f_1(x)$ accurately to d significant digits. Similarly, for $x > 1$, we list the number of even (and odd) convergents of the Legendre continued fraction required to obtain $e^xE_n(x)$ for $n = \langle x\rangle$ to the same accuracy.

It will be noted that near the transition point $x = 1$ the continued fraction evaluation is considerably more time-consuming than the series evaluation. The imbalance could easily be corrected by moving up the transition point. In so doing, however, the evaluation of $f_1(x)$ from (2.2) involves progressively more loss of significant accuracy. In our algorithm, we have decided to leave the

Table I. Number of Partial Sums in (2.2), and Convergents in (2.3), To Meet Specific Accuracy Requirements

x \ d	2	4	6	8	10	12	14	16	18	20	22	24	26
.01	2	2	3	3	3	4	4	4	5	5	5	6	6
.20	2	3	4	4	5	5	6	7	7	8	8	9	9
.40	2	3	4	5	6	6	7	8	8	9	9	10	11
.60	3	4	5	5	6	7	8	8	9	10	10	11	12
.80	3	4	5	6	7	8	8	9	10	10	11	12	13
1.00	3	4	5	6	7	8	9	10	10	11	12	13	13
1.01	4	11	20	31	45	62	81	103	128	155	185	218	251
1.20	4	9	17	27	39	53	70	88	109	132	157	185	214
1.50	4	9	15	23	34	45	59	74	92	110	131	154	177
2.00	3	7	12	19	27	36	46	58	71	86	101	119	137
5.00	2	5	7	11	15	19	24	29	35	42	49	57	65
10.00	2	4	6	8	10	13	16	19	23	27	31	35	40
20.00	2	3	4	6	8	10	12	14	16	19	22	24	27
40.00	2	3	4	5	6	8	9	11	13	14	16	18	20
80.00	1	2	3	4	5	7	8	9	10	12	13	14	16

transition point at $x = 1$, thus sacrificing efficiency in favor of accuracy.

Alternatively, instead of the continued fraction (2.3) we could use a Taylor expansion about $x = n$, when n is moderately large, and asymptotic formulas, when x and n are large. This would result in a more efficient, but larger, program. It would also become necessary to store key values of $E_n(n)$ and thus to fix the precision d.

No provisions are made to test for overflow or underflow conditions which may arise near the singularities $x = 0$ and $x = \infty$ of $f_n(x)$. As for the first, overflow occurs only for extremely small values of x and is likely to be caught by the library subroutine for the logarithm. At the singularity at infinity underflow occurs only for extremely large values of x or n, or both.

3. Tests. Exponential integrals are tabulated by G.F. Miller [5], who gives $(x + n)e^x E_n(x)$ to nine significant digits in the range $0 \leq x \leq 20$ and $0 \leq x^{-1} \leq .05$, generally for $n = 1(1)24$. We tested our algorithm (with *nmax* = 24, $d = 9$) against these tables for selected x-values in the interval (0, 20], and for $x^{-1} = .001$, $x^{-1} = .005(.005).05$. No discrepancies were detected, other than occasional end figure errors of one unit. We also found ourselves in agreement with the initial portion ($x \leq .6$) of the 7-10S table in Kourganoff and Busbridge [4], but observed many end figure discrepancies (of up to 12 units) in the remaining portion of the table. A double check with Miller's table indicates that these discrepancies are due to small errors in the Kourganoff-Busbridge table. John W. Wrench Jr. has kindly supplied the author with 25S values of $E_n(10)$, $n = 1(1)25$, which he computed in 40S arithmetic on a desk calculator. A double precision Fortran version of our algorithm (run with *nmax* = 25, $d = 25$) reproduced these values correctly to all 25 significant digits. The same Fortran version of the algorithm was used with *nmax* = 1, $d = 16$, to compare against the 16S table of $e^x E_1(x)$ given by Miller and Hurst [6]. For the test values $x = .2(.05)1.0$, $x = 1.05$, $x = 1.5$, $x = 2^k$, $k = 1(1)6$, no discrepancies were observed, except for $x = .95$, where the last digit was in error by one unit. All tests were performed on a CDC 6500 computer.

4. *Formal parameter list.*

x	the argument in $f_n(x)$; type real;
nmax	the maximum value N of n; type integer;
d	the desired number of significant decimal digits; type integer;
f	an array of dimension [1:*nmax*] holding the result $f_n(x)$ in $f[n]$.

Acknowledgment. The author is pleased to acknowledge valuable suggestions of the referee, which resulted in a simpler and more flexible algorithm.

References

1. Gautschi, W. Computational aspects of three-term recurrence relations, SIAM Rev. *9* (1967), 24-82.
2. Gautschi, W. Zur Numerik rekurrenter Relationen, Computing *9* (1972), 107-126.
3. Gautschi, W., and Klein, B.J. Remark on Algorithm 282, Derivatives of e^x/x, $\cos(x)/x$, and $\sin(x)/x$. *Comm. ACM 13*, 1 (Jan. 1970), 53.
4. Kourganoff, V., and Busbridge, I.W. *Basic Methods in Transfer Problems.* Clarendon Press, Oxford, 1952.
5. Miller, G.F. Tables of generalized exponential integrals. National Physical Lab. Math. Tables, Vol. 3, H.M. Stationery Office, London, 1960.
6. Miller, J., and Hurst, R.P. Simplified calculation of the exponential integral. *Math. Tables Other Aids Comp. 12* (1958), 187-193.
7. Perron, O. *Die Lehre von den Kettenbrüchen*, Vol. II. B.G. Teubner, Stuttgart, Germany, 1957.
8. Rutishauser, H. *Description of ALGOL 60, Handbook for Automatic Computation, Vol. I, Pt. a.* Springer, New York, 1967.
9. Wrench, J.W., Jr. A new calculation of Euler's constant. *Math. Tables Other Aids Comp. 6* (1952), 255.

Algorithm

```
procedure fsubn(x, nmax, d, f);
  value x, nmax, d;
  integer nmax, d; real x; array f; comment f[1:nmax];
comment This procedure evaluates fn(x) = e↑x En(x) for x > 0,
    n = 1, 2, . . . , nmax, to an accuracy of d significant decimal
    digits. The results are stored in the array f. If x ≤ 0, or nmax ≤ 0,
    the procedure immediately sends control to a procedure recovery
    and exits from the procedure fsubn. A call is made to a real pro-
    cedure f1 which is to return f1(x) for 0 < x ≤ 1, with an accuracy
    of d significant digits. A possible version of such a procedure is
    declared below;
begin
integer n, n1, k, k1;
  real eps, ue, ve, we, we1, uo, vo, wo, wo1, w, r, s;
  real procedure f1(x, d); value x, d; integer d; real x;
  begin
    integer k, k1, k2; real eps, gamma, se, se1, so, so1, s, te, to;
    eps := 10 ↑ (−d);
    comment The constant gamma in the following statement
      should be supplied to at least d significant decimal digits.
      For the first 328 digits see [9];
    gamma := .57721566490153286060651 2;
    se := 0; se1 := −1.0; so := to := x; so1 := 2 × x; s := x/2;
    k1 := 1;
    for k := k1 while so-se > eps × s ∧ se > se1 ∧ so < so1 do
    begin
      se1 := se; so1 := so; k2 := 2 × k;
      te := (k2−1) × x × to/(k2×k2); se := se + to − te;
      to := k2 × x × te/((k2+1) × (k2+1)); so := so − te +
        to;
      s := (se+so)/2; k1 := k1 + 1
    end;
    f1 := (s−gamma−ln(x)) × exp(x)
  end f1;
  if x ≤ 0 ∨ nmax ≤ 0 then begin recovery; go to exit end;
  comment recovery is a procedure which the user has to supply
    and in which he may wish to print appropriate error messages;
  if x ≤ 1 then
  begin
    f[1] := f1(x, d);
    for n := 1 step 1 until nmax − 1 do
      f[n+1] := (1−x×f[n])/n;
    go to exit
  end;
  eps := 10 ↑ (−d);
  n1 := entier(x+.5);
  ue := 1.0; ve := we := 1/(x+n1); we1 := 0;
  uo := 1.0; vo := −n1/(x×(x+n1+1)); wo1 := 1/x; wo :=
    vo + wo1;
    w := (we+wo)/2;
    k1 := 1;
    for k := k1 while wo-we > eps × w ∧ we > we1 ∧ wo < wo1 do
    begin
      we1 := we; wo1 := wo;
      r := n1 + k; s := r + x + k;
      ue := 1/(1−k×(r−1)×ue/((s−2)×s));
      uo := 1/(1−k×r×uo/(s×s−1));
      ve := ve × (ue−1); vo := vo × (uo−1);
      we := we + ve; wo := wo + vo;
      w := (we+wo)/2; k1 := k1 + 1
    end;
    if n1 ≤ nmax then f[n1] := w;
    for n := n1 − 1 step −1 until 1 do
    begin
      w := (1−n×w)/x;
      if n ≤ nmax then f[n] := w
    end;
    for n := n1 step 1 until nmax−1 do
      f[n+1] := (1−x×f[n])/n;
exit: end fsubn
```

Algorithm 472

Procedures for Natural Spline Interpolation [E1]

John G Herriot*
Computer Science Department, Stanford University, Stanford, CA 94305
and Christian H. Reinsch
Mathematisches Institut der Technischen Universität, 8 München 2, Germany

[Recd. 6 Mar. 1972]

Key Words and Phrases: approximation, interpolation, spline, natural spline, spline approximation, cubic natural spline
CR Categories: 5.13
Language: Algol

Description

1. Introduction

The purpose of the procedures presented here is to determine the interpolating natural spline function $S(x)$ of degree $2m - 1$ for the set of data points (x_i, y_i), $i = N1, N1 + 1, \ldots, N2$ where it is assumed that $x_{N1} < x_{N1+1} < \cdots < x_{N2}$. The interpolating natural spline function $S(x)$ with the knots $x_{N1}, \ldots, x_{N2}$ has the properties: (i) $S(x)$ is a polynomial of degree $2m - 1$ in each interval (x_i, x_{i+1}) $i = N1, \ldots, N2 - 1$. (ii) $S(x)$ and its derivatives $D^jS(x)$, $j = 1, 2, \ldots, 2m - 2$ are continuous in (x_{N1}, x_{N2}). (If $m = 1$ the conditions on the derivatives are not applicable.) (iii) $D^jS(x_{N1}) = D^jS(x_{N2}) = 0, j = m, m + 1, \ldots, 2m - 2$ if $m > 1$. (iv) $S(x_i) = y_i$. If $N2 - N1 + 1 \geq m$ then there is a unique natural spline function which has the properties (i)–(iv). (See, e.g. Greville[3, 4].) This spline function can be represented in the form

$$S(x) = A_{i0} + A_{i1}t^2 + A_{i2}t^2 + \cdots + A_{i,2m-1}t^{2m-1} \tag{1}$$

with $t = x - x_i$ for $x_i \leq x < x_{i+1}$, $i = N1, \ldots, N2 - 1$. Evidently $A_{i0} = y_i$. Three of the procedures calculate the other elements A_{ij} of the matrix of the coefficients of (1).

The procedure *NATSPLINE* computes the coefficients of the natural spline in the general case described above. Because the computation requires the calculation of mth order divided differences of the data and these are subject to serious roundoff errors when m is large, it is recommended that this procedure not be used for large values of m, say, greater than seven. Moreover, the condition of the matrix which occurs in the system of equations which must be solved in the computation deteriorates rapidly with increasing m.

Procedure *NATSPLINEEQ* treats the case of equidistant knots x_i. If the knots are known to be equidistant, the use of this procedure results in considerable economy of computational effort. The time required for the calculation of the coefficients using *NATSPLINEEQ* is less than half that required if *NATSPLINE* is used. Note that in the case of equidistant knots it is not necessary to specify the values of x_i. The representation (1) is still used, but now $t = (x - x_i)/h$ where $h = x_{i+1} - x_i$, the constant spacing of the knots.

Since the case of a cubic natural spline is of frequent occurrence, we give also a procedure, *CUBNATSPLINE*, which computes the coefficients in this special case. This procedure is very much faster than either of the other procedures when used with $m = 2$ to produce the same results.

In some applications of cubic natural splines it is more efficient to evaluate the spline approximation by means of the formula

$$S(x) = y_i(1 - t) + y_{i+1}t + V(-2t + 3t^2 - t^3)/6 + W(t^3 - t)/6 \tag{2}$$

with $t = (x - x_i)/h_i$ for $x_i \leq x < x_{i+1}$, $h_i = x_{i+1} - x_i$, $V = h_i^2S''(x_i)$, $W = h_i^2S''(x_{i+1})$, instead of using (1). The procedure *CUBNATSPLINE2D* calculates the second derivatives $S''(x_i)$ and the values of h_i which are the quantities needed to use (2). Since this procedure uses one less array than does *CUBNATSPLINE*, the saving of storage may be significant if the number of data points is arge. It is also slightly faster than *CUBNATSPLINE*.

2. Method of Calculation

(*a*) *General case.* The calculation of the coefficients is carried out in a numerically stable manner following a method described by Anselone and Laurent [1] specialized to the case of the interpolating natural spline as described above. The method is based on the use of minimum support B-splines [2, 4] to form a basis for the class of mth derivatives of the natural splines. For convenience of calculation we use a normalizing factor different from that of Greville [4]. For a fixed m, our B-splines are defined by

$$M_k(x) = M(x; x_k, x_{k+1}, \ldots, x_{k+m}) \tag{3}$$

where

$$M(x;t) = ((-1)^m/(m - 1)!)(t - x)_+^{m-1}. \tag{4}$$

Here $x_+^r = x^r$ if $x > 0$ and 0 otherwise. $M(x;x_k, x_{k+1}, \ldots, x_{k+m})$ denotes the mth divided difference of $M(x;t)$ with respect to t based on the arguments $x_k, x_{k+1}, \ldots, x_{k+m}$. $M_k(x)$ is of constant sign in (x_k, x_{k+m}) and vanishes outside this interval. It is known that a natural spline function $S(x)$ may be extended uniquely over the whole real line by imposing the continuity conditions (ii) at all points. Then outside (x_{N1}, x_{N2}), $S(x)$ is a polynomial of degree $m - 1$, and consequently $D^mS(x)$ vanishes outside (x_{N1}, x_{N2}). It follows that $D^mS(x)$ has a unique representation of the form

$$D^mS(x) = \sum_{k=N1}^{N2-m} d_k(2m - 1)!\, M_k(x). \tag{5}$$

The constants d_k are found by solving the well-conditioned system of equations

$$\sum_{k=N1}^{N2-m} N_{ik}d_k = y_{i,i+1,\ldots,i+m},\ i = N1, \ldots, N2 - m \tag{6}$$

where

$$N_{ik} = N(x_i, x_{i+1}, \ldots, x_{i+m}; x_k, x_{k+1}, \ldots, x_{k+m}) \tag{7}$$

with

$$N(s, t) = (s - t)_+^{2m-1}. \tag{8}$$

Here N_{ik} are the elements of a positive definite band matrix with $N_{ik} = 0$ if $|i - k| \geq m$. The solution of this system is obtained by Gaussian elimination without pivoting.

In order to determine $S(x)$, eq. (5) has to be integrated m times. We introduce two m-fold integrals of $(2m - 1)!\, M_k(x)$:

$$E_k(x) = (2m - 1)! \int_{-\infty}^{x} dx \ldots \int_{-\infty}^{x} dx\, M_k(x), \tag{9}$$

* The work of this author was supported in part by the National Science Foundation under Grant Number GJ-408.

and

$$F_k(x) = (2m-1)! \int_{+\infty}^{x} dx \ldots \int_{+\infty}^{x} dx\, M_k(x). \tag{10}$$

If we use the well-known form of the mth divided difference (see, e.g. Greville [4]) we can use (3) to obtain two alternative explicit formulas for $M_k(x)$. When we substitute these in eqs. (9) and (10), we obtain

$$E_k(x) = \textstyle\sum_{i=0}^{m} (x - x_{k+i})_+^{2m-1}/w_k'(x_{k+i}) \tag{11}$$

and

$$F_k(x) = \textstyle\sum_{i=0}^{m} (x_{k+i} - x)_+^{2m-1}/w_k'(x_{k+i}) \tag{12}$$

where

$$w_k'(x) = D_x[(x - x_k)(x - x_{k+1}) \cdots (x - x_{k+m})]. \tag{13}$$

Equation (11) shows that $E_k(x) = 0$, if $x < x_k$, and

$$E_k(x) = (x - x_k)^{2m-1}/w_k'(x_k), \text{ if } x_k \le x < x_{k+1}. \tag{14}$$

Each time we pass a knot x_{k+i} from left to right, there enters a term $(x - x_{k+i})^{2m-1}/w_k'(x_{k+1})$ which is added to the current polynomial. We can therefore write $E_k(x)$ in the form

$$E_k(x) = \textstyle\sum_{j=0}^{2m-1} e_{k,i,j}(x - x_{k+i})^j \text{ in } x_{k+i} \le x < x_{k+i+1}. \tag{15}$$

From eq. (14) it is clear that

$$\begin{aligned} e_{k,0,j} &= 0 & j &= 0, 1, \ldots, 2m-2, \\ &= 1/w_k'(x_k) & j &= 2m-1. \end{aligned}$$

The other $e_{k,i,j}$ are determined recursively. When $e_{k,i-1,j}$ have been calculated so that $E_k(x)$ is determined by (15) in $x_{k+i-1} \le x < x_{k+i}$, we use the complete Horner scheme to expand the polynomial in powers of $x - x_{k+i}$ and then add the appropriate term required to pass to the interval $[x_{k+i}, x_{k+i+1})$. In the same way $F_k(x)$ may be written in the form

$$F_k(x) = \textstyle\sum_{j=0}^{2m-1} e_{k,-i,j}(x - x_{k+m-i})^j, \quad x_{k+m-i} < x \le x_{k+m-i+1}. \tag{16}$$

Again the $e_{k,-i,j}$ are determined recursively. It suffices to generate $e_{k,i,j}$ and $e_{k,-i,j}$ for only a very limited set of values of k and i as we see below.

By integrating eq. (5) m times, using (9) and (10), and noting that $E_j(x) = 0$ for $x \le x_j$, and $F_j(x) = 0$ for $x \ge x_{j+m}$, we find that

$$S(x) = T(x) + P(x) \tag{17}$$

where

$$T(x) = \textstyle\sum_{x_{j+m}>x}^{k-1} d_jF_j(x) + \sum_{k-1}^{x_j<x} d_jE_j(x) \tag{18}$$

with k arbitrary and $P(x)$ a polynomial of degree $m - 1$ depending on k.

We now let k assume the set of values best described by the Algol 60 for-clause

for $k := N1$ **step** $m - 1$ **until** $N2 - m, N2 - m + 1$ **do**.

For each such value of k we calculate $T(x)$ in the interval $[x_k, x_{k+m-1})$. Then $P(x)$ is uniquely determined by the interpolation conditions

$$y_{k+l} = T(x_{k+l}) + P(x_{k+l}), \quad l = 0, 1, \ldots, m-1.$$

Newton's divided difference formula is used in obtaining $P(x)$. For each value of k it is necessary to calculate the values of $e_{p,i,j}$ only for $p = k, k+1, \ldots, k+m-2$, $i = 0, 1, \ldots, k+m-p-2$, $j = 0, 1, \ldots, 2m-1$, and for $p = k-m+1, k-m+2, \ldots, k-1$, $i = -1, -2, \ldots, k-m-p$, $j = 0, 1, \ldots, 2m-1$. Furthermore, p is restricted to lie between $N1$ and $N2 - m$. More details on the organization of the calculations are given in [5].

Table I. Cubic Natural Spline.
Five nonequidistant knots. Coefficients calculated by *NATSPLINE*

x	$S(x)$	$S'(x)$	$S''(x)/2$	$S'''(x)/3$:
−3.000000	7.000000	−1.999998	0	0.9999998
	11.00000	9.999996	5.999997	0.9999998
−1.000000	11.00000	10.00000	5.999999	−1.000000
	25.99998	18.99998	2.999999	−1.000000
0	26.00000	18.99997	2.999995	−1.999996
	55.99995	−16.99994	−14.99997	−1.999996
3.000000	56.00000	−16.99998	−14.99999	4.999996
	29.00003	−31.99995	0	4.999996
4.000000	29.00000			

(*b*) *Equidistant knots.* The calculation of the coefficients in *NATSPLINEEQ* for the case of equidistant knots is carried out in the same manner as in *NATSPLINE* for the general case. However, there are a number of simplifications which result in considerable economy of computational effort. It is not necessary to specify the x_i. Hence we can assume that $x_i = i$. It is convenient to modify eq. (6) slightly. First of all the right-hand sides reduce to $\Delta^m y_i/m!$ where $\Delta^m y_i$ are ordinary mth differences and require no divisions in their calculation. In the second place it can be shown that N_{ik} is the $2m$th ordinary difference of $s - t_+^{2m-1}/((-1)^m(m!)^2)$ based on the values $s - t = i - k - m, \ldots, i - k + m$. We rescale $M(x;t)$, $M_k(x)$, $E_k(x)$, $F_k(x)$ by multiplying their representations in eqs. (4), (11), and (12) by $(-1)^m m!$. Thus d_k is rescaled by dividing it by $(-1)^m m!$. We denote the rescaled coefficients by d_k^*. If we let N_{ik}^* be the $2m$th difference of j_+^{2m-1} based on the values $j = i - k - m, \ldots, i - k + m$, then $N_{ik}^* = N_{ik}(-1)^m(m!)^2$ and eq. (6) becomes

$$\textstyle\sum_{k=N1}^{N2-m} N_{ik}^* d_k^* = \Delta^m y_i, \quad i = N1, \ldots, N2-m. \tag{19}$$

For large values of m, the calculation of N_{ik}^* by the obvious differencing technique involves serious cancellation and may introduce errors in the computed values of N_{ik}^*. It can be shown that these differences satisfy the recurrence relation

$$\Delta^n(j_+^{n-1}) = (n+j)\Delta^{n-1}((j+1)_+^{n-1}) - j\Delta^{n-1}(j_+^{n-2}). \tag{20}$$

We need to calculate these quantities only for $n = 2m$ at $j = i - k - m$ for $i - k = -m+1, \ldots, 0, 1, \ldots, m-1$, i.e., for $j = -2m+1, \ldots, -2, -1$. In this range, the two weight factors $2m + j$ and $-j$ are both positive, one ranging from 1 to $2m - 1$ and the other from $2m - 1$ to 1. Thus no cancellation can occur when formula (20) is used for calculating N_{ik}^*.

A further simplification occurs because the coefficients of $E_k(x)$ and $F_k(x)$ are independent of k. It therefore suffices to compute the coefficients of $E_0(x)$ and $F_0(x)$. Moreover $F_0(x) = (-1)^m E_0(m - x)$. Thus we have only to calculate the values of an array e_{ij} for $i = -m+1, \ldots, -1, 0, 1, \ldots, m-1$ and $j = 0, 1, \ldots, 2m-1$. This is a major saving over the calculations for the general case. The rest of the calculations are carried out as in the general case.

(*c*) *Cubic spline.* Much computational labor is saved by treating this as a special case instead of using the general program with $m = 2$. We start with eq. (1) setting $m = 2$. By imposing the conditions (ii), (iii) and (iv) at the knots, we get relations between the coefficients, which yield a tridiagonal system of equations for $A_{i2}(= S''(x_i)/2$, the coefficients of t^2. This tridiagonal system is solved by Gaussian elimination. In the procedure *CUBNATSPLINE·2D* the values of $S''(x_i)$ and $h_i = x_{i+1} - x_i$ are output. In the procedure *CUBNATSPLINE* the values of A_{i1}, A_{i2} and A_{i3} are output.

3. *Tests*

These procedures have been tested in Alcor Algol on the Telefunken TR-4 at the Rechenzentrum of the Technischen Universität München and in Algol W on the IBM 360/67 at the Stanford Computation Center. The latter tests included timing tests of the procedures over a range of values of m up to 7 and number of knots $N = N2 - N1 + 1$ up to 100. The time was found to be approximately proportional to the number N of knots and to the square of m. The time T in seconds for the execution of the procedure *NATSPLINE* was found to be approximately

$$T = N/60\ (0.117m^2 - 0.296m + 0.512).$$

For *NATSPLINEEQ* the time was approximately

$$T = N/60\ (0.014m^2 + 0.023m + 0.029).$$

For *CUBNATSPLINE* the time was approximately

$$T = .045N/60 = .00075N.$$

For *CUBNATSPLINE2D* the time was approximately

$$T = .03N/60 = .0005N.$$

In order to check the accuracy of the coefficients calculated for the spline approximation $S(x)$, the values of $D^kS(x)/k!$, $k = 1, 2, \ldots, 2m - 2$ were calculated at the right-hand endpoint of each subinterval $[x_i, x_{i+1})$ and compared with their values (the coefficients in eq. (1)) at the left-hand endpoint of the next subinterval. It was found that the accuracy deteriorated somewhat for larger values of m, although for $m = 7$, with the data used, the largest relative differences were observed to be approximately 0.0018. Table I shows the results of a typical run using *NATSPLINE* for five nonequidistant knots with $m = 2$. The first line of each box gives the tabulated quantities at the given value of x, which is the left-hand endpoint of the subinterval, and the second line of the box gives the tabulated quantities at the right-hand endpoint of the same subinterval. The close agreement of these quantities $D^kS(x)/k!$, $k = 1, 2, \ldots, 2m - 2$ shows that the spline function and its derivatives satisfy the specified continuity conditions. This is a good indication of the correctness of the results.

References

1. Anselone, P.M., and Laurent, P.J. A general method for the construction of interpolating and smoothing spline functions. *Numer. Math. 12* (1968), 66–82.
2. Curry, H.B., and Schoenberg, I.J. On Pólya frequency functions. IV. The fundamental spline functions and their limits. *J. Analyse Math. 17* (1966), 71–107.
3. Greville, T.N.E. Spline functions, interpolation and numerical quadrature. In *Mathematical Methods for Digital Computers, Vol. II.* A. Ralston and H.S. Wilf (Eds.) Wiley, New York, 1967.
4. Greville, T.N.E. Introduction to spline functions. In *Theory and Applications of Spline Functions.* T.N.E. Greville (Ed.) Academic Press, New York, 1969, pp. 1–35. (Pub. No. 22 Mathematics Research Center, U.S. Army, U. of Wisconsin.)
5. Herriot, John G., and Reinsch, Christian H. Algol 60 procedures for the calculation of interpolating natural spline functions. Tech. Rep. STAN-CS-71-200, Comput. Sci. Dep., Stanford U. 1971.

Algorithm

```
procedure NATSPLINE(N1, N2, m, x, A);
  value N1, N2, m;   integer N1, N2, m;
  array x, A;
comment NATSPLINE computes the coefficients of a natural spline
  S(x) of degree (2×m−1), interpolating the ordinates y[i] at
  points x[i], i = N1 through N2. For xx in [x[i], x[i+1]): S(xx) =
  A[i, 0] + A[i, 1] × t + . . . + A[i, 2×m−1] × t ↑ (2×m−1)
  with t = xx − x[i],
  Input:
    N1, N2 subscript of first and last data point
    m 2×m−1 is the degree of the natural spline,
      admissible values range from 1 to N2 − N1 + 1,
      recommended values are not greater than seven (say)
    x[N1:N2] contains the given abscissas x[i] which must be
      strictly monotone increasing
    A[N1:N2, 0:2×m−1] contains the given ordinates as zero-th
      column, i.e. A[i, 0] represents y[i],
  Output:
    A[N1:N2, 0:2×m−1] the coefficients of the natural spline as de-
      scribed above (the zero-th column is unchanged and no val-
      ues are assigned to the last row of A);
if m > 0 ∧ m ≤ N2 − N1 + 1 then
begin
  integer i, j, k, l, l1, m1, m2, mm, n, mk, k1, jj, kk, j1;
  real f, z, w;
  array C[0:2×m], D[N1:N2], E[0:m−1, 1−m:m−1, 0:2×m−1],
    P[0:m], Q[0:m, N1:N2];
  comment i-j-entry of band-matrix stored in A[i, j−i+1], right-
    hand stored in vector D;
  m1 := m − 1; m2 := m − 2;   mm := 2 × m − 1;   n := N2 −m;
  for j := N1 step 1 until N2 do
  begin
    l := j + m;   if l > N2 then l := N2;
    for i := j step 1 until l do Q[i−j, j] := (x[i] − x[j]) ↑ mm
  end;
  for i := N1 + 1 step 1 until N2 do
  begin
    l := i − N1;   if l > m then l := m;
    for j := 0 step 1 until l do P[j] := Q[j, i−j];
    for k := 1 step 1 until m do
    begin
      l1 := i + k − N2;   if l1 < 1 then l1 := 1;
      for j := l step −1 until l1 do
      P[j] := (P[j−1]−P[j])/(x[i−j+k]−x[i−j])
    end;
    for j := l1 step 1 until l do Q[j, i−j] := P[j]
  end;
  for j := N1 step 1 until n do
  begin
    for i := 0 step 1 until m do P[i] := Q[i, j];
    for k := 1 step 1 until m do
    begin
      l1 := N1 − j + k;   if l1 < 1 then l1 := 1;
      for i := m step −1 until l1 do
        P[i] := (P[i]−P[i−1])/(x[i+j]−x[i+j−k])
    end;
    for i := l1 step 1 until m do Q[i, j] := P[i]
  end;
  for j := 1 step 1 until m do
  begin
    l := n − j + 1;
    for i := N1 step 1 until l do A[i, j] := Q[m−j+1, i+j−1]
  end;
  for i := N1 step 1 until N2 do D[i] := A[i, 0];
  for k := 1 step 1 until m do
  begin
    l := N2−k;
    for i := N1 step 1 until l do
      D[i] := (D[i+1]−D[i])/(x[i+k]−x[i])
  end;
  comment Gaussian elimination without pivoting, rational
    Cholesky;
  for i := N1 step 1 until n do
  begin
    l := i + m1;   if l > n then l := n;
    for j := i + 1 step 1 until l do
    begin
      comment f := j-i-entry/i-i-entry, symmetry;
      f := A[i, j−i+1]/A[i, 1];
      D[j] := D[j] − f × D[i];
      for k := j step 1 until l do
        A[j, k−j+1] := A[j, k−j+1] − f × A[i, k−i+1]
    end j
end i;
comment Back substitution;
for i := n step −1 until N1 do
begin
  l := n − i;   if l ≥ m then l := m1;
  f := D[i];
  for j := 1 step 1 until l do f := f − A[i,j+1] × D[i+j];
  D[i] := f/A[i, 1]
end i;
comment Now compute the coefficients of the natural spline;
if m1 = 0 then
begin
  for k := N1 step 1 until n do
    A[k, 1] := −D[k]/(x[k+1]−x[k])
end
else
for k := N1 step m1 until n, N2 − m1 do
begin
  comment Now compute coefficients of the two sets of m-fold
    integrals of the minimum support splines scaled with
    (2×m−1) factorial;
  l := m2;  if l > n − k then l := n − k;
  for kk := 0 step 1 until l do
  begin
    mk := m1 − kk;
    for j := 0 step 1 until mm do C[j] := 0;
    for i := 1 step 1 until mk do
    begin
      k1 := k + kk;
      w := 1;
      for j := 0 step 1 until m do if j ≠ i − 1 then
        w := w × (x[k1+i−1]−x[k1+j]);
      C[mm] := C[mm] + 1/w;
      for j := 0 step 1 until mm do
        E[kk,i−1,j] := C[j];
      if i < mk then
      begin
        z := x[k1+i] − x[k1+i−1];
        for j := 1 step 1 until mm do
          for j1 := mm step −1 until j do
          C[j1−1] := C[j1] × z + C[j1−1]
      end
    end
  end;
  l := m1;  if l > k − N1 then l := k − N1;
  for kk := 1 step 1 until l do
  begin
    mk := m − kk;
    for j := 0 step 1 until mm do C[j] := 0;
    for i := 1 step 1 until mk do
    begin
      k1 := k − kk;
      w := 1;
      for j := 0 step 1 until m do if j ≠ m − i + 1 then
        w := w × (x[k1+m−i+1] − x[k1+j]);
      C[mm] := C[mm] − 1/w;
      z := x[k1+m−i] − x[k1+m−i+1];
      for j := 1 step 1 until mm do
      for j1 := mm step −1 until j do
        C[j1−1] := C[j1] × z + C[j1−1];
      for j := 0 step 1 until mm do E[kk−1,−i,j] := C[j]
    end
  end;
  for l := 0 step 1 until m2 do
  begin
    comment Coefficients of the spline T(x) of degree (2×m−1)
      in the interval [x[k+l], x[k+l+1]) stored as (k+l)-th
      row of A, P(x) = y(x) − T(x) at the points x = x[k]
      through x[k+m−1] stored in P;
      for j := 0 step 1 until mm do C[j] := 0;
      for i := l − m1 step 1 until l do
      begin
        jj := l − i;  j := k + jj;
        if i < 0 then begin j := j − m;  jj := m1 − jj end;
        if j ≥ N1 ∧ j ≤ n then
        begin
          f := D[j];
          for j1 := 0 step 1 until mm do
            C[j1] := C[j1] + f × E[jj,i,j1]
        end j
      end i;
      for j := 1 step 1 until mm do A[k+l,j] := C[j];
      P[l] := A[k+l,0] − C[0]
    end l;
    f := 0;  z := x[k+m1] − x[k+m1−1];
    for j := mm step −1 until 0 do f := f × z + C[j];
    P[m1] := A[k+m1,0] − f;
    comment Compute P(x) from its ordinates at the points
      x = x[k] through x[k+m−1] using Newton's divided
      difference scheme for interpolation;
    for i := 1 step 1 until m1 do
    for j := m1 step −1 until i do
      P[j] := (P[j]−P[j−1])/(x[k+j]−x[k+j−i]);
    for l := 0 step 1 until m2 do
    begin
      comment Add coefficients of P(x) in interval
        [x[k+l],x[k+l+1]) to those of T(x) stored in (k+l)th
        row of A;
      for j := 0 step 1 until m1 do C[j] := P[j];
      for i := m2 step −1 until 0 do
      for j := i step 1 until m2 do
        C[j] := C[j] + (x[k+l]−x[k+i]) × C[j+1];
      for j := 1 step 1 until m1 do
        A[k+l,j] := A[k+l,j] + C[j]
    end l
  end k
end NATSPLINE;


procedure NATSPLINEEQ (N1,N2,m,A);
  value N1, N2, m;  integer N1,N2,m;
  array A;
comment NATSPLINEEQ computes the coefficients of a natural
  spline S(x) of degree (2×m−1), interpolating the ordinates y[i]
  at equidistant points x[i], i = N1 through N2. For xx in
  [x[i], x[i+1]): S(xx) = A[i,0] + A[i,1] × t + . . .
  + A(i,2×m−1] × t↑ (2×m−1) with
  t = (xx−x[i])/(x[i+1]−x[i]) from [0,1),
  Input:
      N1, N2 subscript of first and last data point
      m 2×m−1 is the degree of the natural spline,
        admissible values range from 1 to N2 − N1 + 1,
        recommended values are not greater than seven (say)
      A[N1:N2,0:2×m−1] contains the given ordinates as zero-th
        column, i.e. A[i,0] represents y[i],
  Output:
      A[N1:N2,0:2×m−1] the coefficients of the natural spline
        as described above, (the zero-th column is unchanged and
        no values are assigned to the last row of A);
if m > 0 ∧ m ≤ N2 − N1 + 1 then
begin
  integer i, j, j1, k, l, m1, m2, mm, n;  real f;
  array C[0:2×m], D[N1:N2], E[1−m:m−1,0:2×m−1], P[0:m];
  comment i-j-entry of band-matrix stored in A[i,j−i+1],
    right-hand stored in vector D;
  m1 := m − 1; m2 := m − 2;  mm := 2 × m − 1;
  n := N2 − m;
  for i := 1 step 1 until mm do
  begin
    C[i] := 1;
    for j := i - 1 step -1 until 2 do
      C[j] := (i+1-j) × C[j-1] + j × C[j]
  end i;
  for i := N1 step 1 until N2 do D[i] := A[i,0];
  for j := 1 step 1 until m do
  begin
    f := C[m+1-j];  l := N2 - j;
    for i := N1 step 1 until n do A[i,j] := f;
    for i := N1 step 1 until l do D[i] := D[i+1] - D[i]
  end j;
  comment Gaussian elimination without pivoting, rational
    Cholesky;
  for i := N1 step 1 until n do
  begin
    l := i + m1;  if l > n then l := n;
    for j := i + 1 step 1 until l do
    begin
      comment f:=j-i-entry/i-i-entry, symmetry;
      f := A[i,j-i+1]/A[i,1];
      D[j] := D[j] - f × D[i];
      for k := j step 1 until l do
        A[j,k-j+1] := A[j,k-j+1] - f × A[i,k-i+1]
    end j
  end i;
  comment Back substitution;
  for i := n step -1 until N1 do
  begin
    l := n - i; if l ≥ m then l := m1;
    f := D[i];
    for j := 1 step 1 until l do f := f - A[i,j+1] × D[i+j];
    D[i] := f/A[i,1]
  end i;
  comment Now compute coefficients of the two m-fold integrals
    of the minimum support spline scaled with (2×m-1)
    factorial;
  l := 1;
  for j := 0 step 1 until mm do C[j] := 0;
  for i := 1 step 1 until m1 do
  begin
    C[mm] := C[mm] + l;
    l := l × (i-1-m)/i;
    for j := 0 step 1 until mm do E[i-1,j] := C[j];
    for j := 1 step 1 until mm do
    for k := mm step -1 until j do C[k-1] := C[k-1] + C[k];
    for j := 0 step 1 until mm do E[-i,j] := C[j]
  end i;
  comment Change sign;
  for j := m1 step -2 until 0, m + 1 step 2 until mm do
  for i := -m1 step 1 until -1 do E[i,j] := -E[i,j];
  comment Now compute coefficients of the natural spline;
  if m1 = 0 then
  begin
    for k := N1 step 1 until n do A[k,1] := D[k]
  end
  else
  for k := N1 step m1 until n, N2 - m1 do
  begin
    for l := 0 step 1 until m2 do
    begin
      comment Coefficients of the spline T(x) of degree (2×m-1)
        in the interval [k+l,k+l+1) stored as (k+l)-th row of
        A, P(x) = y(x) - T(x) at the points x = k through
        k + m - 1 stored in P;
      for j := 0 step 1 until mm do C[j] := 0;
      for i := l - m1 step 1 until l do
      begin
        j := k + l - i;  if i < 0 then j := j - m;
        if j ≥ N1 ∧ j ≤ n then
        begin
          f := D[j];
          for j1 := 0 step 1 until mm do
            C[j1] := C[j1] + f × E[i,j1]
        end j
      end i;
      for j := 1 step 1 until mm do A[k+l,j] := C[j];
      P[l] := A[k+l,0] - C[0]
    end l;
    f := 0;
    for j := mm step -1 until 0 do f := f + C[j];
    P[m1] := A[k+m1,0] - f;
    comment Compute P(x) from its ordinates at the points x = k
      through k + m - 1 using Newton's divided difference
      scheme for interpolation;
    for i := 1 step 1 until m1 do
    for j := m1 step -1 until i do P[j] := P[j] - P[j-1];
    f := 1;
    for j := 2 step 1 until m1 do
    begin
      f := f × j; P[j] := P[j]/f
    end j;
    for l := 0 step 1 until m2 do
    begin
      comment Add coefficients of P(x) in interval [k+l,k+l+1)
        to those of T(x) stored in (k+l)-th row of A;
      for j := 0 step 1 until m1 do C[j] := P[j];
      for i := m2 step -1 until 0 do
      for j := i step 1 until m2 do C[j] := C[j] + C[j+1] × (l-i);
      for j := 1 step 1 unfil m1 do A[k+l,j] := A[k+l,j] + C[j]
    end l
  end k
end NATSPLINEEQ;

procedure CUBNATSPLINE(N1,N2,x,y,B,C,D);
  value N1, N2;  integer N1, N2;
  array x, y, B, C, D;
comment CUBNATSPLINE computes the coefficients of a cubic
  natural spline S(x) interpolating the ordinates y[i] at points x[i],
  i = N1 through N2. For xx in [x[i],x[i+1]):
  S(xx) = ((D[i]×t+C[i])×t+B[i]) × t + y[i] with t = xx - x[i],
  Input:
      N1, N2 subscript of first and last data point
      x, y[N1:N2] arrays with x[i] as abscissa and y[i] as ordinate of
        i-th data point. The elements of the array x must be strictly
        monotone increasing,
  Output:
      B, C, D[N1:N2] arrays collecting the coefficients of the
        cubic natural spline S(xx). C[N2] = 0 while B[N2] and
        D[N2] are left undefined;
begin
  integer i, M1, M2;  real R, S;
  M1 := N1 + 1;  M2 := N2 - 1;  S := 0;
  for i := N1 step 1 until M2 do
    begin
      D[i] := x[i+1] - x[i];
      R := (y[i+1]-y[i])/D[i];
      C[i] := R - S;  S := R
    end i;
    R := S := C[N1] := C[N2] := 0;
    for i := M1 step 1 until M2 do
    begin
      C[i] := C[i] + R × C[i-1];
      B[i] := (x[i-1]-x[i+1]) × 2 - R × S;
      S := D[i];  R := S/B[i]
  end i;
  for i := M2 step -1 until M1 do
    C[i] := (D[i]×C[i+1]-C[i])/B[i];
  for i := N1 step 1 until M2 do
  begin
    B[i] := (y[i+1]-y[i])/D[i] - (2×C[i]+C[i+1]) × D[i];
    D[i] := (C[i+1]-C[i])/D[i];
    C[i] := 3 × C[i]
  end i
end CUBNATSPLINE;

procedure CUBNATSPLINE2D(N1,N2,x,y,D,h);
  value N1, N2;  integer N1, N2;
  array x, y, D, h;
comment Construction of a cubic natural spline S(x) interpolating
  the ordinates y[i] at points x[i], i = N1 through N2. For xx in
  [x[i],x[i+1]):
  S(xx) = y[i] × (1-t) + y[i+1] × t + V
           × (-2×t+3×t×t-t×t×t)/6 + W × (t×t×t-t)/6
  with t = (xx-x[i])/h[i], h[i] = x[i+1] - x[i],
  V = h[i] × h[i] × D[i], W = h[i] × h[i] × D[i+1]. This form
  is especially suited for the evaluation of S(x) and its second
  derivative at points corresponding to t = 1/2, 1/4, 3/4, 1/8,
  3/8, . . . ,
  Input:
      N1, N2 subscript of first and last data point
      x, y[N1:N2] arrays with x[i] as abscissa and y[i] as ordinate
        of i-th data point. The elements of the array x must be
        strictly monotone increasing,
  Output:
    D[N1:N2]  D[i] is the second derivative of S(x) at x = x[i],
      i = N1 through N2
    h[N1:N2]  h[i] = x[i+1] - x[i], i = N1 through N2 - 1;
begin
  integer i, M1, M2;  real U, V, W;
  M1 := N1 + 1;  M2 := N2 - 1;  U := y[N1];
  for i := N1 step 1 until M2 do
  begin
    V := y[i+1];  h[i] := x[i+1] - x[i];
    D[i+1] := (V-U)/h[i];  U := V
  end i;
  W := h[N1];  D[N1] := U := 0;
  for i := M1 step 1 until M2 do
  begin
    comment U = h[i-1]/P[i-1], V = h[i-1], W = h[i], P[i]
      stored in h[i], where P[i] denotes diagonal coefficient in the
      Gaussian elimination;
    V := W;  W := h[i];  h[i] := (V+W) × 2 - U × V;
    D[i] := D[i+1] - D[i] - U × D[i-1];  U := W/h[i]
  end i;
  D[N2] := 0;
  for i := M2 step -1 until M1 do
  begin
    comment Back substitution and restore h[i];
    W := x[i+1] - x[i];
    D[i] := (6×D[i]-W×D[i+1])/h[i];
    h[i] := W
  end i
end CUBNATSPLINE2D
```

Algorithm 473

Computation of Legendre Series Coefficients [C6]

Robert Piessens [Recd. 13 Mar. 1972 and 5 Sept. 1972]
Applied Mathematics Division, University of Leuven, Heverlee, Belgium

This work was supported by the Nationaal Fonds voor Wetenschappelijk Onderzoek (Belgium) under Grant No. 10.174.

Key Words and Phrases: Legendre series, Chebyshev series
CR Categories: 5.13
Language: Fortran

Description

LEGSER approximates the first $N + 1$ coefficients B_n of the Legendre series expansion of a function $f(x)$ having known Chebyshev series coefficients A_n. Several algorithms are available for the computation of coefficients A_n of the truncated Chebyshev series expansion on $[-1, 1]$

$$f(x) \simeq \sum_{n=0}^{N}{}' A_n T_n(x), \tag{1}$$

where $\sum'$ denotes a sum whose first term is halved. The commonly used algorithms are based on the orthogonal property of summation of the Chebyshev polynomials [1]. The application of the analogous property of the Legendre polynomials for the calculation of the coefficients B_n of the expansion

$$f(x) \simeq \sum_{n=0}^{N} B_n P_n(x) \tag{2}$$

is less suitable for practical use since it requires the abscissas and weights of the Gauss-Legendre quadrature formulas [2].

We present here a simple method for the calculation of the coefficients B_n, when the coefficients A_n are given. Since

$$B_n = (n + 1/2) \int_{-1}^{+1} P_n(x) f(x)\, dx \tag{3}$$

we have

$$B_n \simeq (n + 1/2) \sum_{k=0}^{N}{}' A_k I_{n,k}, \tag{4}$$

where

$$I_{n,k} = \int_{-1}^{+1} P_n(x) T_k(x)\, dx. \tag{5}$$

The integrals $I_{n,k}$ can be calculated using the recurrence formula

$$I_{n,k+2} = \frac{[(k-1)k - n(n+1)](k+2)}{[(k+3)(k+2) - n(n+1)]k} I_{n,k}, \tag{6}$$

where $I_{n,k} = 0$ if $k < n$, $I_{n,n} = 2^{2n}(n!)^2/(2n+1)!$ if $n > 0$, $I_{0,0} = 2$.

Example. The Chebyshev series coefficients of the function $f(x) = 1/(2 - x)$ are $A_n = 2^n(1 - \sqrt{0.75})^n/\sqrt{0.75}$.

Table I. Coefficients of the Legendre Series Expansion of $f(x) = 1/(2 - x)$

n	Exact B_n	Errors in computed B_n Absolute errors	 Relative errors
0	0.549294E0	0.12E−4	0.22E−4
1	0.295830E0	0.59E−5	0.20E−4
2	0.105917E0	0.20E−5	0.19E−4
3	0.340972E−1	0.56E−6	0.16E−4
4	0.104495E−1	0.17E−6	0.16E−4
5	0.311269E−2	0.42E−7	0.13E−4
10	0.601250E−5	0.41E−10	0.68E−5
15	0.101339E−7	0.29E−12	0.29E−4
20	0.161332E−10	0.63E−12	0.39E−1

In Table I, the exact Legendre series coefficients of this function are compared with the computed values ($N = 20$). The computations are carried out in single precision on an IBM 370 computer.

In this example, the Chebyshev coefficients are known exactly. In most cases, they must be calculated using an algorithm as in [1].

References

1. Smith, L.B. Algorithm 277, Computation of Chebyshev series coefficients. *Comm. ACM*. *9* (Feb. 1966), 86–87.
2. Bakhvalov, N.S., and Vasileva, L.G. Evaluation of the integrals of oscillating functions by interpolation at nodes of Gaussian quadratures. *Z. Vycisl. mat. i mat. Fiz. 8* (1968), 175–181.

Algorithm

```
      SUBROUTINE LEGSER(A, B, N)
C THIS SUBROUTINE CALCULATES THE COEFFICIENTS OF THE
C LEGENDRE SERIES EXPANSION OF A FUNCTIØN HAVING
C KNØWN CHEBYSHEV SERIES EXPANSION.
C INPUT PARAMETERS
C    N  DEGREE OF THE TRUNCATED CHEBYSHEV SERIES
C    A  VECTØR OF DIMENSION N+1  WHICH CØNTAINS THE
C          CHEBYSHEV COEFFICIENTS
C OUTPUT PARAMETER
C    B  VECTØR ØF DIMENSION N+1  WHICH CØNTAINS THE
C          LEGENDRE COEFFICIENTS
      REAL A, AK, AL, B, BB, C, D
      INTEGER K, L, LL, N, N1
      DIMENSIØN A(N), B(N)
      N1 = N + 1
      AK = 0.0E0
C CALCULATIØN ØF THE FIRST LEGENDRE COEFFICIENT
      B(1) = 0.5E0*A(1)
      IF (N-1) 70, 30, 10
   10 DO 20 K=3,N1,2
        AK = AK + 2.0E0
        B(1) = B(1) - A(K)/(AK*AK-1.0E0)
   20 CØNTINUE
   30 C = 2.0E0/3.0E0
      AL = 0.0E0
C START MAIN LOOP
      DO 60 L=2,N1
C CALCULATION OF THE L-TH LEGENDRE COEFFICIENT
        LL = L + 2
        AL = AL + 1.0E0
        BB = C*A(L)
        IF (LL.GT.N1) GØ TO 50
        D = C
        AK = AL
        DO 40 K=LL,N1,2
          D = ((AK-1.0E0)*AK-AL*(AL+1.0E0))*(AK+2.0E0)*D/
     *    (((AK+3.0E0)*(AK+2.0E0)-AL*(AL+1.0E0))*AK)
          BB = BB + A(K)*D
          AK = AK + 2.0E0
   40   CONTINUE
   50   C = 4.0E0*C*(AL+1.0E0)*(AL+1.0E0)/((AL+AL+3.0E0)
     *  *(AL+AL+2.0E0))
        B(L) = (AL+0.5E0)*BB
   60 CØNTINUE
   70 RETURN
      END
```

Algorithm 474

Bivariate Interpolation and Smooth Surface Fitting Based on Local Procedures [E2]

Hiroshi Akima (Recd. 30 Mar. 1972 and 3 Nov. 1972)
U.S. Department of Commerce, Office of Telecommunications, Institute for Telecommunication Sciences, Boulder, CO 80302

Key Words and Phrases: bivariate interpolation, interpolation, partial derivative, polynomial, smooth surface fitting
CR Categories: 5.13
Language: Fortran

Table I. An Example Set of Input Data

		Z(*IX*, *IY*)								
		IY = 1	2	3	4	5	6	7	8	9
IX	*X*(*IX*)	*Y*(*IY*) = 0.0	5.0	10.0	15.0	20.0	25.0	30.0	35.0	40.0
1	0.0	58.2	61.5	47.9	62.3	34.6	45.5	38.2	41.2	41.7
2	5.0	37.2	40.0	27.0	41.3	14.1	24.5	17.3	20.2	20.8
3	10.0	22.4	22.5	14.6	22.5	4.7	7.2	1.8	2.1	2.1
4	15.0	21.8	20.5	12.8	17.6	5.8	7.6	0.8	0.6	0.6
5	20.0	16.8	14.4	8.1	6.9	6.2	0.6	0.1	0.0	0.0
6	25.0	12.0	8.0	5.3	2.9	0.6	0.0	0.0	0.0	0.0
7	30.0	7.4	4.8	1.4	0.1	0.0	0.0	0.0	0.0	0.0
8	35.0	3.2	0.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0
9	40.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
10	45.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
11	50.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Description

Introduction. User information and Fortran listings are given on two subroutines, *ITPLBV* and *SFCFIT*. Each subroutine implements the method of smooth bivariate interpolation based on local procedures [3]. These subroutines are written in ANSI Standard Fortran [4].

Outline of the method. This method interpolates values of a single-valued smooth bivariate function $z = z(x,y)$ and fits a smooth surface to a set of values of the function given at grid points in an x-y plane. These grid points may be unevenly spaced.

The method is an extension of the method of univariate interpolation developed earlier by the author [1,2] and is likewise based on local procedures. It is designed to avoid excessive undulations between grid points.

This method is based on a piecewise function composed of a set of bicubic polynomials in x and y; a bicubic polynomial in x and y is a polynomial that has terms $x^\alpha y^\beta$, where $\alpha = 0, 1, 2, 3$ and $\beta = 0, 1, 2, 3$. Each polynomial is applicable to a rectangle in the x-y plane. In this method, three partial derivatives $\partial z/\partial x$, $\partial z/\partial y$, and $\partial^2 z/\partial x \partial y$ are determined at each data point locally by the coordinates of 13 data points, with the data point in question as the center, two data points on each side of it in the x and y directions, and one data point in each diagonal direction. Each bicubic polynomial corresponding to a rectangle in the x-y plane is determined by the values of the function and its three partial derivatives at four corner points of the rectangle.

When interpolation is made near or on the boundary of the defined range of z, the z values estimated at several grid points outside the range are used to determine the partial derivatives. In this method, this estimation is based on three data points in the x or y direction, the boundary point and two adjacent given data points.

The resulting surface of this method is invariant under a linear-scale transformation of the coordinate system; different scalings of the coordinates result in equivalent surfaces.

This method requires only straightforward procedures, not iterative solutions of equations with preassigned error tolerances, which are required by some methods. No problem concerning computational stability or convergence exists in application of this method.

The ITPLBV subroutine. This subroutine interpolates, from values of the function given at input grid points in an x-y plane and for a given set of points in the plane, the values of a single-valued bivariate function $z = z(x,y)$.

The entrance to this subroutine is achieved by

CALL ITPLBV (*IU*, *LX*, *LY*, *X*, *Y*, *Z*, *N*, *U*, *V*, *W*)

where the input parameters are

IU = logical unit number of standard output unit,

LX = number of input grid points in the x coordinate (must be two or greater),

LY = number of input grid points in the y coordinate (must be two or greater),

X = array of dimension *LX* storing the x coordinates of input grid points (in ascending order),

Y = array of dimension *LY* storing the y coordinates of input grid points (in ascending order),

Z = doubly-dimensioned array of dimension (*LX*,*LY*) storing the values of the function (z values) at input grid points,

N = number of points at which interpolation of the z value is desired (must be one or greater),

U = array of dimension *N* storing the x coordinates of desired points,

V = array of dimension *N* storing the y coordinates of desired points,

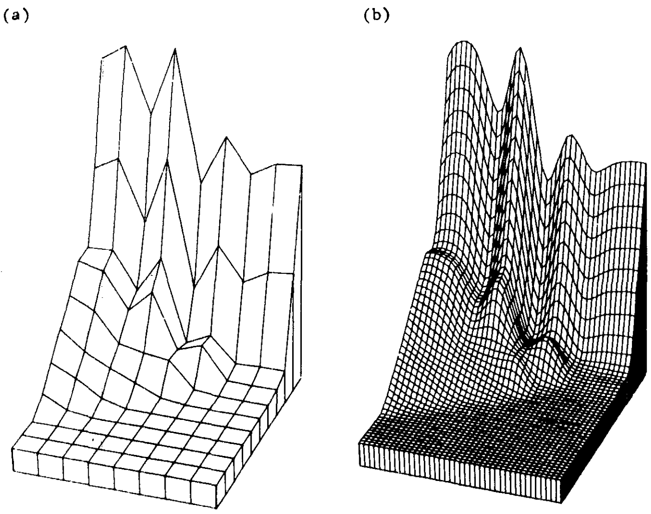
and the output parameter is

Editor's note: *Algorithm 474 described here is available on magnetic tape from the Department of Computer Science, University of Colorado, Boulder, CO 80302. The cost for the tape is $16.00 (U.S. and Canada) or $18.00 (elsewhere). If the user sends a small tape (wt. less than 1 lb.) the algorithm will be copied on it and returned to him at a charge of $10.00 (U.S. only). All orders are to be prepaid with checks payable to ACM Algorithms. The algorithm is recorded as one file of BCD 80 character card images at 556 B.P.I., even parity, on seven track tape. We will supply algorithm at a density of 800 B.P.I. if requested. Cards for algorithms are sequenced starting at 10 and incremented by 10. The sequence number is right justified in column 80. Although we will make every attempt to insure that the algorithm conforms to the description printed here, we cannot guarantee it, nor can we guarantee that the algorithm is correct.—L.D.F. and A.K.C.*

Table II. Output Data Obtained from the Input Data Given in Table I

		W(KX, KY)								
		KY = 1	2	3	4	5	6	7	8	9
KX	*U(KX)*	*V(KY)* = 0.0	2.5	5.0	7.5	10.0	12.5	15.0	17.5	20.0
1	0.0	58.20	61.70	61.50	55.01	47.90	54.82	62.30	48.13	34.60
2	2.5	47.08	50.59	50.40	43.75	36.45	43.73	51.62	36.94	22.94
3	5.0	37.20	40.31	40.00	33.81	27.00	33.86	41.30	27.41	14.10
4	7.5	28.22	30.35	29.90	24.80	19.22	25.03	31.18	19.15	7.49
5	10.0	22.40	23.29	22.50	18.75	14.60	18.45	22.50	13.47	4.70
6	12.5	21.91	22.19	21.02	17.47	13.67	16.39	19.28	12.14	5.23
7	15.0	21.80	21.82	20.50	16.74	12.80	15.07	17.60	11.66	5.80
8	17.5	19.28	18.98	17.48	13.78	10.33	10.92	11.79	9.12	6.12
9	20.0	16.80	16.05	14.40	10.96	8.10	7.40	6.90	6.57	6.20
10	22.5	14.39	12.86	11.12	8.73	6.69	5.61	4.65	3.94	3.49
11	25.0	12.00	9.79	8.00	6.58	5.30	4.10	2.90	1.71	0.60
12	27.5	9.68	7.77	6.15	4.71	3.29	2.05	1.15	0.60	0.17
13	30.0	7.40	6.18	4.80	3.07	1.40	0.45	0.10	0.03	0.00
14	32.5	5.24	3.86	2.57	1.34	0.35	0.04	0.01	0.00	0.00
15	35.0	3.20	1.68	0.70	0.20	0.00	0.00	0.00	0.00	0.00
16	37.5	1.09	0.41	0.08	−0.01	0.00	0.00	0.00	0.00	0.00
17	40.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
18	42.5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
19	45.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
20	47.5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
21	50.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		KY = 9	10	11	12	13	14	15	16	17
		V(KY) = 20.0	22.5	25.0	27.5	30.0	32.5	35.0	37.5	40.0
1	0.0	34.60	40.39	45.50	41.20	38.20	39.80	41.20	41.67	41.70
2	2.5	22.94	29.19	34.69	30.29	27.23	28.95	30.46	30.99	31.08
3	5.0	14.10	19.63	24.50	20.25	17.30	18.84	20.20	20.70	20.80
4	7.5	7.49	11.32	14.73	10.48	7.34	8.35	9.26	9.58	9.68
5	10.0	4.70	6.12	7.20	4.03	1.80	1.96	2.10	2.12	2.10
6	12.5	5.23	6.11	6.60	3.41	1.17	0.93	0.75	0.68	0.62
7	15.0	5.80	6.84	7.60	3.74	0.80	0.66	0.60	0.59	0.60
8	17.5	6.12	4.79	3.61	1.72	0.39	0.28	0.22	0.21	0.22
9	20.0	6.20	3.37	0.60	0.25	0.10	0.04	0.00	−0.01	0.00
10	22.5	3.49	1.77	0.16	0.06	0.02	0.01	0.00	−0.00	−0.00
11	25.0	0.60	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00
12	27.5	0.17	−0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00
13	30.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
14	32.5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
15	35.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
16	37.5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
17	40.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
18	42.5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
19	45.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
20	47.5	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
21	50.0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Fig. 1. Perspective representation of (a) the original data points given in Table I and of (b) the surface fitted by the *SFCFIT* subroutine with $LX = 11$, $LY = 9$, $MX = 5$, $MY = 5$, $NU = 51$, and $NV = 41$.



W = array of dimension *N* where the interpolated *z* values at desired points are to be displayed.

This subroutine occupies 1577 locations on the CDC-3800 computer. Computation time required for this subroutine on the same computer is approximately equal to: $1 + 3.0 * N$ msec for $LX = LY = 10$; $10 + 4.0 * N$ msec for $LX = LY = 100$.

When the function to be interpolated represents a periodic function of *x* and/or *y*, the input data to this subroutine should consist of the data that cover a whole period and two additional grid lines on each side of them.

The SFCFIT subroutine. This subroutine fits a smooth surface of a single-valued bivariate function $z = z(x,y)$ to a set of input data points given at input grid points in an *x*-*y* plane. It generates a set of output grid points by equally dividing the *x* and *y* coordinates in each interval between a pair of input grid points, interpolates the *z* value for the *x* and *y* values of each output grid points, and generates a set of output points consisting of input data points and the interpolated points.

The entrance to this subroutine is achieved by

CALL SFCFIT (IU, LX, LY, X, Y, Z, MX, MY, NU, NV, U, V, W)

where the input parameters are

IU = logical unit number of standard output unit,

LX = number of input grid points in the *x* coordinate (must be two or greater),

LY = number of input grid points in the *y* coordinate (must be two or greater),

X = array of dimension *LX* storing the *x* coordinates of input grid points (in ascending or descending order),

Y = array of dimension *LY* storing the *y* coordinates of input grid points (in ascending or descending order),

Z = doubly-dimensioned array of dimension (*LX*,*LY*) storing the values of the function at input grid points,

MX = number of subintervals between each pair of input grid points in the *x* coordinate (must be two or greater),

MY = number of subintervals between each pair of input grid points in the *y* coordinate (must be two or greater),

NU = number of output grid points in the *x* coordinate = $(LX-1)*MX+1$,

NV = number of output grid points in the *y* coordinate = $(LY-1)*MY+1$,

and the output parameters are

U = array of dimension *NU* where the *x* coordinates of output points are to be displayed,

V = array of dimension *NV* where the *y* coordinates of output points are to be displayed,

W = doubly-dimensioned array of dimension (*NU*,*NV*) where the *z* coordinates of output points are to be displayed,

This subroutine occupies 1333 locations on the CDC-3800 computer. Computation time required for this subroutine on the same computer is approximately

$(1.5 + (0.15 + 0.1 * MX) * MY) * LX * LY$ msec.

When the surface exhibits periodicity with respect to *x* and/or *Y*, the input data to this subroutine should consist of the data that cover a whole period and two additional grid lines on each side of them, and two intervals on each side be discarded from the set of output points.

Test results. All tests were performed on a CDC-3800 computer. An example is shown in Tables I and II. The *X*, *Y*, and *Z* values shown in Table I were given to the *SFCFIT* subroutine as input data with $LX = 11$, $LY = 9$, $MX = 2$, $MY = 2$, $NU = 21$, and $NV = 17$, and the *U*, *V*, and *W* values shown in Table II were obtained. Also, the data in Table I, together with each combination of the *U* and *V* values in Table II, were given to the *ITPLBV* subroutine with $LX = 11$, $LY = 9$, and $N = 1$, and the respective *W* value in Table II was obtained each time. Figure 1(a) depicts the original data points given in Table I, and Figure 1(b) the surface fitted by the *SFCFIT* subroutine with $LX = 11$, $LY = 9$, $MX = 5$, $MY = 5$, $NU = 51$, and $NV = 41$. This example demonstrates one of the properties of this method, that the resulting surface is free from excessive undulations.

Acknowledgments. The author expresses his deep appreciation to L. David Lewis, Rayner K. Rosich, and Jeanne M. Tucker of the U.S. Department of Commerce Boulder Laboratories for their critical review of this paper.

References

1. Akima, Hiroshi. A new method of interpolation and smooth curve fitting based on local procedures. *J. ACM 17*, 4 (Oct. 1970), 589–602.

2. Akima, Hiroshi. Algorithm 433, Interpolation and smooth curve fitting based on local procedures. *Comm. ACM 15*, 10 (Oct. 1972), 914–918.

3. Akima, Hiroshi. A method of bivariate interpolation and smooth surface fitting based on local procedures. *Comm. ACM 17*, 1 (Jan. 1974), 18–20.

4. ANSI Standard Fortran, Publication X3.9—1966. Amer. Nat. Standards Inst., New York. Also reproduced in W.P. Heising, History and summary of FORTRAN standardization development for the ASA. *Comm. ACM 7*,10 (Oct. 1964), 590–625.

Algorithm

```
      SUBROUTINE ITPLBV(IU, LX, LY, X, Y, Z, N, U, V, W)
C BIVARIATE INTERPOLATION
C THIS SUBROUTINE INTERPOLATES, FROM VALUES OF THE FUNCTION
C GIVEN AT INPUT GRID POINTS IN AN X-Y PLANE AND FOR A GIVEN
C SET OF POINTS IN THE PLANE, THE VALUES OF A SINGLE-VALUED
C BIVARIATE FUNCTION Z = Z(X,Y).
C THE METHOD IS BASED ON A PIECE-WISE FUNCTION COMPOSED OF
C A SET OF BICUBIC POLYNOMIALS IN X AND Y.  EACH POLYNOMIAL
C IS APPLICABLE TO A RECTANGLE OF THE INPUT GRID IN THE X-Y
C PLANE.  EACH POLYNOMIAL IS DETERMINED LOCALLY.
C THE INPUT PARAMETERS ARE
C IU  = LOGICAL UNIT NUMBER OF STANDARD OUTPUT UNIT
C LX  = NUMBER OF INPUT GRID POINTS IN THE X COORDINATE
C       (MUST BE 2 OR GREATER)
C LY  = NUMBER OF INPUT GRID POINTS IN THE Y COORDINATE
C       (MUST BE 2 OR GREATER)
C X   = ARRAY OF DIMENSION LX STORING THE X COORDINATES
C       OF INPUT GRID POINTS (IN ASCENDING ORDEP)
C Y   = ARRAY OF DIMENSION LY STORING THE Y COORDINATES
C       OF INPUT GRID POINTS (IN ASCENDING ORDER)
C Z   = DOUBLY-DIMENSIONED ARRAY OF DIMENSION (LX,LY)
C       STORING THE VALUES OF THE FUNCTION (Z VALUES)
C       AT INPUT GRID POINTS
C N   = NUMBER OF POINTS AT WHICH INTERPOLATION OF THE
C       Z VALUE IS DESIRED (MUST BE 1 OR GREATER)
C U   = ARRAY OF DIMENSION N STORING THE X COORDINATES
C       OF DESIRED POINTS
C V   = ARRAY OF DIMENSION N STORING THE Y COORDINATES
C       OF DESIRED POINTS
C THE OUTPUT PARAMETER IS
C W   = ARRAY OF DIMENSION N WHERE THE INTERPOLATED Z
C       VALUES AT DESIRED POINTS ARE TO BE DISPLAYED
C       SOME VARIABLES INTERNALLY USED ARE
C ZA  = DIVIDED DIFFERENCE OF Z WITH RESPECT TO X
C ZB  = DIVIDED DIFFERENCE OF Z WITH RESPECT TO Y
C ZAB = SECOND ORDER DIVIDED DIFFERENCE OF Z WITH
C       RESPECT TO X AND Y
C ZX  = PARTIAL DERIVATIVE OF Z WITH RESPECT TO X
C ZY  = PARTIAL DERIVATIVE OF Z WITH RESPECT TO Y
C ZXY = SECOND ORDER PARTIAL DERIVATIVE OF Z WITH
C       RESPECT TO X AND Y
C DECLARATION STATEMENTS
      DIMENSION X(LX), Y(LY), Z(LX,LY), U(N), V(N), W(N)
      DIMENSION ZA(5,2), ZB(2,5), ZAB(3,3), ZX(4,4), ZY(4,4),
     * ZXY(4,4)
      EQUIVALENCE (Z3A1,ZA(1)), (Z3A2,ZA(2)), (Z3A3,ZA(3)),
     * (Z3A4,ZA(4)), (Z3A5,ZA(5)), (Z4A1,ZA(6)), (Z4A2,ZA(7)),
     * (Z4A3,ZA(8)), (Z4A4,ZA(9)), (Z4A5,ZA(10)), (Z3B1,ZB(1)),
     * (Z3B2,ZB(3)), (Z3B3,ZB(5)), (Z3B4,ZB(7)), (Z3B5,ZB(9)),
     * (Z4B1,ZB(2)), (Z4B2,ZB(4)), (Z4B3,ZB(6)), (Z4B4,ZB(8)),
     * (Z4B5,ZB(10)), (ZA2B2,ZAB(1)), (ZA3B2,ZAB(2)),
     * (ZA4B2,ZAB(3)), (ZA2B3,ZAB(4)), (ZA3B3,ZAB(5)),
     * (ZA4B3,ZAB(6)), (ZA2B4,ZAB(7)), (ZA3B4,ZAB(8)),
     * (ZA4B4,ZAB(9)), (ZX33,ZX(6)), (ZX43,ZX(7)),
     * (ZX34,ZX(10)), (ZX44,ZX(11)), (ZY33,ZY(6)),
     * (ZY43,ZY(7)), (ZY34,ZY(10)), (ZY44,ZY(11)),
     * (ZXY33,ZXY(6)), (ZXY43,ZXY(7)), (ZXY34,ZXY(10)),
     * (ZXY44,ZXY(11)), (PØØ,Z33), (PØ1,ZY33), (P1Ø,ZX33),
     * (P11,ZXY33)
      EQUIVALENCE (LXØ,ZX(1)), (LXM1,ZX(4)), (LXM2,ZX(13)),
     * (LXP1,ZX(16)), (LYØ,ZY(1)), (LYM1,ZY(4)), (LYM2,ZY(13)),
     * (LYP1,ZY(16)), (IX,ZXY(1)), (IY,ZXY(4)), (IXPV,ZXY(13)),
     * (IYPV,ZXY(16)), (IMN,JX), (IMX,JY), (JXM2,JX1),
     * (JYM2,JY1), (UK,DX), (VK,DY), (A1,A5,B1,B5,ZX(2),A,QØ),
     * (A2,ZX(5),B,Q1), (A4,ZX(8),C,Q2), (B2,ZY(2),D,Q3),
     * (B4,ZY(14),E), (X2,ZX(3),A3SQ), (X4,ZX(9)), (X5,ZX(12)),
     * (Y2,ZX(14)), (Y4,ZY(3),B3SQ), (Y5,ZX(15),PØ2),
     * (Z23,ZY(5),PØ3), (Z24,ZY(8),P12), (Z32,ZY(9),P13),
     * (Z34,ZY(12),P2Ø), (Z35,ZY(15),P21), (Z42,ZXY(2),P22),
     * (Z43,ZXY(5),P23), (Z44,ZXY(3),P3Ø), (Z45,ZXY(8),P31),
     * (Z53,ZXY(9),P32), (Z54,ZXY(12),P33), (W2,WY2,W4),
     * (W3,WY3,W1,W5), (WX2,ZXY(14)), (WX3,ZXY(15))
C PRELIMINARY PROCESSING
C SETTING OF SOME INPUT PARAMETERS TO LOCAL VARIABLES
      IUØ = IU
      LXØ = LX
      LXM1 = LXØ - 1
      LXM2 = LXM1 - 1
      LXP1 = LXØ + 1
      LYØ = LY
      LYM1 = LYØ - 1
      LYM2 = LYM1 - 1
      LYP1 = LYØ + 1
      NØ = N
C ERROR CHECK
      IF (LXM2.LT.Ø) GO TO 71Ø
      IF (LYM2.LT.Ø) GO TO 72Ø
      IF (NØ.LT.1) GO TO 73Ø
      DO 1Ø IX=2,LXØ
        IF (X(IX-1)-X(IX)) 1Ø, 74Ø, 75Ø
   1Ø CONTINUE
      DO 2Ø IY=2,LYØ
        IF (Y(IY-1)-Y(IY)) 2Ø, 77Ø, 78Ø
   2Ø CONTINUE
C INITIAL SETTING OF PREVIOUS VALUES OF IX AND IY
      IXPV = Ø
      IYPV = Ø
C MAIN DO-LOOP
      DO 7ØØ K=1,NØ
        UK = U(K)
        VK = V(K)
C ROUTINES TO LOCATE THE DESIRED POINT
C TO FIND OUT THE IX VALUE FOR WHICH
C (U(K).GE.X(IX-1)).AND.(U(K).LT.X(IX))
        IF (LXM2.EQ.Ø) GO TO 8Ø
        IF (UK.GE.X(LXØ)) GO TO 7Ø
        IF (UK.LT.X(1)) GO TO 6Ø
        IMN = 2
        IMX = LXØ
   3Ø   IX = (IMN+IMX)/2
        IF (UK.GE.X(IX)) GO TO 4Ø
        IMX = IX
        GO TO 5Ø
   4Ø   IMN = IX + 1
   5Ø   IF (IMX.GT.IMN) GO TO 3Ø
        IX = IMX
        GO TO 9Ø
   6Ø   IX = 1
        GO TO 9Ø
   7Ø   IX = LXP1
        GO TO 9Ø
   8Ø   IX = 2
C TO FIND OUT THE IY VALUE FOR WHICH
C (V(K).GE.Y(IY-1)).AND.(V(K).LT.Y(IY))
   9Ø   IF (LYM2.EQ.Ø) GO TO 15Ø
        IF (VK.GE.Y(LYØ)) GO TO 14Ø
        IF (VK.LT.Y(1)) GO TO 13Ø
        IMN = 2
        IMX = LYØ
  1ØØ   IY = (IMN+IMX)/2
        IF (VK.GE.Y(IY)) GO TO 11Ø
        IMX = IY
        GO TO 12Ø
  11Ø   IMN = IY + 1
  12Ø   IF (IMX.GT.IMN) GO TO 1ØØ
        IY = IMX
        GO TO 16Ø
  13Ø   IY = 1
        GO TO 16Ø
  14Ø   IY = LYP1
        GO TO 16Ø
  15Ø   IY = 2
C TO CHECK IF THE DESIRED POINT IS IN THE SAME RECTANGLE
C AS THE PREVIOUS POINT.  IF YES, SKIP TO THE COMPUTATION
C OF THE POLYNOMIAL
  16Ø   IF (IX.EQ.IXPV .AND. IY.EQ.IYPV) GO TO 69Ø
        IXPV = IX
        IYPV = IY
C ROUTINES TO PICK UP NECESSARY X, Y, AND Z VALUES, TO
C COMPUTE THE ZA, ZB, AND ZAB VALUES, AND TO ESTIMATE THEM
C WHEN NECESSARY
        JX = IX
        IF (JX.EQ.1) JX = 2
        IF (JX.EQ.LXP1) JX = LXØ
        JY = IY
        IF (JY.EQ.1) JY = 2
        IF (JY.EQ.LYP1) JY = LYØ
        JXM2 = JX - 2
        JXML = JX - LXØ
        JYM2 = JY - 2
        JYML = JY - LYØ
C IN THE CORE AREA, I.E., IN THE RECTANGLE THAT CONTAINS
C THE DESIRED POINT
        X3 = X(JX-1)
        X4 = X(JX)
        A3 = 1.Ø/(X4-X3)
        Y3 = Y(JY-1)
        Y4 = Y(JY)
        B3 = 1.Ø/(Y4-Y3)
        Z33 = Z(JX-1,JY-1)
        Z43 = Z(JX,JY-1)
        Z34 = Z(JX-1,JY)
        Z44 = Z(JX,JY)
        Z3A3 = (Z43-Z33)*A3
        Z4A3 = (Z44-Z34)*A3
        Z3B3 = (Z34-Z33)*B3
        Z4B3 = (Z44-Z43)*B3
        ZA3B3 = (Z4B3-Z3B3)*A3
C IN THE X DIRECTION
        IF (LXM2.EQ.Ø) GO TO 23Ø
        IF (JXM2.EQ.Ø) GO TO 17Ø
        X2 = X(JX-2)
        A2 = 1.Ø/(X3-X2)
        Z23 = Z(JX-2,JY-1)
        Z24 = Z(JX-2,JY)
        Z3A2 = (Z33-Z23)*A2
        Z4A2 = (Z34-Z24)*A2
        IF (JXML.EQ.0) GO TO 180
  170   X5 = X(JX+1)
        A4 = 1.0/(X5-X4)
        Z53 = Z(JX+1,JY-1)
        Z54 = Z(JX+1,JY)
        Z3A4 = (Z53-Z43)*A4
        Z4A4 = (Z54-Z44)*A4
        IF (JXM2.NE.0) GO TO 190
        Z3A2 = Z3A3 + Z3A3 - Z3A4
        Z4A2 = Z4A3 + Z4A3 - Z4A4
        GO TO 190
  180   Z3A4 = Z3A3 + Z3A3 - Z3A2
        Z4A4 = Z4A3 + Z4A3 - Z4A2
  190   ZA2B3 = (Z4A2-Z3A2)*B3
        ZA4B3 = (Z4A4-Z3A4)*B3
        IF (JX.LE.3) GO TO 200
        A1 = 1.0/(X2-X(JX-3))
        Z3A1 = (Z23-Z(JX-3,JY-1))*A1
        Z4A1 = (Z24-Z(JX-3,JY))*A1
        GO TO 210
  200   Z3A1 = Z3A2 + Z3A2 - Z3A3
        Z4A1 = Z4A2 + Z4A2 - Z4A3
  210   IF (JX.GE.LXM1) GO TO 220
        A5 = 1.0/(X(JX+2)-X5)
        Z3A5 = (Z(JX+2,JY-1)-Z53)*A5
        Z4A5 = (Z(JX+2,JY)-Z54)*A5
        GO TO 240
  220   Z3A5 = Z3A4 + Z3A4 - Z3A3
        Z4A5 = Z4A4 + Z4A4 - Z4A3
        GO TO 240
  230   Z3A2 = Z3A3
        Z4A2 = Z4A3
        GO TO 180
C IN THE Y DIRECTION
  240   IF (LYM2.EQ.0) GO TO 310
        IF (JYM2.EQ.0) GO TO 250
        Y2 = Y(JY-2)
        B2 = 1.0/(Y3-Y2)
        Z32 = Z(JX-1,JY-2)
        Z42 = Z(JX,JY-2)
        Z3B2 = (Z33-Z32)*B2
        Z4B2 = (Z43-Z42)*B2
        IF (JYML.EQ.0) GO TO 260
  250   Y5 = Y(JY+1)
        B4 = 1.0/(Y5-Y4)
        Z35 = Z(JX-1,JY+1)
        Z45 = Z(JX,JY+1)
        Z3B4 = (Z35-Z34)*B4
        Z4B4 = (Z45-Z44)*B4
        IF (JYM2.NE.0) GO TO 270
        Z3B2 = Z3B3 + Z3B3 - Z3B4
        Z4B2 = Z4B3 + Z4B3 - Z4B4
        GO TO 270
  260   Z3B4 = Z3B3 + Z3B3 - Z3B2
        Z4B4 = Z4B3 + Z4B3 - Z4B2
  270   ZA3B2 = (Z4B2-Z3B2)*A3
        ZA3B4 = (Z4B4-Z3B4)*A3
        IF (JY.LE.3) GO TO 280
        B1 = 1.0/(Y2-Y(JY-3))
        Z3B1 = (Z32-Z(JX-1,JY-3))*B1
        Z4B1 = (Z42-Z(JX,JY-3))*B1
        GO TO 290
  280   Z3B1 = Z3B2 + Z3B2 - Z3B3
        Z4B1 = Z4B2 + Z4B2 - Z4B3
  290   IF (JY.GE.LYM1) GO TO 300
        B5 = 1.0/(Y(JY+2)-Y5)
        Z3B5 = (Z(JX-1,JY+2)-Z35)*B5
        Z4B5 = (Z(JX,JY+2)-Z45)*B5
        GO TO 320
  300   Z3B5 = Z3B4 + Z3B4 - Z3B3
        Z4B5 = Z4B4 + Z4B4 - Z4B3
        GO TO 320
  310   Z3B2 = Z3B3
        Z4B2 = Z4B3
        GO TO 260
C IN THE DIAGONAL DIRECTIONS
  320   IF (LXM2.EQ.0) GO TO 400
        IF (LYM2.EQ.0) GO TO 410
        IF (JXML.EQ.0) GO TO 350
        IF (JYM2.EQ.0) GO TO 330
        ZA4B2 = ((Z53-Z(JX+1,JY-2))*B2-Z4B2)*A4
        IF (JYML.EQ.0) GO TO 340
  330   ZA4B4 = ((Z(JX+1,JY+1)-Z54)*B4-Z4B4)*A4
        IF (JYM2.NE.0) GO TO 380
        ZA4B2 = ZA4B3 + ZA4B3 - ZA4B4
        GO TO 380
  340   ZA4B4 = ZA4B3 + ZA4B3 - ZA4B2
        GO TO 380
  350   IF (JYM2.EQ.0) GO TO 360
        ZA2B2 = (Z3B2-(Z23-Z(JX-2,JY-2))*B2)*A2
        IF (JYML.EQ.0) GO TO 370
  360   ZA2B4 = (Z3B4-(Z(JX-2,JY+1)-Z24)*B4)*A2
        IF (JYM2.NE.0) GO TO 390
        ZA2B2 = ZA2B3 + ZA2B3 - ZA2B4
        GO TO 390
  370   ZA2B4 = ZA2B3 + ZA2B3 - ZA2B2
        GO TO 390
  380   IF (JXM2.NE.0) GO TO 350
        ZA2B2 = ZA3B2 + ZA3B2 - ZA4B2
        ZA2B4 = ZA3B4 + ZA3B4 - ZA4B4
        GO TO 420
  390   IF (JXML.NE.0) GO TO 420
        ZA4B2 = ZA3B2 + ZA3B2 - ZA2B2
        ZA4B4 = ZA3B4 + ZA3B4 - ZA2B4
        GO TO 420
  400   ZA2B2 = ZA3B2
        ZA4B2 = ZA3B2
        ZA2B4 = ZA3B4
        ZA4B4 = ZA3B4
        GO TO 420
  410   ZA2B2 = ZA2B3
        ZA2B4 = ZA2B3
        ZA4B2 = ZA4B3
        ZA4B4 = ZA4B3
C NUMERICAL DIFFERENTIATION   ---   TO DETERMINE PARTIAL
C DERIVATIVES ZX, ZY, AND ZXY AS WEIGHTED MEANS OF DIVIDED
C DIFFERENCES ZA, ZB, AND ZAB, RESPECTIVELY
  420   DO 480 JY=2,3
          DO 470 JX=2,3
            W2 = ABS(ZA(JX+2,JY-1)-ZA(JX+1,JY-1))
            W3 = ABS(ZA(JX,JY-1)-ZA(JX-1,JY-1))
            SW = W2 + W3
            IF (SW.EQ.0.0) GO TO 430
            WX2 = W2/SW
            WX3 = W3/SW
            GO TO 440
  430       WX2 = 0.5
            WX3 = 0.5
  440       ZX(JX,JY) = WX2*ZA(JX,JY-1) + WX3*ZA(JX+1,JY-1)
            W2 = ABS(ZB(JX-1,JY+2)-ZB(JX-1,JY+1))
            W3 = ABS(ZB(JX-1,JY)-ZB(JX-1,JY-1))
            SW = W2 + W3
            IF (SW.EQ.0.0) GO TO 450
            WY2 = W2/SW
            WY3 = W3/SW
            GO TO 460
  450       WY2 = 0.5
            WY3 = 0.5
  460       ZY(JX,JY) = WY2*ZB(JX-1,JY) + WY3*ZB(JX-1,JY+1)
            ZXY(JX,JY) =
     *      WY2*(WX2*ZAB(JX-1,JY-1)+WX3*ZAB(JX,JY-1)) +
     *      WY3*(WX2*ZAB(JX-1,JY)+WX3*ZAB(JX,JY))
  470     CONTINUE
  480   CONTINUE
C WHEN (U(K).LT.X(1)).OR.(U(K).GT.X(LX))
        IF (IX.EQ.LXP1) GO TO 530
        IF (IX.NE.1) GO TO 590
        W2 = A4*(3.0*A3+A4)
        W1 = 2.0*A3*(A3-A4) + W2
        DO 500 JY=2,3
          ZX(1,JY) = (W1*ZA(1,JY-1)+W2*ZA(2,JY-1))/(W1+W2)
          ZY(1,JY) = ZY(2,JY) + ZY(2,JY) - ZY(3,JY)
          ZXY(1,JY) = ZXY(2,JY) + ZXY(2,JY) - ZXY(3,JY)
          DO 490 JX1=2,3
            JX = 5 - JX1
            ZX(JX,JY) = ZX(JX-1,JY)
            ZY(JX,JY) = ZY(JX-1,JY)
            ZXY(JX,JY) = ZXY(JX-1,JY)
  490     CONTINUE
  500   CONTINUE
        X3 = X3 - 1.0/A4
        Z33 = Z33 - Z3A2/A4
        DO 510 JY=1,5
          ZB(2,JY) = ZB(1,JY)
  510   CONTINUE
        DO 520 JY=2,4
          ZB(1,JY) = ZB(1,JY) - ZAB(1,JY-1)/A4
  520   CONTINUE
        A3 = A4
        JX = 1
        GO TO 570
  530   W4 = A2*(3.0*A3+A2)
        W5 = 2.0*A3*(A3-A2) + W4
        DO 550 JY=2,3
          ZX(4,JY) = (W4*ZA(4,JY-1)+W5*ZA(5,JY-1))/(W4+W5)
          ZY(4,JY) = ZY(3,JY) + ZY(3,JY) - ZY(2,JY)
          ZXY(4,JY) = ZXY(3,JY) + ZXY(3,JY) - ZXY(2,JY)
          DO 540 JX=2,3
            ZX(JX,JY) = ZX(JX+1,JY)
            ZY(JX,JY) = ZY(JX+1,JY)
            ZXY(JX,JY) = ZXY(JX+1,JY)
  540     CONTINUE
  550   CONTINUE
        X3 = X4
        Z33 = Z43
        DO 560 JY=1,5
          ZB(1,JY) = ZB(2,JY)
  560   CONTINUE
        A3 = A2
        JX = 3
  570   ZA(3,1) = ZA(JX+1,1)
        DO 580 JY=1,3
          ZAB(2,JY) = ZAB(JX,JY)
  580   CONTINUE
C WHEN (V(K).LT.Y(1)).OR.(V(K).GT.Y(LY))
  590   IF (IY.EQ.LYP1) GO TO 630
        IF (IY.NE.1) GO TO 680
        W2 = B4*(3.0*B3+B4)
        W1 = 2.0*B3*(B3-B4) + W2
        DO 620 JX=2,3
          IF (JX.EQ.3 .AND. IX.EQ.LXP1) GO TO 600
          IF (JX.EQ.2 .AND. IX.EQ.1) GO TO 600
          ZY(JX,1) = (W1*ZB(JX-1,1)+W2*ZB(JX-1,2))/(W1+W2)
          ZX(JX,1) = ZX(JX,2) + ZX(JX,2) - ZX(JX,3)
          ZXY(JX,1) = ZXY(JX,2) + ZXY(JX,2) - ZXY(JX,3)
  600     DO 610 JY1=2,3
            JY = 5 - JY1
            ZY(JX,JY) = ZY(JX,JY-1)
            ZX(JX,JY) = ZX(JX,JY-1)
            ZXY(JX,JY) = ZXY(JX,JY-1)
  610     CONTINUE
  620   CONTINUE
        Y3 = Y3 - 1.0/B4
        Z33 = Z33 - Z3B2/B4
        Z3A3 = Z3A3 - ZA3B2/B4
        Z3B3 = Z3B2
        ZA3B3 = ZA3B2
        B3 = B4
        GO TO 670
  630   W4 = B2*(3.0*B3+B2)
        W5 = 2.0*B3*(B3-B2) + W4
        DO 660 JX=2,3
```

```
            IF (JX.EQ.3 .AND. IX.EQ.LXP1) GO TO 64Ø
            IF (JX.EQ.2 .AND. IX.EQ.1) GO TO 64Ø
            ZY(JX,4) = (W4*ZB(JX-1,4)+W5*ZB(JX-1,5))/(W4+W5)
            ZX(JX,4) = ZX(JX,3) + ZX(JX,3) - ZX(JX,2)
            ZXY(JX,4) = ZXY(JX,3) + ZXY(JX,3) - ZXY(JX,2)
  64Ø       DO 65Ø JY=2,3
              ZY(JX,JY) = ZY(JX,JY+1)
              ZX(JX,JY) = ZX(JX,JY+1)
              ZXY(JX,JY) = ZXY(JX,JY+1)
  65Ø       CONTINUE
  66Ø     CONTINUE
          Y3 = Y4
          Z33 = Z33 + Z3B3/B3
          Z3A3 = Z3A3 + ZA3B3/B3
          Z3B3 = Z3B4
          ZA3B3 = ZA3B4
          B3 = B2
  67Ø     IF (IX.NE.1 .AND. IX.NE.LXP1) GO TO 68Ø
          JX = IX/LXP1 + 2
          JX1 = 5 - JX
          JY = IY/LYP1 + 2
          JY1 = 5 - JY
          ZX(JX,JY) = ZX(JX1,JY) + ZX(JX,JY1) - ZX(JX1,JY1)
          ZY(JX,JY) = ZY(JX1,JY) + ZY(JX,JY1) - ZY(JX1,JY1)
          ZXY(JX,JY) = ZXY(JX1,JY) + ZXY(JX,JY1) - ZXY(JX1,JY1)
C DETERMINATION OF THE COEFFICIENTS OF THE POLYNOMIAL
  68Ø     ZX3B3 = (ZX34-ZX33)*B3
          ZX4B3 = (ZX44-ZX43)*B3
          ZY3A3 = (ZY43-ZY33)*A3
          ZY4A3 = (ZY44-ZY34)*A3
          A = ZA3B3 - ZX3B3 - ZY3A3 + ZXY33
          B = ZX4B3 - ZX3B3 - ZXY43 + ZXY33
          C = ZY4A3 - ZY3A3 - ZXY34 + ZXY33
          D = ZXY44 - ZXY43 - ZXY34 + ZXY33
          E = A + A - B - C
          A3SQ = A3*A3
          B3SQ = B3*B3
          PØ2 = (2.Ø*(Z3B3-ZY33)+Z3B3-ZY34)*B3
          PØ3 = (-2.Ø*Z3B3+ZY34+ZY33)*B3SQ
          P12 = (2.Ø*(ZX3B3-ZXY33)+ZX3B3-ZXY34)*B3
          P13 = (-2.Ø*ZX3B3+ZXY34+ZXY33)*B3SQ
          P2Ø = (2.Ø*(Z3A3-ZX33)+Z3A3-ZX43)*A3
          P21 = (2.Ø*(ZY3A3-ZXY33)+ZY3A3-ZXY43)*A3
          P22 = (3.Ø*(A+E)+D)*A3*B3
          P23 = (-3.Ø*E-B-D)*A3*B3SQ
          P3Ø = (-2.Ø*Z3A3+ZX43+ZX33)*A3SQ
          P31 = (-2.Ø*ZY3A3+ZXY43+ZXY33)*A3SQ
          P32 = (-3.Ø*E-C-D)*B3*A3SQ
          P33 = (D+E+E)*A3SQ*B3SQ
C COMPUTATION OF THE POLYNOMIAL
  69Ø     DY = VK - Y3
          QØ = PØØ + DY*(PØ1+DY*(PØ2+DY*PØ3))
          Q1 = P1Ø + DY*(P11+DY*(P12+DY*P13))
          Q2 = P2Ø + DY*(P21+DY*(P22+DY*P23))
          Q3 = P3Ø + DY*(P31+DY*(P32+DY*P33))
          DX = UK - X3
          W(K) = QØ + DX*(Q1+DX*(Q2+DX*Q3))
  7ØØ CONTINUE
C NORMAL EXIT
      RETURN
C ERROR EXIT
  71Ø WRITE (IUØ,99999)
      GO TO 8ØØ
  72Ø WRITE (IUØ,99998)
      GO TO 8ØØ
  73Ø WRITE (IUØ,99997)
      GO TO 8ØØ
  74Ø WRITE (IUØ,99996)
      GO TO 76Ø
  75Ø WRITE (IUØ,99995)
  76Ø WRITE (IUØ,99994) IX, X(IX)
      GO TO 8ØØ
  77Ø WRITE (IUØ,99993)
      GO TO 79Ø
  78Ø WRITE (IUØ,99992)
  79Ø WRITE (IUØ,99991) IY, Y(IY)
  8ØØ WRITE (IUØ,9999Ø) LXØ, LYØ, NØ
      RETURN
C FORMAT STATEMENTS
99999 FORMAT(1X/23H  ***   LX = 1 OR LESS./)
99998 FORMAT(1X/23H  ***   LY = 1 OR LESS./)
99997 FORMAT(1X/22H  ***   N = Ø OR LESS./)
99996 FORMAT(1X/27H  ***   IDENTICAL X VALUES./)
99995 FORMAT(1X/33H  ***   X VALUES OUT OF SEQUENCE./)
99994 FORMAT(7H   IX =, I6, 1ØX, 7HX(IX) =, E12.3)
99993 FORMAT(1X/27H  ***   IDENTICAL Y VALUES./)
99992 FORMAT(1X/33H  ***   Y VALUES OUT OF SEQUENCE./)
99991 FORMAT(7H   IY =, I6, 1ØX, 7HY(IY) =, E12.3)
9999Ø FORMAT(7H   LX =, I6, 1ØX, 4HLY =, I6, 1ØX, 3HN =, I7/
     * 36H ERROR DETECTED IN ROUTINE    ITPLBV)
      END
```

```
      SUBROUTINE SFCFIT(IU, LX, LY, X, Y, Z, MX, MY, NU, NV, U,
     * V, W)
C SMOOTH SURFACE FITTING
C THIS SUBROUTINE FITS A SMOOTH SURFACE OF A SINGLE-VALUED
C BIVARIATE FUNCTION Z = Z(X,Y) TO A SET OF INPUT DATA
C POINTS GIVEN AT INPUT GRID POINTS IN AN X-Y PLANE.  IT
C GENERATES A SET OF OUTPUT GRID POINTS BY EQUALLY DIVIDING
C THE X AND Y COORDINATES IN EACH INTERVAL BETWEEN A PAIR
C OF INPUT GRID POINTS, INTERPOLATES THE Z VALUE FOR THE
C X AND Y VALUES OF EACH OUTPUT GRID POINT, AND GENERATES
C A SET OF OUTPUT POINTS CONSISTING OF INPUT DATA POINTS
C AND THE INTERPOLATED POINTS.
C THE METHOD IS BASED ON A PIECE-WISE FUNCTION COMPOSED OF
C A SET OF BICUBIC POLYNOMIALS IN X AND Y.  EACH POLYNOMIAL
C IS APPLICABLE TO A RECTANGLE OF THE INPUT GRID IN THE X-Y
C PLANE.  EACH POLYNOMIAL IS DETERMINED LOCALLY.
C THE INPUT PARAMETERS ARE
C IU  = LOGICAL UNIT NUMBER OF STANDARD OUTPUT UNIT
C LX  = NUMBER OF INPUT GRID POINTS IN THE X COORDINATE
C       (MUST BE 2 OR GREATER)
C LY  = NUMBER OF INPUT GRID POINTS IN THE Y COORDINATE
C       (MUST BE 2 OR GREATER)
C X   = ARRAY OF DIMENSION LX STORING THE X COORDINATES
C       OF INPUT GRID POINTS (IN ASCENDING OR DESCENDING
C       ORDER)
C Y   = ARRAY OF DIMENSION LY STORING THE Y COORDINATES
C       OF INPUT GRID POINTS (IN ASCENDING OR DESCENDING
C       ORDER)
C Z   = DOUBLY-DIMENSIONED ARRAY OF DIMENSION (LX,LY)
C       STORING THE VALUES OF THE FUNCTION AT INPUT
C       GRID POINTS
C MX  = NUMBER OF SUBINTERVALS BETWEEN EACH PAIR OF
C       INPUT GRID POINTS IN THE X COORDINATE
C       (MUST BE 2 OR GREATER)
C MY  = NUMBER OF SUBINTERVALS BETWEEN EACH PAIR OF
C       INPUT GRID POINTS IN THE Y COORDINATE
C       (MUST BE 2 OR GREATER)
C NU  = NUMBER OF OUTPUT GRID POINTS IN THE X COORDINATE
C     = (LX-1)*MX+1
C NV  = NUMBER OF OUTPUT GRID POINTS IN THE Y COORDINATE
C     = (LY-1)*MY+1
C THE OUTPUT PARAMETERS ARE
C U   = ARRAY OF DIMENSION NU WHERE THE X COORDINATES OF
C       OUTPUT POINTS ARE TO BE DISPLAYED
C V   = ARRAY OF DIMENSION NV WHERE THE Y COORDINATES OF
C       OUTPUT POINTS ARE TO BE DISPLAYED
C W   = DOUBLY-DIMENSIONED ARRAY OF DIMENSION (NU,NV)
C       WHERE THE Z COORDINATES OF OUTPUT POINTS ARE TO
C       BE DISPLAYED
C SOME VARIABLES INTERNALLY USED ARE
C ZA  = DIVIDED DIFFERENCE OF Z WITH RESPECT TO X
C ZB  = DIVIDED DIFFERENCE OF Z WITH RESPECT TO Y
C ZAB = SECOND ORDER DIVIDED DIFFERENCE OF Z WITH
C       RESPECT TO X AND Y
C ZX  = PARTIAL DERIVATIVE OF Z WITH RESPECT TO X
C ZY  = PARTIAL DERIVATIVE OF Z WITH RESPECT TO Y
C ZXY = SECOND ORDER PARTIAL DERIVATIVE OF Z WITH
C       RESPECT TO X AND Y
C DECLARATION STATEMENTS
      DIMENSION X(LX), Y(LY), Z(LX,LY), U(NU), V(NV), W(NU,NV)
      DIMENSION ZA(4,2), ZB(5), ZAB(2,3), ZX(2), ZY(2), ZXY(2)
      EQUIVALENCE (Z3A2,ZA(1)), (Z3A3,ZA(2)), (Z3A4,ZA(3)),
     * (Z3A5,ZA(4)), (Z4A2,ZA(5)), (Z4A3,ZA(6)), (Z4A4,ZA(7)),
     * (Z4A5,ZA(8)), (Z4B1,ZB(1)), (Z4B2,ZB(2)), (Z4B3,ZB(3)),
     * (Z4B4,ZB(4)), (Z4B5,ZB(5)), (ZA3B2,ZAB(1)),
     * (ZA4B2,ZAB(2)), (ZA3B3,ZAB(3)), (ZA4B3,ZAB(4)),
     * (ZA3B4,ZAB(5)), (ZA4B4,ZAB(6)), (ZX43,ZX(1)),
     * (ZX44,ZX(2)), (ZY43,ZY(1)), (ZY44,ZY(2)),
     * (ZXY43,ZXY(1)), (ZXY44,ZXY(2)), (PØØ,Z33), (PØ1,ZY33),
     * (P1Ø,ZX33), (P11,ZXY33)
      EQUIVALENCE (IXM1,JX), (IXML,JY), (DU,DV,DX,DY),
     * (FMX,RMX,FMY,RMY,SW,E), (W2,WY2,A,QØ), (W3,WY3,B,Q1),
     * (WX2,C,Q2), (WX3,D,Q3), (Z3A2,PØ2), (Z4A2,PØ3),
     * (Z4B1,P12), (Z4B2,P13), (Z4B4,P2Ø), (Z4B5,P21),
     * (ZA3B2,P22), (ZA3B4,P23)
C PRELIMINARY PROCESSING
C SETTING OF SOME INPUT PARAMETERS TO LOCAL VARIABLES
      IUØ = IU
      LXØ = LX
      LXM1 = LXØ - 1
      LXM2 = LXM1 - 1
      LYØ = LY
      LYM1 = LYØ - 1
      LYM2 = LYM1 - 1
      MXØ = MX
      MXP1 = MXØ + 1
      MXM1 = MXØ - 1
      MYØ = MY
      MYP1 = MYØ + 1
      MYM1 = MYØ - 1
      NUØ = NU
      NVØ = NV
C ERROR CHECK
      IF (LXM2.LT.Ø) GO TO 4ØØ
      IF (LYM2.LT.Ø) GO TO 41Ø
      IF (MXM1.LE.Ø) GO TO 42Ø
      IF (MYM1.LE.Ø) GO TO 43Ø
      IF (NUØ.NE.LXM1*MXØ+1) GO TO 44Ø
      IF (NVØ.NE.LYM1*MYØ+1) GO TO 45Ø
      IX = 2
      IF (X(1)-X(2)) 1Ø, 46Ø, 3Ø
   1Ø DO 2Ø IX=3,LXØ
        IF (X(IX-1)-X(IX)) 2Ø, 46Ø, 47Ø
   2Ø CONTINUE
      GO TO 5Ø
   3Ø DO 4Ø IX=3,LXØ
        IF (X(IX-1)-X(IX)) 47Ø, 46Ø, 4Ø
   4Ø CONTINUE
   5Ø IY = 2
      IF (Y(1)-Y(2)) 6Ø, 49Ø, 8Ø
   6Ø DO 7Ø IY=3,LYØ
        IF (Y(IY-1)-Y(IY)) 7Ø, 49Ø, 5ØØ
   7Ø CONTINUE
      GO TO 1ØØ
   8Ø DO 9Ø IY=3,LYØ
        IF (Y(IY-1)-Y(IY)) 5ØØ, 49Ø, 9Ø
   9Ø CONTINUE
C COMPUTATION OF THE U ARRAY
  1ØØ FMX = MXØ
      RMX = 1.Ø/FMX
      KU = 1
      X4 = X(1)
      U(1) = X4
      DO 12Ø IX=2,LXØ
        X3 = X4
        X4 = X(IX)
        DU = (X4-X3)*RMX
        DO 11Ø JX=1,MXM1
          KU = KU + 1
          U(KU) = U(KU-1) + DU
  11Ø   CONTINUE
        KU = KU + 1
        U(KU) = X4
  12Ø CONTINUE
```

```
C COMPUTATION OF THE V ARRAY
      FMY = MYØ
      RMY = 1.Ø/FMY
      KV = 1
      Y4 = Y(1)
      V(1) = Y4
      DO 14Ø IY=2,LYØ
        Y3 = Y4
        Y4 = Y(IY)
        DV = (Y4-Y3)*RMY
        DO 13Ø JY=1,MYM1
          KV = KV + 1
          V(KV) = V(KV-1) + DV
  13Ø   CONTINUE
        KV = KV + 1
        V(KV) = Y4
  14Ø CONTINUE
C MAIN DO-LOOPS
      JYMX = MYØ
      KVØ = Ø
      DO 39Ø IY=2,LYØ
        IYM2 = IY - 2
        IYM3 = IYM2 - 1
        IYML = IY - LYØ
        IYML1 = IYML + 1
        IX6 = Ø
        IF (IYML.EQ.Ø) JYMX = MYP1
        JXMX = MXØ
        KUØ = Ø
        DO 38Ø IX=1,LXØ
          IXM1 = IX - 1
          IXML = IX - LXØ
          IF (IXML.EQ.Ø) JXMX = MXP1
C ROUTINES TO PICK UP NECESSARY X, Y, AND Z VALUES, TO
C COMPUTE THE ZA, ZB, AND ZAB VALUES, AND TO ESTIMATE THEM
C WHEN NECESSARY
C PRELIMINARY WHEN IX.EQ.1
          IF (IXM1.NE.Ø) GO TO 15Ø
          Y3 = Y(IY-1)
          Y4 = Y(IY)
          B3 = 1.Ø/(Y4-Y3)
          B3SQ = B3*B3
          IF (IYM2.GT.Ø) B2 = 1.Ø/(Y3-Y(IY-2))
          IF (IYM3.GT.Ø) B1 = 1.Ø/(Y(IY-2)-Y(IY-3))
          IF (IYML.LT.Ø) B4 = 1.Ø/(Y(IY+1)-Y4)
          IF (IYML1.LT.Ø) B5 = 1.Ø/(Y(IY+2)-Y(IY+1))
          GO TO 18Ø
C TO SAVE THE OLD VALUES
  15Ø     Z3A2 = Z3A3
          Z4A2 = Z4A3
          X3 = X4
          Z33 = Z43
          Z3B3 = Z4B3
          A3 = A4
          A3SQ = A3*A3
          Z3A3 = Z3A4
          Z4A3 = Z4A4
          ZA3B2 = ZA4B2
          ZA3B3 = ZA4B3
          ZA3B4 = ZA4B4
  16Ø     X4 = X5
          Z43 = Z53
          Z4B1 = Z5B1
          Z4B2 = Z5B2
          Z4B3 = Z5B3
          Z4B4 = Z5B4
          Z4B5 = Z5B5
          A4 = A5
          Z3A4 = Z3A5
          Z4A4 = Z4A5
          ZA4B2 = ZA5B2
          ZA4B3 = ZA5B3
          ZA4B4 = ZA5B4
  17Ø     X5 = X6
          Z53 = Z63
          Z54 = Z64
          Z5B1 = Z6B1
          Z5B2 = Z6B2
          Z5B3 = Z6B3
          Z5B4 = Z6B4
          Z5B5 = Z6B5
C TO COMPUTE THE ZA, ZB, AND ZAB VALUES AND
C TO ESTIMATE THE ZB VALUES
C WHEN (IY.LE.3).OR.(IY.GE.LY-1)
  18Ø     IX6 = IX6 + 1
          IF (IX6.GT.LXØ) GO TO 26Ø
          X6 = X(IX6)
          Z63 = Z(IX6,IY-1)
          Z64 = Z(IX6,IY)
          Z6B3 = (Z64-Z63)*B3
          IF (LYM2.EQ.Ø) GO TO 2ØØ
          IF (IYM2.EQ.Ø) GO TO 19Ø
          Z62 = Z(IX6,IY-2)
          Z6B2 = (Z63-Z62)*B2
          IF (IYML.NE.Ø) GO TO 19Ø
          Z6B4 = Z6B3 + Z6B3 - Z6B2
          GO TO 21Ø
  19Ø     Z65 = Z(IX6,IY+1)
          Z6B4 = (Z65-Z64)*B4
          IF (IYM2.NE.Ø) GO TO 21Ø
          Z6B2 = Z6B3 + Z6B3 - Z6B4
          GO TO 21Ø
  2ØØ     Z6B2 = Z6B3
          Z6B4 = Z6B3
  21Ø     IF (IYM3.LE.Ø) GO TO 22Ø
          Z6B1 = (Z62-Z(IX6,IY-3))*B1
          GO TO 23Ø
  22Ø     Z6B1 = Z6B2 + Z6B2 - Z6B3
  23Ø     IF (IYML1.GE.Ø) GO TO 24Ø
          Z6B5 = (Z(IX6,IY+2)-Z65)*B5
          GO TO 25Ø
  24Ø     Z6B5 = Z6B4 + Z6B4 - Z6B3
  25Ø     IF (IX6.EQ.1) GO TO 17Ø
          A5 = 1.Ø/(X6-X5)
          Z3A5 = (Z63-Z53)*A5
          Z4A5 = (Z64-Z54)*A5
          ZA5B2 = (Z6B2-Z5B2)*A5
          ZA5B3 = (Z6B3-Z5B3)*A5
          ZA5B4 = (Z6B4-Z5B4)*A5
          IF (IX6.EQ.2) GO TO 16Ø
          GO TO 28Ø
C TO ESTIMATE THE ZA AND ZAB VALUES
C WHEN (IX.GE.LX-1).AND.(LX.GT.2)
  26Ø     IF (LXM2.EQ.Ø) GO TO 27Ø
          Z3A5 = Z3A4 + Z3A4 - Z3A3
          Z4A5 = Z4A4 + Z4A4 - Z4A3
          IF (IXML.EQ.Ø) GO TO 29Ø
          ZA5B2 = ZA4B2 + ZA4B2 - ZA3B2
          ZA5B3 = ZA4B3 + ZA4B3 - ZA3B3
          ZA5B4 = ZA4B4 + ZA4B4 - ZA3B4
          GO TO 29Ø
C TO ESTIMATE THE ZA AND ZAB VALUES
C WHEN (IX.GE.LX-1).AND.(LX.EQ.2)
  27Ø     Z3A5 = Z3A4
          Z4A5 = Z4A4
          IF (IXML.EQ.Ø) GO TO 29Ø
          ZA5B2 = ZA4B2
          ZA5B3 = ZA4B3
          ZA5B4 = ZA4B4
C TO ESTIMATE THE ZA AND ZAB VALUES
C WHEN IX.EQ.1
  28Ø     IF (IXM1.NE.Ø) GO TO 29Ø
          Z3A3 = Z3A4 + Z3A4 - Z3A5
          Z3A2 = Z3A3 + Z3A3 - Z3A4
          Z4A3 = Z4A4 + Z4A4 - Z4A5
          Z4A2 = Z4A3 + Z4A3 - Z4A4
          ZA3B2 = ZA4B2 + ZA4B2 - ZA5B2
          ZA3B3 = ZA4B3 + ZA4B3 - ZA5B3
          ZA3B4 = ZA4B4 + ZA4B4 - ZA5B4
          GO TO 3ØØ
C NUMERICAL DIFFERENTIATION   ---   TO DETERMINE PARTIAL
C DERIVATIVES ZX, ZY, AND ZXY AS WEIGHTED MEANS OF DIVIDED
C DIFFERENCES ZA, ZB, AND ZAB, RESPECTIVELY
C TO SAVE THE OLD VALUES WHEN IX.NE.1
  29Ø     ZX33 = ZX43
          ZX34 = ZX44
          ZY33 = ZY43
          ZY34 = ZY44
          ZXY33 = ZXY43
          ZXY34 = ZXY44
C NEW COMPUTATION
  3ØØ     DO 35Ø JY=1,2
            W2 = ABS(ZA(4,JY)-ZA(3,JY))
            W3 = ABS(ZA(2,JY)-ZA(1,JY))
            SW = W2 + W3
            IF (SW.EQ.Ø.Ø) GO TO 31Ø
            WX2 = W2/SW
            WX3 = W3/SW
            GO TO 32Ø
  31Ø       WX2 = Ø.5
            WX3 = Ø.5
  32Ø       ZX(JY) = WX2*ZA(2,JY) + WX3*ZA(3,JY)
            W2 = ABS(ZB(JY+3)-ZB(JY+2))
            W3 = ABS(ZB(JY+1)-ZB(JY))
            SW = W2 + W3
            IF (SW.EQ.Ø.Ø) GO TO 33Ø
            WY2 = W2/SW
            WY3 = W3/SW
            GO TO 34Ø
  33Ø       WY2 = Ø.5
            WY3 = Ø.5
  34Ø       ZY(JY) = WY2*ZB(JY+1) + WY3*ZB(JY+2)
            ZXY(JY) = WY2*(WX2*ZAB(1,JY)+WX3*ZAB(2,JY)) +
     *      WY3*(WX2*ZAB(1,JY+1)+WX3*ZAB(2,JY+1))
  35Ø     CONTINUE
          IF (IXM1.EQ.Ø) GO TO 38Ø
C DETERMINATION OF THE COEFFICIENTS OF THE POLYNOMIAL
          ZX3B3 = (ZX34-ZX33)*B3
          ZX4B3 = (ZX44-ZX43)*B3
          ZY3A3 = (ZY43-ZY33)*A3
          ZY4A3 = (ZY44-ZY34)*A3
          A = ZA3B3 - ZX3B3 - ZY3A3 + ZXY33
          B = ZX4B3 - ZX3B3 - ZXY43 + ZXY33
          C = ZY4A3 - ZY3A3 - ZXY34 + ZXY33
          D = ZXY44 - ZXY43 - ZXY34 + ZXY33
          E = A + A - B - C
          PØ2 = (2.Ø*(Z3B3-ZY33)+Z3B3-ZY34)*B3
          PØ3 = (-2.Ø*Z3B3+ZY34+ZY33)*B3SQ
          P12 = (2.Ø*(ZX3B3-ZXY33)+ZX3B3-ZXY34)*B3
          P13 = (-2.Ø*ZX3B3+ZXY34+ZXY33)*B3SQ
          P2Ø = (2.Ø*(Z3A3-ZX33)+Z3A3-ZX43)*A3
          P21 = (2.Ø*(ZY3A3-ZXY33)+ZY3A3-ZXY43)*A3
          P22 = (3.Ø*(A+E)+D)*A3*B3
          P23 = (-3.Ø*E-B-D)*A3*B3SQ
          P3Ø = (-2.Ø*Z3A3+ZX43+ZX33)*A3SQ
          P31 = (-2.Ø*ZY3A3+ZXY43+ZXY33)*A3SQ
          P32 = (-3.Ø*E-C-D)*B3*A3SQ
          P33 = (D+E+E)*A3SQ*B3SQ
C COMPUTATION OF THE POLYNOMIAL
          DO 37Ø JY=1,JYMX
            KV = KVØ + JY
            DY = V(KV) - Y3
            QØ = PØØ + DY*(PØ1+DY*(PØ2+DY*PØ3))
            Q1 = P1Ø + DY*(P11+DY*(P12+DY*P13))
            Q2 = P2Ø + DY*(P21+DY*(P22+DY*P23))
            Q3 = P3Ø + DY*(P31+DY*(P32+DY*P33))
            DO 36Ø JX=1,JXMX
              KU = KUØ + JX
              DX = U(KU) - X3
              W(KU,KV) = QØ + DX*(Q1+DX*(Q2+DX*Q3))
  36Ø       CONTINUE
  37Ø     CONTINUE
          KUØ = KUØ + MXØ
  38Ø   CONTINUE
        KVØ = KVØ + MYØ
  39Ø CONTINUE
```

```
C NORMAL EXIT
      RETURN
C ERROR EXIT
  400 WRITE (IU0,99999)
      GO TO 520
  410 WRITE (IU0,99998)
      GO TO 520
  420 WRITE (IU0,99997)
      GO TO 520
  430 WRITE (IU0,99996)
      GO TO 520
  440 WRITE (IU0,99995)
      GO TO 520
  450 WRITE (IU0,99994)
      GO TO 520
  460 WRITE (IU0,99993)
      GO TO 480
  470 WRITE (IU0,99992)
  480 WRITE (IU0,99991) IX, X(IX)
      GO TO 520
  490 WRITE (IU0,99990)
      GO TO 510
  500 WRITE (IU0,99989)
  510 WRITE (IU0,99988) IY, Y(IY)
  520 WRITE (IU0,99987) LX0, MX0, NU0, LY0, MY0, NV0
      RETURN
C FORMAT STATEMENTS
99999 FORMAT(1X/23H  ***   LX = 1 OR LESS./)
99998 FORMAT(1X/23H  ***   LY = 1 OR LESS./)
99997 FORMAT(1X/23H  ***   MX = 1 OR LESS./)
99996 FORMAT(1X/23H  ***   MY = 1 OR LESS./)
99995 FORMAT(1X/26H  ***   IMPROPER NU VALUE./)
99994 FORMAT(1X/26H  ***   IMPROPER NV VALUE./)
99993 FORMAT(1X/27H  ***   IDENTICAL X VALUES./)
99992 FORMAT(1X/33H  ***   X VALUES OUT OF SEQUENCE./)
99991 FORMAT(7H   IX =, I6, 10X, 7HX(IX) =, E12.3)
99990 FORMAT(1X/27H  ***   IDENTICAL Y VALUES./)
99989 FORMAT(1X/33H  ***   Y VALUES OUT OF SEQUENCE./)
99988 FORMAT(7H   IY =, I6, 10X, 7HY(IY) =, E12.3)
99987 FORMAT(7H   LX =, I6, 10X, 4HMX =, I6, 10X, 4HNU =, I6/
     * 7H   LY =, I6, 10X, 4HMY =, I6, 10X, 4HNV =, I6/6H ERROR,
     * 30H DETECTED IN ROUTINE    SFCFIT)
      END
```

ACM Transactions on Mathematical Software, Vol. 5, No. 2, June 1979, Pages 241.

REMARK ON ALGORITHM 474

Bivariate Interpolation and Smooth Surface Fitting Based on Local Procedures [E2]

[H. Akima, *Comm. ACM 17*, 1 (Jan. 1974), 26–31]

M.R. Anderson [Recd 14 February 1978 and 5 April 1978]
Department of Physics, University of Michigan, Physics-Astronomy Building,
Ann Arbor, MI 48109

Subroutine *SFCFIT* contains a violation of the Fortran Standard [1] similar to that observed [2] in a previous contribution by the same author [3]. Section 7.1.2.8 states that the initial value of a DO statement must be less than or equal to the value represented by the terminal parameter. When *LX* or *LY* are input as 2, DO statements labeled 10, 30, 60, and 80 violate this rule. Error conditions of

IDENTICAL X VALUES, X VALUES OUT OF SEQUENCE,
IDENTICAL Y VALUES, Y VALUES OUT OF SEQUENCE

may improperly result from comparisons of array variables, subscripts for which are incorrectly generated, within these DO loops.

Subroutine *SFCFIT* may be corrected to avoid the above violation by changing the initial parameters in DO statements labeled 10, 30, 60, and 80 from 3 to 2.

As altered, these carefully written subroutines have been used extensively and successfully.

REFERENCES

1. ANSI Standard Fortran, X3.9-1966. Amer. Nat. Stand. Inst., New York, 1966.
2. Anderson, M.R. Remark on Algorithm 433. *ACM Trans. Math. Software 2*, 2 (June 1976), 208.
3. Akima, H. Algorithm 433. Interpolation and smooth curve fitting based on local procedures. *Comm. ACM 15*, 10 (Oct. 1972), 914–918.

Algorithm 475

Visible Surface Plotting Program [J6]

Thomas Wright [Recd. 18 Apr. 1972, 13 Oct. 1972]
Computing Facility, National Center for Atmospheric Research, Boulder, CO 80302

National Center for Atmospheric Research is sponsored by the National Science Foundation

Key Words and Phrases: hidden line problem, computer graphics, contour surface
CR Categories: 3.65, 4.41, 8.2
Language: Fortran

[This program is not in ANSI Fortran. Nonstandard features are noted in the text. A demonstration driver is included to illustrate use of the subroutines. I/O unit 9 is used by this driver.—LDF.]

Description

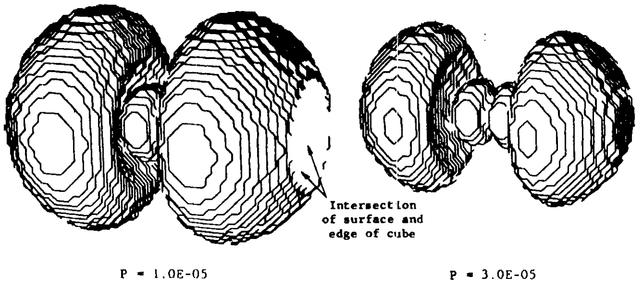
This package of three routines produces a perspective picture of an arbitrary object or group of objects with the hidden parts not drawn. The objects are assumed to be stored in the format described below, a format which was chosen to facilitate the display of functions of three variables (Figure 1) or output from three-dimensional computer simulations (Figure 2). The basic method is to contour cuts through the array, starting with a cut nearest the observer. The algorithm leaves out the hidden parts of the contours by suppressing lines enclosed within lines produced while processing preceding cuts. The technique is described in detail in [2].

The object is defined in a three-dimensional array by setting words to one where the object is, and to zero where it is not. That is, the position in the array corresponds to a position in three-space, and the value of the array tells whether any object is present at that position or not. Because a large array is needed to define objects with good resolution, only a part of the array is passed to the package with each call.

There are three subroutines in the package. *INIT3D* is called at the beginning of a picture. This call can be skipped sometimes if certain criteria are met and certain precautions are taken. See the comment lines for details. *SETORG* (which has an entry point *PERSPC*) does three-space to two-space perspective transformations. It is called by *INIT3D* and need not be called by the user. The mathematical method for the three-space to two-space transformation is due to Kubert, Szabo, and Giulieri [1]. *DANDR* (*d*raw *and r*emember) is called successively to process different parts of the three-dimensional array. For example, in Figure 3, the nearer plane would be processed in the first call to *DANDR*, while the further plane would be processed in a subsequent call. A sample program is provided with the algorithm to illustrate this point.

Although this package was developed using NCAR's CDC machines with locally written systems and compilers, implementation on different machines or systems should not be too difficult regardless of the plotter. The algorithm has been tested on the

Fig. 1. Four contour surfaces of the wave function of a 3-P electron in a one electron atom: 50 × 50 × 50 object cube, 100 × 100 screen model.



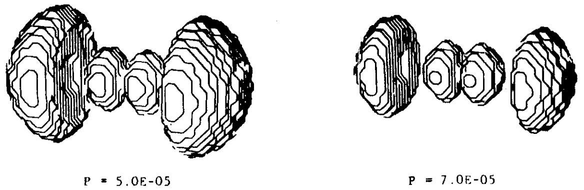


Fig. 2. Output from a three-dimensional cloud model: 100 × 100 × 60 object cube, 200 × 200 screen model.

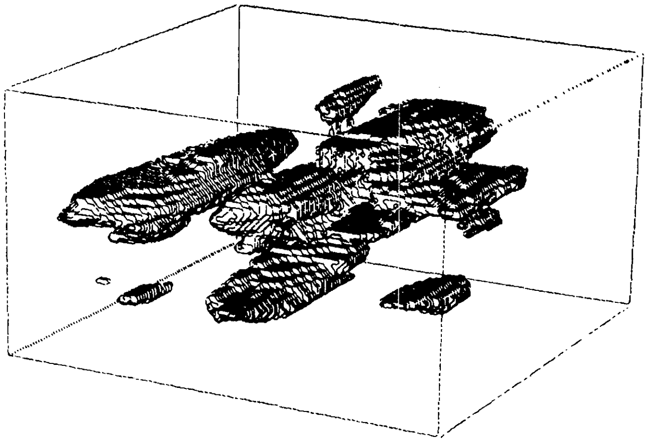
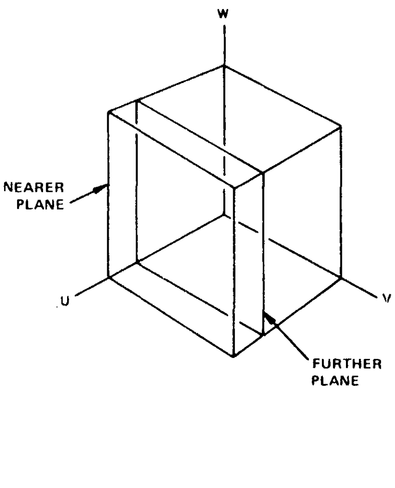


Fig. 3. Processing different parts of a three-dimensional array.



Minnesota Fortran compiler (MNF), and when the following items are taken care of, should be portable.

There is a *PROGRAM* card in the demonstration program There is an *ENTRY* statement in *SETORG*. *ENTRY* statements are nonstandard, but are generally portable. It could be eliminated, but the package would run longer. There are two machine-dependent variables used and described in *DANDR*. There is one system routine, *LINE*, called once and described in *DANDR*, which must be implemented or simulated to use this package. In three statements (which are marked) in *DANDR*, *.OR.* and *.AND.* are used for masking operations. with integer variables. Some compilers may not produce the desired code, so references to machine language functions may have to be substituted. There is a nonstandard but common form of the *DATA* statement in *DANDR*. Functions which are assumed available are *SQRT*, *ACOS*, and *SIN*.

Figures 4 and 5 are referred to in the listing as the first picture and the second picture.

Fig. 4. The first picture produced by the test program.

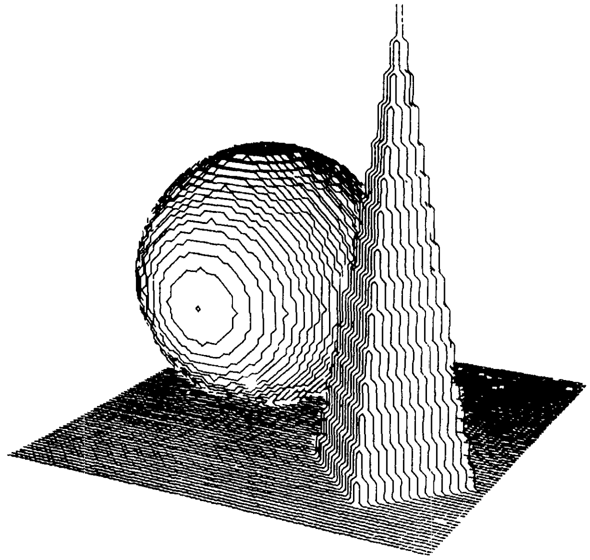
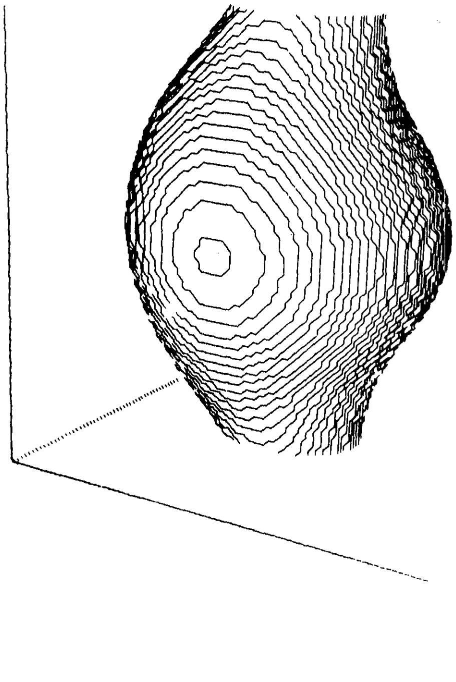


Fig. 5. The second picture produced by the test program.



References

1. Kubert, B., Szabo, J., and Giulieri, S. The perspective representation of functions of two variables. *J. ACM 15*, 2 (Apr. 1968), 193–204.

2. Wright, T. A one-pass hidden-line remover for computer drawn three-space objects. Proc. 1972 Summer Comput. Simulation Conf., pp. 261–267.

Algorithm

```
      PRØGRAM ACMTEST
C DEMONSTRATIØN PRØGRAM
      DIMENSIØN EYE(3), S(4), STI(80,80,2), IS2(3,160)
      DIMENSIØN IOBJ(80,80)
C USE WHØLE FRAME
      S(1) = 0.
      S(2) = 1.
      S(3) = 0.
      S(4) = 1.
C SET EYE PØSITICN
      EYE(1) = 250.
      EYE(2) = 150.
      EYE(3) = 100.
C INITIALIZE PACKAGE
      CALL INIT3D(EYE, 80, 80, 80, STI, 3, 160, IS2, 9, S)
C CREATE AND PLØT TEST ØBJECT
      DO 50 I=1,80
        A = (I-50)**2
        DØ 40 J=1,80
          C = (J-25)**2
          D = IABS(J-63) + IABS(I-25)
          DØ 30 K=1,80
C FLØØR
            IF (K.EQ.1) GØ TØ 10
C BALL
            IF (SQRT(A+C+(FLØAT(K)-25.)**2).LE.25.) GØ TØ 10
C PØINT
            IF (D.GT.FLØAT(80-K)*.1875) GØ TC 20
   10       IØBJ(J,K) = 1
            GØ TØ 30
   20       IØBJ(J,K) = 0
   30     CØNTINUE
   40   CØNTINUE
        CALL DANDR(80, 80, STI, 3, 160, 160, IS2, 9, S, IØBJ,
     *  80)
   50 CONTINUE
C ADVANCE TØ THE NEXT FRAME.
      CALL FRAME
C A SECOND PICTURE WILL NOW BE CALLED USING THE SAME SIZE
C ARRAYS AND EYE PØSITIØN.  THIS MEANS THE CALL TO INIT3D,
C THE BIGGEST TIME CØNSUMER, CAN BE SKIPPED IF THE FOLLØWING
C FOUR LINES ARE INCLUDED.
      REWIND 9
      DØ 70 I=1,3
        DO 60 J=1,160
          IS2(I,J) = 0
   60   CØNTINUE
   70 CØNTINUE
C THIS PICTURE WILL BE THE T=4 CONTCUR SURFACE OF
C T=1/SQRT(U*U+V*V+W*W)+(.5-V)**2/SQRT(U*U+V*V).
      DO 120 I=1,80
        U = (40.5-FLØAT(I))/79.
        UU = U*U
        DØ 110 J=1,80
          V = (FLØAT(J)-40.5)/79.
          VV = V*V
          A = 1./SQRT(UU+VV)
          DO 100 K=1,80
C THE FØLLØWING CARD ADDS AXES.
            IF (I*J.EQ.1 .ØR. I*K.EQ.1 .ØR. J*K.EQ.1) GØ TØ 80
            W = (FLØAT(K)-40.5)/79.
            IF (1./SQRT(UU+VV+W*W)+(.5-V)**2*A.LE.4.) GØ TØ 90
   80       IØBJ(J,K) = 1
            GØ TØ 100
   90       IØBJ(J,K) = 0
  100     CØNTINUE
  110   CØNTINUE
        CALL DANDR(80, 80, STI, 3, 160, 160, IS2, 9, S, IØBJ,
     *  80)
  120 CØNTINUE
C FLUSH PLØT BUFFER
      CALL FRAME
      STØP
      END

      SUBRØUTINE INIT3D(EYE, NU, NV, NW, ST1, LX, NY, IS2, IU,
     * S)
      DIMENSIØN EYE(3), ST1(NV,NW,2), IS2(LX,NY), S(4)
C BY THØMAS WRIGHT
C CØMPUTING FACILITY
C THE NATIØNAL CENTER FØR ATMØSPHERIC RESEARCH
C BØULDER, CØLØRADØ 80302
C NCAR IS SPØNSORED BY THE NATIONAL SCIENCE FØUNDATIØN.
C THE METHØD IS DESCRIBED IN DETAIL IN - A ØNE-PASS HIDDEN-
C LINE REMØVER FØR CØMPUTER DRAWN THREE-SPACE ØBJECTS. PRØC
C 1972 SUMMER CØMPUTER SIMULATIØN CØNFERENCE, 261-267, 1972.
C THIS VERSIØN IS FØR USE ØN CDC 6000 ØR 7000 CØMPUTERS.
C THIS PACKAGE ØF RØUTINES PLØTS 3-DIMENSIØNAL ØBJECTS WITH
C HIDDEN PARTS NØT SHØWN.  ØBJECTS ARE STØRED IN AN ARRAY,
C WITH THE PØSITIØN IN THE ARRAY CØRRESPØNDING TØ A LØCATIØN
C IN 3-SPACE AND THE VALUE ØF THE ARRAY ELEMENT TELLING IF
C ANY ØBJECT IS PRESENT AT THE LØCATIØN.
C INIT3D IS AN INITIALIZATIØN RØUTINE FØR THIS PACKAGE.  IT
C IS CALLED, THEN A SEQUENCE ØF CALLS ARE MADE TØ DANDR TØ
C PRØDUCE A PICTURE.
C EYE    AN ARRAY 3 LØNG CØNTAINING THE U, V, AND W CØØRDI-
C        NATES ØF THE EYE PØSITIØN.  ØBJECTS ARE CØNSIDERED
C        TØ BE IN A BØX WITH 2 EXTREME CØRNERS AT (1,1,1) AND
```

```
C         (NU,NV,NW).  THE EYE POSITION MUST HAVE POSITIVE
C         COORDINATES AWAY FROM THE COORDINATE PLANES U=0,
C         V=0, AND W=0.  WHILE GAINING EXPERIENCE WITH THE
C         PACKAGE, USE EYE(1)=5*NU, EYE(2)=4*NV, EYE(3)=3*NW.
C NU      U DIRECTION LENGTH OF THE BOX CONTAINING THE OBJECTS
C NV      V DIRECTION LENGTH OF THE BOX CONTAINING THE OBJECTS
C NW      W DIRECTION LENGTH OF THE BOX CONTAINING THE OBJECTS
C ST1     A SCRATCH ARRAY AT LEAST NV*NW*2 WORDS LONG.
C LX      FIRST DIMENSION OF A SCRATCH ARRAY, IS2, USED BY THE
C         PACKAGE FOR REMEMBERING WHERE IT SHOULD NOT DRAW.
C         LX=1+NX/NBPW.  SEE DANDR COMMENTS FOR NX AND NBPW.
C NY      SECOND DIMENSION OF IS2.  SEE DANDR COMMENTS.
C IS2     A SCRATCH ARRAY AT LEAST LX*NY WORDS LONG.
C IU      UNIT NUMBER OF SCRATCH FILE FOR THE PACKAGE.  ST1
C         WILL BE WRITTEN NU TIMES ON THIS FILE.
C S       AN ARRAY 4 LONG WHICH CONTAINS THE COORDINATES OF
C         THE AREA WHERE THE PICTURE IS TO BE DRAWN.  THAT IS,
C         ALL PLOTTING COORDINATES GENERATED WILL BE BOUNDED
C         AS FOLLOWS-- X COORDINATES WILL BE BETWEEN S(1) AND
C         S(2), Y COORDINATES WILL BE BETWEEN S(3) AND S(4).
C         TO PREVENT DISTORTION, HAVE S(2)-S(1)=S(4)-S(3).
C IF SEVERAL PICTURES ARE TO BE DRAWN WITH THE SAME SIZE
C ARRAYS AND EYE POSITION AND THE USER REWINDS IU AND FILLS
C IS2 WITH ZEROES, INIT3D NEED NOT BE CALLED FOR OTHER THAN
C THE FIRST PICTURE.
C SET UP TRANSFORMATION ROUTINE FOR THIS LINE OF SIGHT.
      U = NU
      V = NV
      W = NW
      CALL SETORG(U*.5, V*.5, W*.5, EYE(1), EYE(2), EYE(3))
C FIND EXTREMES IN TRANSFORMED SPACE.
      CALL PERSPC(1., 1., W, D, YT, D)
      CALL PERSPC(U, V, 1., D, YB, D)
      CALL PERSPC(U, 1., 1., XL, D, D)
      CALL PERSPC(1., V, 1., XR, D, D)
C ADJUST EXTREMES TO PREVENT DISTORTION WHEN GOING FROM
C TRANSFORMED SPACE TO PLOTTER SPACE.
      DIF = (XR-XL-YT+YB)*.5
      IF (DIF) 10, 30, 20
   10 XL = XL + DIF
      XR = XR - DIF
      GO TO 30
   20 YB = YB - DIF
      YT = YT + DIF
   30 REWIND IU
C FIND THE PLOTTER COORDINATES OF THE 3-SPACE LATTICE POINTS
      C1 = .9*(S(2)-S(1))/(XR-XL)
      C2 = .05*(S(2)-S(1)) + S(1)
      C3 = .9*(S(4)-S(3))/(YT-YB)
      C4 = .05*(S(4)-S(3)) + S(3)
      DO 60 I=1,NU
        U = NU + 1 - I
        DO 50 J=1,NV
          V = J
          DO 40 K=1,NW
            CALL PERSPC(U, V, FLOAT(K), X, Y, D)
            ST1(J,K,1) = C1*(X-XL) + C2
            ST1(J,K,2) = C3*(Y-YB) + C4
   40     CONTINUE
   50   CONTINUE
C WRITE THEM ON UNIT IU.
        WRITE (IU) ST1
   60 CONTINUE
      REWIND IU
C ZERO OUT ARRAY WHERE VISIBILITY IS REMEMBERED.

      DO 80 J=1,NY
        DO 70 I=1,LX
          IS2(I,J) = 0
   70   CONTINUE
   80 CONTINUE
      RETURN
      END

      SUBROUTINE SETORG(X, Y, Z, XT, YT, ZT)
C THIS ROUTINE IMPLEMENTS THE 3-SPACE TO 2-SPACE TRANSFOR-
C MATION BY KUBER, SZABO AND GIULIERI, THE PERSPECTIVE
C REPRESENTATION OF FUNCTIONS OF TWO VARIABLES. J. ACM 15,
C 2, 193-204,1968.
C SETORG ARGUMENTS
C X,Y,Z       ARE THE 3-SPACE COORDINATES OF THE INTERSECTION
C             OF THE LINE OF SIGHT AND THE IMAGE PLANE.  THIS
C             POINT CAN BE THOUGHT OF AS THE POINT LOOKED AT.
C XT,YT,ZT ARE THE 3-SPACE COORDINATES OF THE EYE POSITION.
C PERSPC ARGUMENTS
C X,Y,Z       ARE THE 3-SPACE COORDINATES OF A POINT TO BE
C             TRANSFORMED.
C XT,YT       THE RESULTS OF THE 3-SPACE TO 2-SPACE TRANSFOR-
C             MATION.
C ZT          NOT USED.
C STORE THE PARAMETERS OF THE SETORG CALL FOR USE WHEN
C PERSPC IS CALLED.
      AX = X
      AY = Y
      AZ = Z
      EX = XT
      EY = YT
      EZ = ZT
C AS MUCH COMPUTATION AS POSSIBLE IS DONE DURING EXECUTION
C OF SETORG SINCE PERSPC IS CALLED THOUSANDS OF TIMES FOR
C EACH CALL TO SETORG.
      DX = AX - EX
      DY = AY - EY
      DZ = AZ - EZ
      D = SQRT(DX*DX+DY*DY+DZ*DZ)
      COSAL = DX/D
      COSBE = DY/D
      COSGA = DZ/D
      AL = ACOS(COSAL)
      BE = ACOS(COSBE)
      GA = ACOS(COSGA)
      SINGA = SIN(GA)
C THE 3-SPACE POINT LOOKED AT IS TRANSFORMED INTO (0,0) OF
C THE 2-SPACE.  THE 3-SPACE Z AXIS IS TRANSFORMED INTO THE
C 2-SPACE Y AXIS.  IF THE LINE OF SIGHT IS CLOSE TO PARALLEL
C TO THE 3-SPACE Z AXIS, THE 3-SPACE Y AXIS IS CHOSEN (IN-
C STEAD OF THE 3-SPACE Z AXIS) TO BE TRANSFORMED INTO THE
C 2-SPACE Y AXIS.
      IF (SINGA.LT.0.0001) GO TO 10
      R = 1./SINGA
      ASSIGN 20 TO JUMP
      RETURN
   10 SINBE = SIN(BE)
      R = 1./SINBE
      ASSIGN 30 TO JUMP
      RETURN
C ******************  ENTRY PERSPC  **********************
      ENTRY PERSPC
      Q = D/((X-EX)*COSAL+(Y-EY)*COSBE+(Z-EZ)*COSGA)
      GO TO JUMP, (20,30)
   20 XT = ((EX+O*(X-EX)-AX)*COSBE-(EY+Q*(Y-EY)-AY)*COSAL)*R
      YT = (EZ+O*(Z-EZ)-AZ)*R
      RETURN
   30 XT = ((EZ+Q*(Z-EZ)-AZ)*COSAL-(EX+Q*(X-EX)-AX)*COSGA)*R
      YT = (EY+Q*(Y-EY)-AY)*R
      RETURN
      END



      SUBROUTINE DANDR(NV, NW, ST1, LX, NX, NY, IS2, IU, S,
     * IOBJS, MV)
      DIMENSION ST1(NV,NW,2), IS2(LX,NY), S(4), IOBJS(MV,NW)
C THIS ROUTINE IS CALLED NU TIMES, EACH CALL PROCESSING THE
C PART OF THE PICTURE AT U=NU+1-I WHERE I IS THE NUMBER OF
C THE CALL TO DANDR.  THAT IS, THE PART OF THE PICTURE AT
C U=NU IS PROCESSED DURING THE FIRST CALL, THE PART OF THE
C PICTURE AT U=NU-1 IS PROCESSED DURING THE SECOND CALL, AND
C SO ON UNTIL THE PART OF THE PICTURE AT U=1 IS PROCESSED
C DURING THE LAST CALL.
C NV      SEE INIT3D COMMENTS.
C NW      SEE INIT3D COMMENTS.
C ST1     SEE INIT3D COMMENTS.
C LX      THE NUMBER OF WORDS NEEDED TO HOLD NX BITS.  ALSO,
C         THE FIRST DIMENSION OF IS2.
C NX      NUMBER OF CELLS IN THE X DIRECTION OF A MODEL OF THE
C         IMAGE PLANE.  A SILHOUETTE OF THE PARTS OF THE PIC-
C         TURE PROCESSED SO FAR IS STORED IN THIS MODEL. LINES
C         TO BE DRAWN ARE TESTED FOR VISIBILITY BY EXAMINING
C         THE SILHOUETTE.  LINES IN THE SILHOUETTE ARE HIDDEN.
C         LINES OUT OF THE SILHOUETTE ARE VISIBLE.  THE SOLU-
C         TION IS APPROXIMATE BECAUSE THE SILHOUETTE IS NOT
C         FORMED EXACTLY.  SEE IS2 COMMENT BELOW.
C NY      NUMBER OF CELLS IN THE Y DIRECTION OF THE MODEL OF
C         THE IMAGE PLANE.  ALSO THE SECOND DIMENSION OF IS2.
C IS2     AN ARRAY TO HOLD THE IMAGE PLANE MODEL.  IT IS
C         DIMENSIONED LX BY NY.  THE MODEL IS NX BY NY AND
C         PACKED DENSELY.  IF HIDDEN LINES ARE DRAWN, DECREASE
C         NX AND NY (AND LX IF POSSIBLE).  IF VISIBLE LINES
C         ARE LEFT OUT OF THE PICTURE, INCREASE NX AND NY (AND
C         LX IF NEED BE).  AS A GUIDE, SOME EXAMPLES SHOWING
C         SUCCESSFUL CHOICES ARE LISTED
C            GIVEN  NU  NV  NW   RESULTING NX  NY FROM TESTING
C                  100 100  60              200 200
C                   60  60  60              110 110
C                   40  40  40               75  75
C IU      SEE INIT3D COMMENTS.
C IOBJS A NV BY NW ARRAY (WITH ACTUAL FIRST DIMENSION MV IN
C         THE CALLING PROGRAM) DESCRIBING THE OBJECT.  IF THIS
C         IS CALL NUMBER I TO DANDR, THE PART OF THE PICTURE
C         AT U=NU+1-I IS TO BE PROCESSED.  IOBJS DEFINES THE
C         OBJECTS TO BE DRAWN IN THE FOLLOWING MANNER --
C         IOBJS(J,K)=1 IF ANY OBJECT CONTAINS THE POINT
C         (NU+1-I,J,K) AND IOBJS(J,K)=0 OTHERWISE.
C MV      ACTUAL FIRST DIMENSION OF IOBJS IN THE CALLING
C         PROGRAM.
C************* MACHINE DEPENDANT CONSTANTS ***************
C NBPW NUMBER OF BITS PER WORD
C MASK AN ARRAY NBPW LONG.  MASK(I)=2**(I-1), I=1,2,...,NBPW
C CDC 6000 OR 7000 VERSION
      DIMENSION MASK(60)
      DATA NBPW/60/
      DATA MASK/1B,2B,4B,10B,20B,40B,100B,200B,400B,1000B,
     * 2000B,4000B,10000B,20000B,40000B,100000B,200000B,
     * 400000B,1000000B,2000000B,4000000B,10000000B,
     * 20000000B,40000000B,100000000B,200000000B,400000000B,
     * 1000000000B,2000000000B,4000000000B,10000000000B,
     * 20000000000B,40000000000B,100000000000B,
     * 200000000000B,400000000000B,1000000000000B,
     * 2000000000000B,4000000000000B,10000000000000B,
     * 20000000000000B,40000000000000B,100000000000000B,
     * 200000000000000B,400000000000000B,1000000000000000B,
     * 2000000000000000B,4000000000000000B,
     * 10000000000000000B,20000000000000000B,
     * 40000000000000000B,100000000000000000B,
     * 200000000000000000B,400000000000000000B,
     * 1000000000000000000B,2000000000000000000B,
     * 4000000000000000000B,10000000000000000000B,
     * 20000000000000000000B,40000000000000000000B/
      ASSIGN 120 TO IRET
C RX AND RY ARE USED TO MAP PLOTTER COORDINATES INTO THE
C IMAGE PLANE MODEL.
      RX = (FLOAT(NX)-1.)/(S(2)-S(1))
      RY = (FLOAT(NY)-1.)/(S(4)-S(3))
C READ THE RELATIVE PLOTTER COORDINATES OF THE LATTICE
C POINTS FROM UNIT IU.
      READ (IU) ST1
C DX, DY AND DZ ARE USED TO FIND REQUIRED COORDINATES OF
C NON-LATTICE POINTS.
      NVD2 = NV/2
      NWD2 = NW/2
      DX = (ST1(NV,NWD2,1)-ST1(1,NWD2,1))*.5/(FLOAT(NV)-1.)
      DY = (ST1(1,NWD2,2)-ST1(NV,NWD2,2))*.5/(FLOAT(NV)-1.)
      DZ = (ST1(NVD2,NW,2)-ST1(NVD2,1,2))*.5/(FLOAT(NW)-1.)
C SLOPE IS USED TO DEFORM THE IMAGE PLANE MODEL SO THAT
```

```
C LINES ØF CØNSTANT Y ØF THE IMAGE MØDEL HAVE THE SAME
C SLØPE AS LINES ØF CØNSTANT U AND W IN THE PICTURE.  THIS
C IMPRØVES THE PICTURE.
      SLØPE = DY/DX
C THE FØLLØWING LØØPS THRØUGH STATEMENT 130 GENERATE THE .5
C CØNTØUR LINES IN 2-SPACE FØR THE ARRAY IØBJS (WHICH CØN-
C TAINS ØNLY ZERØES AND ØNES), TESTS THE LINES FØR VISIBIL-
C ITY, AND CALLS A RØUTINE TØ PLØT THE VISIBLE LINES.
      DØ 130 I=2,NV
        JUMP = IØBJS(I-1,1)*8 + IOBJS(I,1)*4 + 1
        DØ 120 J=2,NW
          X = ST1(I,J,1)
          Y = ST1(I,J,2)
C DECIDE WHICH ØF THE 16 PØSSIBILITIES THIS IS.
          JUMP = (JUMP-1)/4 + IOBJS(I-1,J)*8 + IØBJS(I,J)*4 + 1
          GØ TØ (120,20,40,50,70,80,30,100,100,10,80,70,50,40,
     *    20,120),JUMP
C GØING TØ 10 MEANS JUMP=10 WHICH MEANS ØNLY THE LØWER-RIGHT
C AND UPPER-LEFT ELEMENTS ØF THIS CELL ARE SET TO 1.
C TWØ LINES SHØULD BE DRAWN, A DIAGØNAL CØNNECTING THE
C MIDDLE ØF THE BØTTØM TØ THE MIDDLE ØF THE RIGHT SIDE ØF
C THE CELL (LØWER-RIGHT LINE), AND A DIAGØNAL CØNNECTING THE
C MIDDLE ØF THE LEFT SIDE TØ THE MIDDLE OF THE TØP (UPPER-
C LEFT LINE) ØF THE CELL.
   10     ASSIGN 90 TØ IRET
C LØWER-RIGHT LINE
   20     X1 = X
          Y1 = Y - DZ
          X2 = X + DX
          Y2 = Y - DY
          GØ TØ 110
C LØWER-LEFT AND UPPER-RIGHT
   30     ASSIGN 60 TO IRET
C LØWER-LEFT
   40     X1 = X
          Y1 = Y - DZ
          X2 = X - DX
          Y2 = Y + DY
          GØ TØ 110
C HØRIZØNTAL
   50     X1 = X + DX
          Y1 = Y - DY
          X2 = X - DX
          Y2 = Y + DY
          GØ TØ 110
C UPPER-LEFT
   60     ASSIGN 120 TO IRET
   70     X1 = X + DX
          Y1 = Y - DY
          X2 = X
          Y2 = Y + DZ
          GØ TO 110
C VERTICAL
   80     X1 = X
          Y1 = Y - DZ
          X2 = X
          Y2 = Y + DZ
          GØ TØ 110
   90     ASSIGN 120 TØ IRET
C UPPER-LEFT
  100     X1 = X - DX
          Y1 = Y + DY
          X2 = X
          Y2 = Y + DZ
C TEST VISIBILITY ØF THIS LINE SEGMENT.
  110     IX = (X1-S(1))*RX
          IY=MOD(IFIX((Y1-S(3))*RY-SLOPE*FLØAT(IX))+NY,NY)+1
          IBIT = MØD(IX,NBPW) + 1
          IX = IX/NBPW + 1
C *********** .AND. USED AS A MASKING ØPERATØR **************
          IV=IS2(IX,IY).AND.MASK(IBIT)
C IF EITHER END ØF THE LINE IS AT A MARKED SPOT ØN THE IMAGE
C PLANE MØDEL, THE LINE IS HIDDEN
          IF (IV.NE.0) GØ TØ IRET, (60,90,120)
          IX = (X2-S(1))*RX
          IY=MØD(IFIX((Y2-S(3))*RY-SLØPE*FLØAT(IX))+NY,NY)+1
          IBIT = MØD(IX,NBPW) + 1
          IX = IX/NBPW + 1
C *********** .AND. USED AS A MASKING ØPERATØR **************
          IV=IS2(IX,IY).AND.MASK(IBIT)
          IF (IV.NE.0) GØ TØ IRET, (60,90,120)
C *************** UNDEFINED EXTERNAL REFERENCE **************
C SUBRØUTINE LINE(X1,Y1,X2,Y2) IS ASSUMED TØ DRAW A LINE
C FRØM (X1,Y1) TØ (X2,Y2)
          CALL LINE(X1, Y1, X2, Y2)
          GØ TO IRET, (60,90,120)
  120   CØNTINUE
  130 CØNTINUE
C CØDE THRØUGH STATEMENT 150 CREATES AN APPRØXIMATIØN ØF
C THE SILHØUETTE ØF THE PART ØF THE PICTURE JUST DRAWN BY
C MARKING THE IMAGE PLANE MØDEL WHERE THE ØBJECT ØCCURS.
      DØ 150 I=1,NV
        DØ 140 J=1,NW
          IF (IØBJS(I,J).EQ.0) GØ TØ 140
          IX = (ST1(I,J,1)-S(1))*RX + 0.5
          TWK = SLØPE*FLØAT(IX) - 0.5
          IY=MØD(IFIX((ST1(I,J,2)-S(3))*RY-TWK)+NY,NY)+1
          IBIT = MØD(IX,NBPW) + 1
          IX = IX/NBPW + 1
C ************ .ØR. USED AS A MASKING ØPERATØR **************
          IS2(IX,IY)=IS2(IX,IY).ØR.MASK(IBIT)
  140   CØNTINUE
  150 CØNTINUE
      RETURN
      END
```

Remark on Algorithm 475 [J6]

Visible Surface Plotting Program [Thomas Wright, *Comm. ACM 17* (Mar. 1974), 152–155]

Lawrence W. Frederick [Recd 31 May 1974]
Emory University Computing Center, Uppergate House, Emory University, Atlanta, GA 30322

In the initialization phase a significant savings in time may be obtained (as a function of the box dimensions, *NU*, *NV*, *NW*) by integrating subroutine *SETORG* into subroutine *INIT3D*. The time consuming part of *INIT3D* is the 3-space to 2-space transformation done via the call to the *PERSPC* entry of *SETORG*. This transformation is performed in a regular fashion by triply nested *DO* loops ranging over the box dimensions. By algebraically separating the transformation, expressions not depending on inner loop indices may be floated to outer loops. This arrangement eliminates a large number of redundant operations and the nonstandard *ENTRY* statement.

Remark on Algorithm 475 [J6]

Visible Surface Plotting Program [Thomas Wright, *Comm. ACM 17* (Mar. 1974), 152–155]

R.G. Mashburn [Recd 9 Dec. 1974] Computer Sciences Division at Oak Ridge National Laboratory Union Carbide Corporation, Nuclear Division* Oak Ridge, TN 37830

* Prime contractor for the U.S. Energy Research and Development Administration.

The Visible Surface Plotting Program, Algorithm 475, has been modified to run on IBM 360 hardware using the Fortran IV (level H) compiler. Using a modifid version of the demonstration program supplied with the algorithm, the two sample plots were successfully produced. The following documents the changes that were required to convert the programs from CDC 6000 or 7000 programs to IBM 360 programs. In addition to the changes listed below it was, of course, necessary to include a *FRAME* subroutine, a *LINE* subroutine, and other calls to plotting subroutines which support locally available plotting equipment. However, since plotting equipment and its software support vary from one installation to another, only those changes pertinent to the IBM 360 are listed here.

Demonstration program:

1. Remove the *PROGRAM* statement.
2. Change the first *DIMENSION* statement from:

   DIMENSION EYE(3), S(4), ST1(80, 80, 2), IS2(3, 160)

   to:

   DIMENSION EYE(3), S(4), ST1(80, 80, 2), IS2(5, 160)

   Note. The comments in the program indicate the first extent *LX* of the array *IS*2 is calculated as follows:

   LX = 1 + NX/NBPW

This is true so long as *NX* is not an integral multiple of *NBPW*. However, in this case *NX* is 160 and *NBPW* (the number of bits per word) is 32 for the IBM 360. Thus *NX* is an integral multiple of *NBPW*, and *LX* is calculated simply as *NX/NBPW* In general use

```
LX = 1 + (NX-1)/NBPW.
```

3. Change the call to the *INIT3D* subroutine to:

```
CALL INIT3D (EYE, 80, 80, 80, ST1, 5, 160, IS2, 9, S)
```

4. Change the two calls to *DANDR* (one after statement 40, the other after statement 110) to:

```
CALL DANDR (80, 80, ST1, 5, 160, 160, IS2, 9, S, IOBJ
*80)
```

5. Change the *DO* statement following the *REWIND* 9 statement from:

```
DO 70 I = 1, 3   to:   DO 70 I = 1, 5
```

INIT3D subroutine: No changes required.
SETORG subroutine:

1. Because no standard exists for referencing arc cosine, the three statements containing references to the arc cosine subroutine were changed from:

```
AL = ACOS(COSAL) to: AL = ARCOS(COSAL)
BE = ACOS(COSBE)     BE = ARCOS(COSBE)
GA = ACOS(COSBA)     GA = ARCOS(COSGA)
```

2. Because no standard exists for *ENTRY* statements and their syntax differs among compilers, it was necessary to change the *ENTRY* statement from:

```
ENTRY PERSPEC  to:
               ENTRY PERSPC(X, Y, Z, XT, YT, ZT)
```

DANDR subroutine:

1. The *DIMENSION* statement should be changed from:

```
DIMENSION MASK (60)  to:    DIMENSION MASK (32)
```

2. The two *DATA* statements following the *DIMENSION* statement should be changed from:

```
 DATA NBPW/60/
 DATA MASK/1B, 2B, 4B, 10B, 20B, 40B, 100B, 200B, 400B,
 1000B,
* 2000B, 4000B, 10000B, 20000B, etc.,
```

to:

```
 DATA NBPW/32/
 DATA MASK/Z1, Z2, Z4, Z8, Z10, Z20, Z40, Z80, Z100,
* Z200, Z400, Z800, Z1000, Z2000, Z4000, Z8000, Z10000,
* Z20000, Z40000, Z80000, Z100000, Z200000, Z400000,
* Z800000, Z1000000, Z2000000, Z4000000, Z8000000
* Z10000000, Z20000000, Z40000000, Z80000000/
```

3. The two uses of the *.AND.* masking operation and the one use of the *.OR.* masking operation were changed to call assembly language function subprograms *IAND* and *IOR* (programs written locally for the ORNL computing center Fortran library) which return an *INTEGER**4 value which is the logical *AND* and logical *OR* respectively of the two arguments given them.
   Change the two *.AND.* statements from:

   ```
   IV = IS2(IX, IY).AND.MASK (IBIT)  to:

   IV = IAND(IS2(IX, IY), MASK (IBIT))
   ```

   Change the *.OR.* statement from:

   ```
   IS2(X, IY) = IS2(IX, IY).OR.MASK (IBIT)  to:

   IS2(IX, IY) = IOR(IS2(IX, IY), MASK(IBIT))
   ```

   Note. In the original program listing of subroutine *DANDR*, the comment card immediately preceding statement 60 reads:

   C UPPER-LEFT but should say: C UPPER-RIGHT.

Remark on Algorithm 475[J6]

Visible Surface Plotting Program [Thomas Wright, *Comm. ACM 17* (Mar. 1974), 152–155]

C.J. Doran [Recd 22 Oct. 1974], Physics Department, University of Nottingham, England

Algorithm 475 has been successfully implemented on a D.G. Nova 1220 minicomputer and an I.C.L. 1906A, making substitutions for the nonstandard features of the original algorithm.

ENTRY statements are permitted in 1900 Fortran but not by Data General. *SETORG* and *PERSPC* were therefore written as separate subroutines linked by a labelled common area declared as:

COMMON/CSETORG/JUMP, EX, EY, EZ, AX, AY, AZ, D, R, COSBE, COSAL, COSGA

JUMP being declared as a *LOGICAL* variable. The assigned *GO TO* statement in *PERSPC* then becomes

IF (JUMP) GO TO 30

with *JUMP = .FALSE.* replacing the first *ASSIGN* statement in *SETORG*, and *JUMP = .TRUE.* replacing the second.

The *DATA* statement in *DANDR* may easily be standardized by writing decimal literals, but most compilers will not accept an integer 2^{NBPW}. *NBPW* should then be redefined as one less than the number of bits per word.

Logical operations between integers may be performed by portable Fortran functions *IAND* and *IOR* as:

```
FUNCTION IAND(I, J)
LOGICAL BI, BJ
EQUIVALENCE (BI, II), (BJ, JJ)
II = I
JJ = J
BI = BI . AND . BJ
IAND = II
RETURN
END
```

with equivalent coding for *IOR*. The first two masking operations then become:
IV = IAND(IS2(IX, IY), MASK(IBIT))

and the third becomes:

IS2(IX, IY) = IOR(IS2(IX, IY), MASK(IBIT))

ACM Transactions on Mathematical Software, Vol. 1, No. 4, December 1975, Pages 381-382.

CERTIFICATION OF ALGORITHM 475

Visible Surface Plotting Program [J6]
[T. Wright, *Comm. ACM 17*, 3 (March 1974), 152–157]

Gordon E. Bromage [Recd 6 May 1975 and 11 July 1975]
University of Bradford, West Yorkshire, U.K.

Author's present address: S.R.C. Astrophysics Research Division, Culham Laboratory, Abingdon, Oxon., U.K.

This package was modified to remove all the nonstandard features mentioned in the algorithm description, together with one that was not pointed out, namely, two calls in ACMTEST to the system-dependent graph-plotting routine FRAME.

The bit-manipulation (masking) operators .AND. and .OR. and the nonstandard DATA statement (all in DANDR) were dealt with in the following way. The masking operators were replaced by segments IAND and IOR written in an assembly language. Since the array MASK is only used in these bit manipulations, the data statement assigning values to the elements of MASK was removed from DANDR and a corresponding statement inserted into the assembly-language segments, so that only the bit number (IBIT) was referenced from DANDR. Thus, in DANDR, the statement

```
IV = IS2(IX,IY) .AND. MASK(IBIT)
```

was replaced twice by the line

```
CALL IAND(IS2(IX,IY),IBIT,IV)
```

and the line

```
IS2(IX,IY) = IS2(IX,IY) .OR. MASK(IBIT)
```

was replaced once by

```
CALL IOR(IS2(IX,IY),IBIT,IS2(IX,IY)).
```

The package was then tested on an ICL 1904A machine (George 3 system), which uses a word length of 24 bits.

For the system-dependent graph-plotting routines, Calcomp routines were used in place of LINE and FRAME. In fact, to allow for duplication and editing of graphs without having to rerun the package, the plotting routines were separated from the main program. Thus the coordinates (X1,Y1,X2,Y2) of the lines to be plotted were written onto files in DANDR using the statement

```
WRITE (IUX) X1,Y1,X2,Y2      in place of      CALL LINE (X1,Y1,X2,Y2)
```

(where IUX is the I/O unit number assigned to a particular file), and the plotting was performed by a separate program.

It should be emphasized that the number of scratch files needed for assignment of I/O unit IU in INIT3D is also system dependent. For example, on the 1904A more than one file was needed for picture resolutions higher than that corresponding to a 30 × 30 × 30 object cube mesh; for 60 × 60 × 60 mesh, four files were needed, each one storing the information relating to 15 of the 60 image planes.

With the above changes implemented, the package ran successfully on the ICL 1904A for the processing of concave pictures (optimization objective-function surfaces) as well as for pictures of bounded objects and for a wide variety of eye positions. Successful processing was often obtained even when one of the eye-position coordinates was negative (cf. comment lines relating to the array EYE in INIT3D). On this machine, less than 30K 24-bit words were needed at run time for a resolution corresponding to a 60 × 60 × 60 mesh; while 12K words were sufficient for a 30 × 30 × 30 mesh resolution. The run time for the first test picture at the higher resolution was approximately 10 minutes.

ACM Transactions on Mathematical Software, Vol. 2, No. 1, March 1976, Pages 109–110

REMARK ON ALGORITHM 475

Visible Surface Plotting Program [J6]
[T. Wright, *Comm. ACM 17*, 3(March 1974), 152–155]

Lucian D. Duta [Recd 5 Aug. 1975]
Academy of Economic Studies, Str. Dorobanti 15-17, Bucharest, Romania

Algorithm 475 has been modified for running on an IBM 370 computer and on a FELIX C-256 computer, using the Fortran IV compilers. The two sample plots were successfully produced on a BENSON 222 plotter.

The changes in the program are those described by Mashburn [1]. In addition to these changes, we suggest the following.

SETORG Subroutine

1. Because the parameter *ZT* is not used in the PERSPC entry, change the entry statement to

```
      ENTRY PERSPC(X,Y,Z,XT,YT)
```

2. Remove the statements

```
      AL = ACOS(COSAL)
      BE = ACOS(COSBE)
      GA = ACOS(COSGA)
```

3. Change the statement

```
      SINGA = SIN(GA)
```

to

```
      SINGA = SQRT(1. - COSGA*COSGA)
```

4. Change statement 10 from

```
   10 SINBE = SIN(BE)
```

to

```
   10 SINBE = SQRT(1. - COSBE*COSBE)
```

INIT3D Subroutine

1. Modify all statements which call to *PERSPC* entry by removing the last argument:

```
      CALL PERSPC(1.,1.,W,D,YT)
      CALL PERSPC(U,V,1.,D,YB)
      CALL PERSPC(U,1.,1.,XL,D)
      CALL PERSPC(1.,V,1.,XR,D)
```

2. Include an *ENTRY* statement after statement 60:

```
      ENTRY INIS2
```

A call to the *INIS*2 entry will produce the filling of the array *IS*2 with zeros and the rewinding of the *IU* unit. Because the call to the *INIS*2 entry is made only after the call to the *INIT*3*D* subroutine, the *INIS*2 entry need not have parameters.

3. Change the comment cards from

```
C IF SEVERAL PICTURES ARE TO BE DRAWN WITH THE SAME SIZE
C ARRAYS AND EYE POSITION AND THE USER REWINDS IU AND FILLS
C IS2 WITH ZEROES, INIT3D NEED NOT BE CALLED FOR OTHER THAN
C THE FIRST PICTURE.
```

to

```
C IF SEVERAL PICTURES ARE TO BE DRAWN WITH THE SAME SIZE
C ARRAYS AND EYE POSITION, INIT3D NEED NOT BE CALLED FOR
```

```
C OTHER THAN THE FIRST PICTURE. IN THIS CASE, BEFORE EACH
C SUBSEQUENT PICTURE THE INIS2 ENTRY MUST BE CALLED FOR
C REWINDING IU AND FILLING THE ARRAY IS2 WITH ZEROES.
```

Demonstration Program

1. Change the following statements:

```
C FOUR LINES ARE INCLUDED.
      REWIND 9
      DO 70 I = 1,3
        DO 60 J = 1,160
          IS2(I,J) = 0
60      CONTINUE
70    CONTINUE
```

to

```
C LINE IS INCLUDED.
      CALL INIS2
```

REFERENCES

[1] MASHBURN, R.G. Remark on Algorithm 475. *Comm. ACM 18,* 5(May 1975), 276–277.

REMARK ON ALGORITHM 475

Visible Surface Plotting Program [J6]
[T. Wright, *Comm. ACM 17,* 3 (March 1974), 152–155]

A.C.M. van Swieten [Recd 28 July 1976 and 12 Sept. 1978]
VSSG, P.O. Box 3032, Leyden, The Netherlands

and

J.Th.M. de Hosson

Department of Applied Physics, Rijksuniversiteit Groningen, Universiteitscomplex Paddepoel, Nijenborgh 18, 9747 AG Groningen, The Netherlands

This research was supported by the Netherlands Organization for the Advancement of Pure Research (Z.W.O., The Hague).

This remark describes an extension of the visible surface plotting program, ACM Algorithm 475. This program turns out to result in a long plotting time when one is using CALCOMP plot routines. The long plotting time is mainly caused by numerous idle pen movements which are inherent to the structure of the algorithm. Essentially the algorithm does the following: the three-dimensional surface is cut in slices. The slices are separated and then searched in order to produce a perspective image of that slice and to remove the hidden lines; therefore, the algorithm generates a large number of small segments in the search direction. In general, however, the search direction does not coincide with the contour direction. When one is using CALCOMP subroutines there are a lot of idle pen movements due to the fact that the segments are not in an appropriate order. In Figure 1(a) it is shown that numerous idle pen movements are necessary to plot a disklike form. In the improved version only one idle pen movement is made (see Figure 1(b)).

The extension consists of two subroutines: SDLINE and PLTOUT. In the original subroutine DANDR we have to add five statements: Insert

COMMON/TOM1/NSEQ, SS; SS = 0.04, NSEQ = 0

before the statement

SLOPE = DX/DY

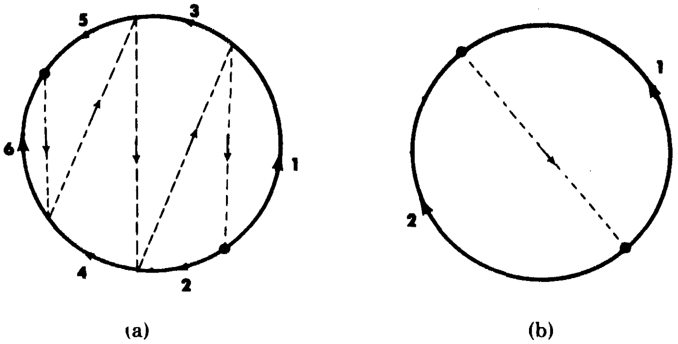


Fig. 1. (a) The pen movements generated by the original version of the plotting program. The idle pen movements are dashed lines. (b) Output of the improved plotting program showing one idle pen movement (dashed line)

which initializes

SDLINE; CALL SDLINE(X1, Y1, X2, Y2)

instead of

LINE(X1, Y1, X2, Y2)

which builds up the sequences and

CALL PLTOUT

after the statement

130 CONTINUE

in DANDR which plots the sequences. The subroutine SDLINE(X1, Y1, X2, Y2) temporarily stores the segments in order to construct the sequences. This is done by comparing the last point of each sequence with the endpoints of a segment. The criterion for the continuation of a sequence is that one of the endpoints of the segment lies within a square with edges of 2SS around the last point of a sequence. The value of SS depends on the plotter precision and it is taken to be equal to 0.04. If there is no continuation point of any sequence a new sequence is started through the segment.

In the present version the length of the sequences is equal to 80 and the number is equal to 20. If a sequence has been filled up completely a new sequence is created. If one needs more than 20 sequences intermediate plotting takes place by calling PLTOUT.

The subroutine PLTOUT plots the sequences taking into account the minimum distance between starting points and ends of sequences. This is done by ordering the sequences in an appropriate way and by indicating whether they should be processed in normal or reversed order.

Finally we give some test results of the revised program compared with the old version. The core size, execution time, and CALCOMP plotting time are compared in the case of the second example (Figure 5) in Algorithm 475. Although this type of surface is not the one that results in the greatest reduction, the saving of plotting time is significant (see Table I). In Table I the time spent in DANDR but not the time spent in PLTOUT is listed. The space of INIT3D + P + DANDR (old version) and of INIT3D + P + DANDR + SDLINE + PLTOUT + TOM are also given in Table I.

Table I

CYBER 74-16	Space	Time	CALCOMP plotting time (minutes)
Old version	1277_8	5.012	31
Revised version	10544_8	5.621	8

REVISED ALGORITHM

```
C     PROGRAM CONES(INPUT,OUTPUT,PLOT,TAPE6=OUTPUT,TAPE5=INPUT,
C    1TAPE99=PLOT,TAPE9)
C
C     DEMONSTRATION PROGRAM
C     BY THOMAS WRIGHT IN:
C     ALGORITHM 475, VISIBLE PLOTTING PROGRAM (J6),
C     COMMUNICATION OF THE ACM, MARCH 1974,VOL.17,NUMBER 3,P 152.
C***********  MACHINE  DEPENDANT  FUNCTIONS  *************************
C     FIRST CARD IS THE PROGRAM CARD FOR CDC 6000 AND CDC 7000 SERIES.
C     CALCOMP PACKAGE WHICH CONTAINS THE SUBROUTINES NAMPLT,ENDPLT,
C     NAMPLT = TO INITIALIZE THE SYSTEM.
C     ENDPLT = TO TERMINATE PLOTTING ON A FILE.
C
      DIMENSION EYE(3),S(4),ST1(80,80,2),IS2(3,160)
      DIMENSION IOBJ(80,80)
      CALL NAMPLT
C     USE WHOLE FRAME
      S(1)=0.
      S(2)=28.
      S(3)=0.
      S(4)=28.
C     SET EYE POSITION
      EYE(1)=200.
      EYE(2)=400.
      EYE(3)=300.
      NX=80
      NY=80
      NZ=80
      NCELLS=2
      MX=NCELLS*NY
      LX=1+MX/60
      MY=MX
C THIS PICTURE WILL BE THE T=4 CONTOURSURFACE OF
C T=1/SQRT(U*U+V*V+W*W)+(.5-V)**2/SQRT(U*U+V*V).
C     THIS IS THE SECOND PICTURE (FIG.5) PRODUCED BY THE TEST PROGRAM
C     OF THOMAS WRIGHT.
      CALL INIT3D(EYE,NX,NY,NZ,ST1,LX,MY,IS2,9,S)
      DO 50 I=1,NX
      U=(40.5-FLOAT(I))/79.
      UU=U*U
      DO 40 J=1,NY
      V=(FLOAT(J)-40.5)/79.
      VV=V*V
      A=1./SQRT(UU+VV)
      DO 30 K=1,NZ
C THE FOLLOWING CARD ADDS AXES.
      IF (I*J.EQ.1 .OR. I*K.EQ.1 .OR. J*K.EQ.1) GO TO 80
      W=(FLOAT(K)-40.5)/79.
      IF (1./SQRT(UU+VV+W*W) + (.5-V)**2*A.LE.4.) GO TO 90
 80   IOBJ(J,K)=1
      GO TO 30
 90   IOBJ(J,K)=0
   30 CONTINUE
   40 CONTINUE
      CALL DANDR(NY,NZ,ST1,LX,MX,MY,IS2,9,S,IOBJ,NY)
   50 CONTINUE
      CALL ENDPLT
      STOP
      END
      SUBROUTINE INIT3D(EYE,NU,NV,NW,ST1,LX,NY,IS2,IU,S)
C     BY THOMAS WRIGHT
C THIS ROUTINE IMPLEMENTS THE 3-SPACE TO 2-SPACE TRANSFORMATION BY
C KUBER,SZABO AND GIULIERI, THE PERSPECTIVE REPRESENTATION OF
C FUNCTIONS OF TWO VARIABLES. J. ACM 15,2, 193-204,1968.
C
      DIMENSION EYE(3),ST1(NV,NW,2),IS2(LX,NY),S(4)
C     THE METHOD IS DESCRIBED IN DETAIL IN - ONE-PASS HIDDEN-
C     LINE REMOVER FOR COMPUTER DRAWN THREE-SPACE OBJECTS. PROC
C     1972 SUMMER COMPUTER SIMULATION CONFERENCE ,261-267,1972.
C     THIS VERSION IS FOR USE ON CDC 6000 OR 7000 COMPUTERS.
C     THIS PACKAGE OF ROUTINES PLOTS 3-DIMENSIONAL OBJECTS WITH
C     HIDDEN PARTS NOT SHOWN.
C     INIT3D IS AN INITIALIZATION ROUTINE FOR THIS PACKAGE. IT IS CALLED
```

```
C     ,THEN A SEQUENCE OF CALLS ARE MADE TO DANDR TO PRODUCE A PICTURE.
C
C EYE AN ARRAY 3 LONG CONTAINING THE U,V,W COORDINATES OF THE EYE
C     POSITION. OBJECTS ARE CONSIDERED TO BE IN A BOX WITH 2 EXTREME
C     CORNERS AT (1,1,1) AND (NU,NV,NW). THE EYE POSITION MUST HAVE POSI
C     TIVE COORDINATES AWAY FROM THE COORDINATE PLANE U=Ø, V=Ø,W=Ø.
C     WHILE GAINING EXPERIENCE WITH THE PACKAGE, USE EYE(1)=5*NU,EYE(2)=
C     4*NV, EYE(3)=3*NW.
C NU  U DIRECTION LENGTH OF THE BOX CONTAINING THE OBJECTS
C NV  V DIRECTION LENGTH OF THE BOX CONTAINING THE OBJECTS
C NW  W DIRECTION LENGTH OF THE BOX CONTAINING THE OBJECTS
C ST1 A SCRATCH ARRAY AT LEAST NV*NW*2 WORDS LONG.
C LX  FIRST DIMENSION OF A SCRATCH ARRAY, IS2, USED BY THE PACKAGE
C     FOR REMEMBERING WHERE IT  SHOULD NOT DRAW.
C     LX=1+NX/NBPW.
C NY  SECOND DIMENSION OF IS2.
C IS2 A SCRATCH ARRAY AT LEAST LX*NY WORDS LONG.
C IU  UNIT NUMBER OF SCRATCH FILE FOR THE PACKAGE. ST1
C     WILL BE WRITTEN NU TIMES ON THIS FILE.
C S   AN ARRAY 4 LONG WHICH CONTAINS THE COORDINATES OF THE
C     AREA WHERE THE PICTURE IS TO BE DRAWN,
C     THAT IS, ALL PLOTTING COORDINATES GENERATED WILL BE BOUNDED AS
C     FOLLOWS-- X COORDINATES WILL BE BETWEEN S(1) AND S(2),
C     Y COORDINATE WILL BE BETWEEN S(3) AND S(4).
C     TO PREVENT DISTORTION, HAVE S(2)-S(1)=S(4)-S(3)
C IF SEVERAL PICTURES ARE TO BE DRAWN WITH THE SAME SIZE
C ARRAYS AND EYE POSITION AND THE USER REWINDS IU AND FILLS IS2
C WITH ZEROES, INIT3D NEED NOT TE BE CALLED FOR OTHER THAN THE
C FIRST PICTURE.
C
C SET UP TRANSFORMATION ROUTINE FOR THIS LINE OF SIGHT.
      U=NU
      V=NV
      W=NW
      AX=U*Ø.5
      AY=V*Ø.5
      AZ=W*Ø.5
      EX=EYE(1)
      EY=EYE(2)
      EZ=EYE(3)
      DX=AX-EX
      DY=AY-EY
      DZ=AZ-EZ
      D=SQRT(DX*DX+DY*DY+DZ*DZ)
      CA=DX/D
      CB=DY/D
      CG=DZ/D
C************ MACHINE DEPENDANT FUNCTION  ******** ACOS  ***************
C     AL=ACOS(CA)
C     BE=ACOS(CB)
C     GA=ACOS(CG)
C     THE MACINE DEPENDANT FUNCTION ACOS CAN BE REPLACED BY ARCCOS
      AL=ARCCOS(CA)
      BE=ARCCOS(CB)
      GA=ARCCOS(CG)
      SINGA=SIN(GA)
C THE 3-SPACE POINT LOOKED AT IS TRANSFORMED INTO (Ø,Ø) OF
C THE 2-SPACE. THE 3-SPACE Z-AXIS IS TRANSFORMED INTO THE
C 2-SPACE Y AXIS.
      IF(SINGA.LT.Ø.ØØØ1)GO TO 11
      R=1./SINGA
C     FIND EXTREMES IN TRANSFORMED SPACE.
      CALL P(1.,1.,W,DUMMY,YT,AX,AY,AZ,EX,EY,EZ,CA,CB,CG,D,R)
      CALL P(U,V,1.,DUMMY,YB,AX,AY,AZ,EX,EY,EZ,CA,CB,CG,D,R)
      CALL P(U,1.,1.,XL,DUMMY,AX,AY,AZ,EX,EY,EZ,CA,CB,CG,D,R)
      CALL P(1.,V,1.,XR,DUMMY,AX,AY,AZ,EX,EY,EZ,CA,CB,CG,D,R)
C ADJUST EXTREMES TO PREVENT DISTORTION WHEN GOING FORM
C TRANSFORMED SPACE TO PLOTTER SPACE.
      DIF=(XR-XL-YT+YB)*.5
      IF(DIF)1Ø,3Ø,2Ø
   1Ø XL=XL+DIF
      XR=XR-DIF
      GO TO 3Ø
   2Ø YB=YB-DIF
      YT=YT+DIF
   3Ø REWIND IU
```

```
C FIND THE PLOTTER COORDINATES OF THE 3-SPACE LATTICE POINTS.
      C1=.9*(S(2)-S(1))/(XR-XL)
      C2=.Ø5*(S(2)-S(1))+S(1)
      C3=.9*(S(4)-S(3))/(YT-YB)
      C4=.Ø5*(S(4)-S(3))+S(3)
      DO 6Ø I=1,NU
      U=NU+1-I
      DO 5Ø J=1,NV
      V=J
      DO 4Ø K=1,NW
      W=K
      Q=D/((U-EX)*CA+(V-EY)*CB+(W-EZ)*CG)
      X =((EX+Q*(U-EX)-AX)*CB-(EY+Q*(V-EY)-AY)*CA)*R
      Y =(EZ+Q*(W-EZ)-AZ)*R
      ST1(J,K,1)=C1*(X-XL)+C2
      ST1(J,K,2)=C3*(Y-YB)+C4
   4Ø CONTINUE
   5Ø CONTINUE
C WRITE THEM ON UNIT IU.
      WRITE(IU)ST1
   6Ø CONTINUE
      REWIND IU
C ZERO OUT ARRAY WHERE VISIBILITY IS REMEMBERED.
      DO 8Ø J=1,NY
      DO 7Ø I=1,LX
      IS2(I,J)=Ø
   7Ø CONTINUE
   8Ø CONTINUE
      RETURN
   11 CONTINUE
      STOP
      END
      SUBROUTINE P(X,Y,Z,XT,YT,AX,AY,AZ,EX,EY,EZ,CA,CB,CG,D,R)
C X,Y,Z ARE THE 3-SPACE COORDINATES OF A POINT TO BE TRANSFORMED.
C XT,YT   THE RESULTS OF THE 3-SPACE TO 2-SPACE TRANSFORMATION.
C
      Q=D/((X-EX)*CA+(Y-EY)*CB+(Z-EZ)*CG)
      XT=((EX+Q*(X-EX)-AX)*CB-(EY+Q*(Y-EY)-AY)*CA)*R
      YT=(EZ+Q*(Z-EZ)-AZ)*R
      RETURN
      END
      SUBROUTINE DANDR(NV,NW,ST1,LX,NX,NY,IS2,IU,S,IOBJS,MV)
C
C     THE PURPOSE OF THE SUBROUTINE AND THE INPUT AS WELL AS THE
C     OUTPUT PARAMETERS ARE THE SAME AS PUBLISHED BEFORE BY WRIGHT.
C     THEY ARE SUMMARIZED AND REPETED IN BEHALF OF THE USERS OF THIS
C     SUBROUTINE DANDR
C
C     THIS SUBROUTINE IS CALLED NU TIMES, EACH CALL PROCESSING THE
C     PART OF THE PICTURE AT U=NU-I+1 WHERE I IS THE NUMBER OF THE CALL
C     TO DANDR. THE PART OF THE PICTURE AT U=NU IS PROCESSED DURING
C     THE FIRST CALL, THE PART OF THE PICTURE AT U=NU-I+1 DURING
C     THE SECOND CALL, AND SO ON UNTIL THE PART OF THE PICTURE AT U=1
C     IS PROCESSED DURING THE LAST CALL.
C       PARAMETERS IN THE CALL
C     NV     V DIRECTION LENGTH OF THE BOX CONTAINING THE OBJECT.
C     NW     W DIRECTION LENGTH OF THE BOX CONTAINING THE OBJECT.
C     ST1    A SCRATCH ARRAY AT LEAST NV*NW*2 WORDS LONG.
C     LX     THE NUMBER OF WORDS NEEDED TO HOLD NX BITS.
C     NX     NUMBER OF CELLS IN THE X DIRECTION OF A MODEL OF THE
C            IMAGE PLANE.
C     NY     NUMBER OF CELLS IN THE Y DIRECTION OF THE MODEL OF THE
C            IMAGE PLANE.
C     IS2    AN ARRAY TO HOLD THE IMAGE PLANE MODEL.
C     IU     UNIT NUMBER OF SCRATCH FILE FOR THE PACKAGE.
C            ST1 WILL BE WRITTEN NU TIMES ON THIS FILE.
C     IOBJS  A NV BY NW ARRAY DESCRIBING THE OBJECT.
C            IF THIS IS CALL NUMBER I TO DANDR, THE PART OF THE PICTURE
C            AT U=NU-I+1 IS TO BE PROCESSED. IOBJS DEFINES THE OBJECTS
C            IOBJS(J,K)=1 IF ANY OBJECT CONTAINS THE POINT (NU-I+1,J,K)
C            AND IOBJS(J,K)=Ø OTHERWISE.
C     MV     THE ACTUAL FIRST DIMENION OF IOBJS IN THE CALLING PROGRAM.
C     S      AN ARRAY WHICH CONTAINS THE COORDINATES OF THE AREA WHERE
C            THE PICTURE IS TO BE DRAWN.
C     THE PROGRAM IS TESTED USING A CDC76ØØ (CYBER 76-16) COMPUTER
C     INSTALLATION, AND CDC 66ØØ INSTALLATION AS WELL.
```

```
C
C     INLINE FUNCTION WHICH ARE ASSUMED TO BE AVAILABLE, ARE=
C     ABS, FLOAT, IFIX, MOD.
C************     MACHINE DEPENDANT  CONSTANTS  ************************
C     NBPW   NUMBER OF BITS PER WORD
C     CDC SERIES (PRESENT CASE) NBPW=60.
C     IBM SERIES, NBPW=32.
C     UNIVAC 1100 SERIES, NBPW=36.
C     MASK   AN ARRAY NBPW LONG.    MASK(I)=2**(I-1),I=1,2,3,..,NBPW.
C
      DIMENSION ST1(NV,NW,2),IS2(LX,NY),S(4),IOBJS(MV,NW)
      DIMENSION MASK(60)
      INTEGER AND,OR
      COMMON/TOM1/NSEQ,SS
C**************** NBPW   **********************************************
      DATA NBPW/60/
      DATA MASK/1B,2B,4B,10B,20B,40B,100B,200B,400B,
     *1000B,2000B,4000B,10000B,
     *20000B,40000B,100000B,200000B,400000B,1000000B,2000000B,4000000B,
     *10000000B,20000000B,40000000B,100000000B,200000000B,400000000B,
     *1000000000B,2000000000B,4000000000B,10000000000B,20000000000B,
     *40000000000B,100000000000B,200000000000B,400000000000B,
     *1000000000000B,2000000000000B,4000000000000B,10000000000000B,
     *20000000000000B,40000000000000B,100000000000000B,200000000000000B,
     *400000000000000B,1000000000000000B,2000000000000000B,
     *4000000000000000B,10000000000000000B,20000000000000000B,
     *40000000000000000B,100000000000000000B,200000000000000000B,
     *400000000000000000B,1000000000000000000B,2000000000000000000B,
     *4000000000000000000B,10000000000000000000B,20000000000000000000B,
     *40000000000000000000B/
      ASSIGN 120 TO IRET
C     INITIALIZATION
      NSEQ=0
      SS=0.04
      RX=(FLOAT(NX)-1.)/(S(2)-S(1))
      RY=(FLOAT(NY)-1.)/(S(4)-S(3))
      READ(IU)ST1
      NVD2=NV/2
      NWD2=NW/2
      DX=(ST1(NV,NWD2,1)-ST1(1,NWD2,1))*.5/(FLOAT(NV)-1.)
      DY=(ST1(1,NWD2,2)-ST1(NV,NWD2,2))*.5/(FLOAT(NV)-1.)
      DZ=(ST1(NVD2,NW,2)-ST1(NVD2,1,2))*.5/(FLOAT(NW)-1.)
      SLOPE=DY/DX
      DO 130 I=2,NV
      JUMP=IOBJS(I-1,1)*8+IOBJS(I,1)*4+1
      DO 120 J=2,NW
      X=ST1(I,J,1)
      Y=ST1(I,J,2)
      JUMP=(JUMP-1)/4+IOBJS(I-1,J)*8+IOBJS(I,J)*4+1
      GO TO(120,20,40,50,70,80,30,100,100,10,80,70,50,40,20,120),JUMP
   10 ASSIGN 90 TO IRET
   20 X1=X
      Y1=Y-DZ
      X2=X+DX
      Y2=Y-DY
      GO TO 110
   30 ASSIGN 60 TO IRET
   40 X1=X
      Y1=Y-DZ
      X2=X-DX
      Y2=Y+DY
      GO TO 110
   50 X1=X+DX
      Y1=Y-DY
      X2=X-DX
      Y2=Y+DY
      GO TO 110
   60 ASSIGN 120 TO IRET
   70 X1=X+DX
      Y1=Y-DY
      X2=X
      Y2=Y+DZ
      GO TO 110
   80 X1=X
      Y1=Y-DZ
      Y2=Y+DZ
      GO TO 11Ø
   9Ø ASSIGN 12Ø TO IRET
  1ØØ X1=X-DX
      Y1=Y+DY
      X2=X
      Y2=Y+DZ
  11Ø IX=(X1-S(1))*RX
      IY=MOD(IFIX((Y1-S(3))*RY-SLOPE*FLOAT(IX))+NY,NY)+1
      IBIT=MOD(IX,NBPW)+1
      IX=IX/NBPW+1
      I1=IS2(IX,IY)
      I2=MASK(IBIT)
      IV=AND(I1,I2)
      IF(IV.NE.Ø)GO TO IRET,(6Ø,9Ø,12Ø)
      IX=(X2-S(1))*RX
      IY=MOD(IFIX((Y2-S(3))*RY-SLOPE*FLOAT(IX))+NY,NY)+1
      IBIT=MOD(IX,NBPW)+1
      IX=IX/NBPW+1
      I1=IS2(IX,IY)
      I2=MASK(IBIT)
      IV=AND(I1,I2)
      IF(IV.NE.Ø)GO TO IRET,(6Ø,9Ø,12Ø)
      CALL SDLINE(X1,Y1,X2,Y2)
      GO TO IRET,(6Ø,9Ø,12Ø)
  12Ø CONTINUE
  13Ø CONTINUE
      CALL PLTOUT
C     SUBROUTINE PLTOUT PLOTS THE SEQUENCES TAKING INTO ACCOUNT
C     THE MINIMUM DISTANCE BETWEEN BEGINNING AND ENDPOINTS OF THE
C     SEQUENCES.
      DO 15Ø I=1,NV
      DO 14Ø J=1,NW
      IF(IOBJS(I,J).EQ.Ø)GO TO 14Ø
      IX=(ST1(I,J,1)-S(1))*RX+Ø.5
      TWK=SLOPE*FLOAT(IX)-Ø.5
      IY=MOD(IFIX((ST1(I,J,2)-S(3))*RY-TWK)+NY,NY)+1
      IBIT=MOD(IX,NBPW)+1
      IX=IX/NBPW+1
      I1=IS2(IX,IY)
      I2=MASK(IBIT)
      IS2(IX,IY)=OR(I1,I2)
  14Ø CONTINUE
  15Ø CONTINUE
      RETURN
      END
C*****MACHINE DEPENDENT  ***********************************************
      INTEGER FUNCTION AND(I,J)
C*****THIS VERSION FOR CDC 6ØØØ SERIES ********************************
C*********  .AND.  USED AS MASKING  OPERATOR.   ************************
      AND=I.AND.J
      RETURN
      END
C*****MACHINE DEPENDENT  ***********************************************
      INTEGER FUNCTION OR(I,J)
C*****THIS VERSION FOR CDC6ØØØ SERIES *********************************
C******  .OR.  USED AS MASKING  OPERATOR.  ***************************
      OR=I.OR.J
      RETURN
      END
      SUBROUTINE SDLINE(X1,Y1,X2,Y2)
C     PEN-UP MINIMIZING VERSION OF THE VISIBLE SURFACE PLOTTING PROGRAM
C     ORIGINAL PROGRAM BY T. WRIGHT , COMMUN. ACM 17, 3(MARCH 1974)
C     PP 152-155.
C     AUTHORS , A.C.M. VAN SWIETEN (*) AND J.TH.M. DE HOSSON (**)
C
C     (*) MATHEMATICAL INSTITUTE, STATE UNIVERSITY GRONINGEN,
C         P.O. BOX 8ØØ, GRONINGEN, THE NETHERLANDS (PRESENT ADDRESS =
C         VSSG,P.O. BOX 3Ø32, LEYDEN, THE NETHERLANDS).
C     (**)NORTHWESTERN UNIVERSITY, DEPT. MATERIALS  SCIENCE AND
C     ENGINEERING, THE TECHNOLOGICAL INSTITUTE, EVANSTON,ILLINOIS 6Ø2Ø1,
C     U.S.A. ( ON LEAVE OF ABSENCE, LABORATORIUM VOOR FYSISCHE METAAL-
C     KUNDE, MATERIALS SCIENCE CENTRE, NIJENBORGH 18, GRONINGEN,
C     THE NETHERLANDS , SEPT.1976- SEPT. 1977).
C
C     IN THE ORIGINAL SUBROUTINE DANDR ONE HAS TO ADD THE
```

```
C     FOLLOWING FIVE  STATEMENTS
C     1)2) SS=Ø.Ø4,NSEQ=Ø,BEFORE SLOPE=DX/DY WHICH INITIALIZE SDLINE.
C     3) ADD  COMMON/TOM1/NSEQ,SS TO  DANDR.
C     4) CALL SDLINE(X1,Y1,X2,Y2) INSTEAD OF LINE(X1,Y1,X2,Y2).
C     5) CALL PLTOUT AFTER STATEMENT 13Ø CONTINUE IN DANDR.
C
C     THE SUBROUTINE SDLINE(X1,Y1,X2,Y2) TEMPERARILY STORES THE
C     SEGMENTS IN ORDER TO BUILT UP THE SEQUENCES. THIS IS DONE BY
C     COMPARING THE LAST POINT OF EACH SEQUENCE WITH THE ENDPOINTS
C     OF A SEGMENT. SDLINE IS ASSUMED TO DRAW A LINE FROM (X1,Y1)
C     TO THE POINT (X2,Y2) UTILIZING THE SUBROUTINES PLTOUT AND PLOT.
C
C     LOGICAL OPERATIONS  .AND.  , .OR.
C************     MACHINE DEPENDANT  CONSTANTS  ***********************
C     CDC 6ØØØ AND CDC 7ØØØ SERIES.
      DIMENSION XX(8Ø,2Ø),YY(8Ø,2Ø),NN(2Ø)
      COMMON/TOM/NN,XX,YY
      COMMON/TOM1/NSEQ,S
      IF(NSEQ.EQ.Ø) GOTO 2Ø
C     SEARCH FOR CONTINUATION POINT.
      DO 1Ø ISEQ=1,NSEQ
      INN=NN(ISEQ)
      ISW=Ø
    5 XL=XX(INN,ISEQ)
      YL=YY(INN,ISEQ)
C     TRUE IN NEXT STATEMENT MEANS CONTINUATION POINT FOUND
C      .AND.   LOGICAL  MULTIPLICATION.
      IF((ABS(X1-XL).LE.S).AND.(ABS(Y1-YL).LE.S)) GOTO 5Ø
      IF((ABS(X2-XL).LE.S).AND.(ABS(Y2-YL).LE.S)) GOTO 4Ø
C      LOGICAL EXPRESSION = INCLUSIVE  .OR.
      IF((INN.GT.2).OR.(ISW.NE.Ø)) GOTO 1Ø
      XBL=XX(1,ISEQ)
      YBL=YY(1,ISEQ)
      XX(1,ISEQ)=XX(2,ISEQ)
      YY(1,ISEQ)=YY(2,ISEQ)
      XX(2,ISEQ)=XBL
      YY(2,ISEQ)=YBL
      ISW=1
      GOTO 5
   1Ø CONTINUE
C     NEW SEQUENCE
   2Ø IF(NSEQ.EQ.2Ø) CALL PLTOUT
      NSEQ=NSEQ+1
      XX(1,NSEQ)=X1
      XX(2,NSEQ)=X2
      YY(1,NSEQ)=Y1
      YY(2,NSEQ)=Y2
      NN(NSEQ)=2
      RETURN
C     CONTINUE OLD SEQUENCE
   4Ø X2=X1
      Y2=Y1
   5Ø INN=INN+1
      IF(INN.GT.8Ø) GOTO 2Ø
      XX(INN,ISEQ)=X2
      YY(INN,ISEQ)=Y2
      NN(ISEQ)=INN
      RETURN
      END
      SUBROUTINE PLTOUT
C
C     INSERT CALL PLTOUT AFTER STATEMENT 13Ø CONTINUE IN
C     THE ORIGINAL DANDR SUBROUTINE.
C
C     THIS SUBROUTINE PLOTS THE SEQUENCES TAKING INTO ACCOUNT
C     THE MINIMUM DISTANCE BETWEEN BEGINNING AND ENDPOINTS OF THE
C     SEQUENCES. THIS IS DONE BY ORDERING THE SEQUENCES IN AN
C     APROPRIATE  WAY AND BY INDICATING WHETHER THEY SHOULD BE PROCESSED
C     IN THE NORMAL ORDER OR REVERSED.
C****** UNDEFINED EXTERNAL REFERENCES  ******************************
C     SUBROUTINE PLOT(X,Y,IND)IS AVAILABLE IN THE CALCOMP PACKAGE.
C     PLOT(X,Y,IND) = TO MOVE THE PEN FROM ITS CURRENT POSITION
C                     TO A NEW POSITION.
C     X = X-COORDINATE, IN CM, OF NEW PEN POSITION RELATIVE TO ORIGIN.
C     Y = Y - COORDINATE, IN CM, OF NEW PEN POSITION RELATIVE TO ORIGIN.
C     IND = IS USED TO CONTROL VERTICAL POSITION OF THE PEN, THE
```

```
C           ESTABLISHING OF NEW ORIGINS, DUMPING OF THE BUFFER, AND
C           THE STARTING OF NEW BLOCKS.
C
      DIMENSION XX(8Ø,2Ø),YY(8Ø,2Ø),NN(2Ø),IND(2Ø),IDR(2Ø)
      COMMON/TOM/NN,XX,YY
      COMMON/TOM1/NSEQ,S
      IF(NSEQ.EQ.Ø) RETURN
      DO 1Ø I=1,NSEQ
      IND(I)=I
   1Ø CONTINUE
      IDR(1)=1
      ITEMP=NN(1)
      OLDX=XX(ITEMP,1)
      OLDY=YY(ITEMP,1)
      DO 3Ø I=2,NSEQ
      DMIN=1ØØØØØØ.
      DO 2Ø J=1,NSEQ
      K=IND(J)
      DX=XX(1,K)-OLDX
      DY=YY(1,K)-OLDY
      D=SQRT(DX*DX+DY*DY)
      IF(D.GE.DMIN) GOTO 15
      DMIN=D
      MINJ=J
      IDRT=1
   15 ITEMP=NN(K)
      DX=XX(ITEMP,K)-OLDX
      DY=YY(ITEMP,K)-OLDY
      D=SQRT(DX*DX+DY*DY)
      IF(D.GE.DMIN) GOTO 2Ø
      DMIN=D
      MINJ=J
      IDRT=-1
   2Ø CONTINUE
      L=IND(MINJ)
      IND(MINJ)=IND(I)
      IND(I)=L
      IDR(I)=IDRT
      IB=1
      IF(IDRT.NE.1) IB=NN(K)
      OLDX=XX(IB,K)
      OLDY=YY(IB,K)
   3Ø CONTINUE
      DO 5Ø I=1,NSEQ
      K=IND(I)
      N=NN(K)
      IB=1
      M3=IDR(I)
      IF(M3.NE.1) IB=N
      M1=IB+M3
C*****  UNDEFINED EXTERNAL REFERENCE  ** PLOT  *************************
      CALL PLOT(XX(IB,K),YY(IB,K),3)
      DO 4Ø L=2,N
C*****  UNDEFINED EXTERNAL REFERENCE  ** PLOT  *************************
      CALL PLOT(XX(M1,K),YY(M1,K),2)
      M1=M1+M3
   4Ø CONTINUE
      NN(K)=Ø
   5Ø CONTINUE
      NSEQ=Ø
      RETURN
      END
      FUNCTION ARCCOS(Y)
C     BECAUSE ACOS IS NOT A STANDARD FORTRAN FUNCTION THE PRESENT
C     FUNCTION ROUTINE IS AN APPROXIMATION FOR IT.
      PI=3.1415926
      X=ABS(Y)
      ARCCOS=(1.57Ø7288-Ø.2121144*X+Ø.Ø74261*X*X-Ø.Ø187293*X*X*X)*SQRT(1
     A.-X)
      IF(Y.LT.Ø.)ARCCOS=PI-ARCCOS
      RETURN
      END
```

Editor's note: *Algorithm 476 described here is available on magnetic tape from the Department of Computer Science, University of Colorado, Boulder, CO 80302. The cost for the tape is $16.00 (U.S. and Canada) or $18.00 (elsewhere). If the user sends a small tape (wt. less than 1 lb.) the algorithm will be copied on it and returned to him at a charge of $10.00 (U.S. only). All orders are to be prepaid with checks payable to ACM Algorithms. The algorithm is recorded as one file of BCD 80 character card images at 556 B.P.I., even parity, on seven track tape. We will supply algorithm at a density of 800 B.P.I. if requested. Cards for algorithms are sequenced starting at 10 and incremented by 10. The sequence number is right justified in column 80. Although we will make every attempt to insure that the algorithm conforms to the description printed here, we cannot guarantee it, nor can we guarantee that the algorithm is correct.—L.D.F. and A.K.C.*

Algorithm 476

Six Subprograms for Curve Fitting Using Splines Under Tension [E2]

A.K. Cline
National Center for Atmospheric Research,* P.O. Box 1470, Boulder, CO 80302

[Recd. 21 Apr. 1972 and 13 June 1973]

* Sponsored by the National Science Foundation.

Author's present address: Institute for Computer Applications in Science and Engineering, Mail Stop 132-C, NASA-Langley Research Center, Hampton, VA 23365

Key Words and Phrases: interpolation, splines, contouring, curve fitting

CR Categories: 5.13, 8.2

Language: Fortran

Description

The spline under tension package includes six subprograms: two in each of three problem areas. These implement the theory presented in [1]. The first pair, *CURV*1 and *CURV*2, solves the standard interpolation problem: determine a real-valued function that assumes values $\{y_i\}_{i=1}^n$ at abscissas $\{x_i\}_{i=1}^n$. The second pair, *KURV*1 and *KURV*2, solves the more general problem of passing a curve through a sequence of pairs $\{x_i, y_i\}_{i=1}^n$ in the plane. The third pair, *KURVP*1 and *KURVP*2, solves the same problem, but the solution curve is closed.

*CURV*1 and *KURV*1 require additional endpoint slope conditions to determine the solution. The user may omit the information in which case values are produced internally based upon the other input information. If three or more points are to be interpolated, these internal slope values are the slopes given by a quadratic polynomial interpolating the first three values for the initial slope and last three values for the terminal slope. If only two points are to be interpolated and no slope information is given, the resulting curve is a straight line. The subprogram *KURVP*1 determines periodic splines under tension, and thus no additional slope information is required.

In each pair of subprograms, the first is called only once, and sets up and solves the tridiagonal system to specify the spline. The second is used for the actual mapping of points. The function *CURV*2 returns an image point for a given real value. The subroutines *KURV*2 and *KURVP*2 return the image pairs in their parameter sequences. Each of these subprograms, *CURV*2, *KURV*2, and *KURVP*2, first must determine which data points are adjacent to the input value. This search usually begins with the leftmost values and proceeds until the correct interval is found. However, if a sequence of input values is to be mapped, the search can be made more efficient by ordering these values left to right. The search can then proceed on one call from where it ended on the previous call. All three subprograms include an efficiency option which in effect says, "You may proceed from where you stopped."

All the subprograms included require a natural exponential function named *EXP*. *KURV*1, *KURV*2, *KURVP*1, and *KURVP*2 require a square root function *SQRT*. The subroutine *KURV*1 requires the sine (*SIN*) and cosine (*COS*) functions, in addition to the function *ATAN*2 of two arguments which when given x and y (not both zero) returns an angle θ which satisfies $x = y \times tan\ (\theta)$. All of these are basic Fortran external functions.

References

1. Cline, A.K. Scalar- and planar-valued curve fitting using splines under tension. *Comm. ACM 17*, 4 (Apr. 1974), 218–220.

Algorithm

```
      SUBROUTINE CURV1(N, X, Y, SLP1, SLPN, YP, TEMP, SIGMA)
      INTEGER N
      REAL X(N), Y(N), SLP1, SLPN, YP(N), TEMP(N), SIGMA
C THIS SUBROUTINE DETERMINES THE PARAMETERS NECESSARY TO
C COMPUTE AN INTERPOLATORY SPLINE UNDER TENSION THROUGH
C A SEQUENCE OF FUNCTIONAL VALUES. THE SLOPES AT THE TWO
C ENDS OF THE CURVE MAY BE SPECIFIED OR OMITTED. FOR ACTUAL
C COMPUTATION OF POINTS ON THE CURVE IT IS NECESSARY TO CALL
C THE FUNCTION CURV2.
C ON INPUT--
C N IS THE NUMBER OF VALUES TO BE INTERPOLATED (N.GE.2),
C X IS AN ARRAY OF THE N INCREASING ABSCISSAE OF THE
C FUNCTIONAL VALUES,
C Y IS AN ARRAY OF THE N ORDINATES OF THE VALUES,(I.E.Y(K)
C IS THE FUNCTIONAL VALUE CORRESPONDING TO X(K)),
C SLP1 AND SLPN CONTAIN THE DESIRED VALUES FOR THE FIRST
C DERIVATIVE OF THE CURVE AT X(1) AND X(N), RESPECTIVELY.
C IF THE QUANTITY SIGMA IS NEGATIVE THESE VALUES WILL BE
C DETERMINED INTERNALLY AND THE USER NEED ONLY FURNISH
C PLACE-HOLDING PARAMETERS FOR SLP1 AND SLPN. SUCH PLACE-
C HOLDING PARAMETERS WILL BE IGNORED BUT NOT DESTROYED,
C YP IS AN ARRAY OF LENGTH AT LEAST N
C TEMP IS AN ARRAY OF LENGTH AT LEAST N WHICH IS USED FOR
C SCRATCH STORAGE,
C AND
C SIGMA CONTAINS THE TENSION FACTOR. THIS IS NON-ZERO AND
C INDICATES THE CURVINESS DESIRED. IF ABS(SIGMA) IS NEARLY
C ZERO (E.G. .001) THE RESULTING CURVE IS APPROXIMATELY A
C CUBIC SPLINE. IF ABS(SIGMA) IS LARGE (E.G. 50.) THE
C RESULTING CURVE IS NEARLY A POLYGONAL LINE. THE SIGN
C OF SIGMA INDICATES WHETHER THE DERIVATIVE INFORMATION
C HAS BEEN INPUT OR NOT. IF SIGMA IS NEGATIVE THE ENDPOINT
C DERIVATIVES WILL BE DETERMINED INTERNALLY. A STANDARD
C VALUE FOR SIGMA IS APPROXIMATELY 1. IN ABSOLUTE VALUE.
C ON OUTPUT--
C YP CONTAINS VALUES PROPORTIONAL TO THE SECOND DERIVATIVE
C OF THE CURVE AT THE GIVEN NODES.
C N,X,Y,SLP1,SLPN AND SIGMA ARE UNALTERED,
      NM1 = N - 1
      NP1 = N + 1
      DELX1 = X(2) - X(1)
      DX1 = (Y(2)-Y(1))/DELX1
C DETERMINE SLOPES IF NECESSARY
      IF (SIGMA.LT.0.) GO TO 50
      SLPP1 = SLP1
      SLPPN = SLPN
```

```
C DENORMALIZE TENSION FACTOR
   1Ø SIGMAP = ABS(SIGMA)*FLOAT(N-1)/(X(N)-X(1))
C SET UP RIGHT HAND SIDE AND TRIDIAGONAL SYSTEM FOR YP AND
C PERFORM FORWARD ELIMINATION
      DELS = SIGMAP*DELX1
      EXPS = EXP(DELS)
      SINHS = .5*(EXPS-1./EXPS)
      SINHIN = 1./(DELX1*SINHS)
      DIAG1 = SINHIN*(DELS*.5*(EXPS+1./EXPS)-SINHS)
      DIAGIN = 1./DIAG1
      YP(1) = DIAGIN*(DX1-SLPP1)
      SPDIAG = SINHIN*(SINHS-DELS)
      TEMP(1) = DIAGIN*SPDIAG
      IF (N.EQ.2) GO TO 3Ø
      DO 2Ø I=2,NM1
        DELX2 = X(I+1) - X(I)
        DX2 = (Y(I+1)-Y(I))/DELX2
        DELS = SIGMAP*DELX2
        EXPS = EXP(DELS)
        SINHS = .5*(EXPS-1./EXPS)
        SINHIN = 1./(DELX2*SINHS)
        DIAG2 = SINHIN*(DELS*(.5*(EXPS+1./EXPS))-SINHS)
        DIAGIN = 1./(DIAG1+DIAG2-SPDIAG*TEMP(I-1))
        YP(I) = DIAGIN*(DX2-DX1-SPDIAG*YP(I-1))
        SPDIAG = SINHIN*(SINHS-DELS)
        TEMP(I) = DIAGIN*SPDIAG
        DX1 = DX2
        DIAG1 = DIAG2
   2Ø CONTINUE
   3Ø DIAGIN = 1./(DIAG1-SPDIAG*TEMP(NM1))
      YP(N) = DIAGIN*(SLPPN-DX2-SPDIAG*YP(NM1))
C PERFORM BACK SUBSTITUTION
      DO 4Ø I=2,N
        IBAK = NP1 - I
        YP(IBAK) = YP(IBAK) - TEMP(IBAK)*YP(IBAK+1)
   4Ø CONTINUE
      RETURN
   5Ø IF (N.EQ.2) GO TO 6Ø
C IF NO DERIVATIVES ARE GIVEN USE SECOND ORDER POLYNOMIAL
C INTERPOLATION ON INPUT DATA FOR VALUES AT ENDPOINTS.
      DELX2 = X(3) - X(2)
      DELX12 = X(3) - X(1)
      C1 = -C(DELX12+DELX1)/DELX12/DELX1
      C2 = DELX12/DELX1/DELX2
      C3 = -DELX1/DELX12/DELX2
      SLPP1 = C1*Y(1) + C2*Y(2) + C3*Y(3)
      DELN = X(N) - X(NM1)
      DELNM1 = X(NM1) - X(N-2)
      DELNN = X(N) - X(N-2)
      C1 = (DELNN+DELN)/DELNN/DELN
      C2 = -DELNN/DELN/DELNM1
      C3 = DELN/DELNN/DELNM1
      SLPPN = C3*Y(N-2) + C2*Y(NM1) + C1*Y(N)
      GO TO 1Ø
C IF ONLY TWO POINTS AND NO DERIVATIVES ARE GIVEN, USE
C STRAIGHT LINE FOR CURVE
   6Ø YP(1) = Ø.
      YP(2) = Ø.
      RETURN
      END
```

```
        FUNCTION CURV2(T, N, X, Y, YP, SIGMA, IT)
        INTEGER N, IT
        REAL T, X(N), Y(N), YP(N), SIGMA
  C THIS FUNCTION INTERPOLATES A CURVE AT A GIVEN POINT
  C USING A SPLINE UNDER TENSION. THE SUBROUTINE CURV1 SHOULD
  C BE CALLED EARLIER TO DETERMINE CERTAIN NECESSARY
  C PARAMETERS.
  C ON INPUT--
  C T CONTAINS A REAL VALUE TO BE MAPPED ONTO THE INTERPO-
  C LATING CURVE.
  C N CONTAINS THE NUMBER OF POINTS WHICH WERE INTERPOLATED
  C TO DETERMINE THE CURVE,
  C X AND Y ARE ARRAYS CONTAINING THE ORDINATES AND ABCISSAS
  C OF THE INTERPOLATED POINTS,
  C YP IS AN ARRAY WITH VALUES PROPORTIONAL TO THE SECOND
  C DERIVATIVE OF THE CURVE AT THE NODES
  C SIGMA CONTAINS THE TENSION FACTOR (ITS SIGN IS IGNORED)
  C IT IS AN INTEGER SWITCH. IF IT IS NOT 1 THIS INDICATES
  C THAT THE FUNCTION HAS BEEN CALLED PREVIOUSLY (WITH N,X,
  C Y,YP, AND SIGMA UNALTERED) AND THAT THIS VALUE OF T
  C EXCEEDS THE PREVIOUS VALUE. WITH SUCH INFORMATION THE
  C FUNCTION IS ABLE TO PERFORM THE INTERPOLATION MUCH MORE
  C RAPIDLY. IF A USER SEEKS TO INTERPOLATE AT A SEQUENCE
  C OF POINTS, EFFICIENCY IS GAINED BY ORDERING THE VALUES
  C INCREASING AND SETTING IT TO THE INDEX OF THE CALL.
  C IF IT IS 1 THE SEARCH FOR THE INTERVAL (X(K),X(K+1))
  C CONTAINING T STARTS WITH K=1.
  C THE PARAMETERS N,X,Y,YP AND SIGMA SHOULD BE INPUT
  C UNALTERED FROM THE OUTPUT OF CURV1.
  C ON OUTPUT--
  C CURV2 CONTAINS THE INTERPOLATED VALUE. FOR T LESS THAN
  C X(1) CURV2 = Y(1). FOR T GREATER THAN X(N) CURV2 = Y(N).
  C NONE OF THE INPUT PARAMETERS ARE ALTERED.
        S = X(N) - X(1)
  C DENORMALIZE SIGMA
        SIGMAP = ABS(SIGMA)*FLOAT(N-1)/S
  C IF IT.NE.1 START SEARCH WHERE PREVIOUSLY TERMINATED,
  C OTHERWISE START FROM BEGINNING
        IF (IT.EQ.1) I1 = 2
  C SEARCH FOR INTERVAL
     1Ø DO 2Ø I=I1,N
          IF (X(I)-T) 2Ø, 2Ø, 3Ø
     2Ø CONTINUE
        I = N
  C CHECK TO INSURE CORRECT INTERVAL
     3Ø IF (X(I-1).LE.T .OR. T.LE.X(1)) GO TO 4Ø
  C RESTART SEARCH  AND RESET I1
  C ( INPUT ''IT'' WAS INCORRECT )
        I1 = 2
        GO TO 1Ø
  C SET UP AND PERFORM INTERPOLATION
     4Ø DEL1 = T - X(I-1)
        DEL2 = X(I) - T
        DELS = X(I) - X(I-1)
        EXPS1 = EXP(SIGMAP*DEL1)
        SINHD1 = .5*(EXPS1-1./EXPS1)
        EXPS = EXP(SIGMAP*DEL2)
        SINHD2 = .5*(EXPS-1./EXPS)
        EXPS = EXPS1*EXPS
        SINHS = .5*(EXPS-1./EXPS)
        CURV2 = (YP(I)*SINHD1+YP(I-1)*SINHD2)/SINHS +
       * ((Y(I)-YP(I))*DEL1+(Y(I-1)-YP(I-1))*DEL2)/DELS
        I1 = I
        RETURN
        END
```

```
      SUBROUTINE KURV1(N, X, Y, SLP1, SLPN, XP, YP, TEMP, S,
     * SIGMA)
C THIS SUBROUTINE DETERMINES THE PARAMETERS NECESSARY TO
C COMPUTE A SPLINE UNDER TENSION PASSING THROUGH A SEQUENCE
C OF PAIRS  (X(1),Y(1)),...,(X(N),Y(N)) IN THE PLANE. THE
C SLOPES AT THE TWO ENDS OF THE CURVE MAY BE SPECIFIED OR
C OMITTED. FOR ACTUAL COMPUTATION OF POINTS ON THE CURVE IT
C IS NECESSARY TO CALL THE SUBROUTINE KURV2.
C ON INPUT--
C N IS THE NUMBER OF POINTS TO BE INTERPOLATED (N.GE.2),
C X IS AN ARRAY CONTAINING THE N X-COORDINATES OF THE
C POINTS,
C Y IS AN ARRAY CONTAINING THE N Y-COORDINATES OF THE
C POINTS,
C SLP1 AND SLPN CONTAIN THE DESIRED VALUES FOR THE SLOPE
C OF THE CURVE AT  (X(1),Y(1)) AND  (X(N),Y(N)), RESPEC-
C TIVELY. THESE QUANTITIES ARE IN DEGREES AND MEASURED
C COUNTERCLOCKWISE FROM THE POSITIVE X-AXIS. THE POSITIVE
C SENSE OF THE CURVE IS ASSUMED TO BE THAT MOVING FROM THE
C POINT 1 TO POINT N. IF THE QUANTITY SIGMA IS NEGATIVE
C THESE SLOPES WILL BE DETERMINED INTERNALLY AND THE USER
C NEED ONLY FURNISH PLACE-HOLDING PARAMETERS FOR SLP1 AND
C SLPN. SUCH PLACE-HOLDING PARAMETERS WILL BE IGNORED BUT
C NOT DESTROYED,
C XP,YP ARE ARRAYS OF LENGTH AT LEAST N,
C TEMP IS AN ARRAY OF LENGTH AT LEAST N WHICH IS USED FOR
C SCRATCH STORAGE,
C AND
C SIGMA CONTAINS THE TENSION FACTOR. THIS IS NON-ZERO AND
C INDICATES THE CURVINESS DESIRED. IF ABS(SIGMA) IS VERY
C LARGE (E.G. 5Ø.) THE RESULTING CURVE IS VERY NEARLY A
C POLYGONAL LINE. THE SIGN OF SIGMA INDICATES WHETHER
C SLOPE IN FORMATION HAS BEEN INPUT OR NOT. IF SIGMA IS
C NEGATIVE THE END-POINT SLOPES WILL BE DETERMINED
C INTERNALLY. A STANDARD VALUE FOR SIGMA IS APPROXIMATELY
C 1. IN ABSOLUTE VALUE.
C ON OUTPUT--
C N,X,Y,SLP1,SLPN, AND SIGMA ARE UNALTERED,
C XP AND YP CONTAIN INFORMATION ABOUT THE CURVATURE OF THE
C CURVE AT THE GIVEN NODES,
C AND
C S CONTAINS THE POLYGONAL ARCLENGTH OF THE CURVE.
      INTEGER N
      REAL X(N), Y(N), XP(N), YP(N), TEMP(N), S, SIGMA
      DEGRAD = 3.141592653589793 2/18Ø.
      NM1 = N - 1
      NP1 = N + 1
      DELX1 = X(2) - X(1)
      DELY1 = Y(2) - Y(1)
      DELS1 = SQRT(DELX1*DELX1+DELY1*DELY1)
      DX1 = DELX1/DELS1
      DY1 = DELY1/DELS1
C DETERMINE SLOPES IF NECESSARY
      IF (SIGMA.LT.Ø.) GO TO 7Ø
      SLPP1 = SLP1*DEGRAD
      SLPPN = SLPN*DEGRAD
C SET UP RIGHT HAND SIDES OF TRIDIAGONAL LINEAR SYSTEM FOR XP
C AND YP
   1Ø XP(1) = DX1 - COS(SLPP1)
      YP(1) = DY1 - SIN(SLPP1)
      TEMP(1) = DELS1
      S = DELS1
      IF (N.EQ.2) GO TO 3Ø
      DO 2Ø I=2,NM1
        DELX2 = X(I+1) - X(I)
        DELY2 = Y(I+1) - Y(I)
        DELS2 = SQRT(DELX2*DELX2+DELY2*DELY2)
        DX2 = DELX2/DELS2
        DY2 = DELY2/DELS2
        XP(I) = DX2 - DX1
        YP(I) = DY2 - DY1
        TEMP(I) = DELS2
        DELX1 = DELX2
        DELY1 = DELY2
        DELS1 = DELS2
        DX1 = DX2
        DY1 = DY2
C ACCUMULATE POLYGONAL ARCLENGTH
        S = S + DELS1
   2Ø CONTINUE
   3Ø XP(N) = COS(SLPPN) - DX1
      YP(N) = SIN(SLPPN) - DY1
C DENORMALIZE TENSION FACTOR
      SIGMAP = ABS(SIGMA)*FLOAT(N-1)/S
C PERFORM FORWARD ELIMINATION ON TRIDIAGONAL SYSTEM
      DELS = SIGMAP*TEMP(1)
      EXPS = EXP(DELS)
      SINHS = .5*(EXPS-1./EXPS)
      SINHIN = 1./(TEMP(1)*SINHS)
      DIAG1 = SINHIN*(DELS*.5*(EXPS+1./EXPS)-SINHS)
      DIAGIN = 1./DIAG1
      XP(1) = DIAGIN*XP(1)
      YP(1) = DIAGIN*YP(1)
      SPDIAG = SINHIN*(SINHS-DELS)
      TEMP(1) = DIAGIN*SPDIAG
      IF (N.EQ.2) GO TO 5Ø
      DO 4Ø I=2,NM1
        DELS = SIGMAP*TEMP(I)
        EXPS = EXP(DELS)
        SINHS = .5*(EXPS-1./EXPS)
        SINHIN = 1./(TEMP(I)*SINHS)
```

```
         DIAG2 = SINHIN*(DELS*(.5*(EXPS+1./EXPS))-SINHS)
         DIAGIN = 1./(DIAG1+DIAG2-SPDIAG*TEMP(I-1))
         XP(I) = DIAGIN*(XP(I)-SPDIAG*XP(I-1))
         YP(I) = DIAGIN*(YP(I)-SPDIAG*YP(I-1))
         SPDIAG = SINHIN*(SINHS-DELS)
         TEMP(I) = DIAGIN*SPDIAG
         DIAG1 = DIAG2
   40 CONTINUE
   50 DIAGIN = 1./(DIAG1-SPDIAG*TEMP(NM1))
      XP(N) = DIAGIN*(XP(N)-SPDIAG*XP(NM1))
      YP(N) = DIAGIN*(YP(N)-SPDIAG*YP(NM1))
C PERFORM BACK SUBSTITUTION
      DO 60 I=2,N
         IBAK = NP1 - I
         XP(IBAK) = XP(IBAK) - TEMP(IBAK)*XP(IBAK+1)
         YP(IBAK) = YP(IBAK) - TEMP(IBAK)*YP(IBAK+1)
   60 CONTINUE
      RETURN
   70 IF (N.EQ.2) GO TO 80
C IF NO SLOPES ARE GIVEN, USE SECOND OPDER INTERPOLATION ON
C INPUT DATA FOP SLOPES AT ENDPOINTS
      DELS2 = SQRT((X(3)-X(2))**2+(Y(3)-Y(2))**2)
      DELS12 = DELS1 + DELS2
      C1 = -C(DELS12+DELS1)/DELS12/DELS1
      C2 = DELS12/DELS1/DELS2
      C3 = -DELS1/DELS12/DELS2
      SX = C1*X(1) + C2*X(2) + C3*X(3)
      SY = C1*Y(1) + C2*Y(2) + C3*Y(3)
      SLPP1 = ATAN2(SY,SX)
      DELNM1 = SQRT((X(N-2)-X(NM1))**2+(Y(N-2)-Y(NM1))**2)
      DELN = SQRT((X(NM1)-X(N))**2+(Y(NM1)-Y(N))**2)
      DELNN = DELNM1 + DELN
      C1 = (DELNN+DELN)/DELNN/DELN
      C2 = -DELNN/DELN/DELNM1
      C3 = DELN/DELNN/DELNM1
      SX = C3*X(N-2) + C2*X(NM1) + C1*X(N)
      SY = C3*Y(N-2) + C2*Y(NM1) + C1*Y(N)
      SLPPN = ATAN2(SY,SX)
      GO TO 10
C IF ONLY TWO POINTS AND NO SLOPES ARE GIVEN, USE STRAIGHT
C LINE SEGMENT FOR CURVE
   80 XP(1) = 0.
      XP(2) = 0.
      YP(1) = 0.
      YP(2) = 0.
      RETURN
      END
```

```
      SUBROUTINE KURV2(T, XS, YS, N, X, Y, XP, YP, S, SIGMA)
      INTEGER N
      REAL T, XS, YS, X(N), Y(N), XP(N), YP(N), S, SIGMA
C THIS SUBROUTINE PERFORMS THE MAPPING OF POINTS IN THE
C INTERVAL (0.,1.) ONTO A CURVE IN THE PLANE. THE SUBROUTINE
C KURV1 SHOULD BE CALLED EARLIER TO DETERMINE CERTAIN
C NECESSARY PARAMETERS. THE RESULTING CURVE HAS A PARAMETRIC
C REPRESENTATION BOTH OF WHOSE COMPONENTS ARE SPLINES UNDER
C TENSION AND FUNCTIONS OF THE POLYGONAL ARCLENGTH PARAMETER
C .
C ON INPUT--
C T CONTAINS A REAL VALUE OF ABSOLUTE VALUE LESS THAN OR
C EQUAL TO 1. TO BE MAPPED TO A POINT ON THE CURVE. THE
C SIGN OF T IS IGNORED AND THE INTERVAL (0.,1.) IS MAPPED
C ONTO THE ENTIRE CURVE. IF T IS NEGATIVE THIS INDICATES
C THAT THE SUBROUTINE HAS BEEN CALLED PREVIOUSLY (WITH ALL
C OTHER INPUT VARIABLES UNALTERED) AND THAT THIS VALUE OF
C T EXCEEDS THE PREVIOUS VALUE IN ABSOLUTE VALUE. WITH
C SUCH INFORMATION THE SUBROUTINE IS ABLE TO MAP THE POINT
C MUCH MORE RAPIDLY. THUS IF THE USER SEEKS TO MAP A
C SEQUENCE OF POINTS ONTO THE SAME CURVE, EFFICIENCY IS
C GAINED BY ORDERING THE VALUES INCREASING IN MAGNITUDE
C AND SETTING THE SIGNS OF ALL BUT THE FIRST, NEGATIVE,
C N CONTAINS THE NUMBER OF POINTS WHICH WERE INTERPOLATED
C TO DETERMINE THE CURVE,
C X AND Y ARE ARRAYS CONTAINING THE X- AND Y-COORDINATES
C OF THE INTERPOLATED POINTS,
C XP AND YP ARE THE ARRAYS OUTPUT FROM KUPV2 CONTAINING
C CURVATURE INFORMATION,
C S CONTAINS THE POLYGONAL ARCLENGTH OF THE CURVE,
C SIGMA CONTAINS THE TENSION FACTOR (ITS SIGN IS IGNORED).
C THE PARAMETERS N,X,Y,XP,YP,S,AND SIGMA SHOULD BE INPUT
C UNALTERED FROM THE OUTPUT OF KURV1.
C ON OUTPUT--
C XS AND YS CONTAIN THE X-ANDY-COORDINATES OF THE IMAGE
C POINT ON THE CURVE.
C T,N,X,Y,XP,YP,S, AND SIGMA ARE UNALTERED.
C DENORMALIZE SIGMA
      SIGMAP = ABS(SIGMA)*FLOAT(N-1)/S
C STRETCH UNIT INTERVAL INTO ARCLENGTH DISTANCE
      TN = ABS(T*S)
C FOR NEGATIVE T START SEARCH WHERE PREVIOUSLY TERMINATED,
C OTHERWISE START FROM BEGINNING
      IF (T.LT.0.) GO TO 10
      I1 = 2
      XS = X(1)
      YS = Y(1)
      SUM = 0.
      IF (T.LE.0.) RETURN
   10 CONTINUE
C DETERMINE INTO WHICH SEGMENT TN IS MAPPED
      DO 30 I=I1,N
        DELX = X(I) - X(I-1)
        DELY = Y(I) - Y(I-1)
        DELS = SQRT(DELX*DELX+DELY*DELY)
        IF (SUM+DELS-TN) 20, 40, 40
   20   SUM = SUM + DELS
   30 CONTINUE
C IF ABS(T) IS GREATER THAN 1., RETURN TERMINAL POINT ON
C CURVE
      XS = X(N)
      YS = Y(N)
      RETURN
C SET UP AND PERFORM INTERPOLATION
   40 DEL1 = TN - SUM
      DEL2 = DELS - DEL1
      EXPS1 = EXP(SIGMAP*DEL1)
      SINHD1 = .5*(EXPS1-1./EXPS1)
      EXPS = EXP(SIGMAP*DEL2)
      SINHD2 = .5*(EXPS-1./EXPS)
      EXPS = EXPS1*EXPS
      SINHS = .5*(EXPS-1./EXPS)
      XS = (XP(I)*SINHD1+XP(I-1)*SINHD2)/SINHS +
     * ((X(I)-XP(I))*DEL1+(X(I-1)-XP(I-1))*DEL2)/DELS
      YS = (YP(I)*SINHD1+YP(I-1)*SINHD2)/SINHS +
     * ((Y(I)-YP(I))*DEL1+(Y(I-1)-YP(I-1))*DEL2)/DELS
      I1 = I
      RETURN
      END
```

```
      SUBROUTINE KURVP1(N, X, Y, XP, YP, TEMP, S, SIGMA)
      INTEGER N
      REAL X(N), Y(N), XP(N), YP(N), TEMP(1), S, SIGMA
C THIS SUBROUTINE DETERMINES THE PARAMETERS NECESSARY TO
C COMPUTE A SPLINE UNDER TENSION FORMING A CLOSED CUPVE IN
C THE PLANE AND PASSING THROUGH A SEQUENCE OF PAIRS
C (X(1),Y(1)),...,(X(N),Y(N)). FOR ACTUAL COMPUTATION OF
C POINTS ON THE CURVE IT IS NECESSARY TO CALL THE SUBPOUTINE
C KURVP2.
C ON INPUT--
C N IS THE NUMBER OF POINTS TO BE INTERPOLATED (N.GE.2),
C X IS AN ARRAY CONTAINING THE N X-COORDINATES OF THE
C POINTS,
C Y IS AN ARRAY CONTAINING THE N Y-COORDINATES OF THE
C POINTS,
C XP,YP ARE ARRAYS OF LENGTH AT LEAST N,
C TEMP IS AN ARRAY OF LENGTH AT LEAST 2*N WHICH IS USED
C FOR SCRATCH STORAGE,
C AND
C SIGMA CONTAINS THE TENSION FACTOR. THIS IS A NON-ZERO
C QUANTITY (WHOSE SIGN IS IGNORED) WHICH INDICATES THE
C CURVINESS DESIRED. IF ABS(SIGMA) IS VERY LARGE (E.G. 50.
C ) THE RESULTING CURVE IS VERY A POLYGON. A STANDARD
C VALUE FOR SIGMA IS APPROXIMATELY 1. IN ABSOLUTE VALUE.
C ON OUTPUT--
C N,X,Y, AND SIGMA ARE UNALTERED,
C XP AND YP CONTAIN INFORMATION ABOUT THE CURVATURE OF THE
C CURVE AT THE GIVEN NODES,
C AND
C S CONTAINS THE POLYGONAL ARCLENGTH OF THE CURVE.
      NM1 = N - 1
      NP1 = N + 1
C SET UP RIGHT HAND SIDES OF TRIDIAGONAL (WITH CORNER
C ELEMENTS) LINEAR SYSTEM FOR XP AND YP
      DELX1 = X(2) - X(1)
      DELY1 = Y(2) - Y(1)
      DELS1 = SQRT(DELX1*DELX1+DELY1*DELY1)
      DX1 = DELX1/DELS1
      DY1 = DELY1/DELS1
      XP(1) = DX1
      YP(1) = DY1
      TEMP(1) = DELS1
      S = DELS1
      DO 10 I=2,N
        IP1 = I + 1
        IF (I.EQ.N) IP1 = 1
        DELX2 = X(IP1) - X(I)
        DELY2 = Y(IP1) - Y(I)
        DELS2 = SQRT(DELX2*DELX2+DELY2*DELY2)
        DX2 = DELX2/DELS2
        DY2 = DELY2/DELS2
        XP(I) = DX2 - DX1
        YP(I) = DY2 - DY1
        TEMP(I) = DELS2
        DELX1 = DELX2
        DELY1 = DELY2
        DELS1 = DELS2
        DX1 = DX2
        DY1 = DY2
C ACCUMULATE POLYGONAL ARCLENGTH
        S = S + DELS1
   10 CONTINUE
      XP(1) = XP(1) - DX1
      YP(1) = YP(1) - DY1
C DENORMALIZE TENSION FACTOR
      SIGMAP = ABS(SIGMA)*FLOAT(N)/S
C PERFORM FORWARD ELIMINATION ON TRIDIAGONAL SYSTEM
      DELS = SIGMAP*TEMP(N)
      EXPS = EXP(DELS)
      SINHS = .5*(EXPS-1./EXPS)
      SINHIN = 1./(TEMP(N)*SINHS)
      DIAG1 = SINHIN*(DELS*.5*(EXPS+1./EXPS)-SINHS)
      DIAGIN = 1./DIAG1
      SPDIG1 = SINHIN*(SINHS-DELS)
      SPDIAG = 0.
      DO 20 I=1,N
        DELS = SIGMAP*TEMP(I)
        EXPS = EXP(DELS)
        SINHS = .5*(EXPS-1./EXPS)
        SINHIN = 1./(TEMP(I)*SINHS)
        DIAG2 = SINHIN*(DELS*(.5*(EXPS+1./EXPS))-SINHS)
        IF (I.EQ.N) GO TO 30
        DIAGIN = 1./(DIAG1+DIAG2-SPDIAG*TEMP(I-1))
        XP(I) = DIAGIN*(XP(I)-SPDIAG*XP(I-1))
        YP(I) = DIAGIN*(YP(I)-SPDIAG*YP(I-1))
        TEMP(N+I) = -DIAGIN*TEMP(NM1+I)*SPDIAG
        IF (I.EQ.1) TEMP(NP1) = -DIAGIN*SPDIG1
        SPDIAG = SINHIN*(SINHS-DELS)
        TEMP(I) = DIAGIN*SPDIAG
        DIAG1 = DIAG2
   20 CONTINUE
   30 TEMP(NM1) = TEMP(N+NM1) - TEMP(NM1)
      IF (N.EQ.2) GO TO 50
C PERFORM FIRST STEP OF BACK SUBSTITUTION
      DO 40 I=3,N
        IBAK = NP1 - I
        XP(IBAK) = XP(IBAK) - TEMP(IBAK)*XP(IBAK+1)
        YP(IBAK) = YP(IBAK) - TEMP(IBAK)*YP(IBAK+1)
        TEMP(IBAK) = TEMP(N+IBAK) - TEMP(IBAK)*TEMP(IBAK+1)
   40 CONTINUE
   50 XP(N) =
     * (XP(N)-SPDIG1*XP(1)-SPDIAG*XP(NM1))/(DIAG1+DIAG2+SPDIG1*T
     * EMP(1)+SPDIAG*TEMP(NM1))
      YP(N) =
     * (YP(N)-SPDIG1*YP(1)-SPDIAG*YP(NM1))/(DIAG1+DIAG2+SPDIG1*T
     * EMP(1)+SPDIAG*TEMP(NM1))
C PERFORM SECOND STEP OF BACK SUBSTITUTION
      DO 60 I=1,NM1
        XP(I) = XP(I) + TEMP(I)*XP(N)
        YP(I) = YP(I) + TEMP(I)*YP(N)
   60 CONTINUE
      RETURN
      END



      SUBROUTINE KURVP2(T, XS, YS, N, X, Y, XP, YP, S, SIGMA)
      INTEGER N
      REAL T, XS, YS, X(N), Y(N), XP(N), YP(N), S, SIGMA
C THIS SUBROUTINE PERFORMS THE MAPPING OF POINTS IN THE
C INTERVAL (0.,1.) ONTO A CLOSED CURVE IN THE PLANE. THE
C SUBROUTINE KURVP1 SHOULD BE CALLED EARLIER TO DETERMINE
C CERTAIN NECESSARY PARAMETERS. THE RESULTING CURVE HAS A
C PARAMETRIC REPRESENTATIONBOTH OF WHOSE COMPONENTS ARE
C PERIODIC SPLINES UNDER TENSION AND FUNCTIONS OF THE POLY-
C GONAL ARCLENGTH PARAMETER.
C ON INPUT--
C T CONTAINS A REAL VALUE OF ABSOLUTE VALUE LESS THAN OR
C EQUAL TO 1. TO BE MAPPED TO A POINT ON THE CURVE. THE
C SIGN OF T IS IGNORED AND THE INTERVAL (0.,1.) IS MAPPED
C ONTO THE ENTIRE CLOSED CURVE. IF T IS NEGATIVE THIS
C INDICATES THAT THE SUBROUTINE HAS BEEN CALLED PREVIOUSLY
C (WITH ALL OTHER INPUT VARIABLES UNALTERED) AND THAT
C THIS VALUE OF T EXCEEDS THE PREVIOUS VALUE IN ABSOLUTE
C VALUE. WITH SUCH INFORMATION THE SUBROUTINE IS ABLE TO
C MAP THE POINT MUCH MORE RAPIDLY. THUS IF THE USER SEEKS
C TO MAP A SEQUENCE OF POINTS ONTO THE SAME CURVE,
C EFFICIENCY IS GAINED BY ORDERING THE VALUES INCREASING
C IN MAGNITUDE AND SETTING THE SIGNS OF ALL BUT THE FIRST,
C NEGATIVE,
C N CONTAINS THE NUMBER OF POINTS WHICH WERE INTERPOLATED
C TO DETERMINE THE CURVE,
C X AND Y ARE ARRAYS CONTAINING THE X- AND Y-COORDINATES
C OF THE INTERPOLATED POINTS,
C XP AND YP ARE THE ARRAYS OUTPUT FROM KURVP1 CONTAINING
C CURVATURE INFORMATION,
C S CONTAINS THE POLYGONAL ARCLENGTH OF THE CURVE,
C SIGMA CONTAINS THE TENSION FACTOR (ITS SIGN IS IGNORED).
C THE PARAMETERS N,X,Y,XP,YP,S AND SIGMA SHOULD BE INPUT
C UNALTERED FROM THE OUTPUT OF KURVP1.
C ON OUTPUT--
C XS AND YS CONTAIN THE X- AND Y-COORDINATES OF THE IMAGE
C POINT ON THE CURVE.
C T,N,X,Y,XP,YP,S AND SIGMA ARE UNALTERED.
C DENORMALIZE SIGMA
      SIGMAP = ABS(SIGMA)*FLOAT(N)/S
C STRETCH UNIT INTERVAL INTO ARCLENGTH DISTANCE
      TN = ABS(T*S)
C FOR NEGATIVE T START SEARCH WHERE PREVIOUSLY TERMINATED,
C OTHERWISE START FROM BEGINNING
      IF (T.LT.0.) GO TO 10
      I1 = 2
      SUM = 0.
   10 IF (I1.EQ.1) GO TO 50
C DETERMINE INTO WHICH SEGMENT TN IS MAPPED
      DO 30 I=I1,N
        DELX = X(I) - X(I-1)
        DELY = Y(I) - Y(I-1)
        DELS = SQRT(DELX*DELX+DELY*DELY)
        IF (SUM+DELS-TN) 20, 40, 40
   20   SUM = SUM + DELS
   30 CONTINUE
      I = 1
      IM1 = N
      DELS = S - SUM
      GO TO 50
   40 IM1 = I - 1
C SET UP AND PERFORM INTERPOLATION
   50 DEL1 = TN - SUM
      DEL2 = DELS - DEL1
      EXPS1 = EXP(SIGMAP*DEL1)
      SINHD1 = .5*(EXPS1-1./EXPS1)
      EXPS = EXP(SIGMAP*DEL2)
      SINHD2 = .5*(EXPS-1./EXPS)
      EXPS = EXPS1*EXPS
      SINHS = .5*(EXPS-1./EXPS)
      XS = (XP(I)*SINHD1+XP(IM1)*SINHD2)/SINHS +
     * ((X(I)-XP(I))*DEL1+(X(IM1)-XP(IM1))*DEL2)/DELS
      YS = (YP(I)*SINHD1+YP(IM1)*SINHD2)/SINHS +
     * ((Y(I)-YP(I))*DEL1+(Y(IM1)-YP(IM1))*DEL2)/DELS
      I1 = I
      RETURN
      END
```

Algorithim 477

Generator of Set-Partitions to Exactly R Subsets [G7]

Gideon Ehrlich [Recd. 11 Dec. 1972 and 26 Feb. 1973]
Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel

Key Words and Phrases: partitions, subset generation, permutations
CR Categories: 5.39
Language: Algol

Description

Purpose. Procedure *PARTEXACT* produces, by successive calls, a sequence of all $S(n,r)$ partitions of a set of n distinct elements into exactly r mutually exclusive subsets. ($S(n,r)$ is the Stirling number of the second kind, see [1].) We assume that $n \geq r > 2$.

There is no distinction of order: neither within subsets nor among them. We assume the elements to be numbers 1, 2, . . . , n. (If this is not the case, we just index the elements.) We also assume that we have a sequence of r numbered cells in which the subsets are located. The first cell contains the number 1 (together with the whole subset to which 1 belongs), then each cell contains the minimal element not contained in the preceding cells. Partitions are represented by an address-array, a, of n components. Every j is located in the cell numbered $a(j)$. It follows that:

1. $a(1) = 1$,
2. $a(j) \leq \min\ (\max\limits_{m<j} a(m) + 1, r)$.

After each call to *PARTEXACT* we receive a new address-array, a, which differs from the old one in, at most, two components. A new partition is received from the old one by transferring s from the os cell to the ns cell, and if $u \neq 0$, then we have also to transfer u from the ou cell to the nu cell. Together with the last a we will get $i = 1$, and we must not call *PARTEXACT* again.

The variables. n, r, k, z are global integers; $p[2{:}n]$, $t[1{:}n]$ are global integer arrays; $a[1{:}n]$ is an integer array. The space required by *PARTEFACT* is thus $3n$ approximately.

Initialization. One can initiate *PARTEXACT* using the following block:

```
begin integer j;
    k := n − r + 1;
    for j = 1 step 1 until k do a[j] := p[j] := 1;
    for j := k + 1 step 1 until n do a[j] := 1 + j − k;
    i := k; t[k] := k − 1; t[k−1] := 0; z := 1
end
```

a defines the first partition. In the case $n = r$ we get $i = 1$, and we stop immediately. The variables must not be changed between calls.

PARTEXACT has the important feature of being loopless, so the computation time of the new partitions is uniformly bounded. There is no dependence on n (or r). The computation time of the whole sequence is thus a linear function of its length—$s(n,r)$. It is to be noted that much computation time is saved, provided the main program deals not with the entire newly generated partition but with the changed element(s) only.

For $r = 2$, *GRAY2* [2] has to be used with "0" and "1" specifying the first and the second cells, respectively. The initial address vector [A = (0, 0, . . . , 0)] must not be used. Together with the last partition *GRAY2* will set $i = 1$ (for the first time).

Algorithm details. z and k are the minimal numbers such that $a(k + 1), a(k+2), \ldots, a(n)$ are $z + 1, z + 2, \ldots, r$, respectively, $a(1), a(2), \ldots, a(i-1)$ are not changed until $a(i)$ takes all available values: that is, if $i > k$ then no other value but its present one, else, all values between 1 and $\min(\max_{m<i} a(m) + 1, r)$. All those values are ordered in a sequence starting at 1 and ending at 2 ("a 1–2 path") or vice versa ("a 2–1 path"). Each sequence can be illustrated as moving a route of i along all available cells each time visiting one new cell.

Each of the seven labels *ONE* . . . *SEVEN*, appearing in *PARTEXACT*, deals with a special segment of one of the two paths. It moves i to the appropriate new cell. *ONE* deals with the first move of an element initially located in the first cell. The roles of the other labels are illustrated in Figures 1 and 2. Each of the arrows describes the effect of the appropriate label.

If i enters the cell $z + 1$, we transfer $k + 1$ from that cell to the first one (from which it starts a 1–2 path). On the other hand, if the move of i empties its old cell, we transfer k to that cell. For each i, $p(i)$ denotes the segment of the i's path according to which i is moved. After each move of i, $i + 1$ moves a whole new route. After each move of $i + 1$, $i + 2$ moves a whole new route, and so on. Between two successive paths of i there will be a single move of some $j < i$.

Fig. 1. 1–2 path.

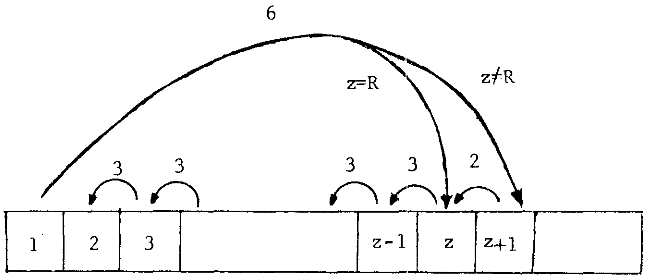
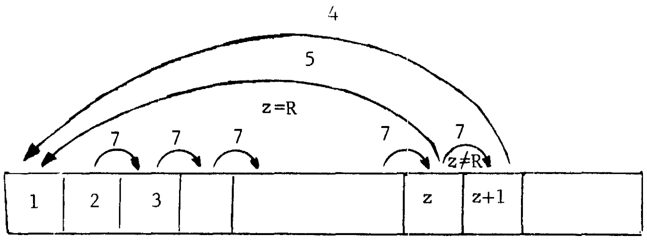


Fig. 2. 2–1 path.



t and i contain the information about the queue of elements to be moved. If i completes a path, then *NOGA* updates t and i.

Otherwise, *OFRA* does the job. Full explanations about *t*, its updating, and a description of the whole method are included in [3].

References

1. Even, S. *Algorithmic Combinatorics.* Macmillan, New York, 1973, Ch. 3.
2. Ehrlich, G. GRAY2—a binary reflected Gray Code Generator. (to be published).
3. Ehrlich, G. Loopless algorithm for generating permutations combinations and other combinatorial configurations. *J. ACM 20* (July 1973), 500–513.

Algorithm

```
procedure PARTEXACT (a, s, os, ns, u, ou, nu, i);
  integer array a;
  integer s, os, ns, u, ou, nu, i;
begin
  switch L := ONE, TWO, THREE, FOUR, FIVE, SIX, SEVEN;
  s := i; os := a[s]; u := 0;
  go to L[p[i]];
 ONE:
  ns := a[i] := z := 2; p[i] := 7;
  if i = k then
  begin
    u := k := k + 1; ou := a[u]; nu := a[u] := 1;
    p[k] = 6
  end;
  go to NOGA;
 TWO:
  ns := a[i] := z := z − 1;
  comment The old cell of i was emptied;
  u := k; ou := a[u]; nu := a[k] := z + 1; k := k − 1;
  if z = 2 then
    begin p[i] := 7; go to NOGA end;
  p[i] := 3; go to OFRA;
 THREE:
  ns := a[i] := a[i] − 1;
  if ns ≠ 2 then go to OFRA;
  p[i] := 7; go to NOGA;
 FOUR:
  u := k; ou := a[u]; nu := a[u] := z;
  z := z −1; k := k − 1;
 FIVE:
  ns := a[i] := 1; p[i] := 6;
 NOGA:
  if i = k then begin i = t[i]; go to EXIT end;
  if t[i] < 1 then
    begin if −t[i] ≠ i − 1 then t[i−1] := t[i]; t[i] := i − 1 end;
  if i ≠ k − 1 then begin t[k] := k − 1; t[k−1] := −i − 1 end;
  t[i+1] := t[i]; i := k;
  go to EXIT;
 SIX:
  if z = r then
    begin ns := a[i] := r; p[i] := 3 end;
  else
  begin
    ns := a[i] := z := z + 1; p[i] := 2;
    u := k := k + 1; ou := a[u]; nu := a[k] := 1; p[k] := 6
  end;
  go to OFRA;
 SEVEN:
  ns := a[i] := a[i] + 1;
  if ns ≥ z then
  begin
    if z = r then p[i] := 5
    else
    if a[i] = z + 1 then
    begin
      comment i enters the cell of k + 1;
      z := z + 1; p[i] := 4;
      u := k := k + 1; ou := a[u]; nu := a[k] := 1;
      p[k] := 6
    end
  end;
OFRA:
  if i = k then go to EXIT;
  t[k] := k − 1;
  if i ≠ k − 1 then t[k−1] = −i;
  i := k;
EXIT:
end PARTEXACT
```

Algorithm 478

Solution of an Overdetermined System of Equations in the l_1 Norm [F4]

I. Barrodale and F.D.K. Roberts, [Recd. 4 Aug. 1972 and 8 May 1973]
Department of Mathematics, University of Victoria, Victoria, B.C., Canada

Key Words and Phrases: **l_1 approximation, l_1 norm, overdetermined system of equations, linear programming, simplex method**
CR Categories: 5.13, 5.41
Language: Fortran

Description

The algorithm calculates an l_1 solution to an overdetermined system of m linear equations in n unknowns, i.e., given equations

$\sum_{j=1}^{n} a_{i,j} x_j = b_i$ for $i = 1, 2, \ldots, m, m \geq n$,

the algorithm determines a vector $x = \{x_j\}$ which minimizes the sum of the absolute values of the residuals

$$e(x) = \sum_{i=1}^{m} | b_i - \sum_{j=1}^{n} a_{i,j} x_j | . \qquad (1)$$

A typical application of the algorithm is that of solving the linear l_1 data fitting problem. Suppose that data consisting of m points with co-ordinates (t_i, y_i) is to be approximated by a linear approximating function $\alpha_1\phi_1 (t) + \alpha_2\phi_2 (t) + \cdots + \alpha_n\phi_n (t)$ in the l_1 norm. This is equivalent to finding an l_1 solution to the system of linear equations

$\sum_{j=1}^{n} \phi_j (t_i)\alpha_j = y_i$ for $i = 1, 2, \ldots, m$.

If the data contains some wild points (i.e. values of the dependent variable that are very inaccurate compared to the overall accuracy of the data), it is advisable to calculate an l_1 approximation rather than an l_2 (least-squares) approximation, or an l_∞ approximation.

The algorithm is a modification of the simplex method of linear programming applied to the primal formulation of the l_1 problem. A feature of the routine is its ability to pass through several simplex vertices at each iteration. The algorithm does not require that the matrix $\{a_{i,j}\}$ satisfy the Haar condition, nor does it require that it be of full rank. Complete details of the method may be found in [1]. Computational experience with this and other algorithms indicates that it is the most efficient yet devised for solving the l_1 problem.

The parameters M and N represent the number of equations and number of unknowns respectively. $M2$ and $N2$ should be set to $M + 2$ and $N + 2$ respectively. The simplex iterations are carried out in the two dimensional array A of size $(M2,N2)$. Initially the coefficients of the matrix $\{a_{i,j}\}$ should be stored in the first M rows and first N columns of A, and the right hand side vector $\{b_i\}$ should be stored in the array B. These values are destroyed by the routine. *TOLER* is a real variable which should be set to a small positive value. Essentially the routine regards any quantity as zero unless its magnitude exceeds *TOLER*. In particular, the routine will not pivot on any number whose magnitude is less than *TOLER*. Computational experience suggests that *TOLER* should be set to approximately $10^{-2d/3}$ where d represents the number of decimal digits of accuracy available (typically we run the routine on an IBM 370 using double precision (16 decimal digits) with *TOLER* set to 10^{-11}). On exit from the routine, the array X contains an l_1 solution $\{x_j\}$ and the array E contains the residuals $\{b_i - \sum_{j=1}^{n} a_{i,j} x_j\}$. The array S is used for workspace. The following information is stored in the array A on exit from the routine:

$A(M+1,N+1)$, the minimum value of (1), i.e. the minimum sum of absolute values of the residuals.

$A(M+1,N+2)$—the rank of the matrix $\{a_{i,j}\}$.

$A(M+2,N+1)$—exit code with the value 1 if a solution has been calculated successfully, and 2 if the calculations are terminated prematurely. This latter condition occurs only when rounding errors cause a pivot to be encountered whose magnitude is less than *TOLER*, and in this event all output information pertains to the last completed simplex iteration. This condition does not occur too frequently in practice, and then only with a large ill-conditioned problem. Since an l_1 solution is not necessarily unique, the routine attempts to determine if other optimal solutions exist. An exit code of 1 indicates that the solution is unique, while an exit code of 0 indicates that the solution almost certainly is not unique (this uncertainty can only be resolved by a close examination of the final simplex tableau contained in A: we do not consider such an examination to be warranted in practice). A solution may be nonunique simply because the matrix $\{a_{i,j}\}$ is not of full rank.

$A(M+2,N+2)$—number of iterations required by the simplex method.

References

1. Barrodale, I., and Roberts, F.D.K. An improved algorithm for discrete l_1 linear approximation. *SIAM J. Numer. Anal. 10*, 5 (1973), 839–848

Algorithm

```
      SUBROUTINE L1(M,N,M2,N2,A,B,TOLEP,X,E,S)
C THIS SUBROUTINE USES A MODIFICATION OF THE SIMPLEX METHOD
C OF LINEAR PROGRAMMING TO CALCULATE AN L1 SOLUTION TO AN
C OVER-DETERMINED SYSTEM OF LINEAR EQUATIONS.
C DESCRIPTION OF PARAMETERS.
C M      NUMBER OF EQUATIONS.
C N      NUMBER OF UNKNOWNS (M.GE.N).
C M2     SET EQUAL TO M+2 FOR ADJUSTABLE DIMENSIONS.
C N2     SET EQUAL TO N+2 FOR ADJUSTABLE DIMENSIONS.
C A      TWO DIMENSIONAL REAL ARRAY OF SIZE (M2,N2).
C        ON ENTRY, THE COEFFICIENTS OF THE MATRIX MUST BE
C        STORED IN THE FIRST M ROWS AND N COLUMNS OF A.
C        THESE VALUES ARE DESTROYED BY THE SUBROUTINE.
C B      ONE DIMENSIONAL REAL ARRAY OF SIZE M. ON ENTRY, B
C        MUST CONTAIN THE RIGHT HAND SIDE OF THE EQUATIONS.
C        THESE VALUES ARE DESTROYED BY THE SUBROUTINE.
C TOLER  A SMALL POSITIVE TOLERANCE. EMPIRICAL EVIDENCE
C        SUGGESTS TOLER=1Ø**(-D*2/3) WHERE D REPRESENTS
C        THE NUMBER OF DECIMAL DIGITS OF ACCURACY AVALABLE
C        (SEE DESCRIPTION).
C X      ONE DIMENSIONAL REAL ARRAY OF SIZE N. ON EXIT, THIS
C        ARRAY CONTAINS A SOLUTION TO THE L1 PROBLEM.
C E      ONE DIMENSIONAL REAL ARRAY OF SIZE M. ON EXIT, THIS
C        ARRAY CONTAINS THE RESIDUALS IN THE EQUATIONS.
C S      INTEGER ARRAY OF SIZE M USED FOR WORKSPACE.
C ON EXIT FROM THE SUBROUTINE, THE ARRAY A CONTAINS THE
C FOLLOWING INFORMATION.
C A(M+1,N+1)  THE MINIMUM SUM OF THE ABSOLUTE VALUES OF
C             THE RESIDUALS.
C A(M+1,N+2)  THE RANK OF THE MATRIX OF COEFFICIENTS.
C A(M+2,N+1)  EXIT CODE WITH VALUES.
C             Ø - OPTIMAL SOLUTION WHICH IS PROBABLY NON-
C                 UNIQUE (SEE DESCRIPTION).
C             1 - UNIQUE OPTIMAL SOLUTION.
C             2 - CALCULATIONS TERMINATED PREMATURELY DUE TO
C                 ROUNDING ERRORS.
C A(M+2,N+2)  NUMBER OF SIMPLEX ITERATIONS PERFORMED.
      DOUBLE PRECISION SUM
      REAL MIN, MAX, A(M2,N2), X(N), E(M), B(M)
      INTEGER OUT, S(M)
      LOGICAL STAGE, TEST
```

```
C BIG MUST BE SET EQUAL TO ANY VERY LARGE REAL CONSTANT.
C ITS VALUE HERE IS APPROPRIATE FOR THE IBM 370.
      DATA BIG/1.E75/
C INITIALIZATION.
      M1 = M + 1
      N1 = N + 1
      DO 10 J=1,N
        A(M2,J) = J
        X(J) = 0.
   10 CONTINUE
      DO 40 I=1,M
        A(I,N2) = N + I
        A(I,N1) = B(I)
        IF (B(I).GE.0.) GO TO 30
        DO 20 J=1,N2
          A(I,J) = -A(I,J)
   20   CONTINUE
   30   E(I) = 0.
   40 CONTINUE
C COMPUTE THE MARGINAL COSTS.
      DO 60 J=1,N1
        SUM = 0.D0
        DO 50 I=1,M
          SUM = SUM + A(I,J)
   50   CONTINUE
        A(M1,J) = SUM
   60 CONTINUE
C STAGE I.
C DETERMINE THE VECTOR TO ENTER THE BASIS.
      STAGE = .TRUE.
      KOUNT = 0
      KR = 1
      KL = 1
   70 MAX = -1.
      DO 80 J=KR,N
        IF (ABS(A(M2,J)).GT.N) GO TO 80
        D = ABS(A(M1,J))
        IF (D.LE.MAX) GO TO 80
        MAX = D
        IN = J
   80 CONTINUE
      IF (A(M1,IN).GE.0.) GO TO 100
      DO 90 I=1,M2
        A(I,IN) = -A(I,IN)
   90 CONTINUE
C DETERMINE THE VECTOR TO LEAVE THE BASIS.
  100 K = 0
      DO 110 I=KL,M
        D = A(I,IN)
        IF (D.LE.TOLER) GO TO 110
        K = K + 1
        B(K) = A(I,N1)/D
        S(K) = I
        TEST = .TRUE.
  110 CONTINUE
  120 IF (K.GT.0) GO TO 130
      TEST = .FALSE.
      GO TO 150
  130 MIN = BIG
      DO 140 I=1,K
        IF (B(I).GE.MIN) GO TO 140
        J = I
        MIN = B(I)
        OUT = S(I)
  140 CONTINUE
      B(J) = B(K)
      S(J) = S(K)
      K = K - 1
C CHECK FOR LINEAR DEPENDENCE IN STAGE I.
  150 IF (TEST .OR. .NOT.STAGE) GO TO 170
      DO 160 I=1,M2
        D = A(I,KR)
        A(I,KR) = A(I,IN)
        A(I,IN) = D
  160 CONTINUE
      KR = KR + 1
      GO TO 260
  170 IF (TEST) GO TO 180
      A(M2,N1) = 2.
      GO TO 350
  180 PIVOT = A(OUT,IN)
      IF (A(M1,IN)-PIVOT-PIVOT.LE.TOLER) GO TO 200
      DO 190 J=KR,N1
        D = A(OUT,J)
        A(M1,J) = A(M1,J) - D - D
        A(OUT,J) = -D
  190 CONTINUE
      A(OUT,N2) = -A(OUT,N2)
      GO TO 120
C PIVOT ON A(OUT,IN).
  200 DO 210 J=KR,N1
        IF (J.EQ.IN) GO TO 210
        A(OUT,J) = A(OUT,J)/PIVOT
  210 CONTINUE
      DO 230 I=1,M1
        IF (I.EQ.OUT) GO TO 230
        D = A(I,IN)
        DO 220 J=KR,N1
          IF (J.EQ.IN) GO TO 220
          A(I,J) = A(I,J) - D*A(OUT,J)
  220   CONTINUE
  230 CONTINUE
      DO 240 I=1,M1
        IF (I.EQ.OUT) GO TO 240
        A(I,IN) = -A(I,IN)/PIVOT
  240 CONTINUE
      A(OUT,IN) = 1./PIVOT
      D = A(OUT,N2)
      A(OUT,N2) = A(M2,IN)
      A(M2,IN) = D
      KOUNT = KOUNT + 1
      IF (.NOT.STAGE) GO TO 270
C INTERCHANGE ROWS IN STAGE I.
      KL = KL + 1
      DO 250 J=KR,N2
        D = A(OUT,J)
        A(OUT,J) = A(KOUNT,J)
        A(KOUNT,J) = D
  250 CONTINUE
  260 IF (KOUNT+KR.NE.N1) GO TO 70
C STAGE II.
      STAGE = .FALSE.
C DETERMINE THE VECTOR TO ENTER THE BASIS.
  270 MAX = -BIG
      DO 290 J=KR,N
        D = A(M1,J)
        IF (D.GE.0.) GO TO 280
        IF (D.GT.(-2.)) GO TO 290
        D = -D - 2.
  280   IF (D.LE.MAX) GO TO 290
        MAX = D
        IN = J
  290 CONTINUE
      IF (MAX.LE.TOLER) GO TO 310
      IF (A(M1,IN).GT.0.) GO TO 100
      DO 300 I=1,M2
        A(I,IN) = -A(I,IN)
  300 CONTINUE
      A(M1,IN) = A(M1,IN) - 2.
      GO TO 100
C PREPARE OUTPUT.
  310 L = KL - 1
      DO 330 I=1,L
        IF (A(I,N1).GE.0.) GO TO 330
        DO 320 J=KR,N2
          A(I,J) = -A(I,J)
  320   CONTINUE
  330 CONTINUE
      A(M2,N1) = 0.
      IF (KR.NE.1) GO TO 350
      DO 340 J=1,N
        D = ABS(A(M1,J))
        IF (D.LE.TOLER .OR. 2.-D.LE.TOLER) GO TO 350
  340 CONTINUE
      A(M2,N1) = 1.
  350 DO 380 I=1,M
        K = A(I,N2)
        D = A(I,N1)
        IF (K.GT.0) GO TO 360
        K = -K
        D = -D
  360   IF (I.GE.KL) GO TO 370
        X(K) = D
        GO TO 380
  370   K = K - N
        E(K) = D
  380 CONTINUE
      A(M2,N2) = KOUNT
      A(M1,N2) = N1 - KR
      SUM = 0.D0
      DO 390 I=KL,M
        SUM = SUM + A(I,N1)
  390 CONTINUE
      A(M1,N1) = SUM
      RETURN
      END
```

Footnote to Algorithm 478

The major portion of the computation performed by the above subroutine is transforming the two-dimensional array A at each iteration. We have experimented with a modified code which transforms the columns of A, one at a time, by passing each column to a second subroutine which involves only one-dimensional arrays. Savings in time of about 25 to 40 percent are normally achieved by this modification. This is because Fortran stores two-dimensional arrays columnwise.

To implement this modification in the above subroutine, the user should: (i) delete the eight lines immediately following statement number 20 up to and including statement number 22; (ii) replace these eight lines by

```
     DO 22 J=KR,N1
         IF(J.EQ.IN) GO TO 22
         CALL COL (A (1,J),A(1,IN),A(OUT,J),M1,OUT)
22       CONTINUE
```

and (iii) include the following subroutine

```
         SUBROUTINE COL (V1,V2,MLT,M1,IOUT)
         REAL V1(M1),V2(M1),MLT
         DO 1 I=1,M1
             IF(I.EQ.IOUT) GO TO 1
             V1(I)=V1(I)−V2(I)*MLT
1            CONTINUE
         RETURN
         END
```

Remark on Algorithm 478[F4]

Solution of an Overdetermined System of Equations in the l_1 Norm [I. Barrodale and F.D.K. Roberts, *Comm. ACM 17* (June 1974), 319–320]

Fred N. Fritsch and Alan C. Hindmarsh [Recd 23 Sept. 1974], Numerical Mathematics Group, Lawrence Livermore Laboratory, University of California, Livermore, CA 94550

This note is to point out an error in the "Footnote to Algorithm 478." To correspond to the published listing, the statement numbers in (i) of the second paragraph of the footnote should be 210 and 230, rather than 20 and 22. To be consistent with the published statement numbering, we would also recommend that statement number 22 be changed to 220 in the three places it occurs in the replacement coding of (ii).

Algorithm 479

A Minimal Spanning Tree Clustering Method [Z]

R.L. Page [Recd. 18 Feb. 1972, 8 Feb. 1973, and 29 Mar. 1973]
Department of Mathematics and Computer Science, Colorado State University, Fort Collins, CO 80521

Key Words and Phrases: clustering, pattern recognition. feature selection, minimal spanning trees

CR Categories: 3.63, 5.39, 5.5

Language: Fortran

Description

Zahn [2] describes a method for automatically detecting clusters in sets of points in N-space. The method is based on the construction of the minimal spanning tree of the complete graph on the input set of points. The motivation for using the minimal spanning tree includes some evidence (cited in [2]) that it is related to human perception of dot pictures in two dimensions, but the method is applicable in any dimension.

Advantages of the method are that it requires little input other than the data points, it is relatively insensitive to permutations in the order of the data points, and the clusters it produces in two dimensions closely parallel clusters detected visually by humans when the data is displayed as a dot picture.

Storage requirements increase linearly with the n, the number of points. The minimal spanning tree is constructed using an algorithm due to Prim and Dijkstra as implemented by Whitney [1]. The time needed is approximately proportional to n^2. (Time also increases slowly with N.) Whitney's algorithm is repeated here because we need to keep some information about the tree structure which his algorithm does not retain in a convenient form.

The basic idea is to detect inherent separations in the data by deleting edges from the minimal spanning tree which are significantly longer than nearby edges. Such an edge is called inconsistent. Zahn suggests the following criterion: an edge is inconsistent if (1) its length is more than f times the average of the length of nearby edges, and (2) its length is more than s standard deviations larger than the average of the lengths of nearby edges (standard deviation computed on the lengths of nearby edges). The real numbers f and s may be adjusted by the user. The question of determining which edges are "nearby" is also answered by the user. We will say point P is nearby point Q if point P is connected to point Q by a path in the minimal spanning tree containing d or fewer edges (d is an integer determined by the user).

Deleting the inconsistent edges breaks up the tree into several connected subtrees. The points of each connected subtree are the members of a cluster.

Use of the program. There are two steps involved in clustering a point set using this Fortran implementation of Zahn's algorithm.

Step 1. Call the subroutine *GROW* to construct the minimal spanning tree of the point set. *GROW* needs four parameters: (1) an array of real numbers specifying the point set; (2) an integer specifying the dimension of the space in which the points lie; (3) an integer specifying the number of points in the set; and (4) a logical value, true if the user would like a description of the minimal spanning tree to be printed on unit 6, and false otherwise. The array of parameter (1) is treated as if it were a matrix (stored by columns) in which each column represents a point in the input point set. To be more specific, the array must be arranged so that its $(K-1)*DIMEN + I$th value is the Ith component of the Kth vector in the point set. ($DIMEN$ stands for the dimension of the space in which the points lie.)

Step 2. Call the subroutine *CLUSTR* to determine the clusters in the point set. *CLUSTR* needs six parameters: (1) the integer d defining the term "nearby"; (2) the real number f described above; (3) the real number s described above; (4) an array to be used for output; (5) the declared length of the output array; and (6) a logical value, true if the user desires a description of the clusters determined to be printed on unit 6, and false otherwise. If parameter (5) is zero, the output array (parameter (4)) will not be used. Otherwise, the output array, which we call C here, will be filled with integers as follows: the first element will be the number of clusters detected; the remaining elements will be arranged in blocks of varying length, each block describing one cluster the first element in each block being the number of points in the cluster, and the remaining elements of the block being the labels of the points in the cluster (a point's label will be its relative position in the input point set; thus the first point in the input has label 1, the second, label 2, etc.).

Once step 1 has been completed for a particular point set, step 2 may be repeated with different parameters without repeating step 1.

Restrictions. (1) As written, the program will handle only 100 data points, but that can be easily changed by increasing the dimensions of three arrays in *GROW* and five arrays in *CLUSTR* (see program for directions). (2) The first parameter in *CLUSTR* must not be larger than 18. This too can be easily changed by increasing the dimension of two arrays in *CLUSTR* (see program). (3) Blank common is used to store the minimal spanning tree.

Tests. The program has been tested on a CDC 6400 with several different input point sets of varying size and dimension, both artificially generated and real data. The artificially generated data included three two dimensional point sets with two, four, and five clusters and one three-dimensional point set with eight clusters as well as some higher-dimensional, larger point sets used for timing analysis. Time to run *GROW* increases like n^2; time to run *CLUSTR* normally increases like n, but in the worst case increases like n^2.

References

1. Whitney, V.K.M. Algorithm 422 Minimal spanning tree. *Comm. ACM 15*, 4 (Apr. 1972), 273-274.

2. Zahn, C.T. Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Trans. on Computers, C-20* (1971), 68-86.

Algorithm

```
C TO CLUSTER A POINT SET USING THIS ALGORITHM, TWO THINGS
C NEED TO BE DONE. (1) BUILD THE MINIMAL SPANNING TREE BY
C CALLING GROW, AND (2) DELETE  ITS INCONSISTENT BRANCHES BY
C CALLING CLUSTR.  ONCE STEP (1) HAS BEEN DONE, STEP (2) CAN
C BE REPEATED OVER AND OVER WITH DIFFERENT PARAMETERS.
C SEE THE BEGINNINGS OF GROW AND CLUSTR FOR EXPLANATIONS OF
C THE PARAMETERS.
C CURRENTLY, THE ARRAYS ARE DIMENSIONED TO HANDLE UP TO 100
C POINTS.  TO CHANGE THIS, SIMPLY CHANGE THE SIZE OF THE
C ARRAYS  MST, NIT, AND UI IN GROW AS DIRECTED BELOW THEIR
C DECLARATIONS.         ALSO, CHANGE THE LENGTHS OF
```

Funds for computer time used in development of this algorithm were provided by National Science Foundation Grant GJ561.

```
C THE ARRAYS EDGE ST, EDGE PT, AVE, SQ, AND NUMNEI AS
C DIRECTED IN THE SUBROUTINE CLUSTR.  IN ADDITION, IF THE
C PARAMETER D IN CLUSTR WILL BE LARGER THAN 18, CHANGE THE
C LENGTHS OF THE ARRAYS NEIG ST AND NEIG PT AS DIRECTED.
      SUBROUTINE GROW(DATA, DIMEN, NUMPTS, PRINT)
      INTEGER DIMEN, NUMPTS
      DIMENSION DATA(1)
      LOGICAL PRINT
C THIS SUBROUTINE COMPUTES THE MINIMAL SPANNING TREE OF THE
C COMPLETE GRAPH ON THE  NUM PTS  POINTS IN ARPAY   DATA
C EACH POINT IS A VECTOR WITH  DIMEN  COMPONENTS STORED IN
C CONTIGUOUS LOCATIONS IN THE ARRAY DATA.  SPECIFICALLY,
C DATA( (K-1)*DIMEN +I ) IS THE I-TH COMPONENT OF THE K-TH
C VECTOR.  THE ARRAY DATA MAY CONTAIN NUMBERS IN EITHER
C INTEGER OR FLOATING POINT FORMAT AS LONG AS THE FORMAT IS
C CONSISTENT WITH THE TYPE SPECIFICATION OF THE PARAMETERS
C IN THE FUNCTION DIST.
C IF THE PARAMETER  PRINT  HAS THE VALUE .TRUE., THEN A
C A DESCRIPTION OF THE MINIMAL SPANNING TREE IS PRINTED ON
C UNIT 6.  EACH NODE IS LABELED WITH AN INTEGER INDICATING
C ITS RELATIVE POSITION IN THE APRAY DATA.
      INTEGER DIM, N, MST(8ØØ), LOC(1), NBR(1), NXT(1)
      REAL WT(1)
      EQUIVALENCE (MST,LOC,NBR,WT,NXT)
      COMMON DIM, N, MST
      INTEGER LASTPT, FREE, PT
C MST (ALIAS LOC, NER, WT, NXT) IS A DESCRIPTION OF THE
C MINIMAL SPANNING TREE. IT CONTAINS ONE LIST FOR EACH NODE.
C THE POINTERS TO THE HEADS OF THESE LISTS ARE STOPED IN THE
C FIRST N=NUM PTS LOCATIONS OF MST AND GO BY THE NAME MST.
C THE FIRST ELEMENT OF EACH LIST CONSISTS OF FOUR FIELDS
C STORED IN CONTIGUOUS WORDS OF MST. EACH FIELD IS CALLED BY
C A NAME WHICH IS AN ALIAS OF MST.
C FIELD 1: LOCATION IN DATA OF THE NODE (LOC)
C FIELD 2: NAME OF NEIGHBORING NODE (NBP)
C FIELD 3: WEIGHT OF THIS BRANCH (WT)
C FIELD 4: POINTER TO NEXT NEIGHBOP OR END MARK=Ø (NXT)
C EACH ADDITIONAL ELEMENT OF THE LIST CONSISTS OF THPEE
C FIELDS.  FIELD 1 ABOVE IS OMITTED.
C THE LENGTH OF THE ARRAY MST MUST BE AT LEAST 8*N .
C THE  MINIMAL SPANNING TREE IS COMPUTED USING THE ALGORITHM
C OF PRIM AND DIJKSTRA AS IMPLEMENTED BY WHITNEY (CACM 15,
C APR 1972).
C EACH COLUMN OF NIT IS A PAIR (NIT(1,I),NIT(2,I),I=1,NITP)
C DENOTING A NODE NOT (YET) IN THE TREE AND ITS NEAFEST
C NEIGHBOR IN THE CURRENT TREE. UI(I) IS THE LENGTH OF THE
C EDGE (NIT(1,I),NIT(2,I)). THE LENGTH OF THE APPAY UI AND
C THE NUMBEP OF COLUMNS OF NIT CANNOT BE LESS THAN N.
      INTEGER NIT(2,1ØØ)
      REAL UI(1ØØ)
      DIM = DIMEN
      N = NUMPTS
C COMPUTE MINIMAL SPANNING TREE USING ALGORITHM OF WHITNEY
C INITIALIZE NODE LABEL ARRAYS AND SET UP LIST FOP NODE N=KP
      NITP = N - 1
      KP = N
      KPDATA = (KP-1)*DIM + 1
      DO 1Ø I=1,NITP
        IDATA = (I-1)*DIM + 1
        NIT(1,I) = I
        UI(I) = DIST(DATA(IDATA),DATA(KPDATA),DIM)
        NIT(2,I) = KP
   1Ø CONTINUE
      FREE = N + 1
      MST(KP) = FPEE
      LOC(FPEE) = (KP-1)*DIM + 1
      FREE = FREE + 1
      NXT(FFEE+2) = Ø
C UPDATE LABEL OF NODES NOT YET IN TREE.
   2Ø KPDATA = (KP-1)*DIM + 1
      DO 3Ø I=1,NITP
        IDATA = (NIT(1,I)-1)*DIM + 1
        D = DIST(DATA(IDATA),DATA(KPDATA),DIM)
        IF (UI(I).LE.D) GO TO 3Ø
        UI(I) = D
        NIT(2,I) = KP
   3Ø CONTINUE
C FIND NODE OUTSIDE TREE NEAREST TO TREE
      UK = UI(1)
      DO 4Ø I=1,NITP
        IF (UI(I).GT.UK) GO TO 4Ø
        UK = UI(I)
        K = I
   4Ø CONTINUE
C ADD NEW EDGE TO MST
C ADD NEIGHBOR TO LIST OF NODE NIT(2,K)
C CHANGE END OF LIST MAPK TO POINT TO NEXT NEIGHBOP
      PT = LASTPT(NIT(2,K))
      NXT(PT) = FREE
C ENTER NAME OF NEIGHBOR
      NBR(FREE) = NIT(1,K)
C ENTER WEIGHT OF THIS BRANCH (OFFSET PICKS UP WT FIELD)
      WT(FREE+1) = UI(K)
C PUT IN END OF LIST MARK (OFFSET PICKS UP POINTER FIELD)
      NXT(FPEE+2) = Ø
      FPEE = FREE + 3
C NEW NODE--CREATE ITS NEIGHBOF LIST
C SET UP HEAD POINTEP
      NODE = NIT(1,K)
      MST(NODE) = FPEE
C ENTER LOCATION OF THIS NODE IN DATA
      LOC(FPEE) = (NODE-1)*DIM + 1
C ENTER NAME OF NEIGHBOPING NODE (OFFSET PICKS UP NBF FIELD)
      NBR(FREE+1) = NIT(2,K)
C ENTER WEIGHT OF THIS BRANCH (OFFSET PICKS UP WT FIELD)
      WT(FREE+2) = UI(K)
C ENTER END OF LIST MARK (OFFSET PICKS UP POINTER FIELD)
      NXT(FREE+3) = Ø
      FREE = FREE + 4
      KP = NIT(1,K)
C DELETE NEW  TREE NODE FROM ARPAY NIT
      UI(K) = UI(NITP)
      NIT(1,K) = NIT(1,NITP)
      NIT(2,K) = NIT(2,NITP)
      NITP = NITP - 1
C THE MST IS FINISHED WHEN IT CONTAINS ALL NODES
      IF (NITP.NE.Ø) GO TO 2Ø
      IF (PRINT) CALL PPTREE
      RETURN
      END
```

```
      SUBROUTINE CLUSTR(D, FACTOR, SPREAD, C, CLEN, PRINT)
      INTEGER D, CLEN, C(CLEN)
      REAL FACTOR, SPREAD
      LOGICAL PRINT
C THIS SUBROUTINE FINDS THE CLUSTEPS OF A POINT SET USING
C A MINIMAL SPANNING TREE CLUSTERING METHOD OF ZAHN.  THE
C MINIMAL SPANNING TREE, COMPUTED BY SUBROUTINE GROW, IS
C STORED IN BLANK COMMON.
C THE ZAHN ALGORITHM FINDS CLUSTERS BY DELETING INCONSISTENT
C EDGES FROM THE MINIMAL SPANNING TREE, AN INCONSISTENT EDGE
C BEING ONE WHOSE WEIGHT IS SIGNIFICANTLY LARGER THAN THE
C AVERAGE WEIGHT OF NEARBY EDGES.
C NEARBY  MEANS CONNECTED TO THE EDGE IN QUESTION BY A
C PATH CONTAINING   D   OR FEWER EDGES.
C SIGNIFICANTLY LARGER   MEANS
C WEIGHT .GT.   FACTOR  * AVERAGE
C AND  WEIGHT .GT. AVEPAGE +  SPREAD  * STANDAPD DEVIATION
C WHERE THE AVERAGE AND STANDARD DEVIATION ARE COMPUTED ON
C THE WEIGHTS OF NEARBY EDGES.
C THE OUTPUT VECTOR   C   DESCRIBES THE CLUSTERS DETERMINED.
C IT IS ARRANGED IN BLOCKS, EACH BLOCK DESCRIBING ONE
C CLUSTER. THE FIRST ELEMENT IN EACH BLOCK IS THE NUMBER
C OF NODES IN THE CLUSTER. THE REMAINING ELEMENTS ARE THE
C LABELS OF THE NODES IN THE CLUSTER, THE LABEL INDICATING
C THE RELATIVE POSITION OF THE NODE IN THE ARRAY DATA. THE
C FIRST BLOCK STARTS AT C(2).
C C(1) IS THE NUMBER OF CLUSTERS FOUND BY THE ALGORITHM.
C THE VALUE OF   C LEN   SHOULD BE THE TRUE SIZE OF
C THE ARRAY C.  IT IS USED TO PREVENT INVALID SUBSCRIPTS.
C IF C LEN IS ZERO, THE ARRAY C WILL NOT BE USED.
C IF THE PARAMETER   PRINT   HAS THE VALUE .TRUE., CLUSTERS
C ARE PRINTED OUT ON UNIT 6.
      INTEGER EDGEST(1Ø1), EDGELN, EDGEPT(1Ø1)
      REAL AVE(1ØØ), SQ(1ØØ), SUPPWT, W
      INTEGER NUMNEI(1ØØ)
      INTEGER NEIGST(2Ø), NEIGLN, NEIGPT(2Ø)
C THE ARRAY EDGE ST (EDGE STACK) IS A STACK OF NODES USED TO
C DIRECT THE SEARCH THROUGH THE TREE FOR INCONSISTENT EDGES.
C ITS LENGTH (EDGE LN) CAN GROW AS LARGE AS ONE MORE THAN
C THE NUMBER OF NODES IN THE TREE.
C THE ARRAY EDGE PT (EDGE POINTERS) IS A STACK OF POINTERS
C TO THE NEXT UNEXAMINED NEIGHBORING NODE OF THE NODE IN THE
C SAME POSITION IN EDGE ST. THUS THE LENGTH OF EDGE PT IS
C ALWAYS THE SAME AS THAT OF EDGE ST.
C THE ARRAY NEIG ST (NEIGHBOR STACK) IS A STACK OF NODES
C USED TO DIRECT THE AVERAGING OF THE WEIGHTS OF NEARBY
C EDGES. ITS LENGTH (NEIG LN) CAN GROW AS LARGE AS D+2.
C THE ARRAY NEIG PT IS USED IN CONJUNCTION WITH NEIG ST. ITS
C LENGTH CAN GROW AS LARGE A D+2.
C THE ARRAYS AVE AND SQ ARE USED TO EXPEDITE THE CALCULATION
C OF AVERAGE WEIGHTS. SPECIFICALLY, AVE(I) STORES THE SUM OF
C THE WEIGHTS OF EDGES EXTENDING FROM THE I-TH NODE AND
C SQ(I) STORES THE SUM OF THE SQUARES. SIMILARLY, NUMNEI(I)
C STORES THE NUMBER OF NEIGHBORS OF THE I-TH NODE. THUS EACH
C OF THESE ARRAYS MUST BE AS LONG AS THE NUMBER OF NODES.
      INTEGER FINDCN, A, B, DLESS1
      INTEGER CLS, INCLS(1), PARENT(1), BAKWRD, BEGCLS
      EQUIVALENCE (INCLS,EDGEST), (PARENT,EDGEPT)
      INTEGER CP, OTHEND
      INTEGER DIM, N, MST(1), LOC(1), NBR(1), NXT(1)
      REAL WT(1)
      EQUIVALENCE (MST,LOC,NBR,WT,NXT)
      COMMON DIM, N, MST
      IF (PRINT) WRITE (6,99998) D, FACTOR, SPREAD
      DLESS1 = D - 1
C COMPUTATION SECTION
C SUM BRANCH WEIGHTS OFF EACH NODE (DEPTH 1)
      DO 2Ø NODE=1,N
        NUMNEI(NODE) = 1
        K = MST(NODE)
        AVE(NODE) = WT(K+2)
        SQ(NODE) = WT(K+2)**2
        K = NXT(K+3)
   1Ø   IF (K.EQ.Ø) GO TO 2Ø
        AVE(NODE) = AVE(NODE) + WT(K+1)
        SQ(NODE) = SQ(NODE) + WT(K+1)**2
        NUMNEI(NODE) = NUMNEI(NODE) + 1
        K = NXT(K+2)
        GO TO 1Ø
   2Ø CONTINUE
C INITIALIZE EDGE STACK WITH NODE 1 SURROUNDED BY ITS FIRST
C TWO NEIGHBORS.  SINCE THE TOP TWO ELEMENTS OF THE STACK
C INDICATE THE DIRECTION OF TRAVEL ALONG A BRANCH, THE
C SEARCH WILL FIRST BE DIRECTED AWAY FROM NODE 1 IN THE
C DIRECTION OF ITS FIRST NEIGHBOR. WHEN ALL THE TPEE IN THAT
C DIRECTION IS SEARCHED, THE SEARCH WILL PROCEDE AWAY FROM
C ITS FIRST NEIGHBOR TOWARD NODE 1.
C THE EDGE PT STACK IS USED TO KEEP TRACK OF THE NEIGHBORS
C OF THE CORRESPONDING NODE IN EDGE ST WHICH HAVE ALREADY
C BEEN SEARCHED.  EDGE PT(I)  POINTS TO THE LOCATION OF
C EDGE ST(I+1)  IN THE LIST OF NEIGHBORS OF   EDGE ST(I)
      EDGELN = 3
      K = MST(1)
      EDGEST(2) = LOC(K)/DIM + 1
      EDGEST(1) = NBR(K+1)
      EDGEST(3) = NBR(K+1)
      EDGEPT(1) = FINDCN(EDGEST(1),EDGEST(2))
      EDGEPT(2) = K + 1
      EDGEPT(3) = -1
C CLIMB TREE TO NEXT UNTESTED BRANCH
   3Ø CALL CLIMB(EDGEPT, EDGEST, EDGELN, N)
      IF (EDGELN.LE.2) GO TO 7Ø
C CHECK THE EDGE BETWEEN NODE EDGE ST(EDGE LN -1) AND
C NODE EDGE ST(EDGE LN) FOR INCONSISTENCY.
      A = EDGEST(EDGELN-1)
      B = EDGEST(EDGELN)
```

```
C SUM WEIGHTS OF ALL BRANCHES NEARBY  BRANCH A--B
      NEARBY = Ø
      AV = Ø.
      STDDEV = Ø.
C INITIALZE NEIG ST TO SUM WEIGHTS HEADING OFF NODE B
      NEIGLN = 2
      NEIGST(1) = A
      NEIGPT(1) = EDGEPT(EDGELN-1)
      NEIGST(2) = B
      NEIGPT(2) = -1
      ASSIGN 5Ø TO OTHEND
C GO OUT TO DEPTH D-1 ALONG BRANCHES NOT YET ADDED
   4Ø CALL CLIMB(NEIGPT, NEIGST, NEIGLN, DLESS1)
C ADD WEIGHTS OF BRANCHES OFF THE TOP NODE LESS THE WEIGHT
C OF THE BRANCH SUPPORTING IT
      K = NEIGPT(NEIGLN-1)
      SUPPWT = WT(K+1)
      K = NEIGST(NEIGLN)
      AV = AV + AVE(K) - SUPPWT
      STDDEV = STDDEV + SQ(K) - SUPPWT**2
      NEARBY = NEARBY + NUMNEI(K) - 1
C WHEN DEPTH OF STACK RETURNS TO 2, ALL BRANCH WEIGHTS OFF
C THIS END HAVE BEEN ADDED
      IF (NEIGLN.LE.2) GO TO OTHEND, (5Ø,6Ø)
      NEIGLN = NEIGLN - 1
      GO TO 4Ø
C INITIALZE NEIG ST TO SUM WEIGHTS HEADING OFF NODE A
   5Ø NEIGLN = 2
      NEIGST(1) = B
      NEIGPT(1) = FINDCN(B,A)
      NEIGST(2) = A
      NEIGPT(2) = -1
      ASSIGN 6Ø TO OTHEND
      GO TO 4Ø
C TEST BRANCH A--B FOR INCONSISTENCY.
   6Ø AV = AV/FLOAT(NEARBY)
      STDDEV = SQRT(ABS(STDDEV/FLOAT(NEARBY)-AV**2))
      K = EDGEPT(EDGELN-1)
      W = WT(K+1)
      EDGELN = EDGELN - 1
      IF (W.LE.AV+SPREAD*STDDEV .OR. W.LE.FACTOR*AV) GO TO 3Ø
C BRANCH A--B IS INCONSISTENT.  DELETE IT.
      NBR(K) = -IABS(NBR(K))
      K = NEIGPT(1)
      NBR(K) = -IABS(NBR(K))
      GO TO 3Ø
C OUTPUT SECTION
C WE COLLECT THE CLUSTERS AS FOLLOWS:  1. START WITH FIRST
C NODE.  2. THROW IN ITS NEIGHBORS.  3. THROW IN NEIGHBORS
C OF NEIGHBORS  UNTIL NO NEW ONES CAN BE FOUND.  4. EACH
C TIME A DELETED BRANCH IS ENCOUNTERED, PUT OTHER END IN A
C LIST OF UNUSED NODES (AT TOP OF ARRAY   IN CLS).  5. WHEN
C A FULL CLUSTER IS COLLECTED , OUTPUT IT.  6. START AGAIN
C AT STEP 2 WITH A NODE FROM THE LIST OF UNUSED NODES.
   7Ø NUMIN = Ø
      CLS = Ø
      CP = 1
      K = MST(1)
      NXTCLS = N
      INCLS(NXTCLS) = LOC(K)/DIM + 1
      PARENT(NXTCLS) = Ø
      BAKWRD = Ø
C START CLUSTER WITH NEXT AVAILABLE UNUSED NODE
   8Ø CLS = CLS + 1
      NUMIN = NUMIN + 1
      BEGCLS = NUMIN
      NXTCN = NUMIN
      NODE = INCLS(NXTCLS)
      INLIST = PARENT(NXTCLS)
      INCLS(NUMIN) = NODE
      NXTCLS = NXTCLS + 1
C LET  K  POINT TO FIRST NEIGHBOR OF  NODE
   9Ø K = MST(NODE) + 1
C ADD NEIGHBOR TO CLUSTER AND RECORD IT ANCESTRY
  1ØØ NXTNBR = NBR(K)
      IF (NXTNBR.LT.Ø) GO TO 11Ø
      IF (NXTNBR.EQ.BAKWRD) GO TO 12Ø
      NUMIN = NUMIN + 1
      INCLS(NUMIN) = NXTNBR
      PARENT(NUMIN) = NODE
      GO TO 12Ø
C THIS NEIGHBOR IS IN A DIFFERENT CLUSTER--ADD TO UNUSED
  11Ø NXTNBR = -NXTNBR
      IF (NXTNBR.EQ.INLIST) GO TO 12Ø
      NXTCLS = NXTCLS - 1
      INCLS(NXTCLS) = NXTNBR
      PARENT(NXTCLS) = NODE
C GET NEXT NEIGHBOR
  12Ø K = NXT(K+2)
      IF (K.NE.Ø) GO TO 1ØØ
C ADD LIST OF NEIGHBORS OF NEXT ELEMENT OF THIS CLUSTER
      NXTCN = NXTCN + 1
      IF (NXTCN.GT.NUMIN) GO TO 13Ø
      NODE = INCLS(NXTCN)
      BAKWRD = PARENT(NXTCN)
      GO TO 9Ø
C END OF CLUSTER--DO OUTPUT
  13Ø CALL STORE(NUMIN-BEGCLS+1, C, CP, CLEN)
      IF (PRINT) WRITE (6,99999) CLS
      DO 14Ø I=BEGCLS,NUMIN
        IF (PRINT) WRITE (6,99997) INCLS(I)
        CALL STORE(INCLS(I), C, CP, CLEN)
  14Ø CONTINUE
      IF (NUMIN.LT.N) GO TO 8Ø
      CP = Ø
      CALL STORE(CLS, C, CP, CLEN)
      CALL FIXMST
      RETURN
99999 FORMAT(1HØ/8HØCLUSTER, I5, 12H CONSISTS OF)
99998 FORMAT(44H1THE TREE HAS BEEN CLUSTERED SEARCHING TO A
     * 8HDEPTH OF, I3/11X, 28HINCONSISTENT EDGES HAVE BEEN,
     * 27H DETERMINED BY A FACTOR OF , G11.4/11X, 1ØHAND A SPRE,
     * 6HAD OF , G11.4, 21H STANDARD DEVIATIONS.)
99997 FORMAT(1ØX, 4HNODE, I5)
      END
```

```
      REAL FUNCTION DIST(A, B, N)
      INTEGER N
      REAL A(N), B(N)
C THIS FUNCTION COMPUTES THE WEIGHT OF THE BRANCH BETWEEN
C NODE A AND NODE B. IT SHOULD BE WRITTEN TO SUIT THE DATA.
C THE TYPE DECLARATION OF A AND B SHOULD MATCH THE DATA.
C THIS VERSION COMPUTES THE USUAL EUCLIDEAN DISTANCE.
      DIST = (A(1)-B(1))**2
      DO 1Ø I=2,N
        DIST = DIST + (A(I)-B(I))**2
   1Ø CONTINUE
      DIST = SQRT(DIST)
      RETURN
      END
```

```
      SUBROUTINE CLIMB(POINTR, STACK, LN, D)
      INTEGER POINTR(1), STACK(1), LN, D
      INTEGER SPACE(2), MST(1), NBR(1), NXT(1)
      EQUIVALENCE (MST,NBR,NXT)
      COMMON SPACE, MST
C STARTING FROM THE NODE ON TOP OF THE STACK, CLIMB OUT
C TO DEPTH D OR TO A TERMINAL NODE, WHICHEVER OCCURS FIRST
   1Ø IF (LN.EQ.D+2) RETURN
      K = POINTR(LN)
      IF (K) 2Ø, 3Ø, 4Ø
C SET POINTER TO FIRST NEIGHBOR OF TOP NODE
   2Ø NODE = STACK(LN)
      POINTR(LN) = MST(NODE) + 1
      GO TO 5Ø
C BACK DOWN FROM TERMINAL NODE
   3Ø LN = LN - 1
C CLIMB OUT ON NEXT NEIGHBOR IF POSSIBLE
   4Ø POINTR(LN) = NXT(K+2)
      IF (POINTR(LN).EQ.Ø) RETURN
C CHECK DIRECTION
   5Ø K = POINTR(LN)
      NEIGHB = IABS(NBR(K))
      IF (NEIGHB.EQ.STACK(LN-1)) GO TO 4Ø
C CLIMB OUT ON NEIGHBORING NODE
      LN = LN + 1
      STACK(LN) = NEIGHB
      POINTR(LN) = -1
      GO TO 1Ø
      END
```

```
      INTEGER FUNCTION LASTPT(NODE)
C THE VALUE OF THIS FUNCTION POINTS TO THE END OF THE LIST
C OF NEIGHBORS OF NODE.
      INTEGER SPACE(2), MST(1), NXT(1)
      EQUIVALENCE (MST,NXT)
      COMMON SPACE, MST
C OFFSET PICKS UP POINTER FIELD
      LASTPT = MST(NODE) + 3
   1Ø IF (NXT(LASTPT).EQ.Ø) RETURN
      LASTPT = NXT(LASTPT) + 2
      GO TO 1Ø
      END
```

```
      INTEGER FUNCTION FINDCN(A, B)
      INTEGER A, B
      INTEGER SPACE(2), MST(1), NBR(1), NXT(1)
      EQUIVALENCE (MST,NBR,NXT)
      COMMON SPACE, MST
C THIS FUNCTION LOCATES NODE B IN THE LIST OF NEIGHBORS OF A
C OFFSET PICKS UP NEIGHBOR FIELD
      FINDCN = MST(A) + 1
   1Ø IF (IABS(NBR(FINDCN)).EQ.B) RETURN
      FINDCN = NXT(FINDCN+2)
      IF (FINDCN.NE.Ø) GO TO 1Ø
      WRITE (6,99999) B, A
99999 FORMAT(5HØNODE, I3, 26H IS NOT A NEIGHBOP OF NODE, I3)
      RETURN
      END
```

```
      SUBROUTINE STORE(VALUE, ARRAY, LOC, N)
      INTEGER VALUE, ARRAY(N), LOC, N
C THIS SUBROUTINE IS USED TO STORE VALUES INTO THE ARRAY
C WHICH IS THE FOURTH PARAMETER OF  CLUSTR.
      IF (N.EQ.Ø) RETURN
      LOC = LOC + 1
      IF (LOC.GT.N) GO TO 1Ø
      ARRAY(LOC) = VALUE
      RETURN
   1Ø WRITE (6,99999) VALUE
99999 FORMAT(41H THE ARRAY USED TO STORE A DESCRIPTION OF/3H TH,
     * 3ØHE CLUSTERS IS NOT LONG ENOUGH /15H ITS NEXT VALUE,
     * 11H SHOULD BE , I1Ø)
      RETURN
      END
```

```
      SUBROUTINE PRTREE
C THE DESCRIPTION OF THE MINIMAL SPANNING TREE PRINTED HERE
C LABELS EACH NODE SEQUENTIALLY AS IT OCCURS IN   DATA
      INTEGER DIM, N, MST(1), LOC(1), NBR(1), NXT(1)
      REAL WT(1)
      EQUIVALENCE (MST,LOC,NBR,WT,NXT)
      COMMON DIM, N, MST
      DO 2Ø NODE=1,N
        WRITE (6,99999) NODE
        K = MST(NODE) + 1
   1Ø   WRITE (6,99998) NBR(K), WT(K+1)
        K = NXT(K+2)
        IF (K.NE.Ø) GO TO 1Ø
   2Ø CONTINUE
      RETURN
```

```
99999 FORMAT(5H0NODE, I3/16H   NEIGHBORS ARE)
99998 FORMAT(10X, 4HNODE, I5, 14H  AT DISTANCE , G11.4)
      END


      SUBROUTINE FIXMST
      INTEGER DIM, N, MST(1), NBR(1), NXT(1)
      EQUIVALENCE (MST,NBR,NXT)
      COMMON DIM, N, MST
      DO 20 I=1,N
        K = MST(I) + 1
   10   NBR(K) = IABS(NBR(K))
        K = NXT(K+2)
        IF (K.NE.0) GO TO 10
   20 CONTINUE
      RETURN
      END
```

Remark on Algorithm 479 [Z]

A Minimal Spanning Tree Clustering Method
[R.L. Page, *Comm. ACM 17* (June 1974), 321–323]

H.S. Magnuski [Recd 19 July 1974] Stanford Electronics Laboratories, Stanford University, Stanford CA 94305

The implementation of this algorithm assumes that both real and integer variables occupy the same amount of storage, which is not true of many Fortran systems. The algorithm assumes that real array *WT* and integer array *MST* are exactly the same length, and intermixes floating point and integer variables in creating the linked lists contained in these arrays. The simplest (but not best) solution is to define array *WT* in its own common block. The correct solution requires rewriting of the algorithm so that the linked lists can properly handle floating point numbers.

[Prof. Page informs me that he has a revised version which follows the suggestion of the last sentence above.--L.D.F.]

ACM Transactions on Mathematical Software, Vol. 2, No. 1, March 1976, Pages 110 - 111

REMARK ON ALGORITHM 479

A Minimal Spanning Tree Clustering Method [Z]
[R.L. Page, *Comm. ACM 17*, 6(June 1974), 321–323]

G.M. White, S. Goudreau, and J.L. Legros [Recd 5 Aug. 1975]
Computer Science Department, University of Ottawa, Ottawa, Ont. Canada K1N 6N5

The algorithm as given generally yields a large number of clusters containing only one point. These are not likely to be of much use. Clusters not containing at least *MINPTS* points can be eliminated by making the following changes to the subroutine *CLUTR*.

1. The first statement should read

```
SUBROUTINE CLUTR(D,FACTOR,SPREAD,C,CLEN,PRINT,MINPTS)
```

2. The statement beginning IF(PRINT) following the COMMON statement

```
COMMON DIM, N, MST
```

   should be removed.

3. The following statements should be inserted immediately after the COMMON statement:

```
      IF(MINPTS.LE.N) GO TO 5
      C(1) = 0
      RETURN
    5 IF(PRINT) WRITE(6,99998)D,FACTOR,SPREAD
      IF(PRINT) WRITE(6,99996)MINPTS
99996 FORMAT(1Hb,10X,39HMINIMUMbNUMBERbOFbPOINTSbPERbCLUSTERb
     * IS,I9)
```

4. Statement number 130 should be replaced by the following:

```
  130 IF((NUMIN-BEGCLS+1).LT.MINPTS) GO TO 150
      CALL STORE (NUMIN-BEGCLS+1,C,CP,CLEN)
```

5. The statement following statement 140 should be replaced by

```
      GO TO 160
  150 CLS=CLS-1
  160 IF(NUMIN.LT.N) GO TO 80
```

With these changes, the program will produce the same results as the original program if *MINPTS* is set equal to 1 at the point of invocation.

The algorithm with the above modifications has been tested successfully using G and H (opt = 2) level Fortran compilers on an IBM 360/65 under o.s. level 21.8. With this configuration, the qualifications mentioned by Magnuski [1] are not applicable.

The program has been used to detect artificially generated clusters superimposed upon a background of noise and to detect stars in nuclear emulsions. The algorithm seems particularly well suited for identifying nuclear events in three dimensions using data obtained automatically from emulsions by flying spot scanners.

REFERENCES

[1] Magnuski, H.S. Remark on Algorithm 479. *Comm. ACM 18,* 2(Feb. 1975), 119.

Algorithm 480

Procedures for Computing Smoothing and Interpolating Natural Splines [E1]

Tom Lyche* and Larry L. Schumaker† [Recd. 18 Oct. 1971 and 9 Apr. 1973]
Department of Mathematics, The University of Texas at Austin, Austin, TX 78712

Key Words and Phrases: approximation, interpolation, spline, natural spline, spline smoothing

CR Categories: 5.13

Language: Algol

```
procedure SPLINECOEFF (m,n,X,Y,W,C,q,S,eps,mach,maxit,fail);
  value m,n,maxit; integer m,n,q,maxit; real S,eps,mach;
  array X,Y,W,C; label fail;
```

comment 1. The purpose of this procedure is to generate the coefficients $\{c_i\}_1{}^n$ in the representation

$$s(x) = \sum_{i=1}^{n} c_i B_i(x) \tag{1}$$

of a natural spline of degree $2m - 1$ (in terms of a local basis $\{B_i(x)\}_1{}^n$) for the splines which solve certain data smoothing and interpolation problems. It is based on algorithms described in [2]. To describe the problems, let m and n be integers ($m,n \geq 1$) and suppose $\{x_i\}_1{}^n$, $\{y_i\}_1{}^n$ and $\{w_i\}_1{}^n$ are prescribed real numbers, with $x_1 < x_2 < \cdots < x_n$ and $w_i > 0$, $i = 1,2,\ldots,n$. Suppose $p > 0$ and $S > 0$. For appropriately smooth f we define

$$J(f) = \int_{-\infty}^{\infty} (f^{(m)}(x))^2 dx \tag{2}$$

$$E(f) = \sum_{i=1}^{n} w_i(y_i - f(x_i))^2. \tag{3}$$

The spline interpolation problem is

$$\text{minimize } J(f) \text{ subject to } E(f) = 0. \tag{4}$$

We can solve either of two data smoothing problems:

$$\text{minimize } [J(f) + pE(f)] \tag{5}$$

or

$$\text{minimize } J(f) \text{ subject to } E(f) \leq S. \tag{6}$$

In all cases, the solutions are certain natural splines of degree $2m - 1$ with knots $\{x_i\}_1{}^n$ which can always be represented in the form (1). We assume that $n \geq 2m$, in which case the solutions are unique, and there is a convenient basis $\{B_i(x)\}_1{}^n$.

Determining the $\{c_i\}_1{}^n$ in problem (4) involves setting up and solving a system of n equations with a $2m - 1$ banded matrix. Similarly (5) leads to a system with a $2m + 1$ banded matrix. Solving problem (6) depends on the fact that for small S there is a unique $p = p(S)$ such that the solution of (5) for this p is the solution of (6). The parameter $p(S)$ is the unique positive solution of

$$f^2(p) = E(s_p) = S, \tag{7}$$

where s_p is the solution of (5) corresponding to p. Equation (7) is solved by Newton's method applied to

$$f^{-1}(p) = S^{-\frac{1}{2}}. \tag{8}$$

Then (6) is solved approximately in the sense that a spline s is determined so that

$$|E(s) - S| < eps\sqrt{n}; \tag{9}$$

comment 2. We describe the parameters of *SPLINECOEFF*. The integers m and n must satisfy $m \geq 1, n \geq 4m - 1$. The real arrays $X[1:n]$, $Y[1:n]$, and $W[1:n]$ must satisfy $X[1] < \cdots < X[n]$ and $W[i] > 0$, $i = 1,2,\ldots,n$. The integer q has nonnegative values. In case $q = 0$, the procedure solves (4)—i.e. produces the coefficients of the natural interpolating spline (1) of degree $2m - 1$ with knots at the $X[i]$'s. The coefficients are returned in the array of real numbers $C[1:n]$.

If $q = 1$, problem (5) is solved with smoothing parameter $p := S$, a specified positive real number. Again the coefficients are returned in array C. Finally, if $q = 2$ the iterative process described in comment 1 is carried out to determine a spline s satisfying (9). S and eps must be positive real numbers. The parameter *maxit* should be a positive integer specifying the maximum number of iterations desired in solving (8).

The parameter *mach* is to be the largest machine number such that $1 + mach = 1$: It is machine dependent, of course. The label *fail* is for the purpose of exiting from *SPLINECOEFF* if certain situations arise (e.g. if *maxit* is exceeded). These are explained in detail in comments 7, 11, and 15;

comment 3. *SPLINECOEFF* calls on four other procedures called *BANDET*, and *BANSOL*, *ENDBASIS*, *MIDBASIS*. It is assumed these procedures are defined in the driver program—we describe their bodies later. The driver program should provide two arrays for workspace, namely, $XXR, XX[1:n,1:2m]$;

```
begin
  integer k,k1; k := m+m; k1 := k--1;
  begin
    integer a,i,j,l,i1,i2,m1,m2,r,v,g,l1,l2;
    real F,FF,f1,s2,p,d,h,h1;
    array E,B,BWE[1:n,-m:m],LB[1:n,1:m],NIK,T[0:n],Z,U[1:k];
    integer array INT [1:n];
    l := n; a := k+k; r := if n>a then a else n;
    for j := 1 step 1 until k do
     begin
      l := l-1; r := r-1;
      for i := 1 step 1 until l do
        XX[i,j] := X[i+j]-X[i];
      for i := 1 step 1 until r do
        XXR[i,j] := XX[n-i-j+1,j];
     end j;
```

* Supported in part by Grant DA-ARO(D)-31-124-61050, Army Research Office, Durham, and National Science Foundation Grant GP-23655.

† Supported in part by Grant USAFOSR 69-1812B.

```
for i := 1 step 1 until n do
for j := −m step 1 until m do
   B[i,j] := 0;
comment 4. The array B is to contain the values of $B_j(x_i)$,
   where $B_j(x)$ are the local basis elements of (1). There are essentially three kinds of basis functions, namely (see [2])
```

$$B_i(x) = \begin{cases} Q_{2m,i}(x), & i=1,2,\ldots,m \\ N_{2m,i}(x), & i=m+1,\ldots,n-m \\ \tilde{Q}_{2m,i}(x), & i=n-m+1,\ldots,n. \end{cases}$$

Let $\tilde{B} = (B_j(x_i))$. Because of the support properties of the $B_j(x)$, $\tilde{B}$ is $2m-1$ banded and we may store it as follows:

$$B = \begin{array}{c|cccccc|c} & -m & 1-m & 0 & & m-1 & & m \\ \hline 1 & 0 & 0 & B_1(x_1) & \cdots & B_m(x_1) & & 0 \\ 2 & & B_1(x_2) & B_2(x_2) & & & & \\ n & 0 & & B_n(x_n) & & & & 0 \end{array}$$

Specifically, $B_{i,j} = B_{i+j}(x_i) = \tilde{B}_{i,i+j}$ for $j = max(1-m, 1-i),\ldots,min(m-1,n-i)$, $i=1,2,\ldots,n$;

```
for l := 1 step 1 until k1 do
begin
   for j := 1 step 1 until l−1 do
      T[j] := XX[j,l−j];
   T[l] := 0;
   l2 := if l+k1 > n then n else l+k1;
   for j := l+1 step 1 until l2 do
      T[j] := XX[l,j−l];
   ENDBASIS (k,l,n,T,XX,NIK);
   l1:= if l > m then l else m;
   for j := l1 step 1 until l2 do
      B[l,j−m−l+1] := NIK[j];
end leftpoints;
for l := k step 1 until n−k do
begin
   for j := l−k1 step 1 until l−1 do
      T[j] := XX[j,l−j];
   T[l] := 0;
   for j := l+1 step 1 until l+k1 do
      T[j] := XX[l,j−l];
   MIDBASIS (k,l,n,T,XX,NIK);
   for j := l−k1 step 1 until l−1 do
      B[l,j+m−l] :=NIK[j];
end midpoints;
for l:=1 step 1 until k do
begin
   for j := 1 step 1 until l−1 do
      T[j] := XXR[j,l−j];
   T[l] := 0;
   l2 := if l+k1 > n then n else l+k1;
   for j := l+1 step 1 until l2 do
      T[j] := XXR[l,j−l];
   ENDBASIS (k,l,n,T,XXR,NIK);
   l1 := if l > m then l else m;
   for j := l1 step 1 until l2 do
      B[n−l+1,m+l−j−1]:=NIK[j]
end rightpoints;
comment 5. When q = 0 or if q was changed from 2 to 3 in
   attempting to do smoothing (see comment 9), the coefficients
   $\{c_i\}_1^n$ of the interpolating spline are computed from the
   linear system $\tilde{B}C = Y$;
interpol:
   if q=0 ∨ q=3 then
   begin
      m1 := m−1;
      for i := 1 step 1 until n do
      for j := −m1 step 1 until m1 do
         BWE[i,j] := B[i,j];
      goto linsol
   end;
   comment 6. For q = 1,2, or 4 (see comment 12) the C array
      is computed from the linear system
```

$$(\tilde{B}+p^{-1}\tilde{E})C = Y, \qquad (10)$$

where

$$\tilde{E}_{lj} = w_l^{-1}\beta_{lj},$$

$$\beta_{lj} = f1_j \prod_{i=max(1,j-m)}^{min(n,j+m)} 1/(x_l-x_i),\ j=1,\ldots,n,\ l=max(1,j-m)\ldots,$$

$min(n,j+m)$, and

$$f1_j = (-1)^m(2m-1)!\begin{cases} 1, & j=1,2,\ldots,m, \\ (x_{j+m}-x_{j-m}), & j=m+1,\ldots,n-m, \\ (-1)^{n+m-j}, & j=n-m+1,\ldots,n. \end{cases}$$

The β's are the coefficients of certain divided differences. The array $\tilde{E}$ is $2m+1$ banded and is stored in E in a form similar to B. The quantity d is an estimate for $\|\tilde{E}\|_1$;

```
   f1 := −1; v := k−1; i1 := 1; i2 := m; d := 0;
   for i := 2 step 1 until m do f1 := −f1 × i;
   for i := m+1 step 1 until v do f1 := f1 × i;
   for j := 1 step 1 until n do
   begin
      if j>n−m then begin f1 := −f1; f:=f1 end
      else if j ≤ m then f := f1
      else f := f1 × XX[j−m,k];
      if j>m+1 then i1 := i1+1;
      if i2<n then i2 := i2+1;
      for l := i1 step 1 until i2 do
      begin
         ff := f; v := l−1;
         for i := i1 step 1 until v do
            ff := ff/XX[i,l−i];
         for i := l+1 step 1 until i2 do
            ff := −ff/XX[l,i−l];
         E[l,j−l] := ff/W[l];
         d := d+abs(E[l,j−l])
      end l;
   end E matrix;
   d := d/n;
   m1 := m; r := −1; s2 := sqrt(S); m2 := m−1;
   if q=2 then p := 10 × mach × d
   else if S<10 × d × mach then
   begin
      q := 7; goto fail
   end
   else p := S;
   comment 7. The matrix $\tilde{E}$ is singular. Hence in the case q = 1,
      if $p<10 \times mach \times \|\tilde{E}\|_1$, the matrix (10) will be very close to
      singular since $\|\tilde{B}\|_1 \approx 1$. In this case we exit and set q = 7;
   comment 8. If q = 2 we need to carry out the iteration described in comment 1. Since $f^{-1}(p)$ in (8) is concave (see [3,
      4]), we want to choose the first guess $p^0$ for Newton's method
      such that $f^{-1}(p^0)<S^{-\frac{1}{2}}$. We choose $p^0 = 10 \times mach \times \|\tilde{E}\|_1$
      (see comment 7);
nextit:
   comment 9. When $p > d/10\,mach$, the matrix $p^{-1}\tilde{E}$ is considered insignificant in (10) and the smoothing problem (5) is replaced by the interpolation problem. In this case we set q = 3;
   if p > d/10/mach then
```

```
    begin
      q := 3; goto interpol
    end;
    r := r+1;
    if r > maxit then
    begin q := 6; goto fail; end;
    for i := 1 step 1 until n do
    for j := −m step 1 until m do
      BWE[i,j] := B[i,j]+E[i,j]/p;
linsol:
    BANDET(BWE,LB,INT,n,m1);
    for i := 1 step 1 until n do
      C[i] := Y[i];
    BANSOL(BWE,LB,C,INT,n,m1);
    if q < 2 ∨ q = 3 then goto exit;
    comment 10. We now calculate $F = f^2(p)$ and check condition (9);
    F := 0; l := m2; i1 := 0;
    for i := 1 step 1 until n do
    begin
      if i>n−m2 then l := l−1;
      if i1>−m2 then i1 := i1−1; FF := − Y[i];
      for j := i1 step 1 until l do
        FF := FF+B[i,j] × C[i+j];
      F := F+FF × FF × W[i]; T[i] := FF;
    end;
    if abs(F−S)<eps × sqrt(n × abs(S) then
    begin S := F; goto exit end;
    comment 11. It may happen that the choice of $p^0$ (see comment
      8) leads to $s_p{}^0$ with $f^{-1}(s_p{}^0) > S^{-\frac{1}{2}}$. In this case we set $q = 5$
      and exit.
      This means the initial choice of S is too large;
    comment 12. In some cases the iteration may lead to $s_p$ with
      $f^{-1}(s_p) > S^{-\frac{1}{2}}$. (Because of the concavity of $f^{-1}$ this is theoretically impossible.) We set $q = 4$ and exit. See also comment
      15;
    if F < S then
    begin
      if r=0 then begin q=5; goto fail end
      else begin q=4; S := F; goto exit end
    end;
    comment 13. We now compute $FF=f(p) \times f'(p)$ and carry out
      one step of the Newton process;
    for i := 1 step 1 until n do
      C[i] := W[i] × T[i];
    BANSOL (BWE,LB,T,INT,n,m);
    FF := 0; l := m2; i1 := 0;
    for i := 1 step 1 until n do
    begin
      if i > n−m2 then l := l−1;
      if i1>−m2 then i1 := i1−1; f1 := 0;
      for j := i1 step 1 until l do
        f1 := f1+B[i,j] × T[i+j];
      FF := FF−C[i] × f1;
    end;
    p := p × (1+F × (s2−sqrt(F))/s2/FF);
    goto nextit;
exit:
  end;
  comment 14. Choice of parameters. It is known that the condition
    number of the system $\tilde{B}C = Y$ for spline interpolation increases at least exponentially with m (see de Boor [1]). It is also
    related to the spacing of the $\{x_i\}_1{}^n$. We have computed splines
    to order 20 (m=10) with knot spacing

    $$\pi = \frac{max_i(x_{i+1}-x_i)}{min_i(x_{i+1}-x_i)}$$

    up to 1000, without difficulty. For many problems a choice of
    a small m is desirable—e.g. m = 2, 3 lead to cubic and quintic
    splines, respectively. The size of the parameter n is naturally
    limited by the storage capability of the machine and the time
    available for computation—it seems to have little or no effect on
    conditioning.
      The choice of $\{w_i\}_1{}^n$ and S for smoothing depends on the
    confidence we have in the data $\{y_i\}_1{}^n$. It has been suggested [3]
    that $w_i$ should be chosen as $\delta y_i^{-2}$, where $\delta y_i$ is an estimate of the
    standard deviation of the ordinate $y_i$. A practical upper bound
    for the choice of $w_i$ is $(mach)^{-2}$, where mach is defined in comment 2. If we have more confidence than this in the data, then it
    is probably accurate to machine word length, and we should set
    q = 0 and do interpolation rather than smoothing. When
    q = 1, the choice of p (input through S) for problem (5) is
    problematical. There really is no dependable scheme for choosing it (see the remarks in [4]) unless more is known about the
    problem. For q = 2, it is recommended [3] that S be chosen
    in the interval $n - (2n)^{\frac{1}{2}} \leq S \leq n + (2n)^{\frac{1}{2}}$. The parameters eps and maxit influence each other. For most applications
    it would seem that eps should not be too small—we often used
    $10^{-1}$;
  comment 15. Summary of output after execution. After the execution of SPLINECOEFF, the values of q,S provide information
    on the computation. If q = 0, 1, 2, then computation proceeded normally, and the desired coefficients are stored in array
    C. If q = 3 (see comment 9) interpolation instead of smoothing has been carried out (if the user insists on doing smoothing,
    S must be increased). If q = 4 (see comment 12) the program
    delivered the solution of problem (6) with the S returned in the
    output. (If the user insists on a solution of (6) with the prescribed S, then the problem can be rerun with a write statement
    providing the values of p and f in each iteration. Then an appropriate p can be estimated by interpolation and the program
    reentered with q = 1.) If q = 5 (see comment 11), the user
    must either reduce S or consider doing a least squares fit. If
    q = 6, maxit has been exceeded. If q = 7 (see comment 7),
    then the initial value of p prescribed for problem (5), i.e. q = 1
    initially, is too small. The value of p can be increased or a least
    squares fit should be used;
end SPLINECOEFF;
real procedure SPLINEDER (v,X,l,C,m,n,arg);
  value v, l, m, n, C, arg;
  integer v, l, m, n; real arg; array X, C;
comment 16. Given a spline s of the form (1) with coefficients $\{c_i\}_1{}^n$
  SPLINEDER produces the value $s^{(v)}(arg)$ of the vth derivative of
  s for the argument arg.
    $s^{(v)}(arg)$ is computed by evaluating certain local basis splines
  corresponding to degree 2m − v. The procedures MIDBASIS
  and ENDBASIS are used here. Then $s^{(v)}(arg)$ is a linear combination of these quantities with coefficients $\{c_i^{(v)}\}_1^{n-v}$ (see [2, Lemmas
  5.1 and 5.2]). The $c_i^{(v)}$ are computed from the $c_i$'s by certain recursions, carried out by procedure CV below;
comment 17. We note that $s^{(2m-1)}$ is piecewise continuous with possible discontinuities at the knots $\{x_i\}_1{}^n$. The procedure always returns $s^{(2m-1)}(x_i+)$ if called with $arg = x_i$ a knot;
comment 18. We describe the parameters of SPLINEDER. The integers m and n and the array X[1:n] are as in procedure SPLINECOEFF. The array C[1:n] is the output of SPLINECOEFF. The
  integer v must satisfy $0 \leq v \leq 2m - 1$. The real number arg and
  the integer l satisfy $1 \leq l \leq n - 1$ and $X[l] \leq arg < X[l+1]$;
begin
  integer k; k := m + m − v;
  begin
    array T, NIK[0:n], Z, QIK, PIK[0:k]; real s;
    integer i, j, i1, i2, pvl, qvl, rvl, mv, lu, l1, l2;
    procedure CV(C, X, r, s, n, m, v); value r, s, n, m, v;
    integer r, s, n, m, v; array C, X;
    comment 19. CV computes $\{c_i{}^{(v)}\}_{i=-r}^{s}$. It should be noted that
```

```
      CV is a recursive procedure;
      begin
        integer j, r1, s1;
        if v = 0 then goto exit else if v ≤ m then
        begin
          CV(C, X, r, s+1, n, m, v−1);
          for j := r step 1 until s do
            C[j] := if j ≤ m − v then −C[j]
              else if j ≤ n − m then
                (C[j+1]−C[j])/(X[m+j]−X[j−m+v])
              else C[j+1]
        end
        else
        begin
          r1 := if r > 1 then r − 1 else 1;
          s1 := if s < n + v − 2×m then s else s − 1;
          CV(C, X, r1, s1, n, m, v−1);
          if s = n + v − 2×m then C[s] := 0;
          for j := s step −1 until r do
            C[j] := (C[j]−C[j−1])/(X[j+2×m−v]−X[j])
        end;
exit:
      end CV;
      comment 20. The numbers pvl and qvl give the range of c^(v)'s
        corresponding to nonzero basis elements in the expansion of
        s^(v)(arg);
      if v < m then
      begin
        pvl := if l < m then 1 else l − m + 1;
        qvl := if n < l + m then n − v else l + m − v;
      end
      else
      begin
        pvl := if l < k then 1 else l − k + 1;
        qvl := if l < n − k then l else n − k;
      end;
      C[0] := 0;
      CV(C, X, pvl, qvl, n, m, v);
      s := 0;
      if v < m then goto vlm;
      for j := pvl step 1 until qvl + k do
        T[j] := abs(arg−X[j]);
      MIDBASIS (k, l, n, T, XX, NIK);
      for j := pvl step 1 until qvl do
        s := s + C[j]×NIK[j];
      goto exit;
vlm:
      if l < k then
      begin
        for j := 1 step 1 until l + k do
          T[j] := abs(X[j]−arg);
        ENDBASIS (k, l, n, T, XX, NIK);
        for j := pvl step 1 until qvl do
          s := s + C[j]×NIK[j+m−1];
      end else
      if l > n − k then
      begin
        for j := 1 step 1 until n − l + k + 1 do
          T[j] := abs(arg−X[n−j+1]);
        l1 := if arg > X[l] then n − l else n − l + 1;
        ENDBASIS (k, l1, n, T, XXR, NIK);
        for j := pvl step 1 until qvl do
          s := s + C[j]×NIK[n+m−v−j];
      end
      else
      begin
        for j := l − k + 1 step 1 until l + k do
          T[j] := abs(X[j]−arg);
```

```
      MIDBASIS (k, l, n, T, XX, NIK);
      for j := pvl step 1 until qvl do
        s := s + C[j]×NIK[i−k+m];
    end;
exit:
    for i := 1 step 1 until v do
      s := s×(m+m−i);
    splineder := s
  end inner block
end splineder;
procedure MIDBASIS (k, l, n, T,XX, NIK);
  value k, l, n; integer k, l, n; array T, XX, NIK;
comment 21. This procedure implements case I of [2]. It computes
  the value of certain normalized B-splines N^k_{i,k}(arg) at an arg which
  enters indirectly through the array T via T[j] = |x[i] − arg|.
  After execution NIK[j] contains N^k_{j,k}(arg), j = max(1,l+1−k),
  ..., l;
begin
  integer i, j, i1, i2;
  NIK[l] := 1; NIK[l+1] := 0;
  i1 := i2 := l;
  for i := 2 step 1 until k do
  begin
    if i ≤ l then
    begin
      i1 := i1 − 1; NIK[l1] := 0;
    end;
    if n − i < l then i2 := i2 − 1;
    for j := i1 step 1 until i2 do
      NIK[j] := T[j]×NIK[j]/XX[j, i−1] + T[i+j]×
      NIK[j+1]/XX[j+1, i−1];
  end;
end midbasis;
procedure ENDBASIS (k, l, n, T, XX, NIK);
  value k, l, n; integer k, l, n; array T, XX, NIK;
comment 22. This procedure implements case II of [2] to compute
  the quantities (7.4) of [2] at an argument arg which enters through
  the array T as in comment 21;
begin
  integer i, j, k1, l1, l2; real temp1, temp2;
  array Q[0:k, −1: k+l];
  k1 := k−1;
  for i := 0 step 1 until k do
  for j := l − 2 step 1 until l + i do
    Q[i, j] := 0;
  Q[1, l] := 1/XX[l, 1]; Q[0, −1] := T[2]/XX[1, 1];
  for i := 2 step 1 until k do
  begin
    for j := l step 1 until i − 2 do
    begin
      temp1 := T[j+1]
      Q[i, j] := Q[i−2, j−2] + (temp1+T[j])×Q[i−2, j−1] +
      temp1×temp1×Q[i−2, j];
    end;
    if i > l then
    begin
      temp1 := T[i]; temp2 := temp1×temp1/XX[1, i−1];
      Q[i, i−1] := Q[i−2, i−3] + (temp1+T[i−1]−temp2)×
      Q[i−2, i−2] + temp2×Q[i−2, i−1];
    end;
    l1 := if i > l then i else l;
    l2 := if l + i − 1 > n − 1 then n − 1 else l + i − 1;
    for j := l1 step 1 until l2 do
      Q[i, j] := (T[j−i+1]×Q[i−1, j−1]+T[j+1]×Q[i−1, j])/
      XX[j−i+1, i];
  end i;
  if l > 1 then NIK[l−1] := 0;
  for j := l step 1 until k1 do
```

```
        NIK[j] := Q[k,j];
      l2 := if k + l - 1 > n - 1 then n - 1 else k + l - 1;
      for j:=k step 1 until l2 do
        NIK[j] := Q[k,j] × XX[j-k+1, k];
end ENDBASIS;
procedure BANDET (A, B, INT, n, m);
  value n, m; integer n, m; array A, B; integer array INT;
comment 23. BANDET decomposes the 2m + 1 banded n × n
  matrix A in an upper triangular matrix A and a lower triangular
  matrix B using Gaussian elimination with complete pivoting. De-
  tails of the interchanges are stored in the array INT. The arrays are
  dimensioned as follows  A[1:n, -m:m],  B[1:n, 1:m],  INT[1:n].
  For further details see [5];
begin
  integer i, j, k, l; real x;
  l := m;
  for i := 1 step 1 until m do
  begin
    for j := 1 - i step 1 until m do
      A[i, j-l] := A[i, j];
    l := l - 1;
    for j := m - l step 1 until m do
      A[i, j] := 0
  end i;
  l := m;
  for k := 1 step 1 until n do
  begin
    x := A[k, -m]; i := k;
    if l < n then l := l + 1;
    for j := k + 1 step 1 until l do
      if abs(A[j, -m]) > abs(x) then
      begin x := A[j, -m]; i := j end;
    INT[k] := i;
    if i ≠ k then
    for j := -m step 1 until m do
    begin
      x := A[k, j]; A[k, j] := A[i, j]; A[i, j] := x
    end j;
    for i := k + 1 step 1 until l do
    begin
      x := A[i, -m]/A[k, -m]; B[k, i-k] := x;
      for j := 1 - m step 1 until m do
        A[i, j-1] := A[i, j] - x × A[k, j];
      A[i, m] := 0
    end i
  end k
end BANDET;
procedure BANSOL (A, B, C, INT, n,m);
  value n, m; integer n, m; array A, B, C; integer array INT;
comment 24. The parameters A, B, INT, n, and m come from
  BANDET. BANSOL solves the system decomposed by BANDET
  with right-hand side C. The solution is returned in {C[i]}1n (see
  [5]);
begin
  integer i, j, k, l; real x;
  l := m;
  for k := 1 step 1 until n do
  begin
    i := INT [k];
    if i ≠ k then
    begin x := C[k]; C[k] := C[i]; C[i] := x end;
    if l < n then l := l + 1;
    for i := k + 1 step 1 until l do
      C[i] := C[i] - B[k, i - k] × C[k]
  end k;
  l := -m;
  for i := n step -1 until 1 do
  begin
    x := C[i]; j := i + m;
    for k := 1 - m step 1 until l do
      x := x - A[i, k] × C[k + j];
    C[i] := x/A[i, -m];
    if l < m then l := l + 1
  end i
end BANSOL;
```

Acknowledgment. We wish to thank Harold Eidson for useful suggestions and for checking the algorithm. The referees were also very helpful.

References

1. de Boor, C. On calculating with B-splines. *J. Approx. Th. 6* (1972), 50–62.
2. Lyche, Tom, and Schumaker, Larry L. Computation of smoothing and interpolating natural splines via local bases. *SIAM J. Numer. Anal. 10* (1973), 1027–1038.
3. Reinsch, C.H. Smoot 'ng by spline functions. *Numer. Math. 10* (1967), 177–183.
4. Reinsch, C.H. Smoothing by spline functions, II. *Numer. Math. 16* (1971), 451–454.
5. Martin, R.S., and Wilkinson, J.H. Solution of symmetric and unsymmetric band equations and the calculation of eigenvectors of band matrices. *Numer. Math. 9* (1967), 279–301.
6. Woodford, C.H. An algorithm for data smoothing using spline functions. *BIT 10* (1971), 501–510.

Algorithm 481

Arrow to Precedence Network Transformation [H]

Keith C. Crandall [Recd. 15 Jan. 1973]
Department of Civil Engineering, University of California, Berkeley, CA 94704

Key Words and Phrases: critical path, networks, precedence networks
CR Categories: 3.23, 5.40
Language: Fortran

Description

Purpose. Many of the recent application programs in the area of critical path scheduling and resource allocation are written for the precedence networking convention [1, 2, 3]. Since only a few of these programs accept networks defined by the arrow convention directly, a method of transforming arrow convention networks into precedence convention is required. This algorithm generates the required transformation by producing a list of followers for each non-dummy arrow activity. New labels are produced for each transformed activity and replace the $(i - j)$ labels associated with arrow networks. (The new label is actually the activity input sequence value, but this can easily be modified to any desired notation by using the input sequence value as a subscript to any array containing the desired notation.)

The logic used in the transformation can also be utilized to produce a list of precedecessors if they are desirable. (This order is required by IBM [3] but is performed internally.) The role of arrays (*II* and *JJ*) would be reversed and the array (*ILOC*) would refer to (*JJ*) vice (*II*).

Method. The values of the arrow $(i - j)$ labels are utilized to trace the followers of a particular activity. Activities which have an (i) label corresponding to the (j) label of the activity under evaluation are logical followers. The major problems rest with the arrow *DUMMY* activities. These activities are not really followers but indicate instead addition nodes that precede logical followers. The transformation routine recursively traces all possible following nodes and determines the input sequence number of all logic followers.

To perform this search with the minimum storage required the following procedure is utilized. First the arrays (*II, JJ, NLOC*) are filled by scanning the description of the arrow network and storing in input order the converted value of the (i) label into array (*II*); the converted value of the (j) label into array (*JJ*); and finally the array (*NLOC*) contains the input sequence value. To aid in determining which activities were dummies, the last two arrays (*JJ, NLOC*) have their values set negative when the corresponding activity was a dummy. Since the minimization of storage was a goal, all incoming $(i - j)$ labels were converted into a numerical sequence starting with one. The algorithm indicates the required modification if this is undesirable. (The actual conversion method is described in the routine *HASH*.) Once the arrays are filled, the transformation routine can be called.

Routine (*TRNFRM*) first sorts the array (*II*) into ascending order, maintaining the same correspondence of each element in array (*NLOC*). A sequential scan is then performed on the sorted array (*II*), and the array is overlayed by an array, (*ILOC*), containing pointers to the beginning of each different (i) value in the sorted array. That is element (1) of the new array points to the start of the value (1) in the sorted array; element (10) to the start of (10), and so forth. Finally the array (*JJ*) is scanned sequentially and the nonnegative values become subscripts to the pointer array (*ILOC*). This yields the beginning location and number of activities that had an (i) label equal to the current (j) value. The values stored in (*NLOC*) are the input sequence numbers of the followers. If the follower was a *DUMMY*, (*NLOC*) negative, a recursive search is performed for additional followers.

Finally for each nonnegative entry in (*JJ*), the description is retrieved from the scratch tape and the activity and its followers are output.

Test Results. Testing was performed by two additional programs which are also included in the algorithm listing in case they are desired. Routine (*TEST*) reads the arrow network filling the arrays (*II, JJ, NLOC*) as described. Routine (*HASH*) performs the required conversion to the $(i - j)$ labels during this process.

Tests include networks with sequential dummies and other unusual conditions. In each case tried, the transformation was correct. The inefficiency of the bubble up sort could adversely affect very large networks and an alternative would be to pre-sort the arrow network and eliminate the sorting portion. The following table indicates execution time versus number of activities for tests run on a CDC 6400.

Execution Times for Various Networks Tested

Number of activities	Execution time in sec.
16	0.42
44	1.68
177	2.08
461	5.81
677	10.76

The routine does not test for logical errors in the arrow network such as loops, so these would be transformed without change into the precedence notation.

References

1. Fondahl, John W. A non-computer approach to the critical path method. Tech. Rep. No. 9, Dep. of Civil Engineering, Stanford U., Stanford, Calif., 1962.
2. Baker, Wilson C. Spread and level CPM. Tech. Rep. No. 56, Dep. of Civil Engineering, Stanford U., Stanford, Calif., 1967.
3. IBM, Project Management System. Application description manual (H20-0210), 1968.

Algorithm

(Note: A sample driver is included to help clarify the use of this algorithm—L.D.F.)

```
C THIS IS THE TEST PROGRAM FOR THE TRANSFORMATION ALGORITHM.
C IT READS THE ARROW NETWORK DESCRIPTION AND ESTABLISHES
C THE INPUT ARRAYS FOR THE ROUTINE (TRNFRM).
C IT IS LIMITED TO 700 ACTIVITIES IN ARROW NOTATION.
C THE ROUTINE (HASH) IS UTILIZED TO CREATE A SEQUENTIAL
C NUMBERING.
C THE ROUTINE (TRNFRM) CREATES THE ACTUAL TRANSFORMATION.
C TAPE(2) -A BINARY SCRATCH TAPE (FILE) WITH ALL DATA TO
C BE INCLUDED WITH THE TRANSFORMED ACTIVITIES.NOTE- CHANGE
C STMT 140 TO CORRESPOND WITH ACTUAL DATA STORED.
C TAPE(4) -A BINARY SCRATCH TAPE FOR TRANSFERING THE TRANS-
C FORMED DATA BACK TO THE MAIN PROGRAM FOR PRINT OUT, OR ANY
C OTHER USE. THE DATA IS IN THE FORM (I,M,FOL) WHERE I IS
```

```
C THE NEW ACTIVITY LABEL AND M IS THE NUMBER OF FOLLOWERS
C AND FOL IS AN ARRAY CONTAINING THE LABELS OF THE M
C FOLLOWERS...
      INTEGER II(7ØØ), JJ(7ØØ), NLOC(7ØØ), ACT(2), DUMMY,
     * HASH, FOL(5Ø)
      DATA DUMMY/5HDUMMY/, IBLNK/1H /
C READ IN ARROW ACTIVITIES ACCORDING TO CURRENT FORMAT.
99999 FØRMAT(1H1, 13H  INPUT ORDER, 6X, 5HLABEL, 5X, 4HDESC,
     * 7HRIPTION, 12X, 3HDUR)
99998 FØRMAT(2A4, 2A1Ø, I3, 3X, I6)
99997 FØRMAT(I14, 4X, A4, 1H-, A4, 3X, 2A1Ø, I6)
99996 FØRMAT(1H1, 19HTRANSFORMED NETWORK//14H  LABEL  DESCR,
     * 6HIPTION, 1ØX, 3HDUR, 3X, 9HFOLLOWERS)
99995 FØRMAT(1H , I7, 2X, 2A1Ø, I4)
99994 FØRMAT(1H+, 36X, 15I5/(37X, 15I5))
      WRITE (6,99999)
      NACT = Ø
      NTAPE2 = Ø
   1Ø READ (5,99998) I, J, ACT, IDUR
C FORMAT (99998) WILL VARY FOR INDIVIDUAL NEEDS.
C THE TEST FOR END OF DATA IS A BLANK CARD.
      IF (I.EQ.IBLNK) GO TO 3Ø
      NACT = NACT + 1
C LIST THE ARROW DATA FOR REFERENCE.
      WRITE (6,99997) NACT, I, J, ACT, IDUR
C CONVERT THE ALPHANUMERIC I-J LABELS INTO SEQUENTIAL
C NUMERIC. (ROUTINE HASH PERFORMS THIS TASK.)
C STORE THE CONVERTED LABELS IN THE ARRAYS (II AND JJ).
C NOTE. THE VALUE STORED IN ARRAY (JJ) IS ALSO SAVED AS
C VARIABLE J TO ALLOW IT TO BE USED AT STMT 2Ø WITHOUT AN
C ARRAY REFERENCE.
      II(NACT) = HASH(I)
      J = HASH(J)
      JJ(NACT) = J
C STORE THE INCOMING INPUT SEQUENCE VALUE IN ARRAY (NLOC)
      NLOC(NACT) = NACT
C EXAMPLE OF USER CREATED LABELING, SEE ALSO COMMENTS AFTER
C STMT 14Ø IN ROUTINE TRNFRM.
C LABLS(NACT)=CONCATENATION OF INPUT (I-J)
C THE CONCATENATION IS PERFORMED IN ACCORDANCE WITH VALID
C FORTRAN FOR THE COMPILER IN USE.
C TEST FOR A DUMMY ACTIVITY AS IT WILL NOT BE TRANSFORMED.
      IF (ACT(1).EQ.DUMMY) GO TO 2Ø
C SAVE ON TAPE (2) ALL INFORMATION RELATING TO THE ACTIVITY
C JUST READ THAT IS TO BE ASSOCIATED WITH THE TRANSFORMED
C ACTIVITY.(FOR THE EXAMPLES ONLY THE DESCRIPTION AND DUR
C ARE SAVED,ACTUAL USERS WILL HAVE INDIVIDUAL REQUIREMENTS)
      NTAPE2 = NTAPE2 + 1
      WRITE (2) ACT, IDUR
      GO TO 1Ø
C IF AN ACTIVITY WAS A DUMMY, SO NOTE BY SETTING THE
C LOCATION AND JJ LABEL VECTORS NEGATIVE.
   2Ø NLOC(NACT) = -NACT
      JJ(NACT) = -J
C RETURN FOR NEXT INPUT ACTIVITY. TRANSFER WILL BE MADE TO
C STMT 3Ø WHEN LAST INPUT IS RECOGNIZED.
      GO TO 1Ø
   3Ø REWIND 2
C CALL THE TRANSFORMATION ROUTINE.,DESCRIPTION OF INPUT
C ARRAYS IS FOUND IN THE (TRNFRM) ROUTINE.
      CALL TRNFRM(NACT, II, JJ, NLOC)
C PRINT OUT THE TRANSFORMED NETWORK...
      WRITE (6,99996)
      DO 4Ø N=1,NTAPE2
C RECOVER THE REQUIRED DATA RELATING TO THE TRANSFORMED
C ACTIVITY FROM TAPE(2) AND TAPE (4).
        READ (2) ACT, IDUR
        READ (4) I, M, FOL
        WRITE (6,99995) I, ACT, IDUR
        IF (M.LE.Ø) GO TO 4Ø
        WRITE (6,99994) (FOL(MM),MM=1,M)
   4Ø CONTINUE
      STOP
      END
```

```
      INTEGER FUNCTION HASH(N)
C THIS ROUTINE CONVERTS THE ALPHANUMERIC ARROW LABELS INTO A
C SEQUENTIAL NUMERIC EQUIVALENT. THE MAXIMUM NUMBER OF
C SEPARATE ACTIVITY LABELS IS 5ØØ FOR THIS TEST PACKAGE.
C THE ACTUAL INCOMING LABEL IS STORED IN ARRAY (HOLD) AND
C THE SEQUENTIAL NUMERIC EQUIVALENT IS STORED IN ARRAY
C  (SAVE)
C VARIABLE (NUM) PROVIDES THE SEQUENTIAL NUMBERS.
      INTEGER HOLD(5ØØ), SAVE(5ØØ)
      DATA NUM/Ø/, HOLD/5ØØ*Ø/
C USE A MODIFIED HASHING ROUTINE TO FIND AND STORE THE
C EQUIVALENT VALUES.
C NN IS A HASHED VALUE FOR THE INPUT VARIABLE N.
99999 FORMAT(34H EXCEEDED THE EVENT TABLE CAPACITY)
      NN = MOD(IABS(N/68719476736),375)
   1Ø DO 2Ø I=NN,5ØØ
C THE ARRAY (HOLD) IS EXAMINED STARTING WITH THE HASHED
C VALUE, IF THE ARRAY ELEMENT CONTAINS THE INPUT VARIABLE N,
C TRANSFER IS MADE TO STMT 4Ø AND THE EQUIVALENT SEQUENTIAL
C NUMBER IS RECALLED FROM ARRAY (SAVE). IF THE ARRAY ELEMENT
C CONTAINS A ZERO,TRANSFER IS MADE TO STMT 3Ø AND A
C  NUMERICAL
C EQUIVALENT IS ASSIGNED. THE SEARCH OF (HOLD) CONTINUES
C UNTIL AN OPEN ELEMENT IS FOUND...
        IF (HOLD(I).EQ.N) GO TO 4Ø
        IF (HOLD(I).EQ.Ø) GO TO 3Ø
   2Ø CONTINUE
C IF NO OPEN ELEMENT WAS FOUND AND NN=1 THERE ARE NO OPEN
C ELEMENTS IN THE ENTIRE ARRAY. IF NN IS NOT EQUAL TO 1, SET
C IT TO 1 AND SEARCH LOWER PART OF (HOLD)...
      IF (NN.EQ.1) GO TO 6Ø
      NN = 1
      GO TO 1Ø
C FOUND A NEW LABEL-GIVE IT AN EQUIVALENT SEQUENTIAL NUMBER
   3Ø HOLD(I) = N
      NUM = NUM + 1
      IVAL = NUM
      SAVE(I) = IVAL
C TRANSFER TO STMT 5Ø AND SAVE A REDUNDANT RECALL FROM
C  (SAVE)
      GO TO 5Ø
   4Ø IVAL = SAVE(I)
   5Ø HASH = IVAL
      RETURN
C AN ERROR MESSAGE IS GENERATED IF THE NUMBER OF EVENTS
C EXCEEDS THE DIMENSION ALLOWED.
   6Ø WRITE (6,99999)
      STOP
      END
```

```
      SUBROUTINE TRNFRM(NACT, II, JJ, NLOC)
C ALL DATA WAS STORED IN THE ARRAYS (II-JJ-NLOC) BY THE
C CALLING ROUTINE AND COMFORMS TO THE FOLLOWING DESCPIPTION
C (NACT) -THE  NUMBER OF ARROW ACTIVITIES INCLUDING DUMMIES.
C (II) -AN ARRAY OF CONVERTED -I- LABELS STORED IN THE ARROW
C NETWORK INPUT ORDER.REFER TO THE COMMENTS AFTER STMT 14Ø
C IF USER GERERATED LABELS ARE DESIRED.SEE ALSO COMMENTS IN
C MAIN ROUTINE.
C (JJ) -AN ARRAY LIKE (II) FOR -J- LABELS EXCEPT THAT THE
C VALUE IS NEGATIVE FOR ALL DUMMY ACTIVITIES.
C (NLOC) -AN ARRAY INDICATING INPUT ORDER.(A SEQUENTIAL LIST
C SUCH THAT THE ABSOLUTE VALUES WOULD RANGE FROM ONE TO NACT
C )  NOTE- THE VALUE STORED IN (NLOC) IS NEGATIVE WHEN THE
C CORRESPONDING ARROW ACTIVITY WAS A -DUMMY- .
C TAPE(4) -A BINARY SCRATCH TAPE FOR TRANSFERING THE TRANS-
C FORMED DATA BACK TO THE MAIN PROGRAM FOR PRINT OUT, OR ANY
C OTHER USE. THE DATA IS IN THE FORM (I,M,FOL) WHERE I IS
C THE NEW ACTIVITY LABEL AND M IS THE NUMBER OF FOLLOWERS
C AND FOL IS AN ARRAY CONTAINING THE LABELS OF THE M
C FOLLOWERS...
C STORAGE FOR THE ARRAYS IS ALSO SPECIFIED IN THE CALLING
C PROGRAM.
      INTEGER II(1), JJ(1), NLOC(1)
      INTEGER STACK(5Ø), FOL(5Ø)
C THE DIMENSION STAMENTS FOR (II-JJ-NLOC) MUST BE MODIFIED
C FOR USE WITH SOME FORTRAN COMPILERS.
C DIMENSIONS ON STACK AND FOL LIMIT THE NUMBER OF FOLLOWING
C ACTIVITIES TO 5Ø.
C STATEMENT FUNCTION TO PROVIDE OVERLAYING ARRAY (II) WITH
C ARRAY (ILOC).REFER TO THE WARNING AFTER STMT 3Ø,IF A
C SEPERATE ARRAY (ILOC) IS UTILIZED THE STATEMENT FUNCTION
C WOULD BE DELETED.
99999 FORMAT(41H THE FOLLOWING ACTIVITY APPEARS TO HAVE M,
     * 22HORE THAN 5Ø FOLLOWERS )
99998 FORMAT(41H SUSPECT THE FOLLOWING ACTIVITY IS INVOLV,
     * 41HED IN A NETWORK LOOP - CHECK INPUT DATA. /I5)
      ILOC(I) = II(I)
C REWIND TAPE 4 FOR TRANSFER OF TRANSFORMED DATA.
      REWIND 4
C PLACE THE ARRAYS (II-NLOC) IN ASENDING ORDER USING (II)
C AS THE SORT VARIABLE. (THIS IS A BUBBLE UP SORT.)
      LIMIT = NACT - 1
      DO 2Ø M=1,LIMIT
        LL = M + 1
        DO 1Ø N=LL,NACT
          IF (II(M).LE.II(N)) GO TO 1Ø
          IHOLD = II(N)
          II(N) = II(M)
          II(M) = IHOLD
          IHOLD = NLOC(N)
          NLOC(N) = NLOC(M)
          NLOC(M) = IHOLD
   1Ø   CONTINUE
   2Ø CONTINUE
C REPLACE THE ARRAY (II) WITH AN INTEGER POINTER SUCH THAT
C THE (K TH) ELEMENT OF THE POINTER POINTS TO THE FIRST
C LOCATION IN THE SORTED ARRAY (II) WHICH CONTAINS THE VALUE
C (K).THE POINTER ARRAY WILL BE CALLED (ILOC) SINCE IT
C INDICATES THE BEGINNING OF SORTED ARROW NODES (ARRAY II)
C AND THESE NODES ARE NORMALLY REFERRED TO AS (I) NODES.
C THE VARIABLE (N) IS SET TO THE MINIMUM VALUE IN ARRAY (II)
C N IS ALSO A VARIABLE THAT INDICATES THE CURRENT VALUE
C UNDER INVESTIGATION IN ARRAY (II).
C L IS A POINTER TO THE ARRAY (ILOC),INDICATING THE LOCATION
C OF THE NEXT ELEMENT.IN ADDITION L ALSO INDICATES THE NEXT
C SEQUENTIAL NUMBER,AND IS USED TO FIND THE END NODES.(NODES
C WHERE THERE EXISTS NO -I- IN THE (I-J) PAIRS,AND THERE-
C FORE NO ENTRY IN THE SORTED (II) ARRAY..)
      N = 1
      L = 2
      DO 5Ø I=2,NACT
```

```
        IF (II(I).EQ.N) GO TO 50
        N = II(I)
   30   IF (N.EQ.L) GO TO 40
C THIS TEST FINDS THE REFERENCES TO THE END NODE WHICH WILL
C NOT BE IN THE SORTED ARRAY OF (I) NODES.
C WARNING -- ALTHOUGH INPUT ORDER IS NOT NORMALLY IMPORTANT
C REFERENCE TO END NODES,THAT IS (I-J) PAIRS WITH -J- EQUAL
C TO AN END NODE,SHOULD BE POSITIONED IN THE LATER PORTION
C OF THE INPUT DATA.THIS RESTRICTION CAN BE ELIMINATED BY
C USING A SEPARATE ARRAY FOR (ILOC).
C II(L) IS SET TO ZERO TO INDICATE THAT NODE -L- IS AN END
C NODE IN THE ARROW INPUT NETWORK.
        II(L) = 0
        L = L + 1
        GO TO 30
C STORE THE SUBSCRIPT VALUE OF THE ARRAY (II) IN TO THE
C OVERLAYED ARRAY (ILOC).
   40   II(L) = I
        L = L + 1
   50 CONTINUE
C SET THE NEXT LOCATION OF THE POINTER TO ONE PAST THE LAST
C ACTIVITY NUMBER.
      MAXLST = L - 1
      II(L) = NACT + 1
C FOR ALL NON DUMMY ACTIVITIES,TRANSFORM THE ARROW LOGIC
C CONSTRAINTS INTO THE PRECEDENCE NOTATION BY GIVING THE
C ACTIVITY A LABEL EQUAL TO ITS INPUT ORDER,THEN LIST ALL
C TRANSFORMED FOLLOWERS.
      DO 160 I=1,NACT
        L = 0
        M = 0
C L INDICATES THE LENGTH OF THE STACK AND M IS THE NUMBER OF
C FOLLOWERS.THE STACK IS USED TO RECURSIVELY TRACE ALL
C DUMMIES TO FIND LOGICAL FOLLOWERS.
        N = JJ(I)
C IF N IS NEGATIVE THE ARROW ACTIVITY WAS A DUMMY.
        IF (N.LE.0) GO TO 160
   60   LOC = N
        IF (LOC.GT.MAXLST) GO TO 110
C LOC HAS A VALUE EQUAL TO THE -J- LABEL OF ACTIVITY UNDER
C TRANSFORMATION. ILOCR POINTS TO THE BEGINNING OF THAT SAME
C VALUE IN THE SORTED ARRAY (II).WHEN (LOC) EXCEEDS THE
C VALUE OF (MAXLST) THE -J- LABEL ON THE ARROW NETWORK WAS
C THE END NODE,THEREFORE THERE ARE NO FOLLOWERS.
        ILOCR = ILOC(LOC)
        IF (ILOCR.LE.0) GO TO 110
C IF ILOCR IS NEG OR ZERO THE ACTIVITY HAS NO FOLLOWERS.
   70   LOC = LOC + 1
        NN = ILOC(LOC) - ILOCR
C NN INDICATES THE NUMBER OF ELEMENTS IN ARRAY (II) WITH THE
C VALUE.
        IF (NN.LE.0) GO TO 70
        DO 100 LOOP=1,NN
          LOCS = NLOC(ILOCR)
          IF (LOCS.EQ.0) GO TO 90
          IF (LOCS.GT.0) GO TO 80
C LOCS NEGATIVE INDICATES A DUMMY AND THESE ARE HELD IN THE
C STACK FOR LATER CONTINUED SEARCH OF FOLLOWERS.
          L = L + 1
          IF (L.GT.50) GO TO 130
          STACK(L) = -LOCS
          GO TO 90
    80    M = M + 1
C A FOLLOWER HAS BEEN FOUND.STORE IT IN THE ARRAY (FOL).
          IF (M.GT.50) GO TO 120
          FOL(M) = LOCS
C INCREASE THE POINTER TO NEXT POTENTIAL FOLLOWER.
   90     ILOCR = ILOCR + 1
  100   CONTINUE
  110   IF (L.LE.0) GO TO 140
C IF (L) IS NON-ZERO,THERE ARE DUMMY LINKAGES TO BE CONSIDER
C ED. (N) WILL INDICATE FIRST OF THESE AND THE SEARCH FOR
C FOLLOWERS WILL CONTINUE.
        K = STACK(L)
        N = IABS(JJ(K))
        L = L - 1
        GO TO 60
C ERROR MESSAGES IF DIMENSIONS EXCEEDED- LOOP ASSUMED.
  120   WRITE (6,99999)
  130   WRITE (6,99998) I
  140   WRITE (4) I, M, FOL
C IF USER LABELS ARE USED THEY WOULD BE RETRIEVED THUSLY --
C       I = LABLS(I)
C       DO 150 LOOP=1,M
C         ISUB = FOL(LOOP)
C         FOL(LOOP) = LABLS(ISUB)
C 150   CONTINUE
C WHERE LABLS WOULD BE AN ARRAY PASSED IN THE ARGUMENT LIST
  160 CONTINUE
      REWIND 4
      RETURN
      END
```

Algorithm 482

Transitivity Sets [G7]

John McKay and E. Regener* [Recd. 21 May 1973]
School of Computer Science, McGill University, Montreal, Quebec, Canada

Key Words and Phrases: transitivity, sets
CR Categories: 5.39
Language: Algol

Let $P = \{P_1, P_2, \ldots, P_k\}$ be a set of k permutations on the set $\Omega = \{1, 2, \ldots, n\}$. The transitivity set containing i (or orbit of i) under P is the set of images of i under the action of products of elements of P. This procedure computes these orbits.

On entry, *im*[*i,j*] is assumed to contain the image of i under P_j, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, k$. The procedure numbers the orbits consecutively starting at 1. On exit *ind*[*i*] contains the number of the orbit to which i belongs. The orbits appear in order in *orb*[1:*n*]. In *orb* the first element of each orbit is tagged negative. If only one permutation is input, the array *orb* contains it (tagged) in disjoint cycle form on exit.

The algorithm, which involves no searching, is related to one for finding a spanning tree of a graph [1]. The set P need not, in general, generate a group—it is sufficient that it generate a semigroup on Ω.

References

1. Cannon, J. Ph.D. Th., Sydney U., Sydney, N.S.W., Australia, 1969.

Algorithm

```
procedure orbits (ind, orb, im, n, k);
  value n, k; integer n, k;
  integer array ind, orb, im;
begin
  integer q, r, s, j, nt, ns, norb;
  for j := 1 step 1 until n do ind[j] := 0;
  norb := 0; ns := 1;
  for r := 1 step 1 until n do if ind[r] = 0 then
  begin
    norb := norb + 1; ind[r] := norb;
    nt := ns; orb[ns] := -r; s := r;
a:
    ns := ns + 1;
    for j := 1 step 1 until k do
    begin
      q := im[s,j];
      if ind[q] = 0 then
      begin
        nt := nt + 1; orb[nt] := q; ind[q] := norb
      end
    end;
    if ns ≤ nt then
    begin s := orb[ns]; go to a end
  end
end
```

* Now at Faculté de Musique, University de Montréal, Montréal, P.Q., Canada.

Editors' note: *Algorithm 483 described here is available on magnetic tape from the Department of Computer Science, University of Colorado, Boulder, CO 80302. The cost for the tape is $16.00 (U.S. and Canada) or $18.00 (elsewhere). If the user sends a small tape (wt. less than 1 lb.) the algorithm will be copied on it and returned to him at a charge of $10.00 (U.S. only). All orders are to be prepaid with checks payable to ACM Algorithms. The algorithm is recorded as one file of BCD 80 character card images at 556 B.P.I., even parity, on seven track tape. We will supply algorithms at a density of 800 B.P.I. if requested. Cards for algorithms are sequenced starting at 10 and incremented by 10. The sequence member is right justified in column 80. Although we will make every attempt to insure that the algorithm conforms to the description printed here, we cannot guarantee it, nor can we guarantee that the algorithm is correct.— L.D.F. and A.K.C.*

Algorithm 483

Masked Three-Dimensional Plot Program with Rotations [J6]

Steven L. Watkins [Recd. 26 March 1973] Applied Research Laboratories, The University of Texas at Austin, Austin, TX 78712

Key Words and Phrases: plotting, three-dimensional plotting
CR Categories: 4.41
Language: Fortran

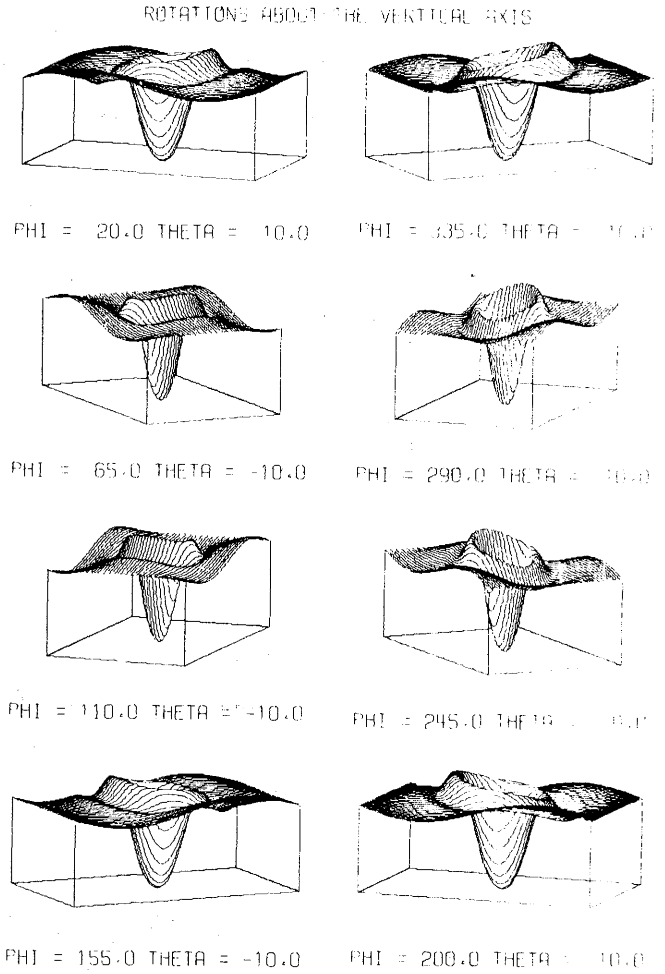
Description

PLOT3D will accept three-dimensional data in various forms, rotate it in three-space, and plot the projection of the resulting figure onto the *x-y* plane. Those lines or portions of lines which should be hidden by previous lines are masked.

Each call to *PLOT3D* causes one line to be plotted. A line consists of a sequence of points in three-space which will be connected using linear interpolation between adjacent points. This sequence of points is specified by three sequences of real numbers, the *x*, *y*, and *z* components of each point. Each of these sequences of real numbers can be specified either as being equally spaced, and therefore denoted by an initial value and an increment, or as being contained in a real array. There is no restriction that any of the three component sequences be either increasing or decreasing, and the number of points may change between successive calls.

The masking technique used by *PLOT3D* is based on two premises: (1) lines in the foreground (positive *z* direction) are plotted before lines in the background; and (2) a line or portion of a line is masked (hidden) if it lies within the region bounded by previously plotted lines. Masking is then achieved by maintaining a visible maximum function and a visible minimum function. Those portions of each line falling within the region bounded by these functions are considered to be hidden. Any line which exceeds user specified limits is truncated without the loss of the plotter origin. A call to *PLOT3D* before initiating a new figure can be used to simulate a line drawn at the bottom of the paper; therefore, only those portions of each line lying above all previous lines will be drawn.

Fig. 1.



The data are transformed by a three-dimensional rotation determined by two user specified angles. *PLOT3D* assumes a right-hand coordinate system with *x* running the length of the paper, *y* running across the width, and *z* coming out of it. The figure is first rotated by an angle of θ degrees clockwise about the *x*-axis. The resultant figure is then rotated by an angle of $\varnothing$ degrees about its *y*-axis. The plotted figure is the projection of this final figure onto the *x-y* plane. Figure 1 demonstrates rotations about the vertical or *y*-axis, and Figure 2 demonstrates rotations about the horizontal or *x*-axis. *Warning:* Some rotations will alter the foreground/background relationships between the lines, and thus the order in which they should be plotted to avoid violating the first masking premise.

As an option, the coordinates of the vertices of the figure and the projection of these vertices onto the $y = 0$ plane of the figure will be returned in a user supplied array. This information can then be used to put a frame on the figure, as is done in the example program, or to connect the endpoints of each line, or to plot axes, etc.

Crosshatched figures are easily obtained as is demonstrated by the example program which generated Figure 3. Some perspective can be achieved by modifying the data scaling parameters between successive calls. *PLOT3D* attempts to minimize plotter movement by beginning at the alternate end of successive lines. A more detailed description of the parameters is contained in the comments at the beginning of the program listing.

This routine was developed at the Applied Research Laboratories on their Control Data Corporation 3200 computer system. The following system routines were utilized:

IROUND(X) returns the rounded integer value of its floating point argument.

IPLOT(IX, IY, J) moves the pen to the point *(IX, IY)* where:

- *IX* is the number of plotter increments along the length of the paper from the origin
- *IY* is the number of plotter increments across the width of the paper from the origin
- *J* is the pen status
  - 2—lower pen before moving
  - 3—raise pen before moving
  - If *J* is negative, the origin will be reset at *(IX, IY)*.

Fig. 2.

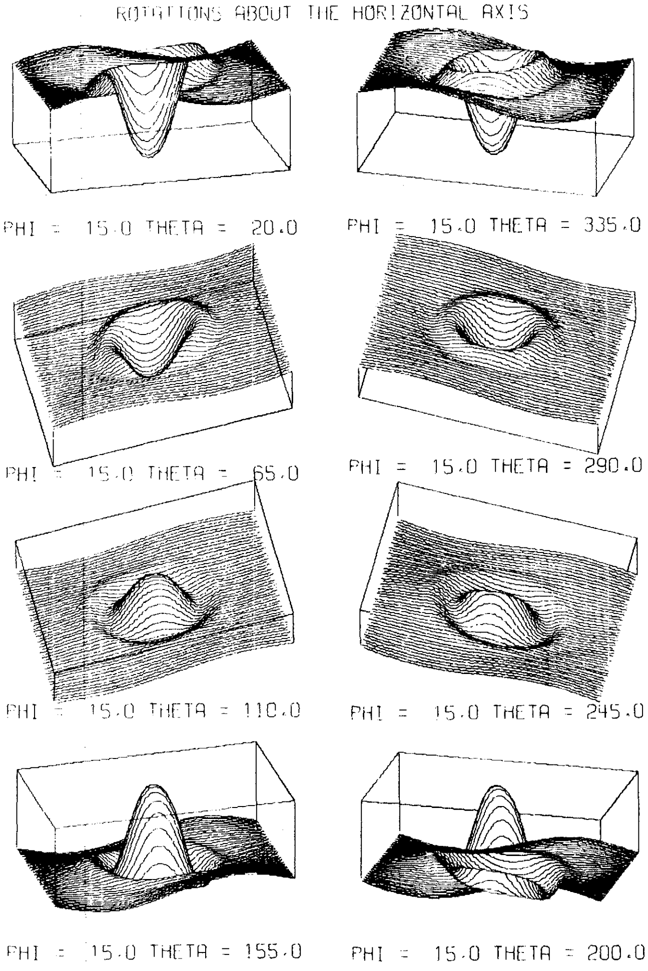
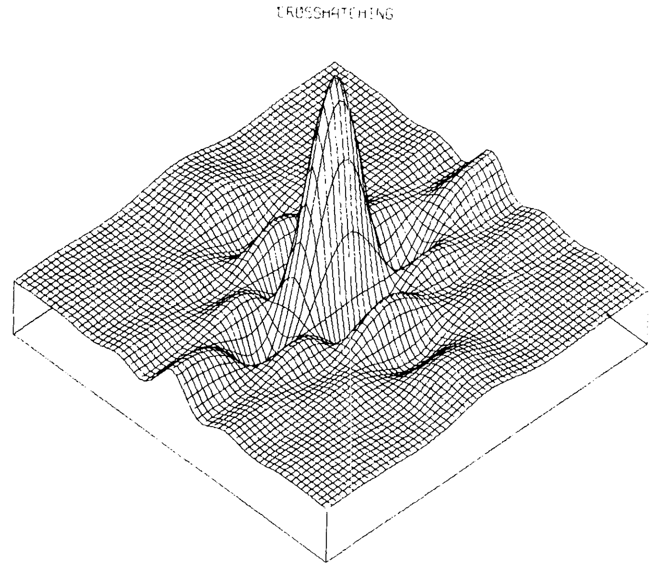


Fig. 3.



Algorithm

(A sample driver has been included to illustrate the use of this algorithm—L.D.F. and A.K.C.)

```
C THIS PROGRAM GENERATES AN EXAMPLE OF A CROSSHATCHED
C FIGURE, THAT IS, ONE FIGURE WHOSE LINES RUN PAPALLEL TO
C THE X-AXIS OVERLAYED BY ANOTHER FIGURE WHOSE LINES RUN
C PARALLEL TO THE Z-AXIS.  THE FUNCTION IS A PRODUCT TO TWO
C SINC (I.E. SINF(X)/X) FUNCTIONS.
      DIMENSION MASK(2ØØØ), VERTEX(16), OUTBUF(61), Z(61)
C FIRST FIGURE
C GENERATE DATA RUNNING PARALLEL TO X-AXIS
      DO 2Ø NLINE=1,61
        BEAMV = SINC(15.Ø*SINF((3*NLINE-93)*Ø.Ø17453293))
        DO 1Ø NPOINT=1,61
          OUTBUF(NPOINT) =
     *    BEAMV*SINC(7.5*SINF((3*NP        3)*Ø.Ø17453293)) +
     *    Ø.25
   1Ø   CONTINUE
C PLOT EACH LINE AS IT IS COMPUTED
        CALL PLOT3D(1Ø, Ø.Ø, OUTBUF, Ø.Ø, Ø.1, 4.Ø, -Ø.1,
     *  NLINE, 61, -45., -45., 5.Ø, 3.Ø, 1Ø.Ø, MASK, Ø)
   2Ø CONTINUE
C SECOND FIGURE
C GENERATE ARRAY OF Z-COMPONENTS
      DO 3Ø NLINE=1,61
        Z(NLINE) = -Ø.1*(NLINE-1)
   3Ø CONTINUE
C GENERATE DATA RUNNING PARALLEL TO Z-AXIS
      DO 5Ø NLINE=1,61
        X = Ø.1*(NLINE-1)
        BEAMV = SINC(7.5*SINF((3*NLINE-93)*Ø.Ø17453293))
        DO 4Ø NPOINT=1,61
          OUTBUF(NPOINT) =
     *    BEAMV*SINC(15.Ø*SINF((3*NPOINT-93)*Ø.Ø17453293)) +
     *    Ø.25
   4Ø   CONTINUE
C PLOT EACH LINE AS IT IS COMPUTED
        CALL PLOT3D(1Ø11, X, OUTBUF, Z, Ø.Ø, 4.Ø, 1.Ø,
     *  NLINE, 61, -45., -45., 5.Ø, 3.Ø, 1Ø.Ø, MASK, VERTEX)
   5Ø CONTINUE
C DRAW A FRAME ON THE FIGUPE
      CALL FRAMER(3, VERTEX, MASK)
      STOP
      END


      SUBROUTINE PLOT3D(IVXYZ, XDATA, YDATA, ZDATA, XSCALE,
     * YSCALE, ZSCALE, NLINE, NPNTS, PHI, THETA, XREF,
     * YREF, XLENTH, MASK, VERTEX)
C MASKED 3-DIMENSIONAL PLOT PROGRAM WITH ROTATIONS
C THIS ROUTINE WILL ACCEPT 3-DIMENSIONAL DATA IN VARIOUS
C FORMS AS INPUT, ROTATE IT IN 3-SPACE TO ANY ANGLE,
C AND PLOT THE PPOJECTION OF THE RESULTING FIGURE ONTO THE
C XY PLANE.  LINEAR INTERPOLATION IS USED BETWEEN DATA
C POINTS.  THOSE LINES OF A FIGURE WHICH SHOULD BE HIDDEN BY
C A PREVIOUS LINE ARE MASKED.
C THE MASKING TECHNIQUE USED BY THIS ROUTINE IS BASED ON
C TWO PREMISES -
C         LINES IN THE FOREGROUND (POSITIVE Z DIRECTION)
C         ARE PLOTTED BEFORE LINES IN THE BACKGROUND.
C         A LINE OR PORTION OF A LINE IS MASKED (HIDDEN) IF
C         IT LIES WITHIN THE REGION BOUNDED BY PREVIOUSLY
```

```
C         PLOTTED LINES.
C EACH CALL TO PLOT3D CAUSES ONE LINE OF A FIGURE TO BE
C PLOTTED.
C TWO PARAMETERS OF THE PLOTTER ARE SET ON THE INITIAL CALL
C FOR EACH FIGURE -
C (PIPI) IS THE NUMBER OF PLOTTER INCREMENTS PER INCH.
C (NYPI) IS THE NUMBER OF INCREMENTS AVAILABLE ACROSS THE
C WIDTH OF THE PAPER (Y-DIRECTION).
C WHEN A NEW FIGURE IS INITIATED, THE PLOTTER ORIGIN IS SET
C AT THE BOTTOM OF THE PAPER BY PLOT3D AND SHOULD NOT BE
C MOVED UNTIL THE FIGURE IS COMPLETED.
C INPUT PARAMETERS -
C (IVXYZ) IS A FOUR DIGIT DECIMAL INTEGER WHICH IS USED TO
C SELECT VARIOUS INPUT/OUTPUT OPTIONS.  THESE DIGITS, IN
C DECREASING ORDER OF MAGNITUDE, WILL BE REFERRED TO AS V,
C X, Y, AND Z.
C IF V .NE. 0, THE VERTICES OF THE CURRENT FIGURE AND THEIR
C PROJECTION ONTO THE Y=0 PLANE, WILL BE STORED IN A 16
C ENTRY REAL ARRAY (VERTEX), AND WILL BE UPDATED AS EACH
C LINE IS PLOTTED.  THESE COORDINATES ARE IN INCHES AND
C RELATIVE TO THE CURRENT PLOTTER ORIGIN.  THE X Y PAIRS
C ARE ORDERED SO THAT THE FIRST PAIR COORESPONDS TO THE
C FIRST POINT OF THE FIGURE, THE SECOND PAIR COORESPONDS
C TO THE LAST POINT OF THE FIRST LINE, AND THE FOLLOWING
C PAIRS ARE ORDERED IN A CIRCULAR FASHION.  THE PAIRS ON THE
C Y=0 PLANE OF THE FIGURE, THEN FOLLOW IN THE SAME ORDER.
C IF V=0, THE VERTEX PARAMETER IS IGNORED, BUT SHOULD NOT
C BE DELETED
C IF X=0, THE X-COMPONENTS OF THIS LINE ARE ASSUMED TO BE
C EQUALLY SPACED, AND ARE COMPUTED BY
C                 X(I)=XDATA+(I-1)*XSCALE
C WHERE (XDATA) IS THE INITIAL VALUE IN INCHES AND (XSCALE)
C IS THE SPACING BETWEEN POINTS IN INCHES.  IF X .NE. 0, THE
C X-COMPONENTS OF THIS LINE ARE READ FROM AN ARRAY AND
C MODIFIED BY
C                 X(I)=XDATA(I)*XSCALE
C WHERE (XSCALE) IS A SCALE FACTOR.
C THE SAME RELATIONS HOLD FOR THE Y-COMPONENTS, THAT IS, IF
C Y=0
C                 Y(I)=YDATA+(I-1)*YSCALE
C AND IF Y .NE. 0
C                 Y(I)=YDATA(I)*YSCALE
C IF Z=0, THE Z-COMPONENTS OF THIS LINE ARE ALL ASSUMED TO
C BE EQUAL, AND ARE COMPUTED BY
C                 Z(I)=ZDATA+(NLINE-1)*ZSCALE
C WHERE (NLINE) IS SOME INTEGER ASSOCIATED WITH THIS LINE.
C IF Z .NE. 0, AGAIN WE HAVE
C                 Z(I)=ZDATA(I)*ZSCALE
C WHEN (NLINE) IS EQUAL TO ONE, IT INDICATES THE BEGINNING
C OF A NEW FIGURE.  A CALL TO PLOT3D WITH (NLINE) EQUAL TO
C ZERO BEFORE INITIATING A NEW FIGURE SIMULATES A LINE DRAWN
C AT THE BOTTOM OF THE PAGE.  THEREFORE ONLY THOSE PORTIONS
C OF A LINE LYING ABOVE ALL PREVIOUS LINES WILL BE PLOTTED.
C ALL OTHER PARAMETERS ARE IGNORED ON SUCH A CALL.
C (NPNTS) IS THE NUMBER OF POINTS ON THIS LINE, AND MAY BE
C ALTERED FROM LINE TO LINE.
C (PHI) AND (THETA) ARE THE TWO ANGLES (IN DEGREES) USED TO
C SPECIFY THE DESIRED 3-DIMENSIONAL ROTATION.  THE FOLLOWING
C TWO DEFINIATIONS OF THESE ROTATIONS ARE EQUIVALENT -
C IN TERMS OF ROTATIONS OF AXES, THE INITIAL SYSTEM OF AXES,
C XYZ, IS ROTATED BY AN ANGLE (PHI) COUNTERCLOCKWISE ABOUT
C THE Y-AXIS, AND THE RESULTANT SYSTEM IS LABELED THE TUV
C AXES.  THE TUV AXES ARE THEN ROTATED BY AN ANGLE (THETA)
C COUNTERCLOCKWISE ABOUT THE T-AXIS, AND THIS FINAL SYSTEM
C IS LABELED THE PQR AXES.  THE PLOTTED FIGURE IS THE
C PROJECTION OF THE ORIGINAL FIGURE ONTO THE PQ-PLANE.
C IN TERMS OF ROTATIONS OF COORDINATES, THE FIGURE IS FIRST
C ROTATED BY AN ANGLE (THETA) CLOCKWISE ABOUT THE X-AXIS.
C THE RESULTANT FIGURE IS THEN ROTATED BY AN ANGLE (PHI)
C CLOCKWISE ABOUT ITS Y-AXIS.  THE PLOTTED FIGURE IS THE
C PROJECTION OF THIS FINAL FIGURE ONTO THE XY-PLANE.
C WARNING. SOME ROTATIONS WILL ALTER THE FOREGROUND/
C BACKGROUND RELATIONSHIPS BETWEEN THE LINES, AND
C THUS THE ORDER IN WHICH THEY SHOULD BE PLOTTED.
C (XREF) AND (YREF) ARE THE COORDINATES, IN INCHES,
C RELATIVE TO THE PLOTTER ORIGIN, TO BE USED AS THE ORIGIN
C OF THE FIGURE.
C (XLENTH) IS THE LENGTH, IN INCHES, TO WHICH THE PLOT IS
C RESTRICTED.  ANY POINT WHICH EXCEEDS THIS LIMIT, OR THE
C LIMITS OF THE PAPER IN THE Y DIRECTION (NYPI), WILL BE
C SET TO THAT LIMIT.
C (MASK) IS AN INTEGER ARRAY OF 2*XLENTH*PIPI ENTRIES WHICH
C IS USED TO STORE THE MASK.  THE CONTENTS OF THIS ARRAY
C SHOULD NOT BE ALTERED DURING THE PLOTING OF ANY GIVEN
C FIGURE.
C ALL PARAMETERS EXCEPT (MASK) AND (VERTEX) ARE RETURNED
C UNCHANGED.
C BETWEEN ANY TWO CALLS FOR THE SAME FIGURE, ANY PARAMETER
C CAN BE MEANINGFULLY CHANGED EXCEPT (XLENTH), (MASK), AND
C (VERTEX).
      INTEGER HIGH, OLDHI, OLDLOW
      DIMENSION XDATA(1), YDATA(1), ZDATA(1), MASK(1),
     * VERTEX(1)
      DATA INIT, JVXYZ, SPHI, STHETA/-1, -1, -1.0E99,
     * -1.0E99/
C INITIALIZATION PROCEDURES
C INITIALIZATION PROCEDURE FOR A NEW FIGURE
C TEST FOR SPECIAL MASK MODIFYING CALL
      IF (NLINE.EQ.0) GO TO 550
C DETERMINE IF INITIALIZATION IS REQUIRED
      IF (NLINE.NE.1) GO TO 20
C SET PLOTTER PARAMETERS
      PIPI = 100.0
      NYPI = 1090
C RESET PLOTTER ORIGIN TO BOTTOM OF PLOT PAGE
      I = NYPI + 100
      CALL IPLOT(0, -I, -3)
C COMPUTE LENGTH OF PLOT PAGE IN INCREMENTS
      LIMITX = XLENTH*PIPI + 0.5
      I = LIMITX + LIMITX
C INITIALIZE MASKING ARRAY OVER THE LENGTH OF THE PLOT PAGE
      DO 10 K=1,I
        MASK(K) = INIT
   10 CONTINUE
      INIT = -1
C SET THE NECESSARY INDICATORS FOR THE FIRST LINE OF A NEW
C FIGURE
      INCI = -1
      I = 0
C INPUT TYPE AND VERTEX INITIALIZATION
C DETERMINE IF INITIALIZATION IS REQUIRED
   20 IF (JVXYZ.EQ.IVXYZ) GO TO 70
C SET INDICATORS FOR TYPES OF INPUT DATA AND SAVING VERTICES
      JVXYZ = IVXYZ
      INDZ = 1
      INDY = 1
      INDX = 1
      INDV = 1
      IF (JVXYZ.LT.1000) GO TO 30
      INDV = 2
      JVXYZ = JVXYZ - 1000
   30 IF (JVXYZ.LT.100) GO TO 40
      INDX = 2
      JVXYZ = JVXYZ - 100
   40 IF (JVXYZ.LT.10) GO TO 50
      INDY = 2
      JVXYZ = JVXYZ - 10
   50 IF (JVXYZ.LT.1) GO TO 60
      INDZ = 2
   60 JVXYZ = IVXYZ
C ROTATION INITIALIZATION
C DETERMINE IF INITIALIZATION IS REQUIRED
   70 IF (PHI.EQ.SPHI .AND. THETA.EQ.STHETA) GO TO 80
C COMPUTE ROTATION FACTORS
      SPHI = SINF(0.0174532925*PHI)
      CPHI = COSF(0.0174532925*PHI)
      STHETA = SINF(0.0174532925*THETA)
      CTHETA = COSF(0.0174532925*THETA)
      A11 = CPHI
      A13 = -SPHI
      A21 = STHETA*SPHI
      A22 = CTHETA
      A23 = STHETA*CPHI
      SPHI = PHI
      STHETA = THETA
C PROCESSING PROCEDURES
C SET FLAG TO MOVE THROUGH THE DATA ARRAYS IN THE OPPOSITE
C DIRECTION
   80 INCI = -INCI
C SET INDICATOR TO THE FIRST POINT TO BE PROCESSED
      IF (I.NE.0) I = NPNTS + 1
C LOOP TO PROCESS EACH POINT IN THE DATA ARRAYS
      DO 530 K=1,NPNTS
C DATA CALCULATION
        I = I + INCI
        GO TO (90,100), INDX
   90   X = XDATA + (I-1)*XSCALE
        GO TO 110
  100   X = XDATA(I)*XSCALE
  110   GO TO (120,130), INDY
  120   Y = YDATA + (I-1)*YSCALE
        GO TO 140
  130   Y = YDATA(I)*YSCALE
  140   GO TO (150,160), INDZ
  150   Z = ZDATA + (NLINE-1)*ZSCALE
        GO TO 170
  160   Z = ZDATA(I)*ZSCALE
C DATA ROTATION
  170   XXX = A11*X + A13*Z + XREF
        XX = XXX
        IX = IROUND(XX*PIPI)
        YYY = A21*X + A23*Z + YREF
        YY = YYY + A22*Y
        IY = IROUND(YY*PIPI)
C RESTRICT FIGURE TO PLOT PAGE
        IF (IX.LE.0) IX = 1
        IF (IX.GT.LIMITX) IX = LIMITX
        IF (IY.LT.10) IY = 10
        IF (IY.GT.NYPI) IY = NYPI
        IF (K.NE.1) GO TO 250
C (LOC) IS THE POSITION OF THE PREVIOUS POINT WITH RESPECT
C TO THE MASK
C         +1  ABOVE THE MASK
C          0  WITHIN THE LIMITS OF THE MASK
C         -1  BELOW THE MASK
C PROCEDURE FOR INITIAL POINT OF EACH LINE
C LOCATE INITIAL POINT WITH RESPECT TO THE MASK THEN
C UPDATE THE MASK
        LOW = IX + IX
        HIGH = LOW - 1
        MLOW = MASK(LOW)
        MHIGH = MASK(HIGH)
        IF (MHIGH-IY) 200, 210, 180
  180   IF (MLOW-IY) 190, 230, 220
  190   LOCOLD = 0
        GO TO 240
  200   MASK(HIGH) = IY
        IF (MLOW.EQ.-1) MASK(LOW) = IY
  210   LOCOLD = +1
        GO TO 240
  220   MASK(LOW) = IY
  230   LOCOLD = -1
C MOVE THE RAISED PEN TO THIS INITIAL POINT
  240   CALL IPLOT(IX, IY, 3)
        JX = IX
        JY = IY
        IYREF = IY
C STORE VERTICES IF REQUESTED
        IF (INDV.EQ.1) GO TO 530
        INDEX = INCI + 6
        VERTEX(INDEX) = XX
        VERTEX(INDEX+1) = YY
        VERTEX(INDEX+8) = XXX
        VERTEX(INDEX+9) = YYY
        IF (NLINE.NE.1) GO TO 530
        VERTEX(1) = XX
        VERTEX(2) = YY
        VERTEX(9) = XXX
        VERTEX(10) = YYY
        GO TO 530
C SPECIAL CASE WHERE CHANGE IN X COORDINATE IS ZERO
C A SPECIAL PROVISION IS MADE AT THIS POINT SO THAT A LINE
C WILL NOT MASK ITSELF AS LONG AS THE X COORDINATE REMAINS
C CONSTANT
  250   IF (IX.NE.JX) GO TO 260
        JY = IY
        GO TO 280
C COMPUTE CONSTANTS FOR LINEAR INTERPOLATION
  260   YINC = FLOAT(IY-JY)/ABS(FLOAT(IX-JX))
        INCX = (IX-JX)/IABS(IX-JX)
        YJ = JY
C PREFORM LINEAR INTERPOLATION AT EACH INCREMENTAL STEP ON
C THE X AXIS
  270   JX = JX + INCX
        YJ = YJ + YINC
        JY = IROUND(YJ)
C LOCATE THE CURRENT POINT WITH RESPECT TO THE MASK AT THAT
C POINT THEN PLOT THE INCREMENT AS A FUNCTION OF THE
C LOCATION OF THE PREVIOUS POINT WITH RESPECT TO ITS MASK
        LOW = JX + JX
        HIGH = LOW - 1
        MLOW = MASK(LOW)
        MHIGH = MASK(HIGH)
  280   IF (MHIGH-JY) 300, 300, 290
  290   IF (MLOW-JY) 310, 320, 320
C THE CURRENT POINT IS ABOVE THE MASK
  300   LOC = +1
        IF (LOCOLD) 360, 370, 430
C THE CURRENT POINT IS WITHIN THE MASK
  310   LOC = 0
        IF (LOCOLD) 340, 350, 330
C THE CURRENT POINT IS BELOW THE MASK
  320   LOC = -1
        IF (LOCOLD) 510, 450, 440
C PLOT FROM ABOVE THE MASK TO WITHIN THE MASK
  330   IF (MHIGH.LE.IYREF) CALL IPLOT(JX, MHIGH, 2)
        GO TO 350
C PLOT FROM BELOW THE MASK TO WITHIN THE MASK
  340   IF (MLOW.GE.IYREF) CALL IPLOT(JX, MLOW, 2)
C PLOT FROM WITHIN THE MASK TO WITHIN THE MASK
  350   CALL IPLOT(JX, JY, 3)
        GO TO 520
C PLOT FROM BELOW THE MASK TO ABOVE THE MASK
  360   IF (MLOW-IYREF) 370, 380, 380
C PLOT FROM WITHIN THE MASK TO ABOVE THE MASK
  370   IF (MHIGH-IYREF) 400, 390, 390
  380   CALL IPLOT(JX, MLOW, 2)
  390   CALL IPLOT(JX, MHIGH, 3)
        GO TO 430
  400   IF (MHIGH.EQ.-1) GO TO 430
        OLDHI = HIGH - 2*INCX
        IF (MASK(OLDHI)-JY) 420, 420, 410
  410   CALL IPLOT(JX, JY, 3)
        GO TO 430
  420   CALL IPLOT(JX-INCX, MASK(OLDHI), 3)
C PLOT FROM ABOVE THE MASK TO ABOVE THE MASK
  430   MASK(HIGH) = JY
        IF (MLOW.EQ.-1) MASK(LOW) = JY
        CALL IPLOT(JX, JY, 2)
        GO TO 520
C PLOT FROM ABOVE THE MASK TO BELOW THE MASK
  440   IF (MHIGH-IYREF) 460, 460, 450
C PLOT FROM WITHIN THE MASK TO BELOW THE MASK
  450   IF (MLOW-IYREF) 470, 470, 480
  460   CALL IPLOT(JX, MHIGH, 2)
  470   CALL IPLOT(JX, MLOW, 3)
        GO TO 510
  480   OLDLOW = LOW - 2*INCX
        IF (MASK(OLDLOW)-JY) 490, 500, 500
  490   CALL IPLOT(JX, JY, 3)
        GO TO 510
  500   CALL IPLOT(JX-INCX, MASK(OLDLOW), 3)
C PLOT FROM BELOW THE MASK TO BELOW THE MASK
  510   MASK(LOW) = JY
        CALL IPLOT(JX, JY, 2)
  520   IYREF = JY
        LOCOLD = LOC
        IF (JX.NE.IX) GO TO 270
  530 CONTINUE
C RAISE PEN
      CALL IPLOT(JX, JY, 3)
C STORE VERTICES IF REQUESTED
      IF (INDV.EQ.1) GO TO 540
      INDEX = -INCI + 6
      VERTEX(INDEX) = XX
      VERTEX(INDEX+1) = YY
      VERTEX(INDEX+8) = XXX
      VERTEX(INDEX+9) = YYY
      IF (NLINE.NE.1) GO TO 540
      VERTEX(3) = XX
      VERTEX(4) = YY
      VERTEX(11) = XXX
      VERTEX(12) = YYY
  540 I = I - 1
C RETURN TO CALLING PROGRAM
      RETURN
C OPTION TO MODIFY THE MASKING TECHNIQUE TO BE USED ON THE
C FOLLOWING FIGURE SO AS TO PLOT ONLY ABOVE ALL PREVIOUS
C LINES.
  550 INIT = 0
      RETURN
      END
```

```
      SUBROUTINE FRAMER(IHCOR, VERTEX, MASK)
C ROUTINE TO PLOT A FRAME ON THE PROJECTION OF A
C 3-DIMENSIONAL FIGURE AS DRAWN BY PLOT3D.
C INPUT PARAMETERS -
C     IHCOR     NUMBER OF THE VERTEX OF THE FIGURE WHICH
C               APPEARS TO BE FURTHEST IN THE BACKGROUND
C               (MINUS Z DIRECTION).
C     VERTEX    ARRAY CONTAINING THE COORDINATES OF THE
C               VERTICES OF THIS FIGURE AS RETURNED FROM
C               PLOT3D ON THE LAST CALL.
C     MASK      ARRAY CONTAINING THE MASK FOR THIS FIGURE
C               AS RETURNED BY PLOT3D ON THE LAST CALL.
C THE VERTICES OF THE FRAME ARE NUMBERED (1-4) IN THE SAME
C ORDER AS THEIR COORDINATES APPEAR IN VERTEX.
C THE MASK ARRAY IS ALTERED BY THIS ROUTINE,
C BUT THE PLOTTER ORIGIN IS NOT MOVED.
      DIMENSION VERTEX(1), MASK(1), ARRAY(14)
      I = 2*IHCOR
      IF (I.LT.2) I = 2
      IF (I.GT.8) I = 8
C THE VERTICES WHICH MAY BE HIDDEN
C ARE DRAWN BY A CALL TO PLOT3D.
      ARRAY(1) = VERTEX(I-1)
      ARRAY(8) = VERTEX(I)
      ARRAY(2) = VERTEX(I+7)
      ARRAY(9) = VERTEX(I+8)
      ARRAY(4) = ARRAY(2)
      ARRAY(11) = ARRAY(9)
      ARRAY(6) = ARRAY(2)
      ARRAY(13) = ARRAY(9)
      ARRAY(7) = ARRAY(1)
      ARRAY(14) = ARRAY(8)
      I = I - 2
      IF (I.EQ.0) I = 8
      ARRAY(3) = VERTEX(I+7)
      ARRAY(10) = VERTEX(I+8)
      I = I + 4
      IF (I.GT.8) I = I - 8
      ARRAY(5) = VERTEX(I+7)
      ARRAY(12) = VERTEX(I+8)
      CALL PLOT3D(110, ARRAY, ARRAY(8), 0.0, 1.0, 1.0, 0.0,
     * 2, 7, 0.0, 0.0, 0.0, 0.0, 0.0, MASK, 0)
C THE REMAINING VERTICES ARE DRAWN BY CALLS TO PLOT.
      CALL PLOT(VERTEX(I-1), VERTEX(I), 3)
      I = I - 2
      DO 10 J=1,3
        I = I + 2
        IF (I.EQ.10) I = 2
        CALL PLOT(VERTEX(I+7), VERTEX(I+8), 2)
   10 CONTINUE
      CALL PLOT(VERTEX(I-1), VERTEX(I), 2)
      I = I - 2
      IF (I.EQ.0) I = 8
      CALL PLOT(VERTEX(I-1), VERTEX(I), 3)
      CALL PLOT(VERTEX(I+7), VERTEX(I+8), 2)
      RETURN
      END
```

ACM Transactions on Mathematical Software, Vol. 1, No. 3, September 1975.

REMARK ON ALGORITHM 483

Masked Three-Dimensional Plot Program with Rotations [J6]
[S. L. Watkins, *Comm. ACM 17*, 9 (Sept. 1974), 520–523]

Robert Feinstein [Recd 28 April 1975]
The Marine Biomedical Institute, The University of Texas Medical Branch at Galveston, 200 University Boulevard, Galveston, TX 77550

In the sample main program of Algorithm 483, line 13 should read:

```
*      BEAMV*SINC(7.5*SINF((3*NPOINT—93)*0.017453293))+
```

Further, the algorithm does not define subroutine PLOT which is called by FRAMER. Whereas IPLOT accepts coordinates in increments, PLOT accepts coordinates in inches.

I have modified this algorithm to run on a PDP 11/45-GOULD 5000 and would be happy to supply a listing to anyone who desires it.

Algorithm 484

Evaluation of the Modified Bessel Functions K₀(z) and K₁(z) for Complex Arguments [S17]

Keith H. Burrell [Recd. 30 Mar. 1972] California Institute of Technology, Pasadena, CA 91109

Work supported by the Atomic Energy Commission under contract AT(04-3)767.

Key Words and Phrases: Bessel functions, Hankel functions, modified Bessel functions, Gauss-Hermite quadrature
CR Categories: 5.12
Language: Fortran

Description

Introduction. This procedure evaluates the real and imaginary parts of the modified Bessel functions $K_0(z)$ and $K_1(z)$ for values of the complex argument $z = x + iy$ in the half plane $x \geq 0$. (The notation $K_n(z)$ is fairly standard; the exact definition of the function is given in [1]).

Methods for the published algorithm. Many previous methods of calculating these functions have simply used the series expansion for arguments z of small magnitude (i.e. $|z| \lesssim 11$) and the asymptotic expansion for larger arguments. Rewriting eqs. 9.6.11 and 9.7.2 of [1] in a form more suitable for recursive computation, the series expansion may be expressed as

$$K_0(z) = \sum_{j=0}^{\infty} t_{0j}(z)(z/2)^{2j}/(j!)^2 \tag{1}$$

$$K_1(z) = 1/z - z/2\sum_{j=0}^{\infty} t_{1j}(z)(z/2)^{2j}/(j!)^2, \tag{2}$$

$$\begin{aligned} t_{00} &= -(\ln(z/2) + \gamma), \\ t_{0j} &= t_{0(j-1)} + 1/j, \quad j > 0, \\ t_{1j} &= [t_{0j} + 1/(2j + 2)]/(j + 1), \quad j \geq 0, \end{aligned}$$

where $\gamma = 0.577\ldots$ is Euler's constant; the asymptotic expansion may be written as

$$K_n(z) \sim \left(\frac{\pi}{2z}\right) \cdot e^{-z}[a_{n0} + a_{n1}/1!(8z) + a_{n2}/2!(8z)^2 + a_{n3}/3!(8z)^3 + \cdots]$$

$$k_0 = -1, \; k_j = k_{f-j} - 8j,$$

$$a_{00} = 1, \quad a_{0j} = (k_j - 4)a_{0(j-1)},$$

$$a_{10} = 1, \quad a_{1j} = k_j a_{1(j-1)}.$$

Methods based solely on these expansions tend to be inefficient because of the large number of terms in the series that must be evaluated when $|z|$ gets as large as 10. Further, they are of limited accuracy due to the loss of significant digits in summing the series when $y \ll x$ and $x \gtrsim 5$. To overcome these difficulties, the integral representation developed by Hunter [2] can be used.

$$K_n(z) = \sqrt{\pi}e^{-z}/(\Gamma(n+\tfrac{1}{2})(2z)^n) \int_{-\infty}^{+\infty} e^{-t^2}t^{2n}(2z+t^2)^{n-\frac{1}{2}}\, dt, \qquad |\arg z| < \pi. \tag{3}$$

Hunter suggests evaluation of this integral by means of the trapezoidal rule, which is well suited to integrands of this type, but one can achieve equivalent accuracy with fewer evaluations of the integrand by using Gauss-Hermite quadrature [3].

To have a fast, accurate algorithm, the functions must be evaluated by different methods in different regions of the complex plane. Owing to the singularity at the origin for $K_n(z)$, only the series expansions will be useful near $z = 0$. For moderate values of $|z|$, the integral representations will be the most useful, while for $|z|$ large, calculation of the asymptotic expansions will be faster than that of the integral. To decide exactly where each method should be used and how good the resulting algorithm is, one must be able to assess the speed and accuracy of each method. This could be done from first principles; but since close estimates of the error tend to involve considerable mathematical labor, I thought it easier to write a test algorithm which, although very slow, would evaluate $K_0(z)$ and $K_1(z)$ quite accurately.

Test algorithm. For $|z| < 3$, the test algorithm uses the series expansions; otherwise, the integral representation in eq. (3) is evaluated using the trapezoidal rule. To find the error in this algorithm, consider first the truncation error caused by stopping after n terms of the series in eqs. (1) and (2).

Using the integral representation (eq. 9.6.17 in [1])

$$K_0(z) = -1/\pi\int_0^{\pi} d\theta e^{z\cos\theta}[\gamma + \ln(2z\sin^2\theta)]$$

and the identities

$$K_0'(z) = -K_1(z) \text{ and}$$

$$e^z = \sum_{m=0}^{n-1} z^m/m! + z^n/(n-1)!\int_0^1 dt(1-t)^{n-1}e^{tz}$$

it is easy to show that

$$K_0(z) = \sum_{j=0}^{n-1} t_{0j}(z)(z/2)^{2j}/(j!)^2 + T_{0n}(z) \text{ and}$$

$$K_1(z) = 1/z - z/2\sum_{j=0}^{n-1} t_{1j}(z)(z/2)^{2j}/(j!)^2 + T_{1n}(z) \text{ where}$$

$$T_{0n}(z) = -z^{2n}/(\pi(2n-1)!)\int_0^1 dt(1-t)^{2n-1}\int_0^{\pi} d\theta e^{tz\cos\theta} (\gamma + \ln(2z\sin^2\theta))$$

Editors' note: *Algorithm 484 described here is available on magnetic tape from the Department of Computer Science, University of Colorado, Boulder, CO 80302. The cost for the tape is $16.00 (U.S. and Canada) or $18.00 (elsewhere). If the user sends a small tape (wt. less than 1 lb.) the algorithm will be copied on it and returned to him at a charge of $10.00 (U.S. only). All orders are to be prepaid with checks payable to ACM Algorithms. The algorithm is recorded as one file of BCD 80 character card images at 556 B.P.I., even parity, on seven track tape. We will supply algorithms at a density of 800 B.P.I. if requested. Cards for algorithms are sequenced starting at 10 and incremented by 10. The sequence number is right justified in column 80. Although we will make every attempt to insure that the algorithm conforms to the description printed here, we cannot guarantee it, nor can we guarantee that the algorithm is correct.—L.D.F. and A.K.C.*

$$T_{1n}(z) = z^{2n+1}/(\pi(2n+1)!) \int_0^1 dt(1-t)^{2n+1} \int_0^\pi d\theta e^{tz\cos\theta} \cdot (1 + (2n+2+zt\cos\theta)(\gamma+\ln(2z\sin^2\theta)).$$

At least four terms in each sum are taken by the test algorithm, thus

$$| T_{0n}(z) | \leq \frac{\sqrt{\pi}}{2} e^x \left| t_{0n}(z) \frac{(z/2)^{2n}}{(n!)^2} \right| \frac{| \gamma + \ln(2z) | + \ln 4}{| t_{0n}(z) |}$$

$$| T_{1n}(z) | \leq \frac{\sqrt{\pi}}{2} e^x \left| t_{1n}(z) \frac{(z/2)^{2n+1}}{(n!)^2} \right| \frac{(| z |+2n+2)(| \gamma+\ln(2z) |+\ln 4)+1}{2(n+1) | t_{0n}(z)+1/(2n) |}.$$

Evaluation continues in the test program until $| t_{0n}(z)(z/2)^{2n}(n!)^{-2}/K_0(z) | < 10^{-17}$ and $| t_{1n}(z)(z/2)^{2n+1}(n!)^{-2}/K_1(z) | < 10^{-17}$. Thus, defining $\mathcal{E}_0(z)$ and $\mathcal{E}_1(z)$ to be the absolute values of the relative errors in the computation of $K_0(z)$ and $K_1(z)$, we obtain the limits $\mathcal{E}_0(z) \leq 1.115 \times 10^{-15}$ and $\mathcal{E}_1(z) \leq 1.278 \times 10^{-15}$.

The errors in evaluating eq. (3) by the trapezoidal rule have been analyzed by Hunter [2]. Expressing the trapezoidal rule as

$$\int_0^\infty F(t)\,dt = h\left[\frac{1}{2}F(0) + \sum_{r=1}^{\infty} F(rh)\right] - \frac{1}{2}E(h) \tag{4}$$

he obtains bounds for $E_0(z, h)$ and $E_1(z, h)$, the errors in $K_0(z)$ and $K_1(z)$, respectively. The test algorithm uses $h = 0.25$. For this, Hunter's formulas yield $| E_0(z, h) | \leq 3.047 \times 10^{-18}$ and $| E_1(z, h) | \leq 4.008 \times 10^{-18}$.

By taking 32 terms in the sum in eq. (4), the truncation error can be made much smaller than the $E_n(z, h)$, so that $\mathcal{E}_0(z) \leq 4.236 \times 10^{-18}$ and $\mathcal{E}_1(z) \leq 5.435 \times 10^{-17}$. (Round-off error is not a problem for the test algorithm. The series is not subject to it for $| z | \leq 3.0$, and all the terms in the sum in eq. (4) have the same sign.)

Results of testing. The goal was to make the published algorithm accurate to a few parts in 10^{10}. On this scale, the test algorithm can be viewed as exact, at least for purposes of computing the modulus of the relative errors. Using the test algorithm, the published algorithm was found to be most efficient if the series are used for $| z | < 4.3$; the integrals in eq. (3), evaluated with 15 point Gauss-Hermite quadrature, are used for $4.3 \leq | z | \leq 14.0$; and the asymptotic expansions are used otherwise.

During the check runs to find these points of division, it was noticed that the number of terms needed in the series could be predicted approximately by two simple functions of $| z |$. With this in mind, the error expression for the asymptotic expansions (eq. 9.7.2 in [1]) was used to generate a similar function for these expansions. By predicting the number of terms needed, instead of making convergence tests in the loops that sum the expansions, an appreciable reduction in the number of computations can be achieved. This amounts to a 30 percent saving, for example, for the series expansions.

The most extensive test runs were done for $z = \rho e^{i\phi}$ having the values $\phi = 0°(5°)90°$ and $\rho = 0.1, 0.5(0.5)\ 120.0$. Another test run with $\rho = 0.1, 0.5(0.5)30.0$ verified that

$$K_n(\rho e^{i\phi}) = K^*_n(\rho e^{-i\phi})$$

by checking the values $\phi = -90°(5°)90°$. All tests were made using double precision arithmetic on an IBM 370/155. They showed that

$$\mathcal{E}_0(z) \leq 3.55 \times 10^{-10} \text{ and}$$
$$\mathcal{E}_1(z) \leq 3.93 \times 10^{-10}.$$

Finally, it should be noted that the algorithm actually returns the values of $e^x K_0(z)$ and $e^x K_1(z)$. For $| z |$ large, $| K_n(z) | \sim e^{-x}(\pi/2 | z |)^{\frac{1}{2}}$ so that such a return expands the range of $| z |$ over which this procedure may be used.

References

1. Abramowitz, M., and Stegun, I.A. (Eds.) *Handbook of Mathematical Functions.* Applied Math. Series 55, National Bureau of Standards, U.S. Gov. Print. Off., Washington, D.C., 1964.
2. Hunter, D.B. The calculation of certain Bessel functions. *Math. Comp. 18* (1964), 123–128.
3. Salzer, H.E., Zucker, R., Capuano, R. Tables of the zeros and weight factors of the first twenty Hermite polynomials. *J. Res. Nat. Bur. Standards 48* (1952), 111–116.

Algorithm

```
      SUBROUTINE KZEONE(X, Y, REØ, IMØ, RE1, IM1)
C THE VARIABLES X AND Y ARE THE REAL AND IMAGINARY PARTS OF
C THE ARGUMENT OF THE FIRST TWO MODIFIED BESSEL FUNCTIONS
C OF THE SECOND KIND,KØ AND K1.  REØ,IMØ,RE1 AND IM1 GIVE
C THE REAL AND IMAGINARY PARTS OF EXP(X)*KØ AND EXP(X)*K1,
C RESPECTIVELY.  ALTHOUGH THE REAL NOTATION USED IN THIS
C SUBROUTINE MAY SEEM INELEGANT WHEN COMPARED WITH THE
C COMPLEX NOTATION THAT FORTRAN ALLOWS, THIS VERSION RUNS
C ABOUT 3Ø PERCENT FASTER THAN ONE WRITTEN USING COMPLEX
C VARIABLES.
      DOUBLE PRECISION X, Y, X2, Y2, REØ, IMØ, RE1, IM1,
     * R1, R2, T1, T2, P1, P2, RTERM, ITERM, EXSQ(8), TSQ(8)
      DATA TSQ(1) /Ø.ØDØ/, TSQ(2) /3.19303633920635D-1/,
     * TSQ(3) /1.29075862295915DØ/, TSQ(4)
     * /2.95837445869665DØ/, TSQ(5) /5.40903159724444DØ/,
     * TSQ(6) /8.80407957805676DØ/, TSQ(7)
     * /1.34685357432515D1/, TSQ(8) /2.02499163658709D1/,
     * EXSQ(1) /Ø.564100308726DØ/, EXSQ(2)
     * /Ø.412028687498DØ/, EXSQ(3) /Ø.158488915795DØ/,
     * EXSQ(4) /Ø.307800338725D-1/, EXSQ(5)
     * /Ø.277806884291D-2/, EXSQ(6) /Ø.100004441232D-3/,
     * EXSQ(7) /Ø.105911554771D-5/, EXSQ(8)
     * /Ø.152247580425D-8/
C THE ARRAYS TSQ AND EXSQ CONTAIN THE SQUARE OF THE
C ABSCISSAS AND THE WEIGHT FACTORS USED IN THE GAUSS-
C HERMITE QUADRATURE.
      R2 = X*X + Y*Y
      IF (X.GT.Ø.ØDØ .OR. R2.NE.Ø.ØDØ) GO TO 1Ø
      WRITE (6,99999)
      RETURN
   1Ø IF (R2.GE.1.96D2) GO TO 5Ø
      IF (R2.GE.1.849D1) GO TO 3Ø
C THIS SECTION CALCULATES THE FUNCTIONS USING THE SERIES
C EXPANSIONS
      X2 = X/2.ØDØ
      Y2 = Y/2.ØDØ
      P1 = X2*X2
      P2 = Y2*Y2
      T1 = -(DLOG(P1+P2)/2.ØDØ+Ø.5772156649Ø15329DØ)
C THE CONSTANT IN THE PRECEDING STATEMENT IS EULER*S
C CONSTANT
      T2 = -DATAN2(Y,X)
      X2 = P1 - P2
      Y2 = X*Y2
      RTERM = 1.ØDØ
      ITERM = Ø.ØDØ
      REØ = T1
      IMØ = T2
      T1 = T1 + Ø.5DØ
      RE1 = T1
      IM1 = T2
      P2 = DSQRT(R2)
      L = 2.1Ø6DØ*P2 + 4.4DØ
      IF (P2.LT.8.ØD-1) L = 2.129DØ*P2 + 4.ØDØ
      DO 2Ø N=1,L
        P1 = N
        P2 = N*N
        R1 = RTERM
        RTERM = (R1*X2-ITERM*Y2)/P2
        ITERM = (R1*Y2+ITERM*X2)/P2
        T1 = T1 + Ø.5DØ/P1
        REØ = REØ + T1*RTERM - T2*ITERM
        IMØ = IMØ + T1*ITERM + T2*RTERM
        P1 = P1 + 1.ØDØ
        T1 = T1 + Ø.5DØ/P1
        RE1 = RE1 + (T1*RTERM-T2*ITERM)/P1
        IM1 = IM1 + (T1*ITERM+T2*RTERM)/P1
   2Ø CONTINUE
      R1 = X/R2 - Ø.5DØ*(X*RE1-Y*IM1)
      R2 = -Y/R2 - Ø.5DØ*(X*IM1+Y*RE1)
      P1 = DEXP(X)
      REØ = P1*REØ
      IMØ = P1*IMØ
      RE1 = P1*R1
      IM1 = P1*R2
      RETURN
C THIS SECTION CALCULATES THE FUNCTIONS USING THE INTEGRAL
C REPRESENTATION, EQN 3, EVALUATED WITH 15 POINT GAUSS-
C HERMITE QUADRATURE
   3Ø X2 = 2.ØDØ*X
      Y2 = 2.ØDØ*Y
      R1 = Y2*Y2
      P1 = DSQRT(X2*X2+R1)
      P2 = DSQRT(P1+X2)
      T1 = EXSQ(1)/(2.ØDØ*P1)
      REØ = T1*P2
      IMØ = T1/P2
      RE1 = Ø.ØDØ
      IM1 = Ø.ØDØ
      DO 4Ø N=2,8
        T2 = X2 + TSQ(N)
        P1 = DSQRT(T2*T2+R1)
        P2 = DSQRT(P1+T2)
        T1 = EXSQ(N)/P1
        REØ = REØ + T1*P2
        IMØ = IMØ + T1/P2
        T1 = EXSQ(N)*TSQ(N)
        RE1 = RE1 + T1*P2
        IM1 = IM1 + T1/P2
   4Ø CONTINUE
      T2 = -Y2*IMØ
      RE1 = RE1/R2
      R2 = Y2*IM1/R2
      RTERM = 1.41421356237309DØ*DCOS(Y)
      ITERM = -1.41421356237309DØ*DSIN(Y)
C THE CONSTANT IN THE PREVIOUS STATEMENTS IS,OF COURSE,
C SQRT(2.Ø).
      IMØ = REØ*ITERM + T2*RTERM
      REØ = REØ*RTERM - T2*ITERM
      T1 = RE1*RTERM - R2*ITERM
      T2 = RE1*ITERM + R2*RTERM
      RE1 = T1*X + T2*Y
      IM1 = -T1*Y + T2*X
      RETURN
C THIS SECTION CALCULATES THE FUNCTIONS USING THE
C ASYMPTOTIC EXPANSIONS
   5Ø RTERM = 1.ØDØ
      ITERM = Ø.ØDØ
      REØ = 1.ØDØ
      IMØ = Ø.ØDØ
      RE1 = 1.ØDØ
      IM1 = Ø.ØDØ
      P1 = 8.ØDØ*R2
      P2 = DSQRT(R2)
      L = 3.91DØ+8.12D1/P2
      R1 = 1.ØDØ
      R2 = 1.ØDØ
      M = -8
      K = 3
      DO 6Ø N=1,L
        M = M + 8
        K = K - M
        R1 = FLOAT(K-4)*R1
        R2 = FLOAT(K)*R2
        T1 = FLOAT(N)*P1
        T2 = RTERM
        RTERM = (T2*X+ITERM*Y)/T1
        ITERM = (-T2*Y+ITERM*X)/T1
        REØ = REØ + R1*RTERM
        IMØ = IMØ + R1*ITERM
        RE1 = RE1 + R2*RTERM
        IM1 = IM1 + R2*ITERM
   6Ø CONTINUE
      T1 = DSQRT(P2+X)
      T2 = -Y/T1
      P1 = 8.86226925452758D-1/P2
C THIS CONSTANT IS SQRT(PI)/2.Ø, WITH PI=3.14159...
      RTERM = P1*DCOS(Y)
      ITERM = -P1*DSIN(Y)
      R1 = REØ*RTERM - IMØ*ITERM
      R2 = REØ*ITERM + IMØ*RTERM
      REØ = T1*R1 - T2*R2
      IMØ = T1*R2 + T2*R1
      R1 = RE1*RTERM - IM1*ITERM
      R2 = RE1*ITERM + IM1*RTERM
      RE1 = T1*R1 - T2*R2
      IM1 = T1*R2 + T2*R1
      RETURN
99999 FORMAT (42H  ARGUMENT OF THE BESSEL FUNCTIONS IS ZERO,
     * 35H OR LIES IN LEFT HALF COMPLEX PLANE)
      END
```

Algorithm 485

Computation of g-Splines via a Factorization Method [E2]

Harold D. Eidson and Larry L. Schumaker [Recd. 19 Oct. 1972] Department of Mathematics and Center for Numerical Analysis, University of Texas, Austin, TX 78712

This work was supported in part by AFOSR-69-1812-C.

Key Words and Phrases: approximation, spline approximation
CR Categories: 5.13
Language: Fortran

Abstract

Fortran subroutines are presented for the purpose of computing and evaluating g-splines interpolating Hermite-Birkoff data. The subroutines are based on a factorization method for computing g-splines discussed by Munteanu and Schumaker (*Math. Comp. 27* (1973), 317–325).

Description

1. Introduction. In the following we present subroutines for calculating polynomial spline functions solving Hermite-Birkhoff (HB) interpolation problems. The subroutines are based on algorithms described in [9].

We begin by reviewing the definition of an HB-interpolation problem. Let $N \geq 2$ and $x_1 < x_2 < \cdots < x_N$ be prescribed. Suppose for each j, $1 \leq j \leq N$, that z_j is a positive integer, $IM_{1,j} < IM_{2,j} < \cdots < IM_{z_j,j}$ are positive integers, and $y_{1,j}, y_{2,j}, \ldots, y_{z_j,j}$ are prescribed real numbers. The HB-interpolation problem is to determine s such that

$$s^{(IM_{ij}-1)}(x_j) = y_{i,j},\ i = 1, 2, \ldots, z_j, j = 1, 2, \ldots, N. \tag{1}$$

We see that z_j describes the number of derivatives prescribed at x_j while the vector $(IM_{1,j}, \ldots, IM_{z_j,j})$ describes which derivatives. If $z_j = 1$, $j = 1, \ldots, N$, we have a simple interpolation problem.

We are concerned with solving HB-interpolation problems with polynomial splines. Let M be an integer, $M \geq IM_{z_j,j}$, $j = 1, 2, \ldots, N$. Then (cf. [4]) there exists a function s satisfying (1) and

$$s^{(2M)}(t) = 0,\ x_j < t < x_{j+1},\ j = 1, 2, \ldots, N - 1; \tag{2}$$

$$s^{(M)}(t) = 0,\ t < x_1.\ t > x_N; \tag{3}$$

$$s \in C^{(M-1)}(-\infty, \infty); \tag{4}$$

$$s^{(2M-l)}(x_j+) = s^{(2M-l)}(x_j-), \tag{5}$$
$$l \in \{1, \ldots, M\} \setminus \{IM_{1,j}, \ldots, IM_{z_j,j}\}$$
$$j = 1, 2, \ldots, N.$$

The function s is called a g-spline. It is a polynomial spline of degree $2M - 1$; i.e. it is piecewise a polynomial of degree $2M - 1$. The way in which the pieces tie together is described by (4) and (5).

If the only polynomial of degree $M - 1$ which solves the homogeneous HB-interpolation problem (i.e. satisfies (1) with zero right-hand side) is the identically zero polynomial, then we say the HB-problem is M-poised. In this case there is a unique g-spline of degree $2M$-1 solving the HB-problem (1). We consider constructing g-splines only for M-poised HB-problems.

Given an M-poised HB-interpolation problem, the unique g-spline interpolant s satisfying (1)–(5) can be represented as

$$s(t) = \begin{cases} p_1(t), & t \leq x_1 \\ p_j(t), & x_{j-1} < t \leq x_j, j = 2, 3, \ldots, N, \\ p_{N+1}(t), & t > x_N, \end{cases} \tag{6}$$

where for $j = 1, 2, \ldots, N$, $p_j(t)$ is a polynomial of the form

$$p_j(t) = \sum_{l=1}^{2M} C_{l,j}(t - x_j)^{l-1} \text{ and} \tag{7}$$

$$p_{N+1}(t) = \sum_{l=1}^{M} C_{l,N}\,(t - x_N)^{l-1}. \tag{8}$$

For later use we introduce the notation $C_j = (C_{1,j}, \ldots, C_{2M,j})^T$. Several algorithms were discussed in [9] for computing the coefficients $\{C_{l,j}\}_{l=1}^{2M}\,{}_{j=1}^{N}$ of s. We give a subroutine *GSF* below which implements Method 3 of [9]. We also include a function *GVAL* for evaluating s or its various derivatives (For a sketch of the organization of these algorithms, see Section 2 below.)

2. Organization of the algorithms. GSF consists of: (i) a forward march during which certain matrices U_K, V_K, and A_K are set up for $K = 2, 3, \ldots, N - 1$; (ii) the solution of a $2M$-system for C_N; and (iii) a backward march in which the $C_{N-1}, C_{N-2}, \ldots, C_1$ are computed recursively. This proceeds as follows. With appropriate $2M \times 2M$ Taylor matrices TB_K and $Z_K \times 2M$ matrices $INTCON_K$, the interpolating conditions (1) at $X(K)$ can be written as $INTCON_K\ TB_{K+1}C_{K+1} = B_K$. Similarly with $2M - Z_K \times 2M$ matrices $SMOCON_K$ the smoothing conditions (5) at $X(K)$ can be written as $SMOCON_KC_K = SMOCON_KTB_{K+1}C_{K+1}$. Finally, the end conditions (3) at $X(1)$ and $X(N)$ can be written as $ENDCON_1\ TB_1C_2 = D_1$ and $ENDCON_NC_N = D_N$. To compute U_2, V_2, and A_2 the matrix

$$\begin{bmatrix} SMOCON_2 & -SMOCON_2 \\ INTCON_1 TB_1 & 0 \\ ENDCON_1 TB_1 & 0 \end{bmatrix} \begin{bmatrix} C_2 \\ TB_2 C_3 \end{bmatrix} = \begin{bmatrix} 0 \\ B_1 \\ D_1 \end{bmatrix}$$

is triangularized by *TRISYS* to the form

$$\begin{bmatrix} U_2 & V_2 \\ 0 & W_2 \end{bmatrix} \begin{bmatrix} C_2 \\ TB_2 C_3 \end{bmatrix} = \begin{bmatrix} A_2 \\ D_2 \end{bmatrix}.$$

To get U_3, V_3, A_3 we triangularize

$$\begin{bmatrix} SMOCON_3 & -SMOCON_3 \\ INTCON_2 TB_2 & 0 \\ W_2 TB_2 & 0 \end{bmatrix} \begin{bmatrix} C_3 \\ TB_3 C_4 \end{bmatrix} = \begin{bmatrix} 0 \\ B_2 \\ D_2 \end{bmatrix}$$

Editors' note: *Algorithm 485 described here is available on magnetic tape from the Department of Computer Science, University of Colorado, Boulder, CO 80302. The cost for the tape is \$16.00 (U.S. and Canada) or \$18.00 (elsewhere). If the user sends a small tape (wt. less that 1 lb.) the algorithm will be copied on it and returned to him at a charge of \$10.00 (U.S. only). All orders are to be prepaid with checks payable to ACM Algorithms. The algorithm is recorded as one file of BCD 80 character card images at 556 B.P.I., even parity, on seven track tape. We will supply algorithms at a density of 800 B.P.I. if requested. Cards for algorithms are sequenced starting at 10 and incremented by 10. The sequence number is right justified in column 80. Although we will make every attempt to insure that the algorithm conforms to the description printed here, we cannot guarantee it, nor can we guarantee that the algorithm is correct.—L.D.F. and A.K.C.*

to the form

$$\begin{bmatrix} U_3 & V_3 \\ 0 & W_3 \end{bmatrix} \begin{bmatrix} C_3 \\ TB_3C_4 \end{bmatrix} = \begin{bmatrix} A_3 \\ D_3 \end{bmatrix}$$

Continuing yields U_K, V_K, A_K for $K = 2, 3, \ldots, N-1$. Then the system

$$\begin{bmatrix} INTCON_{N-1}TB_{N-1} \\ W_{N-1}TB_{N-1} \\ INTCON_N \\ ENDCON_N \end{bmatrix} \begin{bmatrix} \\ C_N \\ \\ \end{bmatrix} = \begin{bmatrix} B_{N-1} \\ D_{N-1} \\ B_N \\ D_N \end{bmatrix}$$

is solved for C_N using *TRISYS* and back substitution. In the backward march $C_{N-1}, C_{N-2}, \ldots, C_2$ are obtained successively from the stored arrays U_K, V_K, A_K via the recursion $U_KC_K = -V_KTB_K \cdot C_{K+1} + A_K$, where TB_K is another $2M$-Taylor matrix. Since U_K is upper triangular, to determine C_K we perform a matrix multiplication and a back substitution. Finally we set the first M components of C_1 equal to the first M components of TB_1C_2, and the last M components to zero.

The organization of *GVAL* is very simple. First a simple search is performed to determine the integer *KNOT* such that $X_{KNOT-1} < T \leq X_{KNOT}$. Then Horner's scheme is used to evaluate the $(ID\text{-}1)$-th derivative of the polynomial P_{KNOT}.

3. Numerical Experience. Table I below shows the results of using *GSF* to compute a cubic spline interpolating simple data and of using *GVAL* to evaluate it (and its derivatives) at various points. The table should be of use in verifying that the subroutines are operating correctly on the reader's machine. The data in Table I is taken from Greville [3, p. 20].

Table II below shows the results of using *GSF* and *GVAL* on simple, Hermite, and Hermite-Birkhoff interpolation problems. For comparison, we give the maximum interpolation error,

$$\max_{1 \leq j \leq N} \max_{1 \leq i \leq z_j} | Y_{ij} - s^{(IM_{ij}-1)}(x_j) |,$$

the root mean square error

$$\left(\sum_{j=1}^{N} \sum_{i=1}^{z_j} [Y_{ij} - s^{(IM_{ij}-1)}(x_j)]^2 \Big/ \sum_{j=1}^{N} z_j \right)^{\frac{1}{2}},$$

and the relative central processing times for each interpolation problem.

Tables I and II were computed on the CDC 6600 at The University of Texas, Austin. In addition to these examples, we tested the subroutines on a wide variety of simple, Hermite, and HB interpolation problems for $1 \leq M \leq 10$, $2 \leq N \leq 100$. We tested data from standard functions as well as random data with equally spaced and unequally spaced knots with knot mesh ratios ($\sigma = \max (x_{j+1} - x_j)/\min (x_{j+1} - x_j)$) up to $\sigma = 10^4$. The results were comparable in accuracy with the procedures in [8] for computing simple interpolating splines and the subroutines in [2] for computing g-splines. For small $M(M = 2,3)$ *GSF* and *GVAL* are as fast or faster than these other algorithms; for larger M the reverse is usually true.

4. Discussion. The subroutines presented below can be applied to compute g-splines interpolating HB-data whenever the HB-interpolation problem is M-poised. The question of when an HB problem is M-poised is a difficult one, and has been the subject of intensive research recently. For a survey of results, see Karlin/Karon [6]. An obvious necessary condition for M-poisedness is that $\sum_{j=1}^{N} z_j \geq M$. For Hermite interpolation problems ($IM_{1,j} = 1, \ldots, IM_{z_{j},j} = z_j - 1$), this is also sufficient. For simple interpolation ($z_j \equiv 1$, all j), this reduces to $N \geq M$. For nonpoised HB-interpolation problems, the subroutines may or may not produce g-splines interpolating the data. Thus the algorithm cannot be used as a test for M-poisedness.

There are a large variety of algorithms in the literature for computing splines interpolating simple data (cf. [7, 10] and references therein). In this special case the subroutines given here can be simplified (see Eidson [1]). There are few practical schemes for computing g-splines (see [2, 5] and references therein). The only other subroutines we know of for g-splines are those in [2] based on local support bases. The algorithms underlying the subroutines given here are valid also for Lg-splines, see [9], and for *EHB*-data (see [4, 9]). We hope to prepare subroutines for the more general case.

Table I. Cubic Spline Interpolating Simple Data

DATA					
($M = 2, N = 9$)		Values of the spline			
x_j	y_j	t	$s(t)$	$s'(t)$	$s''(t)$
266.8	1250	273.16	1346.2	15.076	−.02575
283.5	1500	303.16	1782.9	14.614	.03996
300.9	1750	323.16	2073.1	13.992	−.06902
318.0	2000	373.16	2706.4	11.665	−.03218
355.9	2500	423.16	3254.7	10.357	−.02148
399.2	3000	473.16	3749.0	9.481	−.01359
500.1	4000	523.16	4209.3	8.998	−.00574
555.7	4500	573.16	4655.2	8.884	−.00047
612.0	5000				

Table II. *G*-spline Interpolants for Various Types of Data

	Input data						Results		
Type of data	N	x_j $j = 1(1)N$	z_j $j = 1(1)N$	IM_{ij} $j = 1(1)N$	Y_{ij}	M	Maximum interpolation error	RMS interpolation error	Time of computation (sec)
Simple	50	$j/32$	1	$1, i = 1$	$\sin x_j, i = 1$	2	7.1(−15)	2.7(−15)	.260
						3	7.1(−15)	4.5(−15)	.608
						4	1.1(−14)	6.0(−15)	1.198
Hermite	10	$\exp(j/5)$	2	$\begin{cases}1, i = 1\\2, i = 2\end{cases}$	$\sin x_j, i = 1$ $\cos x_j, i = 2$	2	2.0(−14)	6.6(−15)	.040
						3	1.1(−14)	5.4(−15)	.095
						4	1.4(−14)	1.4(−09)	.200
Hermite-Birkhoff	40	$j/10$	3	$\begin{cases}1, i = 1\\3, i = 2\\5, i = 3\end{cases}$	e^{x_j}, $i = 1, 2, 3$	5	8.8(−09)	1.4(−09)	1.543

Acknowledgments. We wish to thank the referees for their extremely thorough consideration of our paper, and for several helpful suggestions.

References

1. Eidson, Harold D. Computation of interpolating splines via a factorization method. CNA report, to appear.

2. Eidson, Harold D., and Schumaker, L.L. Computation of g-splines via local bases. CNA report, Center for Numerical Analysis, U. of Texas, Austin, 1972, to appear.

3. Greville, T.N.E. *Data fitting by spline functions.* MRC report 893, U. of Wisconsin, 1968.

4. Jerome, J.W., and Schumaker, L.L. *On Lg-splines, J. Approx. Th. 2* (1969), 29-49.

5. Jerome, J.W., and Schumaker, L.L. Local bases and computation of g-splines. *Methoden und Verfahren der Mathematische Physik 5* (1971), 171-199.

6. Karlin, S., and Karon, J.M. On Hermite-Birkhoff interpolation. *J. Approx. Th. 6* (1972), 90-115.

7. Lyche, T., and Schumaker, L.L. Computation of smoothing and interpolating natural splines via local bases. *SIAM J. Numer. Anal. 10* (1937), 1027-1038.

8. Lyche, T., and Schumaker, L.L. ALGOL procedures for computing smoothing and interpolating natural splines. *Comm. ACM 17*, 8 (Aug. 1974), 465-469.

9. Munteanu, M.J., and Schumaker, L.L. On a method of Carasso and Laurent for constructing interpolating splines. *Math. Comp. 27* (1973), 317-325.

10. Schumaker, L.L. Some algorithms for the computation of interpolating and approximating spline functions. In *Theory and Application of Spline Functions*, Academic Press, New York, 1968, pp. 87–102.

Algorithm

```
      SUBROUTINE GSF(N, M, X, Y, Z, IM, C, IDET)
C INPUT   N,M,X,Z,IM,Y--
C N IS A POSITIVE INTEGER GIVING THE NUMBER OF KNOTS
C M IS A POSITIVE INTEGER DETERMINING THE DEGREE
C 2*M-1 OF THE SPLINE
C X IS AN ARRAY OF REAL NUMBERS WITH
C     X(1).LT.X(2).LT.....LT.X(N)
C Z IS AN ARRAY OF INTEGERS SUCH THAT
C     Ø.LT.Z(I).LE.M, I=1,2,...,N
C IM IS AN INTEGER ARRAY WITH
C     1.LE.M(1,J).LT.....LT.IM(Z(J),J).LE.M,
C J=1,2,...,N. Y IS AN ARRAY OF REAL NUMBERS
C OUTPUT  C,IDET--
C THE COLUMN VECTOR C(1,J),...,C(2*M,J) CONTAINS THE
C COEFFICIENTS OF THE SPLINE IN THE INTERVAL X(J-1)
C TO X(J). IDET IS SET TO ZERO IF A SINGULAR SYSTEM
C IS ENCOUNTERED OTHERWISE, IDET IS 1.
C THE SUBROUTINE  GSF(N,M,X,Y,Z,IM,C,IDET)  COMPUTES
C THE COEFFICIENTS OF THE INTERPOLATING G-SPLINE. THE
C PARAMETERS N,M,X,Y,Z, AND  IM  ARE INPUT. N  AND  M
C ARE THE POSITIVE INTEGERS OF SECTION 1  WHICH GIVE
C THE NUMBER OF X S AND DETERMINE THE DEGREE OF THE
C SPLINE, RESPECTIVELY. X  MUST BE AN ARRAY OF REAL
C NUMBERS WITH X(1).LT.X(2).LT.....LT.X(N)  AND Z  IS
C AN ARRAY OF POSITIVE INTEGERS NONE OF WHICH SHOULD
C EXCEED M. X CONTAINS THE POINTS WHERE HB-DATA IS
C PRESCRIBED AND  Z DESCRIBES THE NUMBER OF PIECES OF
C DATA AT EACH SUCH POINT. IM  IS AN INTEGER ARRAY
C WITH
C  1.LE.IM(1,J).LT.IM(2,J).LT.....LT.IM(Z(J),L).LE.M,
C               J=1,2,...,N.
C THE J TH COLUMN OF  IM  IS A LIST OF WHICH
C DERIVATIVES (SHIFTED UP BY 1) ARE SPECIFIED AT
C X(J).  THE DATA FOR THE HB-INTERPOLATION PROBLEM IS
C ENTERED IN THE ARRAY  Y. Y(I,J) SHOULD CONTAIN THE
C VALUE ASSIGNED TO THE IM(I,J)-1 ST DERIVATIVE OF
C THE SPLINE EVALUATED AT  X(J). THE PARAMETERS  C
C AND  IDET  ARE OUTPUT OF GSF. AFTER EXECUTION, THE
C ARRAY C WILL CONTAIN THE COEFFICIENTS OF THE
C SPLINE. IN PARTICULAR, THE COLUMN VECTOR
C C(1,J),...,C(2*M,J) CONTAINS THE COEFFICIENTS OF
C THE POLYNOMIALS P(J)  DESCRIBED BY EQUATIONS (6),
C J=1,2,...,N+1. SUBROUTINE GSF CALLS ON SUBROUTINE
C INTCON,SMOCON, AND TRISYS WHICH MUST BE LOADED WITH
C THE MAIN PROGRAM.
      INTEGER Z, ZK
      DIMENSION X(1ØØ), Y(4,1ØØ), Z(1ØØ), IM(4,1ØØ),
     * C(8,1ØØ)
      DIMENSION D(12,17), UV(8,17,1ØØ)
      DOUBLE PRECISION SUM
C INITIALIZE CONSTANTS
      IDET = 1
      M2 = 2*M
      M2P1 = M2 + 1
      M2M1 = M2 - 1
      M4 = 4*M
      M4P1 = M4 + 1
      NM1 = N - 1
C GENERATE FACTORIALS FOR TAYLOR MATRIX
      C(1,1) = 1.Ø
      DO 1Ø J=2,M
        JM1 = J - 1
        C(1,J) = FLOAT(JM1)*C(1,JM1)
   1Ø CONTINUE
C BEGIN FORWARD MARCH
      ZK = Z(1)
      MMZ = M - ZK
      M2MZ = M2 - ZK
C SET UP INTERPOLATION MATRIX AT X(1)
      CALL INTCON(1, ZK, M2, IM, D)
C SET END CONDITIONS AT X(1)
      IF (MMZ.NE.Ø) CALL SMOCON(-1, ZK, M, M2, M4P1,
     * IM, D)
C BEGIN K LOOP
      DO 25Ø K=2,N
        KM1 = K - 1
        LZK = ZK
        ZK = Z(K)
        LMMZ = MMZ
        MMZ = M - ZK
        M2MZ = M2 - ZK
        M3MZ = M2MZ + M
        H = X(KM1) - X(K)
C TAYLOR MATRIX RIGHT MULTIPLICATION
        IROW = M2MZ
        IF (K.EQ.N) IROW = M
        DO 2Ø I=1,M
          MU = IROW + I
          D(MU,1) = D(I,1)
   2Ø   CONTINUE
        D(1,M2P1) = 1.Ø
        DO 9Ø I=2,M2
          IM1 = I - 1
          DO 4Ø J=1,IM1
            DO 3Ø II=1,M
              D(II,J) = D(II,J)*H
   3Ø       CONTINUE
   4Ø     CONTINUE
          D(I,M2P1) = 1.Ø
          IF (2.GT.IM1) GO TO 6Ø
          T = D(1,M2P1)
          DO 5Ø II=2,IM1
            V = D(II,M2P1) + T
            T = D(II,M2P1)
            D(II,M2P1) = V
   5Ø     CONTINUE
   6Ø     DO 8Ø J=1,M
            SUM = Ø.Ø
            DO 7Ø II=1,I
              SUM = SUM + D(J,II)*D(II,M2P1)
   7Ø       CONTINUE
            MU = IROW + J
            D(MU,I) = SUM
   8Ø     CONTINUE
   9Ø   CONTINUE
C ON LAST STEP JUMP TO SET INTERPOLATION CONDITIONS
        IF (K.EQ.N) GO TO 24Ø
C SET UP SMOOTHING MATRIX AT X(K)
        CALL SMOCON(K, ZK, M, M2, M4P1, IM, D)
        DO 11Ø I=1,M
          DO 1ØØ J=M2P1,M4
            MU = M2MZ + I
            D(MU,J) = Ø.Ø
  1ØØ     CONTINUE
  11Ø   CONTINUE
C ADJUST RHS OF SYSTEM TO CORRESPOND WITH DIFFERENT
C Z(K)
        IF (LMMZ.EQ.Ø) GO TO 16Ø
        IF (LZK-ZK) 13Ø, 13Ø, 12Ø
  12Ø   II = M2 + LMMZ + 1
        JJ = -1
        III = M3MZ + 1
        GO TO 14Ø
  13Ø   II = M2
        JJ = +1
        III = M2MZ + LZK
  14Ø   DO 15Ø I=1,LMMZ
          MU = III + I*JJ
          NU = II + I*JJ
          D(MU,M4P1) = D(NU,M4P1)
  15Ø   CONTINUE
C FILL IN INTERPOLATION DATA
  16Ø   DO 17Ø I=1,LZK
          J = IM(I,KM1)
          MU = M2MZ + I
          D(MU,M4P1) = Y(I,KM1)/C(1,J)
  17Ø   CONTINUE
C TRIANGULARIZE SYSTEM AT Z(K)
        CALL TRISYS(D, M4P1, M3MZ, M2, IDET)
        IF (IDET) 19Ø, 18Ø, 19Ø
  18Ø   RETURN
C FILL UV MATRIX
  19Ø   DO 21Ø I=1,M2
          DO 2ØØ J=1,M4P1
            UV(I,J,K) = D(I,J)
  2ØØ     CONTINUE
  21Ø   CONTINUE
C COUPLE M-Z(K) ROWS WITH INTERP CONDITIONS AT NEXT
C STEP
        IF (MMZ.EQ.Ø) GO TO 24Ø
        DO 23Ø I=1,MMZ
          DO 22Ø J=1,M2
            LAMDA = ZK + I
            MU = M2 + I
            NU = M2 + J
            D(LAMDA,J) = D(MU,NU)
  22Ø     CONTINUE
  23Ø   CONTINUE
C SET UP INTERPOLATION MATRIX AT X(K)
  24Ø   CALL INTCON(K, ZK, M2, IM, D)
  25Ø CONTINUE
C END OF K LOOP
C SET END CONDITIONS AT X(N)
      IF (MMZ.NE.Ø) CALL SMOCON(-N, ZK, M, M2, M2P1,
     * IM, D)
C FILL IN INTERPOLATION DATA AT X(N-1)
      DO 26Ø I=1,LZK
        J = IM(I,NM1)
        MU = M + I
        D(MU,M2P1) = Y(I,NM1)/C(1,J)
  26Ø CONTINUE
C ADJUST RHS TO CORRESPOND WITH Z(N) DATA
      IF (LMMZ.EQ.Ø) GO TO 28Ø
      DO 27Ø I=1,LMMZ
        MU = M + LZK + I
        NU = M2 + I
        D(MU,M2P1) = D(NU,M4P1)
  27Ø CONTINUE
C FILL INTERPOLATION DATA AT X(N)
  28Ø DO 29Ø I=1,ZK
        J = IM(I,N)
        D(I,M2P1) = Y(I,N)/C(1,J)
  29Ø CONTINUE
C TRIANGULARIZE MATRIX SYSTEM AT X(N)
```

```
      CALL TRISYS(D, M2P1, M2, M2, IDET)
      IF (IDET.EQ.0) RETURN
C BACK SOLVE FOR C(N)
      I = M2P1
      DO 320 II=1,M2
        IP1 = I
        I = I - 1
        SUM = 0.0
        IF (IP1.GT.M2) GO TO 310
        DO 300 J=IP1,M2
          SUM = SUM + D(I,J)*C(J,N)
  300   CONTINUE
  310   V = -SUM + D(I,M2P1)
        C(I,N) = V/D(I,I)
  320 CONTINUE
C END FORWARD MARCH
C BEGIN BACKWARD MARCH
      K = N
C BEGIN KB LOOP
      DO 430 KB=2,N
        KP1 = K
        K = K - 1
        ZK = Z(K)
        H = X(K) - X(KP1)
C TAYLOR MATRIX LEFT MULTIPLICATION
        DO 330 I=1,M2
          D(1,I) = 1.0
  330   CONTINUE
        D(M2,M4P1) = C(M2,KP1)
        DO 370 I=1,M2M1
          IP1 = I + 1
          T = C(M2,KP1)*D(1,M2)
          M2MI = M2 - I
          DO 340 II=1,M2MI
            J = M2 - II
            T = T*H + C(J,KP1)*D(1,J)
  340     CONTINUE
          D(I,M4P1) = T
          T = 1.0
          IF (IP1.GT.M2M1) GO TO 360
          DO 350 II=IP1,M2M1
            D(1,1) = D(1,II)
            D(1,II) = T
            T = D(1,1) + D(1,II)
  350     CONTINUE
  360     D(1,M2) = T
  370   CONTINUE
C IF K = 1 JUMP OUT TO DETERMINE C(1)
        IF (KB.EQ.N) GO TO 440
        DO 390 I=1,M2
C SET UP RHS OF SYSTEM FOR C(K)
          SUM = 0.
          DO 380 J=1,M2
            MU = M2 + J
            SUM = SUM + UV(I,MU,K)*D(J,M4P1)
  380     CONTINUE
          UV(I,M4P1,K) = -SUM + UV(I,M4P1,K)
  390   CONTINUE
C BACK SOLVE FOR COEFFS C(K) USING TRIANGULAR PART OF
C UV(K)
        I = M2P1
        DO 420 II=1,M2
          IP1 = I
          I = I - 1
          SUM = 0.0
          IF (IP1.GT.M2) GO TO 410
          DO 400 J=IP1,M2
            SUM = SUM + UV(I,J,K)*C(J,K)
  400     CONTINUE
  410     V = -SUM + UV(I,M4P1,K)
          C(I,K) = V/UV(I,I,K)
  420   CONTINUE
  430 CONTINUE
C END K3 LOOP
C SET COEFFICIENTS C(1)
  440 DO 450 I=1,M
        MU = M + I
        C(MU,K) = 0.0
        C(I,K) = D(I,M4P1)
  450 CONTINUE
C END BACKWARD MARCH
      RETURN
      END
      SUBROUTINE INTCON(K, ZK, M2, IM, D)
C FILLS INTERPOLATION MATRIX AT X(K) USING
C INFORMATION OBTAINED FROM ARRAYS  Z(K) AND  IM(I,K)
      INTEGER ZK
      DIMENSION D(12,17), IM(4,100)
      DO 20 I=1,ZK
        DO 10 J=1,M2
          D(I,J) = 0.0
   10   CONTINUE
        II = IM(I,K)
        D(I,II) = 1.0
   20 CONTINUE
      RETURN
      END
      SUBROUTINE SMOCON(KK, ZK, M, M2, ICOL, IM, D)
C FILLS SMOOTHING MATRIX AT KNOTS 2 THROUGH N-1
C AND THE END CONDITIONS AT K = 1,N
      INTEGER ZK
      DIMENSION D(12,17), IM(4,100)
C IF KK IS NEGATIVE THEN SET END CONDITIONS
      K = IABS(KK)
      IF (KK.LT.0) GO TO 140
C SMOOTHING FIRST M DERIVATIVES
      DO 20 I=1,M
        DO 10 J=1,M2
          DUM = 0.0
          IF (I.EQ.J) DUM = 1.0
          D(I,J) = DUM
   10   CONTINUE
   20 CONTINUE
      IROW = M
      IDUP = 1
C SMOOTHING HIGHER DERIVATIVES
   30 IF (ZK.GE.M) GO TO 80
      J = M
      I = ZK
   40 IF (IM(I,K)-J) 60, 50, 60
   50 J = J - 1
      I = I - 1
      IF (I.LT.1) I = 1
      IF (J) 80, 80, 40
   60 IROW = IROW + 1
      DO 70 II=1,M2
        D(IROW,II) = 0.0
   70 CONTINUE
      J = J - 1
      MU = M2 - J
      D(IROW,MU) = 1.0
      IF (J) 80, 80, 40
   80 GO TO (90, 120), IDUP
   90 M2MZ = M2 - ZK
      DO 110 I=1,M2MZ
        D(I,ICOL) = 0.0
        DO 100 J=1,M2
          MU = M2 + J
          D(I,MU) = -D(I,J)
  100   CONTINUE
  110 CONTINUE
      RETURN
  120 MMZ = M - ZK
      DO 130 I=1,MMZ
        MU = MM + I
        D(MU,ICOL) = 0.0
  130 CONTINUE
      RETURN
C SET END CONDITIONS
  140 IROW = ZK
      IDUP = 2
      MM = ZK
      IF (K.EQ.1) MM = M2
      GO TO 30
      END
      SUBROUTINE TRISYS(D, N, L, M2, IDET)
C TRIANGULARIZATION OF NON-SQUARE MATRIX USING LU
C DECOMPOSITION WITH PIVOTING
      DIMENSION D(12,17)
      DOUBLE PRECISION SUM
      IDET = 1
      DO 150 K=1,M2
        KP1 = K + 1
        KM1 = K - 1
        PIVOT = 0.0
        DO 40 I=K,L
          IF (KM1.EQ.0) GO TO 20
          SUM = 0.0
          DO 10 J=1,KM1
            SUM = SUM + D(I,J)*D(J,K)
   10     CONTINUE
          D(I,K) = -SUM + D(I,K)
   20     T = ABS(D(I,K))
          IF (T-PIVOT) 40, 40, 30
   30     PIVOT = T
          IPIV = I
   40   CONTINUE
        IF (PIVOT) 60, 50, 60
   50   IDET = 0
        RETURN
   60   IF (IPIV-K) 70, 90, 70
   70   DO 80 J=1,N
          T = D(K,J)
          D(K,J) = D(IPIV,J)
          D(IPIV,J) = T
   80   CONTINUE
   90   T = D(K,K)
        IF (KP1-L) 100, 100, 120
  100   DO 110 I=KP1,L
          D(I,K) = D(I,K)/T
  110   CONTINUE
  120   IF (KM1.EQ.0 .OR. KP1.GT.N) GO TO 150
        DO 140 J=KP1,N
          SUM = 0.0
          DO 130 I=1,KM1
            SUM = SUM + D(K,I)*D(I,J)
  130     CONTINUE
          D(K,J) = -SUM + D(K,J)
  140   CONTINUE
  150 CONTINUE
      LAST = L - M2
      IF (LAST.EQ.0) GO TO 190
      K = M2
      M2P1 = M2 + 1
      DO 18Ø I=1,LAST
        K = K + 1
        DO 17Ø J=M2P1,N
          SUM = Ø.Ø
          DO 16Ø II=1,M2
            SUM = SUM + D(K,II)*D(II,J)
  16Ø     CONTINUE
          D(K,J) = -SUM + D(K,J)
  17Ø   CONTINUE
  18Ø CONTINUE
  19Ø RETURN
      END
      FUNCTION GVAL(T, ID, N, M, X, C)
C INPUT T,ID,N,M,X,C
C THE PARAMETERS N,M,X,C ARE AS IN GSF  AND
C COMPLETELY DESCRIBE THE G-SPLINE.
C T IS A REAL NUMBER AND ID A POSITIVE INTEGER.
C GVAL PRODUCES THE ID-1 ST DERIVATIVE OF THE SPLINE
C AT T. GVAL AUTOMATICALLY PRODUCES Ø IF ID.GT.M*2
      DIMENSION X(1ØØ), C(8,1ØØ), S(8)
      IORD = 2*M
      IF (ID.GT.IORD) GO TO 13Ø
C BINARY SEARCH FOR KNOT SUCH THAT
C X(KNOT-1).LT.T(KNOT)
      KNOT = 1
      IF (T-X(KNOT)) 7Ø, 7Ø, 1Ø
   1Ø KNOT = N
      IF (T-X(KNOT)) 2Ø, 6Ø, 6Ø
   2Ø KUP = N
      KLO = 1
   3Ø IF ((KUP-KLO).EQ.1) GO TO 7Ø
      KNOT = (KUP+KLO)/2
      IF (T-X(KNOT)) 5Ø, 7Ø, 4Ø
   4Ø KLO = KNOT
      KNOT = KUP
      GO TO 3Ø
   5Ø KUP = KNOT
      GO TO 3Ø
C EVALUATION OF THE SPLINE
   6Ø IORD = M
      IF (ID.GT.IORD) GO TO 13Ø
   7Ø Y = T - X(KNOT)
      IORD1 = IORD + 1
C SET UP SPLINE COEFFICIENTS
      DO 8Ø I=1,IORD
        MU = IORD1 - I
        S(I) = C(MU,KNOT)
   8Ø CONTINUE
C HORNERS SCHEME
      DO 1ØØ K=1,ID
        IORD = IORD1 - K
        DO 9Ø I=2,IORD
          S(I) = S(I-1)*Y + S(I)
   9Ø   CONTINUE
  1ØØ CONTINUE
      FACT = 1.Ø
      IF (ID.EQ.1) GO TO 12Ø
      IDM1 = ID - 1
      DO 11Ø I=1,IDM1
        FACT = FACT*FLOAT(I)
  11Ø CONTINUE
  12Ø GVAL = S(IORD)*FACT
      RETURN
  13Ø GVAL = Ø.Ø
      RETURN
      END
```

Algorithm 486

Numerical Inversion of Laplace Transform [D5]

Francoise Veillon [Recd. 26 Sept. 1972]
Mathematiques Appliquees Informatique, Universite de Grenoble, B.P. 53, Cedex 53, 38 Grenoble-Gare, France

Key Words and Phrases: Laplacε transform inversion
CR Categories: 5.16
Language: Algol

Description

This work forms part of a thesis presented in Grenoble in March 1972. Improvements made to the Dubner and Abate algorithm for numerical inversion of the Laplace transform [1] have led to results which compare favorably with theirs and those of Bellmann [2], and Stehfest [3]. The Dubner method leads to the approximation formula:

$$f(t) = 2e^{at}/T[\tfrac{1}{2}Re\{F(a)\} + \sum_{k=1}^{\infty} Re\{F(a + ik\pi/T)\}\cos(k\pi t/T)], \quad (1)$$

where $F(s)$ is the Laplace transform of $f(t)$ and a is positive and greater than the real parts of the singularities of $f(t)$.

Definition of the calling parameters. Assume that $f(t)$ is a function which has real values and that $F(s)$ is its Laplace transform. The procedure *laplaceinverse* calculates, for a programmer-chosen set of values of t, the corresponding values of $f(t)$. The parameters are as follows:

rf1f is a real procedure with two parameters which are, respectively, real part of s and imaginary part of s. Its value is the real part of $F(s)$.

ntf is the number of values of t for which we want to calculate $f(t)$.

tf is a one-dimensional array, the bounds of which are 1 and *ntf*. It contains the values of t.

naf is the number of values taken by the parameter a (see eq. (1)). In the following examples, *naf* is equal to 5.

af is a one-dimensional array, the bounds of which are 0 and *naf* − 1. At the time of the call this array must contain the values of a. In the following examples, these values are, in order: 1.15, 1.20, 1.25, 1.30, 1.35. These values have been experimentally chosen as the best over the whole set of functions that have been calculated (approximately 30, as different as possible), but they are not the best for each particular function.

iterf is 1/8 of the number of terms considered in the infinite sum of the approximation formula, eq. (1). In the example, *iterf* is equal to 8.

resultatf is a one-dimensional array, the bounds of which are 1 and *ntf*. At the end of the procedure it contains the *ntf* values of $f(t)$.

ecri is a procedure with one real parameter (time). It must print the value of the parameter, an error message (see later) and be written with local conventions.

A few examples of functions which have been calculated by means of this procedure, and then compared with other methods are given in Tables I and II.

Outline of the method. The program first evaluates $f(t)$ using eq. (1) for *naf* values of a. The sum in eq. (1) is evaluated in *iterf* groups of eight terms by the ϵ-algorithm (procedure *epsalgor*) which corresponds to an iteration of the Aitken Δ^2 process. This accelerates the convergence of the sum. The grouping of terms by eight results in either using fewer calculations for the same results or, for the same volume of calculations, using more terms in the sequence of the partial sum, and consequently obtaining better precision. It also smooths this sequence. If *iterf* is equal to eight, this leads to the use of 64 terms in the sum. That is satisfactory to proceed with the ϵ-algorithm.

If *naf* is different from one (and greater than three, otherwise the spline approximation is meaningless), then the program fits to the *naf* estimates of $f(t)$ a cubic spline $S(a)$ whose second derivatives vanish at the endpoints. The spline representation employs second derivatives, and the system of linear equations satisfied by these derivatives is solved using the double-sweep method.

We want an a such that $f(t)$ is the least dependent on a. The program then attempts to find an a for which $S'(a) = 0$. If no such a exists, then the program attempts to find one for which $S'(a)$ is

Table I.

t	$f(t) = \frac{1}{\sqrt{t\pi}}$ exact	$f(t)$ Stehfest	$f(t)$ Dubner*	$f(t)$ *Laplace-inverse*
1	0.56419	0.56555	0.73172	0.56419
2	0.39894	0.39912	0.40035	0.39894
3	0.32574	0.32655	0.26343	0.32573
4	0.28209	0.28278	0.28286	0.28209
5	0.25231	0.25174	0.29365	0.25231
6	0.23033	0.22989	0.22901	0.23033
7	0.21324	0.21322	0.18062	0.21324
8	0.19947	0.19956	0.20112	0.19947
9	0.18806	0.18814	0.21609	0.18806
10	0.17841	0.17796	0.17650	0.17841

* The Dubner method has been performed with $aT = 10$ and 500 terms for the sum.

Table II.

t	$f(t) = e^{-t/2}$ exact	$f(t)$ Bellmann	$f(t)$ *Laplace-inverse*
4.140186	0.126174	0.120527	0.126174
2.501126	0.286329	0.288195	0.286329
1.643438	0.439675	0.439084	0.439675
1.085084	0.581269	0.581308	0.581269
0.693147	0.707107	0.707318	0.707107
0.412298	0.813712	0.813401	0.813712
0.214821	0.898157	0.898482	0.898158
0.085541	0.958131	0.957847	0.958135
0.016048	0.991008	0.992205	0.992015

minimum. Using this a, the program evaluates $f(t)$ from eq. (1) (unless the chosen a is among the original set of values of a) to obtain the final approximation. As it is not possible to calculate the best a for an unknown function, the values of a have been experimentally chosen so as to give the best global result over a set of about thirty known functions, as different as possible.

Although it is very rare, a zero divide may occur in procedure *epsalgor* because of the division between two terms which may become equal. Then the program calls the procedure *ecri* and jumps to the next value of t. The value of $f(t)$, which has not been evaluated because of this, will be zero.

It must be said that the algorithm can be applied only to functions whose inverses are expected to be reasonably smooth.

Implementation. This program has been run on an IBM/360 computer, using compiler F under Operating System, version 18.6. The computing time per t-value, irrespective of the time needed to evaluate $Re(F(s))$, is 0.7 sec. The number of calls of procedure *rf1f* is less than or equal to $ntf(naf - 1)\ (8 \times iterf + 1)$. The object module size is about 15K bytes. The effective memory occupied during the execution step is 66K bytes.

References

1. Dubner, H., and Abate, J. Numerical inversion of Laplace transforms and the Finite Fourier Transform. *J. ACM 15*, 1 (Jan. 1968), 115–123.

2. Bellmann, R., Kalaba, R., and Lockett, J. *Numerical Inversion of the Laplace Transform.* American Elsevier, New York, 1966.

3. Stehfest, H. Algorithm 368. Numerical inversion of Laplace transform. *Comm. ACM 13*, 1 (Jan. 1970), 47–49.

4. Veillon, F. Quelques méthodes nouvelles pour le calcul numérique de la transformée inverse de Laplace. Th. U. de Grenoble, Mar. 1972.

Algorithm

```
procedure laplaceinverse (rf1f,tf,ntf,af,naf,iterf,resultatf,ecri);
  real procedure rf1f; real array tf,af,resultatf; procedure ecri;
  integer iterf,ntf,naf;
begin
  procedure epsalgor (eps, neps, resuleps, teps);
  array eps, resuleps; integer neps; real teps; boolean bool;
  begin
    array epstamp[1: neps − 1]; integer i, j, k;
    for i := 1 step 1 until neps ÷ 2 do resuleps[i] := 0.0;
    for i := 1 step 1 until neps − 1 do
    begin
      if eps[i + 1] = eps[i] then
      begin ecri(teps); bool := true; go to fin; end;
      epstamp[i] := 1.0/(eps[i + 1] − eps[i])
    end;
    resuleps[1] := eps[neps];
    k := 2;
    for j := neps − 2 step −1 until 1 do
    begin
      for i := 1 step 1 until j do
      begin
        eps[i] := epstamp[i];
        if epstamp[i + 1] = epstamp[i] then
        begin ecri(tps); bool := true; go to fin; end;
        epstamp[i] := eps[i+1] + 1.0/(epstamp[i+1] −
        epstamp[i]);
      end;
      if (k÷2) × 2 = k then resuleps[(k÷2)+1] := epstamp[j];
      k := k + 1
    end;
fin:
  end epsalgor;
  procedure laplinv (rf1g,tg, iterg,ag,resultatg);
  real procedure rf1g; real tg,resultatg,ag;
  integer iterg;
  begin
  real somme; integer i,j;
  real array ftab[0:8×iterg], ep[1 : iterg], resulep[1 : iterg÷2];
  for i := 0 step 1 until 8 × iterg do
    ftab[i] := rf1g(ag,i×3.1415926536/(8.0×tg));
  somme := 0.0;
  for i := 1 step 1 until iterg do
  begin
    for j := 1 step 1 until 8 do
      somme := somme + ftab[j + 8 × (i−1)] × cos((j + 8 ×
      (i−1)) × 3.1415926536/8.0);
    ep[i] := somme
  end;
  epsalgor (ep,iterg,resulep,tg);
  if ¬ bool then
    resultatg := 2.0 × exp(ag×tg)/(8.0×tg) × (resulep
    [iterg÷2] + 0.5 × ftab[0]);
end laplinv;
procedure coefsplinetrois (n,x,y,m);
value n,x,y; integer n; array x,y,m;
begin
  integer i; array d[1 : n−1]; real a,b,c,e;
  for i := n−1 step −1 until 1 do
  begin
    a := x[i+1] − x[i]; b := x[i] − x[i−1];
    c := y[i+1] − y[i]; e := y[i] − y[i−1];
    if i = n − 1 then
    begin
      d[i] := (x[i+1] − x[i−1])/3.0; m[i] := c/a − e/b
    end
    else
    begin
      d[i] := (12×d(i+1)×(x[i+1]−x[i−1])−a×a)/(36×
      d[i+1]);
      m[i] := c/a − e/b − a × m[i+1]/(6.0 × d[i+1])
    end
  end;
  m[0] := m[n] := 0.0;
  for i := 1 step 1 until n − 1 do
    if i = 1 then m[i] := m[i]/d[i]
    else
    m[i] := (6×m[i]−(x[i]−x[i−1])×m[i−1])/(6 × d[i])
end coefsplinetrois;
boolean bool, bool1, bool2;
real delta, a1, b1, c1, zero, x1, x2, dzero, v, u;
integer i, j;
real array x, m, z[0:naf−1];
real array y[0:naf];
for i := 1 step 1 until ntf do
begin
  bool := false; resultatf[i] := 0.0;
  for j := 0 step 1 until naf − 1 do
  begin
    x[j] := af[j]/tf[i];
    laplinv (rf1f, tf[i], iterf, x[j], y[j]);
    if ¬ bool then resultatf[i] := y[j]
    else
    go to e;
  end;
  if naf ≠ 1 then
  begin
    coefsplinetrois (naf−1, x, y, m); u := 0.0;
    for j := 0 step 1 until naf−2 do
    begin
      a1 := (m[j+1] − m[j])/6.0/(x[j+1] − x[j]);
      b1 := (m[j] − 6.0 × a1 × x[j])/2.0;
      c1 := (y[j+1] − y[j])/(x[j+1] − x[j]) − a1 ×
        (x[j] × x[j] + x[j+1] × x[j+1] + x[j] ×
        x[j+1]) − b1 × (x[j] + x[j+1]);
      delta := b1 × b1 − 3.0 × a1 × c1;
      bool1 := false; bool2 := false;
      if delta ≥ 0.0 then
      begin
        if a1 = 0.0 then
        begin
          if b1 ≠ 0.0 then
          begin x2 := −c1/2.0/b1; bool2 := true end;
        end
        else
        begin
          x1 := (−b1 + sqrt(b1 × b1 − 3.0 × a1 × c1))/
            a1/3.0;
          x2 := (−b1 − sqrt(b1 × b1 − 3.0 × a1 × c1))/
            a1/3.0;
          bool1 := true; bool2 := true;
        end
      end;
      if bool1 then
      begin
        if (x[j] ≤ x1 ∧ x1 < x[j+1]) then u := x1
      end
      else
      if bool2 then
      begin
        if (x[j]) ≤ x2 ∧ x2 < x[j+1]) then u := x2
      end
    end;
    if ¬ bool1 ∧ ¬ bool2 ∨ u = 0.0 then
    for j := 0 step 1 until naf−2 do
    begin
      if j = 0 then z[j] := abs ((3.0 × a1 × x[j] + 2.0 ×
        b1) × x[j] + c1);
      z[j+1] := abs ((3.0 × a1 × x[j+1] + 2.0 × b1) ×
        x[j+1] + c1);
      if j = 0 then
        begin
          if z[j] < z[j+1] then
          begin u := x[j]; v := z[j] end
          else
          begin u := x[j+1]; v := z[j+1] end;
        end
        else
        if v > z[j+1] then
        begin v := z[j+1]; u := x[j+1] end;
        zero := −b1/3.0/a1;
        dzero := abs ((3.0 × a1 × zero + 2.0 × b1) ×
          zero + c1);
        if (x[j] ≤ zero ∧ zero < x[j+1] ∧ dzero < z[j]) then
        begin u := zero; v := dzero end
      end;
      j := 0;

      if u = x[j] then resultatf[i] := y[j]
      else
      if u < x[j+1] then
      begin
        laplinv (rf1f, tf[i], iterf, u, y[naf]);
        if ¬ bool then resultatf[i] := y[naf]
        else
        resultatf[i] := y[j];
      end
      else
      if j < naf−2 then
      begin j := j + 1; go to l end
      else
      if u = x[j+1] then resultatf[i] := y[j+1]
    end;
e:
  end
end laplaceinverse;
```

ACM Transactions on Mathematical Software, Vol. 2, No. 4, December 1976, Pages 395 396.

REMARK ON ALGORITHM 486

Numerical Inversion of Laplace Transform [D5]
[F. Veillon, *Comm. ACM 17*, 10 (Oct. 1974), 587–589]

Henk Koppelaar and Peter Molenaar [Recd 12 Feb. 1976 and 11 May 1976]
Department of Psychology, Division MPS, State University of Utrecht, Oudenoord 6, Utrecht, The Netherlands.

The following changes were made in the algorithm:

(1) Within the body of the procedure *epsalgor* the last call of *ecri* was changed to read: *ecri*(*teps*).

(2) Within the body of the procedure *coefsplinetrois* the assignment to *d*[*i*] was changed to read: *d*[*i*] := (12×*d*[*i*+1]×, etc.

(3) Tests show the increasing inaccuracy of the approximation by *Laplace-inverse* if *t* gets in the vicinity of zero. In fact if $t = 0$, overflow occurs at various places. The first spot where it occurs is after declaration of *coefsplinetrois* in the inner do-loop:

```
for j := 0 step 1 until naf−1 do begin
x[j] := af[j]/tf[i]
```

if *t* = *tf*[*i*] is zero for some *i*. In order to avoid this overflow, one may compute *Laplaceinverse* at $t \neq 0$ or insert in the algorithm the precaution: **if** *tf*[*i*] = 0 **then begin** *ecri* (*tf*[*i*]); **go to** *e* **end**:

```
for j := 0 step 1 until naf−1 do begin
if tf[i] = 0 then begin ecri(tf[i]);
go to e end; x[j] := af[j]/tf[i]
```

Though this precaution prevents overflow, it is appropriate to add a comment in the heading of *Laplaceinverse* concerning problems if $t = 0$. Also, in the description of the algorithm a warning against $t = 0$ is necessary.

(4) In the heading of the procedure *epsalgor* the declaration **boolean** *bool*; was erased.

With these modifications the algorithm *Laplaceinverse* was translated for the CDC-6500 using the Control Data Algol 3 compiler.

The program was used on the following five tests, computing the inverse of $F(s)$, $s = a + ib$, which is $f(t)$, while the program is supplied with Re $\{F(s)\}$:

Test	$F(s)$	$f(t)$	Re $\{F(s)\}$
a	$1/\sqrt{s}$	$1/\sqrt{(t\pi)}$	$\sqrt{[(a+p)/2]/p}$, $p = \sqrt{[(a^2+b^2)]}$
b	$1/(s+0.5)$	$\exp(-t/2)$	$(0.5+a)/((0.5+a)^2+b^2)$
c	$s/(s^2+1)^2$	$(t/2) \sin(t)$	$a(x^2+4b^2(1-b^2))/(x^2+4b^2(a^2))^2$, $x = a^2-b^2+1$
d	$1/(s^2+s+1)$	$(2/\sqrt{3})\exp(-t/2)\sin(t/(2/\sqrt{3}))$	$x/(x^2+y^2)$, $x = a^2-b^2+a+1$, $y = b(2a+1)$
e	$s^{-1}\exp(-25s)$	$U(t-25)$	$\exp(-25a)(a\times\cos(25b)-b\times\sin(25b))/$ (a^2+b^2)

Except for tests b and e the results were accurate to about four decimal places. For $t \approx 0.01$ the results for test b were accurate to about two decimal places, while test e showed accuracy to only one decimal place at $t \approx 25$.

ACM Transactions on Mathematical Software, Vol. 3, No. 1, March 1977. Page—111

REMARK ON ALGORITHM 486

Numerical Inversion of Laplace Transform [D5]
[Francoise Veillon, *Comm. ACM* 17, 10 (Oct. 1974), 587–589]

Francoise Veillon [Recd 21 April and 30 July 1976]
Mathématiques Appliquées Informatique, U.S.M.G. B.P. 53, 38041 Grenoble, France

A significant improvement in efficiency can be obtained by using call by value rather than call by name where appropriate. Thus the following three changes are suggested:

(1) **value** *tf, ntf, af, naf, iterf*;
inserted between the heading of the procedure *laplaceinverse* and its specifications.

(2) **value** *eps, neps, teps*;
inserted between the heading of the procedure *epsalgor* and its specifications.

(3) **value** *tg, iterg, ag*;
inserted between the heading of the procedure *laplinv* and its specifications.

As the procedures needed to evaluate $Re(F(s))$ and the true values of the results are the responsibility of the user, two kinds of tests have been performed:

(a) The modifications (1), (2), and (3) are included in the procedure *laplaceinverse.*

(b) Calls by value are also used in the user supplied function *rf1f*. (Call by value is used in only three of the seven true value functions because it is not worthwhile using it when the parameter is referred to only once.)

The computing times (in seconds) are given in Table I. They concern the calculation of ten t-values for seven functions; the last column, to the right, concerns the mean time for one t-value.

The programs were run on an IBM/360/67 computer, using an F compiler, under Operating System MVT, version 20.1/asp 2.6.

Table I

$t1$: Computing time needed to evaluate $Re(F(s))$.
$t2$: Computing time irrespective of the time needed to evaluate $Re(F(s))$.
$t3$: Computing time per t-value irrespective of the time needed to evaluate $Re(F(s))$.

	Full computing time	Computing time $t1$	Computing time $t2$	Computing time $t3$
No call by value	61.99	28.18	33.81	0.48
Call by value only in *laplace-inverse*	48.42	28.18	20.24	0.29
Call by value in *laplaceinverse* and user supplied function *rf1f*	41.16	20.92	20.24	0.29

Algorithm 487

Exact Cumulative Distribution of the Kolmogorov-Smirnov Statistic for Small Samples [S14]

John Pomeranz [Recd. 13 Mar. 1973]
Computer Sciences Department, Mathematical Sciences Building, Purdue University, West Lafayette, IN 47907*

Key Words and Phrases: Kolmogorov-Smirnov test, K-S statistic, goodness-of-fit testing
CR Categories: 8.1, 5.5
Language: Fortran

Description

The algorithm calculates the exact cumulative distribution of the two-sided Kolmogorov-Smirnov statistic for samples with few observations. The general problem for which the formula is needed is to assess the probability that a particular sample comes from a proposed distribution. The problem arises specifically in data sampling and in discrete system simulation. Typically, some finite number of observations are available, and some underlying distribution is being considered as characterizing the source of the observations.

The statistic used here simply measures the maximum deviation between the proposed distribution and the empirical distribution derived from the sample. Elementary rules for calculating this deviation can be found in, e.g. Knuth [4, p. 41], Brunk [2, p. 267], or Miller and Freund [5, p. 222]. Simply put, let $S_N(x)$ be the fraction of the N observations which are less than x. Let $F(x)$ be the proposed cumulative distribution of the source. Let

$$K_N = \sqrt{N} \times \max_x | S_N(x) - F(x) |.$$

Usually K_N is called a two-sided Kolmogorov-Smirnov statistic. Omitting the absolute value signs gives a one-sided statistic. For computational ease we let $D_N = K_N/\sqrt{N}$ be the observed deviation, unweighted by $\sqrt{N}$.

The inputs to the function are the sample size N and a critical value D. The function value is the exact probability $\Pr\{D_N < D\} = \Pr\{K_N < D\sqrt{N}\}$.

The formulas used in the function are obtained directly from Durbin [3, formulas (23) and (24)]. To validate the function, another was coded using matrices determined by Pomeranz [7], and the two were identical to eight decimal places. Then the function was used to generate Birnbaum's Table 1 [1, pp. 428–30] for $D = 1/N, 2/N, \ldots, J/N, J = \min\{N, 15\}, 1 \leq N \leq 100$. Eight entries differed by 10^{-5}, apparently from roundoff error [1, p. 440]. The final test was of Miller's Table 1 [6, pp. 113–15] of critical values in the extreme tail for $1 \leq N \leq 100$. (Miller's approximation is based on the one-sided statistic with doubled tail probabilities, which is accurate in the extreme tail.) Newton's method was used to determine the values of D, which yield cumulative probabilities of .8, .9, .95, .98 and .99, for each N. Miller's entries agreed within one in the fifth decimal place for probabilities other than .8 and within four in the fifth decimal place for the .8 probability. This supports Miller's claim [6, p. 120] and further allows the use of the column $\alpha = .10$ ($P = .80$) in his Table 1 when an error in D of 4×10^{-5} is acceptable. However, the two-sided statistic and the one-sided statistic [4, p. 44] are significantly different outside the tail. For example, with a sample size of 10, $\Pr\{K_{10} < .54\}$ is approximately .12, but at the same critical value for the one-sided statistic, the cumulative probability is .50.

Finally, using a CDC 6500, values were computed up to $N = 140$. The major limitation is the magnitude of the exponent required to represent N^N. Rearranging sums produced no changes.

References

1. Birnbaum, Z.W. Numerical tabulation of the distribution of Kolmogorov's statistic for finite sample size. *J. Amer. Stat. Assoc. 47*, 259 (Sept. 1952), 425–4].
2. Brunk, H.D. *An Introduction to Mathematical Statistics.* Ginn and Company, Lexington, Mass., 1960.
3. Durbin, J. The probability that the sample distribution function lies between two parallel straight lines. *Ann. Math. Statist. 39*, 2 (Apr. 1968), 398–411.
4. Knuth, Donald E. *The Art of Computer Programming Volume 2/Seminumerical Algorithms.* Addison-Wesley, Reading, Mass., 1969.
5. Miller, Irwin, and Freund, John E. *Probability and Statistics for Engineers.* Prentice-Hall, Englewood Cliffs, N.J., 1965.
6. Miller, Leslie H. Table of percentage points of Kolmogorov statistics. *J. Amer. Stat. Assoc. 5*, 273 (Mar. 1956), 111–21.
7. Pomeranz, John E. Exact values of the two-sided Kolmogorov-Smirnov cumulative distribution for finite sample size. Tech. Rep. 88, Computer Sciences Department, Purdue U., Feb. 1973.

Algorithm

```
      REAL FUNCTION PKS2(N, D)
      INTEGER N
C N IS THE SAMPLE SIZE USED.
      REAL D
C D IS THE MAXIMUM MAGNITUDE (OF THE DISCREPANCY
C BETWEEN THE EMPIRICAL AND PROPOSED DISTRIBUTIONS)
C IN EITHER THE POSITIVE OR NEGATIVE DIRECTION.
C PKS2 IS THE EXACT PROBABILITY OF OBTAINING A
C DEVIATION NO LARGER THAN D.
C THESE FORMULAS APPEAR AS (23) AND (24) IN
C J. DURBIN.  THE PROBABILITY THAT THE SAMPLE
C DISTRIBUTION FUNCTION LIES BETWEEN TWO PARALLEL
C STRAIGHT LINES. ANNALS OF MATHEMATICAL STATISTICS
C 39, 2(APRIL 1968),398-411.
      DOUBLE PRECISION Q(141), FACT(141), SUM, CI,
     * FT, FU, FV
      IF (N.EQ.1) GO TO 90
      FN = FLOAT(N)
      FND = FN*D
      NDT = IFIX(2.*FND)
      IF (NDT.LT.1) GO TO 1ØØ
      ND = IFIX(FND)
      NDD = MINØ(2*ND,N)
      NDP = ND + 1
      NDDP = NDD + 1
      FACT(1) = 1.
      CI = 1.
      DO 1Ø I=1,N
        FACT(I+1) = FACT(I)*CI
        CI = CI + 1.
      1Ø CONTINUE
         Q(1) = 1.
         IF (NDD.EQ.Ø) GO TO 5Ø
         CI = 1.
         DO 2Ø I=1,NDD
           Q(I+1) = CI**I/FACT(I+1)
           CI = CI + 1.
      2Ø CONTINUE
         IF (NDP.GT.N) GO TO 8Ø
```

* Present address: A.T. Kearney, Inc., 100 South Wacker Drive, Chicago, IL 60606.

```
      FV = FLOAT(NDP) - FND
      JMAX = IDINT(FV) + 1
      DO 40 I=NDP,NDD
         SUM = 0.
         FT = FND
         K = I
         FU = FV
         DO 30 J=1,JMAX
            SUM = SUM + FT**(J-2)/FACT(J)*FU**K/
     *       FACT(K+1)
            FT = FT + 1.
            FU = FU - 1.
            K = K - 1
   30    CONTINUE
         Q(I+1) = Q(I+1) - 2.*FND*SUM
         JMAX = JMAX + 1
         FV = FV + 1.
   40 CONTINUE
      IF (NDD.EQ.N) GO TO 80
   50 DO 70 I=NDDP,N
         SUM = 0.
         SIGN = 1.
         FT = 2.*FND
         DO 60 J=1,NDT
            FT = FT - 1.
            K = I - J + 1
            SUM = SUM + SIGN*FT**J/FACT(J+1)*Q(K)
            SIGN = -SIGN
   60    CONTINUE
         Q(I+1) = SUM
   70 CONTINUE
   80 PKS2 = Q(N+1)*FACT(N+1)/FN**N
      RETURN
   90 PKS2 = 2.*D - 1.
      RETURN
  100 PKS2 = 0.
      RETURN
      END
```

```
      SUBPOUTINE PRFAC
      DOUBLE PRECISION PF(4,40)
      DIMENSION DXA(4)
      COMMON DX, DXA, PF, J
      DATA I /1/
      DO 10 J=1,4
         IF (DXA(J).EQ.DX) RETURN
   10 CONTINUE
      J = I
      I = I + 1
      IF (I.EQ.5) I = 1
      DXA(J) = DX
      PF(J,1) = 1.
      DO 20 K=2,38
         PF(J,K) = (PF(J,K-1)*DX)/FLOAT(K-1)
   20 CONTINUE
      RETURN
      END
```

```
      FUNCTION CEIL(X)
      IF (X.GE.0.) GO TO 10
      I = -X
      CEIL = -I
      RETURN
   10 I = X + .99999999
      CEIL = I
      RETURN
      END
```

```
      FUNCTION PKS(N, EPS)
C CALCULATE THE CUMULATIVE DISTRIBUTION OF THE
C KOLMOGOROV-SMIRNOV STATISTIC USING THE FORMULAS OF
C JOHN POMERANZ. EXACT VALUES OF THE TWO-SIDED
C KOLMOGOROV- SMIRNOV CUMULATIVE DISTRIBUTION FOR
C FINITE SAMPLE SIZE. TECHNICAL REPORT NUMBER 88,
C COMPUTER SCIENCES DEPARTMENT, PURDUE UNIVERSITY,
C FEBRUARY 1973.
      DOUBLE PRECISION PF(4,40), U(40), V(40)
      DOUBLE PRECISION SUM
      DIMENSION DXA(4)
      COMMON DX, DXA, PF, L
      DATA MNP /40/
      FN = N
      RN = 1./FN
      K = EPS*FN + .00000001
      FK = K
      IF (ABS(FK-EPS*FN).GT..00000001) GO TO 10
      K = K - 1
      FK = K
   10 CONTINUE
      DEL = EPS - FK*RN
      XUP = RN - DEL
      XLO = DEL
      IF (ABS(XUP-XLO).LT..00000001) XUP = XLO
      XPREV = 0.
      DO 20 I=1,MNP
         U(I) = 0.
   20 CONTINUE
      U(K+1) = 1.
      IMIN = -K
   30 X = AMIN1(XUP,XLO)
      IF (X.GT..999999) X = 1.
      DX = X - XPREV
      JMIN = CEIL((X-EPS)*FN-.00000001)
      IF (ABS(FLOAT(JMIN)-(X-EPS)*FN).LT..00000001)
     * JMIN = JMIN + 1
      JMAX = (X+EPS)*FN + .00000001
      IF (ABS(FLOAT(JMAX)-(X+EPS)*FN).LT..00000001)
     * JMAX = JMAX - 1
      JMAX = JMAX - JMIN + 1
      CALL PRFAC
      DO 60 J=1,MNP
         SUM = 0.
         IF (J.GT.JMAX) GO TO 50
         I = 1
   40    IP = J - I + 1 + JMIN - IMIN

         SUM = SUM + U(I)*PF(L,IP)
         I = I + 1
         IF ((IMIN+I).LE.(JMIN+J)) GO TO 40
   50    V(J) = SUM
   60 CONTINUE
      DO 70 I=1,MNP
         U(I) = V(I)
   70 CONTINUE
      IMIN = JMIN
      XPREV = X
      IF (X.EQ.XUP) XUP = XUP + RN
      IF (X.EQ.XLO) XLO = XLO + RN
      IF (X.LT.1.) GO TO 30
      DO 80 I=1,N
         U(K+1) = U(K+1)*FLOAT(I)
   80 CONTINUE
      PKS = U(K+1)
      RETURN
      END
```

ACM Transactions on Mathematical Software, Vol. 2, No. 1, March 1976, Page 111

REMARK ON ALGORITHM 487

Exact Cumulative Distribution of the Kolmogorov-Smirnov Statistic for Small Samples [S14]
[J. Pomeranz, *Comm. ACM 17*, 12 (Dec. 1974), 703–704]

Subroutine PRFAC, function subprogram CEIL, and function subprogram PKS, which were published as a part of Algorithm 487, were test routines that were inadvertently printed along with the main algorithm.

Algorithm 488

A Gaussian Pseudo-Random Number Generator [G5]

Richard P. Brent [Recd. 9 Nov. 1973, and 19 Dec. 1973]
Computer Centre, Australian National University,
Canberra, Australia

Key Words and Phrases: random numbers, pseudo-random numbers, Gaussian distribution, normal distribution
CR Categories: 5.39, 5.5
Language: Fortran

Description

Introduction. Successive calls to the Fortran function *GRAND* return independent, normally distributed pseudo-random numbers with zero mean and unit standard deviation. It is assumed that a Fortran function *RAND* is available to generate pseudo-random numbers which are independent and uniformly distributed on [0, 1). Thus, *GRAND* may be regarded as a function which converts uniformly distributed numbers to normally distributed numbers.

Outline of the method. GRAND is based on the following algorithm (Algorithm A) for sampling from a distribution with density function $f(x) = K \exp(-G(x))$ on $[a, b)$, where

$$0 \leq G(x) \leq 1 \tag{1}$$

on $[a, b)$, and the function $G(x)$ is easy to compute:

Step 1. If the first call, then take a sample u from the uniform distribution on [0, 1); otherwise u has been saved from a previous call.
Step 2. Set $x \leftarrow a + (b - a)u$ and $u_0 \leftarrow G(x)$.
Step 3. Take independent samples $u_1, u_2, \ldots$ from the uniform distribution on [0, 1) until, for some $k \geq 1$, $u_{k-1} \leq u_k$.
Step 4. Set $u \leftarrow (u_k - u_{k-1})/(1 - u_{k-1})$.
Step 5. If k is even go to Step 2, otherwise return x.

The reason why Algorithm A is correct is explained in Ahrens and Dieter [2], Forsythe [4], and Von Neumann [6]. The only point which needs explanation here is that, at Step 4, we can form a new uniform variate u from u_{k-1} and u_k, thus avoiding an extra call to the uniform random number generator. This is permissible since at Step 4 it is clear (from Step 3) that $(u_k - u_{k-1})/(1 - u_{k-1})$ is distributed uniformly and independent of x and k. (The fact that it is dependent on u_k is irrelevant.)

Let a_i be defined by $(2/\pi)^{\frac{1}{2}} \int_{a_i}^{\infty} \exp(-\tfrac{1}{2}t^2)dt = 2^{-i}$ for $i = 0, 1, \ldots$. To sample from the positive normal distribution (Algorithm B), we may choose $i \geq 1$ with probability 2^{-i} (easily done by inspecting the leading bits in a uniformly distributed number) and then use Algorithm A to generate a sample from $[a_{i-1}, a_i)$, with $G(x) = \tfrac{1}{2}(x^2 - a_{i-1}^2)$. It is easy to verify that condition (1) is satisfied, in fact

$$\tfrac{1}{2}(a_i^2 - a_{i-1}^2) < \log(2). \tag{2}$$

Finally, to sample from the normal distribution (Algorithm C), we generate a sample from the positive normal distribution and then attach a random sign.

Comments on the method. The algorithm is exact, apart from the inevitable effect of computing with floating-point numbers with a finite word-length. Thus, the method is preferable to methods which depend on the central limit theorem or use approximations to the inverse distribution function.

Let N be the expected number of calls to a uniform random number generator when Algorithm A is executed. If the expected value of k at Step 3 is E, and the probability that k is even is P, then $N = E + N P$, so $N = E/(1 - P)$. From Forsythe [4, eq. (11)], $E = (b - a)^{-1} \int_a^b \exp(G(x))dx$ and

$$1 - P = \frac{1}{b - a} \int_a^b \exp(-G(x))\, dx, \quad \text{so}$$

$$N = \int_a^b \exp(G(x))\, dx \Big/ \int_a^b \exp(-G(x))\, dx. \tag{3}$$

From (3) and the choice of a_i, the expected number of calls to a uniform random number generator when Algorithm C is executed is

$$\sum_{i=1}^{\infty} 2^{-i} \int_{a_{i-1}}^{a_i} \exp(\tfrac{1}{2}(x^2 - a_{i-1}^2))\, dx \Big/ \int_{a_{i-1}}^{a_i} \exp(-\tfrac{1}{2}(x^2 - a_{i-1}^2))\, dx$$
$$\simeq 1.37446. \tag{4}$$

This is lower than 4.03585 for the algorithm given in Forsythe [4], or 2.53947 for the improved version (*FT*) given in Ahrens and Dieter [2]. It is even slightly lower than 1.38009 for the algorithm FL_4 of [2], and FL_4 requires a larger table than Algorithm C. Thus, Algorithm C should be quite fast, and comparable to the best algorithms described by Ahrens and Dieter [1]. The number (4) could be reduced by increasing the table size (as in the algorithms FL_4, FL_5, and FL_6 of [2]), but this hardly seems worthwhile. Exact timing comparisons depend on the machine and uniform random number generator used. (If a very fast uniform generator is used, then Step 4 of Algorithm A may take longer than generating a new uniform deviate.)

The loss of accuracy caused by Step 4 of Algorithm A is not serious. We may say that $\log_2(1 - u_{k-1})^{-1}$ "bits of accuracy" are lost, and in our application we have, from (2) and Step 3 of Algorithm A, $\log(2) > u_0 > \cdots > u_{k-1}$, so the number of bits lost is less than $\log_2(1 - \log(2))^{-1} < 2$.

Test results. If x is normally distributed then $u = (2\pi)^{-\frac{1}{2}} \int_{-\infty}^{x} \exp(-\tfrac{1}{2}t^2)\, dt$ is uniformly distributed on (0, 1). Hence, standard tests for uniformity may be applied to the transformed variate u. Several statistical tests were performed, using a Univac 1108 with both single-precision (27-bit fraction) and double-precision (60-bit fraction). For example, we tested two-dimensional uniformity by taking 10^6 pairs (u, u'), plotting them in the unit square, and performing the Chi-squared test on the observed numbers falling within each of 100 by 100 smaller squares. This test should show up any lack of independence in pairs of successive uniform deviates. We tested one-dimensional uniformity similarly, taking 10^6 trials and subdividing (0, 1) into 1,000 smaller intervals. The values of χ^2 obtained were not significant at the 5 percent level. It is worth noting that the method of summing 12 numbers distributed uniformly on (−1/2, 1/2) failed the latter test, giving $\chi_{999}^2 = 1351$. (The probability of such a value being exceeded by chance is less than 10^{-11}.)

Naturally, test results depend on the particular uniform generator *RAND* which is used. *GRAND* will not produce independent normally distributed deviates unless *RAND* supplies it with independent uniformly distributed deviates! For our tests we used an additive uniform generator of the form $u_n = u_{n-1} + u_{n-127} \pmod{2^w}$ with $w = 27$ or 60 (see Brent [3] and Knuth [5]), but a good linear congruential generator should also be adequate for most applications.

Comparison with Algorithm 334. The fastest exact method previously published in Communications is Algorithm 334 [7]. We timed function *GRAND*, subroutine *NORM* (a Fortran translation of Algorithm 334), and function *RAND* (the uniform random number generator called by *GRAND* and *NORM*). The mean execution times obtained from 500,000 trials on a Univac 1108 were 172, 376 and 59 μsec respectively. Since *NORM* returns two normally distributed numbers, *GRAND* was effectively 9 percent faster than *NORM*. Based on comparisons in [2], we estimate that the saving would be greater if both routines were coded in assembly language, for much of the execution time of *NORM* is taken up in evaluating a square-root and logarithm which are already coded in assembly language.

GRAND requires about 1.38 uniform deviates per normal deviate, and *NORM* requires $4/\pi + 1/2 \simeq 1.77$. Thus, we may estimate that if a uniform generator taking U μsec per call were used, the time per normal deviate would be $(91 + 1.38U)$ μsec for *GRAND* and $(83 + 1.77U)$ μsec for *NORM*. Hence, *GRAND* should be faster for $U \geq 20$.

References

1. Ahrens, J.H., and Dieter, U. Computer methods for sampling from the exponential and normal distributions. *Comm. ACM 15*, 10 (Oct. 1972), 873–882.
2. Ahrens, J.H., and Dieter, U. Pseudo-random Numbers (preliminary version). Preprint of book to be published by Springer, Part 2, Chs. 6–8.
3. Brent, R.P. *Algorithms for Minimization Without Derivatives.* Prentice-Hall, Englewood Cliffs, N.J., 1973, pp. 163–164.
4. Forsythe, G.E. Von Neumann's comparison method for random sampling from the normal and other distributions. *Math. Comp. 26*, 120 (Oct. 1972), 817–826.
5. Knuth, D.E. *The Art of Computer Programming, Vol. 2.* Addison-Wesley, Reading, Mass., 1969, pp. 26, 34, 464.
6. Von Neumann, J. Various techniques used in connection with random digits. In *Collected Works, Vol. 5*, Pergamon Press, New York, 1963, pp. 768–770.
7. Bell, J.R. Algorithm 334, Normal random deviates. *Comm. ACM 11*, 7 (July 1968), 498.

Algorithm

```
      FUNCTION GRAND(N)
C EXCEPT ON THE FIRST CALL GRAND RETURNS A
C PSEUDO-RANDOM NUMBER HAVING A GAUSSIAN (I.E.
C NORMAL) DISTRIBUTION WITH ZERO MEAN AND UNIT
C STANDARD DEVIATION.  THUS, THE DENSITY IS  F(X) =
C EXP(-Ø.5*X**2)/SQRT(2.Ø*PI). THE FIRST CALL
C INITIALIZES GRAND AND RETURNS ZERO.
C THE PARAMETER N IS DUMMY.
C GRAND CALLS A FUNCTION RAND, AND IT IS ASSUMED THAT
C SUCCESSIVE CALLS TO RAND(Ø) GIVE INDEPENDENT
C PSEUDO- RANDOM NUMBERS DISTRIBUTED UNIFORMLY ON (Ø,
C 1), POSSIBLY INCLUDING Ø (BUT NOT 1).
C THE METHOD USED WAS SUGGESTED BY VON NEUMANN, AND
C IMPROVED BY FORSYTHE, AHRENS, DIETER AND BRENT.
C ON THE AVERAGE THERE ARE 1.37746 CALLS OF RAND FOR
C EACH CALL OF GRAND.
C WARNING - DIMENSION AND DATA STATEMENTS BELOW ARE
C           MACHINE-DEPENDENT.
C DIMENSION OF D MUST BE AT LEAST THE NUMBER OF BITS
C IN THE FRACTION OF A FLOATING-POINT NUMBER.
C THUS, ON MOST MACHINES THE DATA STATEMENT BELOW
C CAN BE TRUNCATED.
C IF THE INTEGRAL OF SQRT(2.Ø/PI)*EXP(-Ø.5*X**2) FROM
C A(I) TO INFINITY IS 2**(-I), THEN D(I) = A(I) -
C A(I-1).
      DIMENSION D(6Ø)
      DATA D(1), D(2), D(3), D(4), D(5), D(6), D(7),
     * D(8), D(9), D(1Ø), D(11), D(12), D(13),
     * D(14), D(15), D(16), D(17), D(18), D(19),
     * D(2Ø), D(21), D(22), D(23), D(24), D(25),
     * D(26), D(27), D(28), D(29), D(3Ø), D(31),
     * D(32) /Ø.67448975Ø,Ø.47585963Ø,Ø.383771164,
     * Ø.328611323,Ø.291142827,Ø.263684322,
     * Ø.2425Ø8452,Ø.225567444,Ø.211634166,
     * Ø.199924267,Ø.18991Ø758,Ø.181225181,
     * Ø.1736Ø14ØØ,Ø.1668419Ø9,Ø.16Ø796729,
     * Ø.155349717,Ø.15Ø4Ø9384,Ø.1459Ø2577,
     * Ø.14177ØØ33,Ø.137963174,Ø.134441762,
     * Ø.13117215Ø,Ø.128125965,Ø.12527909Ø,
     * Ø.12261Ø883,Ø.12Ø1Ø356Ø,Ø.1177417Ø7,
     * Ø.115511892,Ø.1134Ø2349,Ø.1114Ø272Ø,
     * Ø.1Ø95Ø3852,Ø.1Ø7697617/
      DATA D(33), D(34), D(35), D(36), D(37), D(38),
     * D(39), D(4Ø), D(41), D(42), D(43), D(44),
     * D(45), D(46), D(47), D(48), D(49), D(5Ø),
     * D(51), D(52), D(53), D(54), D(55), D(56),
     * D(57), D(58), D(59), D(6Ø)
     * /Ø.1Ø5976772,Ø.1Ø4334841,Ø.1Ø27666Ø12,
     * Ø.1Ø1265Ø52,Ø.Ø99827234,Ø.Ø98448282,
     * Ø.Ø971243Ø9,Ø.Ø95851778,Ø.Ø94627461,
     * Ø.Ø93448407,Ø.Ø923119Ø9,Ø.Ø91215482,
     * Ø.Ø9Ø156838,Ø.Ø89133867,Ø.Ø88144619,
     * Ø.Ø87187293,Ø.Ø8626Ø215,Ø.Ø85361834,
     * Ø.Ø8449Ø7Ø6,Ø.Ø83645487,Ø.Ø82824924,
     * Ø.Ø82Ø27847,Ø.Ø81253162,Ø.Ø8Ø499844,
     * Ø.Ø79766932,Ø.Ø79Ø53527,Ø.Ø78358781,
     * Ø.Ø77681899/
C END OF MACHINE-DEPENDENT STATEMENTS
C U MUST BE PRESERVED BETWEEN CALLS.
      DATA U /Ø.Ø/
C INITIALIZE DISPLACEMENT A AND COUNTER I.
      A = Ø.Ø
      I = Ø
C INCREMENT COUNTER AND DISPLACEMENT IF LEADING BIT
C OF U IS ONE.
   1Ø U = U + U
      IF (U.LT.1.Ø) GO TO 2Ø
      U = U - 1.Ø
      I = I + 1
      A = A - D(I)
      GO TO 1Ø
C FORM W UNIFORM ON Ø .LE. W .LT. D(I+1) FROM U.
   2Ø W = D(I+1)*U
C FORM V = Ø.5*((W-A)**2 - A**2). NOTE THAT Ø .LE. V
C .LT. LOG(2).
      V = W*(Ø.5*W-A)
C GENERATE NEW UNIFORM U.
   3Ø U = RAND(Ø)
C ACCEPT W AS A RANDOM SAMPLE IF V .LE. U.
      IF (V.LE.U) GO TO 4Ø
C GENERATE RANDOM V.
      V = RAND(Ø)
C LOOP IF U .GT. V.
      IF (U.GT.V) GO TO 3Ø
C REJECT W AND FORM A NEW UNIFORM U FROM V AND U.
      U = (V-U)/(1.Ø-U)
      GO TO 2Ø
C FORM NEW U (TO BE USED ON NEXT CALL) FROM U AND V.
   4Ø U = (U-V)/(1.Ø-V)
C USE FIRST BIT OF U FOR SIGN, RETURN NORMAL VARIATE.
      U = U + U
      IF (U.LT.1.Ø) GO TO 5Ø
      U = U - 1.Ø
      GRAND = W - A
      RETURN
   5Ø GRAND = A - W
      RETURN
      END
```

Algorithm 489

The Algorithm SELECT—for Finding the *i*th Smallest of *n* Elements [M1]

Robert W. Floyd [Recd 26 Sept. 1974]
Computer Science Department, Stanford University,
Stanford, CA 94305
and
Ronald L. Rivest, M.I.T. Project MAC,
545 Technology Square, Cambridge, MA 02139

Key Words and Phrases: selection, medians, quantiles
CR Categories: 5.30, 5.39

Language: Algol (not strictly Algol 60)

Description

SELECT will rearrange the values of array segment $X[L : R]$ so that $X[K]$ (for some given K; $L \leq K \leq R$) will contain the $(K-L+1)$-th smallest value, $L \leq I \leq K$ will imply $X[I] \leq X[K]$, and $K \leq I \leq R$ will imply $X[I] \geq X[K]$. While *SELECT* is thus functionally equivalent to Hoare's algorithm *FIND* [1], it is significantly faster on the average due to the effective use of sampling to determine the element T about which to partition X. The average time over 25 trials required by *SELECT* and *FIND* to determine the median of n elements was found experimentally to be:

n	500	1000	5000	10000
SELECT	89 ms.	141 ms.	493 ms.	877 ms.
FIND	104 ms.	197 ms.	1029 ms.	1964 ms.

The arbitrary constants 600, .5, and .5 appearing in the algorithm minimize execution time on the particular machine used. *SELECT* has been shown to run in time asymptotically proportional to $N + \min(I, N-I)$, where $N = L - R + 1$ and $I = K - L + 1$. A lower bound on the running time within 9 percent of this value has also been proved [2]. Sites [3] has proved *SELECT* terminates.

The neater Algol 68 construct:
while ⟨boolean expression⟩ **do** ⟨statement⟩
is used here instead of the Algol 60 equivalent:
for *dummy* := 1 **while** ⟨boolean expression⟩ **do** ⟨statement⟩

References

1. Hoare, C.A.R. Algorithm 63 (*PARTITION*) and Algorithm 65 (*FIND*), *Comm. ACM 4* (July 1961;, 321.

2. Floyd, Robert W., and Ronald L. Rivest. Expected time bounds for selection. Stanford CSD Rep. No. 349, Apr., 1973).

3. Sites, Richard. Some thoughts on proving clean termination of programs. Stanford CSD Rep. 417, May 1974.

Algorithm

```
procedure SELECT (X,L,R,K);
  value L,R,K;  integer L,R,K;  array X;
begin
  integer N,I,J,S,SD,LL,RR;  real Z, T;
  while R > L do
  begin
    if R - L > 600 then
    begin
      comment Use SELECT recursively on a sample of size S
        to get an estimate for the (K-L+1)-th smallest element
        into X[K], biased slightly so that the (K-L+1)-th
        element is expected to lie in the smaller set after partition-
        ing;
      N: = R - L + 1;
      I := K - L + 1;
      Z := ln(N);
      S := .5 × exp(2×Z/3);
      SD := .5 × sqrt(Z×S×(N-S)/N) × sign(I-N/2);
      LL := max(L,K-I×S/N+SD);
      RR := min(R,K+(N-I) × S/N+SD);
      SELECT(X,LL,RR,K)
    end;
    T := X[K];
    comment The following code partitions X[L : R] about T. It
      is similar to PARTITION but will run faster on most ma-
      chines since subscript range checking on I and J has been
      eliminated.;
    I := L;
    J := R;
    exchange(X[L],X[K]);
    if X[R] > T then exchange(X[R],X[L]);
    while I < J do
    begin
      exchange(X[I],X[J]);
      I := I + 1; J := J - 1;
      while X[I] < T do I := I + 1;
      while X[J] > T do J := J - 1;
    end;
    if X[L] = T then exchange(X[L],X[J])
      else begin J := J + 1;  exchange(X[J],X[R]) end;
    comment Now adjust L, R so they surround the subset con-
      taining the (K-L+1)-th smallest element;
    if J ≤ K then L := J + 1;
    if K ≤ J then R := J - 1;
  end
end SELECT
```

ACM Transactions on Mathematical Software, Vol. 3, No. 2, September 1976. Pages 301–304

REMARK ON ALGORITHM 489

The Algorithm SELECT—for Finding the ith Smallest of n Elements [M1]
[R.W. Floyd and R.L. Rivest, *Comm. ACM 18*, 3 (March 1975), 173.]

Theodore Brown [Recd 2, Oct. 1975]
Department of Computer Science, Queens College of the City of New York,
Flushing, NY 11367

Algorithm 489, *SELECT*, is an effective algorithm for finding the kth smallest of n elements. The authors, Floyd and Rivest, have analyzed its properties in a companion paper [1].

The description of the algorithm given here is different from that given by Floyd and Rivest [1] and is truer to the actual implementation. The description, furthermore, leads to a simple modification of the algorithm that, as is shown, improves its performance for finding values near the median. It is also shown that a small constant multiplying the standard deviation term is beneficial. Finally, a basic error in Floyd and Rivest's analysis is pointed out.

SELECT can be viewed as a descendant of *FIND* [3], an earlier algorithm for finding the kth smallest element. A major component of *SELECT* is an improved coding of the partitioning algorithm *PARTITION* [3] used by *FIND*. This also is the partitioning algorithm used by the familiar *QUICKSORT* [3]. The partitioning works by dividing the n elements into two parts: those greater than a chosen element and those less than it. (Equality is ignored here. The analyses are based on uniquely valued elements.) In *FIND* (and in *SELECT*) the partitioning is reapplied repeatedly to the partition that contains the required kth smallest element until this value is determined.

FIND chooses the partitioning element randomly from the available candidates. The improved performance of *SELECT* is based on the use of a sample of the available candidates to determine the partitioning element. As described in the following paragraphs, the jth smallest of the sample, say $\mathbf{S}_{(j)}$ (found by recursively calling *SELECT*), is chosen so as to reduce the subsequent size of the required partition.

Writing the kth smallest of the original n elements as $x_{(k)}$, for a sample of size $s(n) \equiv s$ the probability that the jth smallest sample value is the ith smallest of the original n is

$$\Pr\{\mathbf{S}_{(j)} = x_{(i)}\} = \binom{i-1}{j-1}\binom{n-i}{s-j}\Big/\binom{n}{s}, \tag{1}$$

as $j - 1$ elements of the sample must be less than $x_{(i)}$ and $s - j$ greater. The mean and variance for this distribution are, respectively,

$$\mu_{(j)} = j(n+1)/(s+1) \tag{2}$$

$$\sigma_{(j)}^2 = j(s-j+1)(n+1)(n-s)/(s+1)^2(s+2) \tag{3a}$$

$$\leq \tfrac{1}{4}(n+1)(n-s)/s. \tag{3b}$$

Equation (2) can be interpreted as the mean size of the partition of the n elements which contains $\mathbf{S}_{(j)}$ and the values smaller than it.

Floyd and Rivest [1] suggest a value for j of $u = \mu_{(k)} - 2d(n)\sigma_{(k)}$ if $k > n/2$ or $v = \mu_{(k)} + 2\,d(n)\sigma_{(k)}$ if $k \leq n/2$, $d(n)$ a slowly increasing function of n ($(\ln n)^{1/2}$ is used). They suggest this value for j to make sure that the kth smallest falls in the partition either greater than u (if u is used) or less than v (if v is used). A better criterion is *to keep the partition that will contain the k-th as small as possible.* Their stated criterion is contrary to this for very small k, values of k near n, and for values of k near the median; for intermediate values of k, their criterion is consistent with this one.

Notice that from eq. (1), $\Pr\{\mathbf{S}_{(j)} \geq x_{(k)}\} = 1$ for $k \leq j$. So for very small values of k, it does not pay to choose $j > k$ (or for k near n, $j < k$). The coded version of

Table I. Times (in msec) To Find Median *SELECT*

N	Algorithm 489	Our Fortran version	Difference
500	89	44	45
1,000	141	89	52
5,000	493	363	130
10,000	877	666	211

SELECT takes care of these conditions in the *MIN* and *MAX* functions. Notice too that when finding a median it pays to choose $j = \mu_{(k)} = s/2$. Any other choice will cause k to be most likely in the larger partition.

In fact, for any j it never pays to choose a value of u less than $s/2$ or a value of v greater than $s/2$. It is proposed that the calculation of u and v be modified to $\mu_{(k)} + 2d(n)f(n)\sigma_{(k)}$ and $\mu_{(k)} - 2d(n)f(n)\sigma_{(k)}$, respectively, with $f(n)$ a function that monotonically goes to zero from each side of the median. We used a linear function, replacing the *SIGN* function in the coded calculation of **SD** by the factor $(2 \times I/N - 1)$.

A Fortran version of *SELECT* was written for an XDS Sigma 7. Table I compares the times published by Floyd and Rivest in Algorithm 489 with those obtained here. Unfortunately, Floyd and Rivest only give times for finding a median. Notice, however, that not only is our version faster but that it gives proportionately better results for larger n. Our Fortran program was run first with no modification, then with the proposed modification. Figure 1 shows the timing of our Fortran version of *SELECT* without the modification (labeled 1) and with the proposed modification (labeled 2). As expected, the most substantial improvement occurs at the median.

Additional improvement was obtained by reducing the size of $d(n)$. This is true for several reasons. For $n = 5000$, $d(n) = 2.9$. With a normal approximation, the probability that k is more than 5.8 standard deviations away from the mean is less than 10^{-6}. This is a much stricter bound than required, and can be substantially reduced without adverse effects. One does not need to be so careful that the kth smallest element does not end up in the smaller partition. Even if the kth smallest ends up in the larger partition but near the boundary, the reduced problem can be done efficiently. This can be seen in Figure 1. Furthermore, the algorithm's use of the bound (3b) in place of the true deviation overestimates the true standard deviation. Floyd and Rivest recognized this and used a 0.5 multiplier for the standard deviation in the coded version. It was found that a multiplier of 0.1 produced even better results. The modified standard deviation with a 0.1 multiplier gave the results labeled 3 in Figure 1.

Floyd and Rivest [1] assert that their choices of $s(n)$, u, and v make the probability of $o(1/n)$ that k will fall in the partition less than u if u is used or in the partition greater than v if v is used. This is incorrect. It is not possible for any u or v for their choice of $s(n)$. Even the choice of $v = \mathbf{S}_{(1)}$ or $u = \mathbf{S}_{(s)}$ is not adequate, for from eq. (1),

$$\Pr\{\mathbf{S}_{(1)} > x_{(x)}\} = \binom{n-k}{s}/\binom{n}{s} = (n - s)_k/(n)_k,$$

where $(n)_k = n(n - 1) \ldots (n - k + 1)$, is clearly not $o(1/n)$. The best choice of $s(n)$ is an open question. The sorting method of Frazier and McKellar [2] has similarities to *SELECT*—it uses sampling and the partitioning of *PARTITION* [2]. Frazier and McKellar suggest a sample of $0.1n$ for their procedure. No appreciable change in the times resulted from using this sample size. The values differed by less than 10 percent. Further experiments showed that the modifications made here made the running time of *SELECT* rather insensitive to changes in the parameters that Floyd and Rivest [1] suggest tuning for the particular computer: the sample size and the cutoff point below which the algorithm does not do sampling.

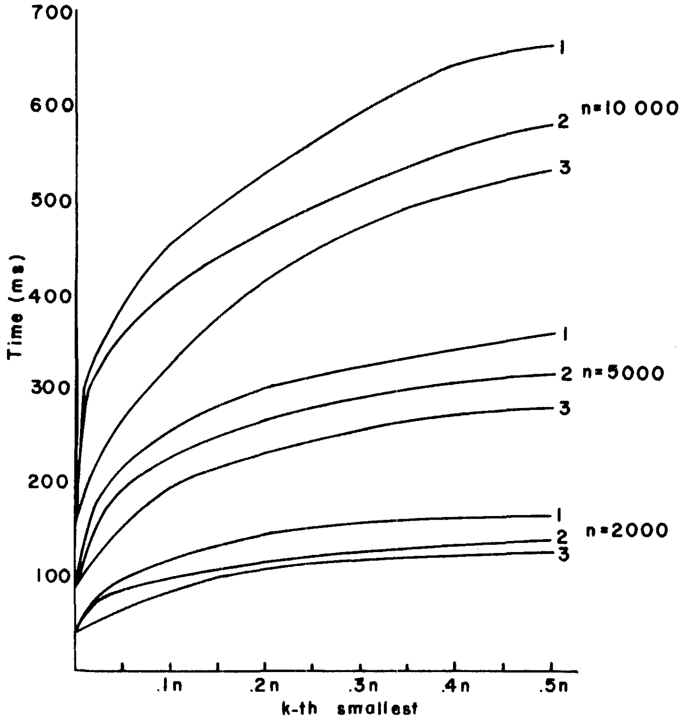


Fig. 1

ACKNOWLEDGMENT

The author acknowledges the help of Robert Pfeffer in writing the timing routine.

REFERENCES

1. Floyd, R.W., and Rivest, R.L. Expected time bounds for selection. *Comm. ACM 18,* 3 (March 1975), 165–172.
2. Frazier, W.D., and McKellar, A.C. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM 17,* 3 (July 1970), 496–507.
3. Hoare, C.A.R. Algorithm 63, *PARTITION*; Algorithm 64, *QUICKSORT*; and Algorithm 65, *FIND. Comm. ACM 4,* 7 (July 1961), 321-322.

Algorithm 490

The Dilogarithm Function of a Real Argument [S22]

Edward S. Ginsberg* [Recd 22 June 1973]
Department of Physics, University of Massachusetts at Boston, Boston, MA 02125
and
Dorothy Zaborowski†
Information Processing Center, Massachusetts Institute of Technology, Cambridge, MA 02139

Key Words and Phrases: dilogarithm
CR Categories: 5.12
Language: Fortran

Description

The dilogarithm function [1-3], defined by

$$Li_2(x) = -\int_0^x (1/z)\ln(1-z)\,dz, \tag{1}$$

occurs in several different applications in physics and engineering, ranging from quantum electrodynamics, to network analysis, to the thermodynamics of ideal ferromagnets, to the structure of polymers. A new function subroutine is developed which computes the dilogarithm function of a real argument to an accuracy of a few parts in 10^{15}. This program was designed to be included in the usual package of library subprograms relied upon by most users. It employs an alternative computational approach to a previously published algorithm [4].

The dilogarithm function is real for real argument $x \leq 1$ and complex for $x > 1$. However, the imaginary part of the dilogarithm is just an ordinary logarithm, $-i\pi\ln(x)$, when $x > 1$, which does not require special means for computation. Therefore, the following algorithm and comments are concerned only with the computation of the real part of the dilogarithm function for real argument.

Briefly, the method consists of transforming the usual series definition

$$Li_2(x) = \sum_1^\infty (x^n/n^2),\ |x| \leq 1, \tag{2}$$

into a more highly convergent power series by means of partial fractions. The identity

$$\frac{1}{n(n+1)(n+2)} = \frac{1}{2}\left(\frac{1}{n} - \frac{2}{n+1} + \frac{1}{n+2}\right) \tag{3}$$

leads immediately to the relation

$$\begin{aligned}(1 + 4x + x^2)Li_2(x) = 4x^2 \sum_1^\infty (x^n/[n(n+1)(n+2)]^2)&\\ + 4x + \frac{23}{4}x^2 + 3(1 - x^2)\ln(1 - x),& \qquad (4)\\ |x| \leq 1.&\end{aligned}$$

This equation permits the evaluation of $Li_2(x)$ for $|x| \leq 1$ using a series which converges like x^n/n^6 instead of x^n/n^2. Of course, more partial fractions can be employed to increase the rate of convergence even further, but then the resulting equation for $Li_2(x)$ is not so simple. The "optimal" number of partial fractions is a question requiring further study.

By the use of well-known functional identities, it is possible to relate the real part of $Li_2(x)$, for any real argument, to values of the function in the restricted range $0 < x \leq \frac{1}{2}$. With $x = \frac{1}{2}$, the maximum relative error in $Li_2(\frac{1}{2})$ after only 25 terms from eq. (4) is roughly

$$\frac{2}{3}\left(\frac{1}{2}\right)^{25}\left(\frac{1}{25}\right)^6 \simeq 10^{-16}.$$

In many cases, far fewer terms are actually needed to achieve this relative accuracy. The various ranges of argument and the corresponding identities used in the Fortran program listing below are:

for $x \geq 2$
$$\mathrm{Re}[Li_2(x)] = \pi^2/3 - \tfrac{1}{2}(\ln x)^2 - Li_2(1/x),$$
for $2 > x > 1$
$$\mathrm{Re}[Li_2(x)] = \pi^2/6 - (\ln x)(\ln(x-1) - \tfrac{1}{2}\ln x) + Li_2(1 - 1/x),$$
for $1 > x > \frac{1}{2}$
$$Li_2(x) = \pi^2/6 - (\ln x)\ln(1-x) - Li_2(1-x),$$
for $0 > x \geq -1$
$$Li_2(x) = -\tfrac{1}{2}[\ln(1-x)]^2 - Li_2(x/(x-1))$$
for $-1 > x$
$$Li_2(x) = \pi^2/6 - \tfrac{1}{2}\ln(1-x)[2 \times \ln(-x) - \ln(1-x)] + Li_2(1/(1-x)).$$

The inherent limitations of floating point arithmetic forced certain modifications and are the only serious sources of error. For example, when $|x|$ is small, the argument of the natural logarithm in eq. (4) is close to unity. The error in *DLOG* (the library subprogram) then determines the accuracy of *DILOG*. It was found that for $0 < |x| \leq 10^{-2}$, the original series, eq. (2), with eight terms, provided 16-place accuracy. Also excluded is a small region around $x_0 \simeq 12.595\ldots$,[1] which is a zero for the real part of the dilogarithm. Here, a Taylor series is used for the calculation. The relative accuracy of *DILOG* suffers accordingly, because the closer x is to x_0, the more significant figures are lost in computing the difference $(x/x_0) - 1$ used in the expansion. (In addition, the value of x_0 probably cannot be expressed exactly in floating point or hexadecimal form.) It is possible to recoup some relative accuracy by computing $(x/x_0) - 1$ to higher than machine precision [5]. However, this would require calculating x_0 to more significant figures than presently known.

The most accurate tables [2] (nine decimal places) published thus far are not adequate to check the values computed by *DILOG*.

* This work was supported by NSF grant GP 29705. † Present address: 54 Dwight Street, Boston, MA 02118.

[1] The best value for x_0 obtained by the authors so far is 12.5951703698450184. . . .

Instead, the program was tested at a selection of arguments for separate ranges of x as follows:

(a) For certain special arguments, the dilogarithm function can be expressed entirely in terms of elementary functions. These are: $1, -1, 2, \frac{1}{2}, 2+q, 1+q, q, 1-q, -q$, and $-1-q$, where $q = \frac{1}{2}((5)^{\frac{1}{2}} - 1)$. For example, $Li_2(1) = \pi^2/6$, and $Li_2(q) = -\ln^2 q + \pi^2/10$.

(b) For values of $|x|$ close to unity, *DILOG* can be checked against a Taylor series expansion. Most of the discrepancy for this class of argument is associated with the computation of $1 - x$ when x is near unity.

(c) For very small values of x, an exact calculation by hand is practical with eq. (2).

(d) For very large values of x, an exact hand calculation for the difference $Li_2(x) - Li_2(-x)$ is possible. In this case, of course, there is cancellation between the two terms so that fewer than 16 places of accuracy are to be expected in evaluating the difference. (Since $Li_2(x) \to -\frac{1}{2}\ln^2|x|$, for $|x| \to \infty$, the values shown in the table below for $Li_2(x) - Li_2(-x)$ are consistent with 16-place accuracy for *DILOG*.)

It can be seen that the worst case in the table represents a relative error of only 2.4 parts in 10^{15}. Thus, 15 to 16 significant figures are correct, representing a slight gain over Kölbig's algorithm [4]. Moreover, a test on an IBM 370/165 of the time required for 1,000 calls to *DILOG*, for randomly generated arguments of absolute value less than 100, revealed that the present algorithm is twice as fast as Kölbig's (0.21 vs. 0.43 sec).

```
                 SELECTED VALUES OF CILOG FCR VARICUS ARGUMENTS
SPECIAL VALUES EXPRESSIBLE IN ELEMENTARY TERMS
          X                        DILCG(X)                    CHECK
  0.10000000000000000D+01   0.16449340668482260D+01   0.16449340668482260D+01
 -0.10000000000000000D+01  -0.82246703342411300D+00  -0.82246703342411320D+00
  0.20000000000000000D+01   0.24674011002723370D+01   0.24674C110027233390D+01
  0.50000000000000000D+00   0.58224052646501230D+00   0.58224C52646501250D+00
  0.26180339887498950D+01   0.24003296863799660D+01   0.24003296863799670D+01
  0.16180339887498950D+01   0.24186901038761120D+01   C.24186901038761140D+01
  0.61803398874989480D+00   0.75539561953174130D+00   0.75539561953174140D+00
  0.38196601125010520D+00   0.42640880616209610D+0C   0.4264C880616209610D+00
 -0.61803398874989480D+00  -0.54219121645069340D+00  -0.54219121645069340D+00
 -0.16180339887498950D+01  -0.12185252606861280D+01  -0.12185252606861300D+01
ARGUMENTS CLOSE TO UNITY
          X                        CILCG(X)                    CHECK
  0.10000100000000000D+01   0.16450591955022320D+01   C.16450591955022340D+01
  0.99999000000000000D+00   0.16448089369929260D+01   0.16448089369929260D+01
  0.10000000001000000D+01   0.16449340692508070D+01   0.16449340692508110D+01
  0.99999999990000000D+00   0.16449340644456410D+01   0.16449340644456410D+01
  0.10000000000000001D+01   0.16449340668482580D+01   0.16449340668482620D+01
  0.99999999999999990D+00   0.16449340668481910D+01   0.16449340668481910D+01
 -0.99999000000000000D+00  -0.82246010194265020D+00  -0.82246010194265020D+00
 -0.10000100000000000D+01  -0.82247396488626100D+00  -0.82247396488626150D+00
 -0.99999999990000000D+00  -0.82246703335479830D+00  -0.82246703335479850D+00
 -0.10000000001000000D+01  -0.82246703349342740D+00  -C.82246703349342790D+00
 -0.99999999999999990D+00  -0.82246703342411240D+00  -0.82246703342411250D+00
 -0.10000000000000001D+01  -0.82246703342411350D+00  -0.82246703342411390D+00
VERY SMALL ARGUMENTS
          X                        DILCG(X)                    CHECK
  0.10000000000000001D-01   0.10025111740139110D-01   C.10025111740139100D-01
  0.99999999999999990D-02   0.10025111740139080D-01   0.10025111740139080D-01
 -0.99999999999999990D-02  -0.99751104900835260D-02  -0.99751104900835260D-02
 -0.10000000000000001D-01  -0.99751104900835450D-02  -0.99751104900835460D-02
  0.10000000000000000D-04   0.1C000025000111110D-04   0.1000002500011111C-04
 -0.10000000000000000D-04  -0.99999750001111100D-05  -0.99999750001111100D-05
  0.10000000000000000D-09   0.1000000000025C000D-09   0.10000000000250000D-09
 -0.10000000000000000D-09  -0.99999999997500000D-10  -0.9999999999975C0000D-10
  0.10000000000000000D-14   0.10000000000000000D-14   0.10000000000000000D-14
 -0.10000000000000000D-14  -0.99999999999999970D-15  -0.99999999999999970D-15
  0.10000000000000000D-29   0.10000000000000000D-29   0.1C000000000000000D-29
 -0.10000000000000000D-29  -0.1C000000000000000D-29  -0.1000000000000000D-29
VERY LARGE ARGUMENTS
          X                   DILOG(X)-CILCG(-X)               CHECK
  0.10000000000000000D+03   0.49148019783144580D+01   0.49148019783144570D+01
  0.10000000000000000D+06   0.49347822005446640D+01   0.49347822005446790D+01
  0.10000000000000000D+11   0.49348022003446200D+01   0.49348022003446790D+01
  0.10000000000000000D+16   0.49348022005445960D+01   0.49348022005446770D+01
  0.10000000000000000D+31   0.49348022005445390D+01   0.49348022005446790D+01
```

Author Ginsberg would like to acknowledge the hospitality of the Center for Theoretical Physics at M.I.T. Both authors are indebted to W.J. Cody of Argonne National Laboratory for suggesting many improvements to the original program.

References

1. Lewin, L. *Dilogarithms and Associated Functions.* MacDonald, London, 1958.
2. Mitchell, K. *Phil. Mag. 40*, (1949), 351-368.
3. Abramowitz, M., and Stegun, I.A., Eds. *Handbook of Mathematical Functions, etc.* Nat. Bur. Stand. App. Math. Ser. #55, Supt. of Documents, U.S. Gov. Print. Off. 1964.
4. Kölbig, K.S. Collected Algorithms from CACM, 327-P 1-0.
5. Paciorek, K.A. Collected Algorithms from CACM, 385-P 1-0.

Algorithm

```
      DOUBLE PRECISION FUNCTION DILOG(X)
C REAL PART OF THE DILOGARITHM FUNCTION FOR A REAL
C ARGUMENT. REF. NO. 1=L. LEWIN, *DILOGARITHMS +
C ASSOCIATED FUNCTIONS*
C                           (MAC-DONALD, LONDON, 1958).
C NUMERICAL CONSTANTS USED ARE C(N)=(N(N+1)(N+2))**2
C FOR N=1 TO 30, (PI**2)/3=3.289868...,
C (PI**2)/6=1.644394..., AND ZERO OF DILOG ON THE
C POSITIVE REAL AXIS, X0=12.59517...
      DOUBLE PRECISION A, B, BY, C, C1, C2, C3, C4,
     * DX, DY, TEST, W, X, X0, Y, Z
      DIMENSION C(30)
      DATA C(1), C(2), C(3), C(4), C(5), C(6), C(7),
     * C(8), C(9), C(10), C(11), C(12), C(13),
     * C(14), C(15), C(16), C(17), C(18), C(19),
     * C(20), C(21), C(22), C(23), C(24), C(25),
     * C(26), C(27), C(28), C(29), C(30)
     * /36.D0,576.D0,36.D2,144.D2,441.D2,112896.D0,
     * 254016.D0,5184.D2,9801.D2,17424.D2,2944656.D0,
     * 4769856.D0,74529.D2,112896.D2,166464.D2,
     * 23970816.D0,33802596.D0,467856.D2,636804.D2,
     * 853776.D2,112911876.D0,147476736.D0,19044.D4,
     * 24336.D4,3080025.D2,386358336.D0,480661776.D0,
     * 5934096.D2,7273809.D2,8856576.D2/
      IF (X.GT.12.6D0) GO TO 10
      IF (X.GE.12.59D0) GO TO 100
      IF (X.GE.2.D0) GO TO 10
      IF (X.GT.1.D0) GO TO 20
      IF (X.EQ.1.D0) GO TO 30
      IF (X.GT..5D0) GO TO 40
      IF (X.GT.1.D-2) GO TO 50
      IF (X.LT.-1.D0) GO TO 60
      IF (X.LT.-1.D-2) GO TO 70
C DILOG COMPUTED FROM REF. NO. 1, P.244, EQ(1).
      DILOG = X*(1.D0+X*(.25D0+X*(1.D0/9.D0+X*
     * (625.D-4+X*(4.D-2+X*(1.D0/36.D0+X*(1.D0/
     * 49.D0+X/64.D0)))))))
      RETURN
C DILOG COMPUTED FROM REF. NO. 1, P.244, EQ(6),
C AND DESCRIPTION OF THIS ALGORITHM, EQ(4).
   10 Y = 1.D0/X
      BY = -1.D0 - Y*(4.D0+Y)
      DILOG = 3.28986813369645287D0 -
     * .5D0*DLOG(X)**2 + (Y*(4.D0+5.75D0*Y)+3.D0*
     * (1.D0+Y)*(1.D0-Y)*DLOG(1.D0-Y))/BY
      IF (DILOG+4.D0*Y.EQ.DILOG) RETURN
      GO TO 80
C DILOG COMPUTED FROM REF. NO. 1, P.244, EQ(7) WITH
C X=1/X + EQ(6), AND DESCRIPTION OF THIS ALGORITHM,
C EQ(4).
   20 Y = 1.D0 - 1.D0/X
      DX = DLOG(X)
      BY = 1.D0 + Y*(4.D0+Y)
      DILOG = 1.64493406684822643D0 +
     * DX*(.5D0*DX-DLOG(X-1.D0)) +
     * (Y*(4.D0+5.75D0*Y)-3.D0*(1.D0+Y)*DX/X)/BY
      GO TO 80
C DILOG COMPUTED FROM REF. NO. 1, P.244, EQ(2).
   30 DILOG = 1.64493406684822643D0
      RETURN
C DILOG COMPUTED FROM REF. NO. 1, P.244, EQ(7),
C AND DESCRIPTION OF THIS ALGORITHM, EQ(4).
   40 Y = 1.D0 - X
      DX = DLOG(X)
      BY = -1.D0 - Y*(4.D0+Y)
      DILOG = 1.64493406684822643D0 - DX*DLOG(Y) +
     * (Y*(4.D0+5.75D0*Y)+3.D0*(1.D0+Y)*DX*X)/BY
     .GO TO 80
C DILOG COMPUTED FROM DESCRIPTION OF THIS ALGORITHM,
C EQ(4)
   50 Y = X
      BY = 1.D0 + Y*(4.D0+Y)
      DILOG = (Y*(4.D0+5.75D0*Y)+3.D0*(1.D0+Y)*
     * (1.D0-Y)*DLOG(1.D0-Y))/BY
      GO TO 80
C DILOG COMPUTED FROM REF. NO. 1, P.245, EQ(12) WITH
C X=-X, AND DESCRIPTION OF THIS ALGORITHM, EQ(4).
   60 Y = 1.D0/(1.D0-X)
      DX = DLOG(-X)
      DY = DLOG(Y)
      BY = 1.D0 + Y*(4.D0+Y)
      DILOG = -1.64493406684822643D0 +
     * .5D0*DY*(DY+2.D0*DX) + (Y*(4.D0+5.75D0*Y)
     * +3.D0*(1.D0+Y)*(1.D0-Y)*(DX+DY))/BY
      IF (DILOG+4.D0*Y.EQ.DILOG) RETURN
      GO TO 80
C DILOG COMPUTED FROM REF. NO. 1, P.244, EQ(8),
C AND DESCRIPTION OF THIS ALGORITHM, EQ(4).
   70 Y = X/(X-1.D0)
      DX = DLOG(1.D0-X)
      BY = -1.D0 - Y*(4.D0+Y)
      DILOG = (Y*(4.DØ+5.75DØ*Y)-3.DØ*(1.DØ+Y)*
     * (1.DØ-Y)*DX)/BY - .5DØ*DX*DX
   8Ø B = 4.DØ*Y*Y/BY
      DO 9Ø N=1,3Ø
         B = B*Y
         A = B/C(N)
         TEST = DILOG
         DILOG = DILOG + A
         IF (DILOG.EQ.TEST) RETURN
   9Ø CONTINUE
      RETURN
C DILOG COMPUTED FROM TAYLOR SERIES ABOUT ZERO OF
C DILOG, XØ.
  1ØØ XØ = 12.595170369845Ø184DØ
      Y = X/XØ - 1.DØ
      Z = 1.DØ/11.595170369845Ø184DØ
      W = Y*Z
      C1 = (3.DØ*XØ-2.DØ)/6.DØ
      C2 = ((11.DØ*XØ-15.DØ)*XØ+6.DØ)/24.DØ
      C3 = (((5.D1*XØ-1Ø4.DØ)*XØ+84.DØ)*XØ-24.DØ)/
     * 12.D1
      C4 = ((((274.DØ*XØ-77.D1)*XØ+94.D1)*XØ-54.D1)*
     * XØ+12.D1)/72.D1
      DILOG = Y*(1.DØ-Y*(.5DØ-Y*(1.DØ/3.DØ-Y*
     * (.25DØ-Y*(.2DØ-Y/6.DØ)))))*DLOG(Z) -
     * W*XØ*Y*(.5DØ-W*(C1-W*(C2-W*(C3-W*C4))))
      RETURN
      END
```

ACM Transactions on Mathematical Software, Vol. 2, No. 1, March 1976, Page 112

REMARK ON ALGORITHM 490

The Dilogarithm Function of a Real Argument [S22]
[E.S. Ginsberg and D. Zaborowski, *Comm. ACM 18*, 4 (April 1975), 200–202]

Robert Morris [Recd 11 July 1975]
Bell Laboratories, Murray Hill, NJ 07974

The necessary value for the zero of the dilogarithm function is

$$12.5951703698450161286398965\ldots$$

to 25 decimal places, all correct. The value given in Algorithm 490 is in error in the last two digits.

The identity stated for values of x less than -1 is incorrect and should read

$$Li_2(x) = -\pi^2/6 - \tfrac{1}{2}\ln(1-x)[2 \times \ln(-x) - \ln(1-x)] + Li_2(1/(1-x)).$$

Algorithm 491

Basic Cycle Generation [H]

Norman E. Gibbs [Recd 13 July 1971]
Department of Mathematics, College of William and Mary, Williamsburg, VA 23185

This work was partially supported by NASA under Grant NGL-47-006-058.

Key Words and Phrases: Graph, basic cycle, fundamental cycle, spanning tree, vertex adjacency matrix
CR Categories: 5.32, 3.24
Language: PL/I

Description

The PL/I procedure *BASIC_GENERATOR* is an implementation of Paton's algorithm [1] for finding a set of basic (fundamental) cycles of a finite undirected graph from its vertex adjacency matrix.

The input parameters to the procedure are:

(1) A modified form of the vertex adjacency matrix, called A (see assumption 3 below).

(2) The number of vertices of the graph, called *N*.

(3) The number of edges of the graph, called *EDGES*.

The output of the procedure is an array of bit strings, called *B*. The *j*th bit of B_i is 1 if and only if the *i*th basic cycle contains the edge labeled *j*.

The following assumptions are made by the procedure:

(1) The graph is finite, connected, undirected, and without loops or multiple edges.

(2) The vertices are labeled 1, 2, . . ., *N*.

(3) The vertex adjacency matrix A has an edge table coded into its lower triangular part. The following PL/I code could be used to generate the table:

```
E = 0;
DO I = 2 TO N;
  DO J = 1 TO I − 1;
    IF A(I, J)¬ = 0 THEN
      DO;
        E = E + 1;
        A(I, J) = E;
      END;
  END;
END;
```

(4) *A* is not the vertex adjacency matrix of a tree.

The algorithm is:

Step 1. Let vertex 1 be the root of the spanning tree. Start forming the spanning tree by placing all edges of the form {1, *W*} into the tree. At the same time, place all vertices *W* into a push-down list called *STACK*.

Step 2. Let *Z* be the last vertex added to *STACK* (i.e., the top of the stack). If *STACK* is empty, then stop. If *STACK* is not empty, then remove *Z* from *STACK* and go to step 3.

Step 3. Consider all edges {*Z*, *W*} which have not been examined. If all edges have been examined, go to step 2. Otherwise, for each edge {*Z*, *W*} do the following:

(a) If *W* is in the tree, generate the basic cycle formed by adding {*Z*, *W*} to the tree and repeat step 3.

(b) If *W* is not in the tree, add {*Z*, *W*} to the tree, *W* to *STACK*, and repeat step 3.

For details on the algorithm and the production of the basic cycles, Paton's original paper should be consulted. This paper also discusses two other algorithms for basic cycle generation and contains performance statistics.

BASIC_GENERATOR has been implemented using the IBM PL/I F-level compiler (version 5.1) and has been tested on approximately 200 graphs.

Reference

1. Paton, K. An algorithm for finding a fundamental set of cycles of a graph. *Comm. ACM 12,* 9 (Sept. 1969), 514–518.

Algorithm

```
BASIC_GENERATOR:
PROCEDURE (A,N,EDGES,B);
/*  BASIC_GENERATOR GENERATES A SET OF BASIC (FUNDAMENTAL)
    CYCLES FROM THE VERTEX ADJACENCY MATRIX OF A CONNECTED
    UNDIRECTED GRAPH WITHOUT LOCPS OR MULTIPLE EDGES.  THE
    PROCEDURE IS A PL/I IMPLEMENTATION OF KEITH PATON'S
    ALGORITHM DESCRIBED IN CACM 12, 9 (SEPTEMBER 1969),
    514-518.                                               */
  DECLARE
    (A(*,*),N,EDGES) BINARY FIXED (15,0),
    B(*) BIT (EDGES),
    BASIC BINARY FIXED (15,0) INITIAL (0),
    T BIT (N) INITIAL ('0'B),
    STACK CONTROLLED BINARY FIXED (15,0),
    (Z,W,J) BINARY FIXED (15,0),
    PREV(N) BINARY FIXED (15,0) INITIAL ((N)0);
/*      A IS AN N BY N VERTEX ADJACENCY MATRIX OF THE GRAPH.
          THE LOWER TRIANGULAR PORTION CONTAINS AN EDGE
          TABLE.  IF J>K AND A(J,K)=M, THEN EDGE M JOINS
          VERTICES J AND K IN THE GRAPH.  IF A(J,K)¬=0 AND
          J>K THEN A(K,J)=1.  THE UPPER TRIANGULAR PART OF A
          IS DESTROYED IN THE PROCESS,  BUT CAN BE EASILY
          RECOVERED FROM THE LOWER TRIANGULAR PART.  (INPUT)
        N IS THE NUMBER OF VERTICES IN THE GRAPH. (INPUT)
    EDGES IS THE NUMBER OF EDGES IN THE GRAPH. (INPUT)
        B WILL BE THE SET OF BASIC CYCLES GENERATED.  THE
          K TH BIT OF B(J) IS 1 IF AND ONLY IF THE J TH
          BASIC CYCLE CONTAINS THE EDGE LABELED K (OUTPUT).
    BASIC IS USED TO INDEX THE BASIC CYCLES AS THEY ARE
          GENERATED.
        T IS USED TO KEEP TRACK OF THE VERTICES CURRENTLY
          IN THE SPANNING TREE.
    STACK IS A PUSHDOWN LIST USED TO HOLD THE VERTICES OF
          THE SPANNING TREE WHICH HAVE NOT YET BEEN
          EXAMINED.
        Z IS THE VERTEX OF THE SPANNING TREE CURRENTLY BEING
          EXAMINED.
        W IS USED TO FIND EDGES WHICH CONNECT TO Z.
     PREV IS AN ARRAY USED IN THE PRODUCTION OF THE BASIC
          CYCLES.  IF PREV(K)=J THEN (K,J) IS AN EDGE OF THE
          TREE WITH J NEARER THE ROOT.                     */
/*  INITIALIZATION SECTION--NOTE THAT VERTEX 1 IS ALWAYS
    THE ROOT.                                              */
  B='0'B;
  SUBSTR(T,1,1)='1'B;
  ALLOCATE STACK;
  STACK=0;
  ALLOCATE STACK;
  STACK=1;
NEW_Z:
  Z=STACK;
  IF Z=0 THEN RETURN;
  ELSE
    DO;
      FREE STACK;
      DO W=2 TO N;
        IF A(MIN(Z,W),MAX(Z,W))=1 THEN
          DO:
            IF SUBSTR(T,W,1) THEN
/*  THE EDGE CONNECTING Z AND W CREATES A BASIC CYCLE.     */
```

```
              DO;
                BASIC=BASIC+1;
                SUBSTR(B(BASIC),A(MAX(W,PREV(W)),
                  MIN(W,PREV(W))),1)='1'B;
                SUBSTR(B(BASIC),A(MAX(Z,W),MIN(Z,W)),
                  1)='1'B;
                A(MIN(Z,W),MAX(Z,W))=0;
                J=Z;
                DO WHILE(J¬=PREV(W));
                  SUBSTR(B(BASIC),A(MAX(PREV(J),J),
                    MIN(PREV(J),J)),1)='1'B;
                  J=PREV(J):
                END;
              END;
            ELSE
/*  THE EDGE CONNECTING Z AND W SHOULD BE PLACED IN THE
    TREE.                                                */
              DO;
                PREV(W)=Z;
                SUBSTR(T,W,1)='1'B;
                ALLOCATE STACK;
                STACK=W;
                A(MIN(Z,W),MAX(Z,W))=0;
              END;
          END;
      END;
      GO TO NEW_Z;
    END:
END BASIC_GENERATOR;
```

Algorithm 492 Generation of All the Cycles of a Graph from a Set of Basic Cycles [H]

Norman E. Gibbs [Recd 13 July 1971]
Department of Mathematics, College of William and Mary, Williamsburg, VA 23185

This work was partially supported by NASA under Grant NGL-47-006-058.

Key Words and Phrases: basic cycle, cycle, graph
CR Categories: 5.32, 3.24 Language: PL/I

Description

The PL/I procedure *CYCLE_GENERATOR* is an implementation of Gibbs' algorithm [1] for finding all the cycles in a graph from a set of basic cycles.

The input parameters are:

(1) An array of bit strings B, where the jth bit of B_i is 1 if and only if the ith basic cycle includes the edge labeled j.

(2) The number of basic cycles, called *BASIC*.

(3) The number of edges in the graph, called *EDGES*.

The output from the procedure consists of:

(1) An array of bit strings Q, where the jth bit of Q_i is 1 if and only if the ith cycle contains the edge labeled j.

(2) The number of cycles, called *CYCLES*.

The algorithm is:

Step 1. Set $C = \{B_1\}$, $Q = C$, $D = R = \emptyset$, $i = 2$. If $BASIC = 1$, stop.

Step 2. For all $T \in Q$, if $T \cap B_i = \emptyset$, then set $D = D \cup \{T \oplus B_i\}$, otherwise set $R = R \cup \{T \oplus B_i\}$. $(A \oplus B = A \cup B - A \cap B)$.

Step 3. For all $U, V \in R$, if $U \subset V$, set $D = D \cup \{V\}$ and $R = R - \{V\}$.

Step 4. Set $C = C \cup R \cup \{B_i\}$, $Q = C \cup D$, $R = \emptyset$, $i = i + 1$.

Step 5. If $i > BASIC$, stop. C is the set of all cycles. If $i \leq BASIC$, go to step 2.

In *CYCLE_GENERATOR*, $C = \{Q(I) : QFLAG(I) = \text{'0'}B\}$, $D = Q - C$, and $R = \{Q(LOWER), \mathrm{Q}(LOWER+1), \ldots, Q(UPPER)\}$. The procedure assumes that $BASIC > 0$ and that the dimension of Q is $2^{BASIC} - 1$. *CYCLE_GENERATOR* has been implemented using the IBM PL/I F-level compiler (version 5.1) and has been tested on approximately 200 graphs.

Reference

1. Gibbs, N.E. A cycle generation algorithm for finite undirected linear graphs. *J. ACM 16*, 4 (Oct. !969), 564–568.

Algorithm

```
CYCLE_GENERATOR:
PROCEDURE(B,BASIC,Q,CYCLES,FDGES);
/* CYCLE_GENERATOR GENERATES ALL THE CYCLES OF A GRAPH
   FROM A SET OF BASIC (FUNDAMENTAL) CYCLES.  THIS
   PROCEDURE IS A PL/I IMPLEMENTATION OF NORM GIBBS'
   ALGORITHM FOR GENERATING ALL THE CYCLES OF A GRAPH
   WHICH APPEARED IN JACM 16, 4 (OCTOBER 1969), 564-568. */
  DECLARE
    (B(*),Q(*)) BIT (EDGES),
    (BASIC,CYCLES,EDGES) BINARY FIXED (15,0),
    QFLAG(2**BASIC-1) BIT (1) INITIAL((2**BASIC-1)(1)'0'B),
    (QINDEX,I,J,K,UPPER,LOWER) BINARY FIXED (15,0);
/*       B IS THE SET OF BASIC CYCLES WHERE THE K TH BIT OF
           B(J) IS 1 IF AND ONLY IF EDGE K IS AN ELEMENT OF
           THE J TH BASIC CYCLE.  (INPUT).
         Q IS THE SET OF ALL CYCLES GENERATED.  THE K TH BIT
           OF Q(J) IS 1 IF AND ONLY IF EDGE K IS AN ELEMENT
           OF THE J TH CYCLE.  (OUTPUT).
     BASIC IS THE NUMBER OF BASIC CYCLES IN B.  (BASIC > 0).
           (INPUT).
    CYCLES IS THE NUMBER OF CYCLES GENERATED.  (OUTPUT).
     EDGES IS THE NUMBER OF EDGES IN THE GRAPH.  (INPUT).
     QFLAG IS A LOGICAL ARRAY USED TO MARK EDGE-DISJOINT
           UNIONS OF CYCLES.
    OTHER IDENTIFIERS ARE USED AS COUNTERS OR POINTERS.    */
/* INITIALIZATION STEP.  THE PROCEDURE ASSUMES THAT
    BASIC>0.                                               */
  Q(1)=B(1);
  IF BASIC=1 THEN
    DO;
      CYCLES=BASIC;
      RETURN;
    END;
/* FORM ALL LINEAR COMBINATIONS OF THE BASIC CYCLES IN Q.*/
  DO I=2 TO BASIC;
    LOWER=2**(I-1);
    UPPER=2**I-1;
/* IF B(I) INTERSECT Q(QINDEX) IS EMPTY, THEN THE SYMMETRIC
   DIFFERENCE OF B(I) AND Q(QINDEX) IS THE UNION OF DIS-
   JOINT CYCLES AND THE APPROPRIATE ELEMENT OF QFLAG IS
   SET TO '1'B.  OTHERWISE THE SYMMETRIC DIFFERENCE IS
   PLACED INTO A SET (INDEXED BY LOWER AND UPPER) FOR
   FURTHER TESTING.                                        */
    DO QINDEX=1 TO LOWER-1;
      IF B(I) & Q(QINDEX) THEN
        DO;
          Q(UPPER)=(B(I)|Q(QINDEX))&(¬B(I)|¬Q(QINDEX));
          UPPER=UPPER-1;
        END;
      ELSE
        DO;
          Q(LOWER)=(B(I)|Q(QINDEX))&(¬B(I)|¬Q(QINDEX));
          QFLAG(LOWER)='1'B;
          LOWER=LOWER+1;
        END;
    Q(LOWER)=B(I);
    END;
/* WE NOW TEST THE SET Q(LOWER), Q(LOWER+1),...,Q(UPPER)
   TO SEE IF ANY ELEMENT OF THIS SET PROPERLY CONTAINS
   ANY OTHER ELEMENT.  IF SO, THE CONTAINING ELEMENT IS
   MARKED AS THE EDGE-DISJOINT UNION OF CYCLES AND THE
   APPROPRIATE ELEMENT OF QFLAG IS SET TO '1'B.            */
    DO J=LOWER+1 TO 2**I-2;
      DO K=J+1 TO 2**I-1;
        IF QFLAG(J) THEN GO TO NEXT_J;
        ELSE IF QFLAG(K) THEN GO TO NEXT_K;
        IF (Q(J)|Q(K))=Q(J) THEN QFLAG(J)='1'B;
        ELSE IF (Q(J)|Q(K))=Q(K) THEN QFLAG(K)='1'B;
NEXT_K:
      END;
NEXT_J:
    END;
  END;
/* BEFORE RETURNING, WE WANT TO MOVE ALL THE CYCLES (THOSE
   ELEMENTS OF Q FOR WHICH QFLAG IS '0'B)  TO Q(1), Q(2),
   ..., Q(CYCLES) AND SET CYCLES EQUAL TO THE NUMBER OF
   CYCLES IN Q.                                            */
  CYCLES=0;
HOUSEKEEPING:
  DO I=1 TO 2**BASIC-1;
    IF QFLAG(I) THEN GO TO NEXT_I;
    ELSE
      DO;
        Q(CYCLES+1)=Q(I);
        CYCLES=CYCLES+1;
      END;
NEXT_I:
  END HOUSEKEEPING;
END CYCLE_GENERATOR;
```