Algorithms Supplement

Previously published algorithms

The following algorithms have recently appeared in the Algorithms Sections of the specified journals.

(*a*) **Communications of the ACM** (April–June 1969)

348 MATRIX SCALING BY INTEGER PROGRAMMING

Uses scaling to precondition matrices so as to improve subsequent computational characteristics.

349 POLYGAMMA FUNCTIONS WITH ARBITRARY PRECISION

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 $-\ln(z)$

350 SIMPLEX METHOD PROCEDURE EMPLOYING LU DECOMPOSITION

Attacks the linear programming problem

$$\text{maximise } d^T x$$
$$\text{subject to } Gx = b \text{ and } x \geqslant 0$$

351 MODIFIED ROMBERG QUADRATURE

Calculates the approximate value of the definite integral

$$I = \int_A^B F(X)dX$$

together with an error bound, by a modified form of Romberg quadrature which is less sensitive to the accumulation of rounding errors than the customary method.

(*b*) **BIT** (January 1969)

PARTITION FUNCTIONS (MODULO *d*)

Computes $p(k)$ (mod d), where $p(k) = p_{-1}(k)$, and $p_n(k)$ where

$$\sum_{k=0}^{\infty} p_n(k)x^k = (\phi(x))^n$$

and $$\phi(x) = \prod_{k=1}^{\infty}(1 - x^k)$$

(*c*) **Applied Statistics** (September–December 1969)

AS18 EVALUATION OF MARGINAL MEANS

Transfers values to an n-way table (array) allotting space for margins which are then filled with marginal means.

AS19 ANALYSIS OF VARIANCE FOR A FACTORIAL TABLE

Given an n-way table with margins filled with the marginal means, produces tables of corrected sums of squares and associated degrees of freedom for all main effects and interactions. A minor modification of part of this procedure gives the Yates' algorithm for forming effects from a 2^n table.

AS20 THE EFFICIENT FORMATION OF A TRIANGULAR ARRAY WITH RESTRICTED STORAGE FOR DATA

All $\binom{N}{2}$ pairs of N equal vectors may be required when only $M(<N)$ can be held simultaneously in core. This algorithm comes near to minimising the number of transfers from backing to core store and, when the backing store is magnetic tape, is particularly good in minimising tape winding.

AS21 SCALE SELECTION FOR COMPUTER PLOTS

Chooses a reasonable scale for plotting one- and two-dimensional data for a given number of intervals.

AS22 THE INTERACTION ALGORITHM

Computes one cycle in transforming an n-way array (factorial table) into effects according to a specified orthonormal contrast matrix. The full transformation to effects can be achieved by using the subroutine n times.

AS23 CALCULATION OF EFFECTS

Similar to AS22, *using procedures described in* AS1. *Complete algorithms are given for both the* 1*-cycle operation and the calculation of effects using all n-cycles.*

AS24 FROM NORMAL INTEGRAL TO DEVIATE

Computes the deviate corresponding to a given area under a normal curve.

AS25 CLASSIFICATION OF MEANS FROM ANALYSIS OF VARIANCE

Uses Tukey's method to divide a set of means into distinguishable groups at a chosen significance level.

The following papers, containing useful algorithms, have recently appeared in the specified journals.

(*a*) **BIT** (January–April 1969)

A PROOF OF HAMBLIN'S ALGORITHM FOR TRANSLATION OF ARITHMETIC EXPRESSIONS FROM INFIX TO POSTFIX FORM (Bind 9, Hefte Nr. 1, pp.59–68)

SMOOTH CURVE INTERPOLATION (Bind 9, Hefte Nr. 1, pp. 69–77)

ALGORITHMS OVER PARTIALLY ORDERED SETS (Bind 9, Hefte Nr. 2, pp. 97–118)

SOME EFFICIENT FOURTH ORDER MULTIPOINT METHODS FOR SOLVING EQUATIONS (Bind 9, Hefte Nr. 2, pp. 119–124)

COMPUTER CARTOGRAPHY RANGE MAP (Bind 9, Hefte Nr. 2, pp. 157–166)

New algorithms

Algorithm 43

A LISTED RADIX SORT

A. D. Woodall
Reading College of Technology

Author's Note:

Radix sorting, which is the method used to sort punched cards mechanically, has some attractions when programmed for a computer. In particular, the time taken is a linear function of the number of items to be sorted: the sort can be very fast for large numbers of items.

As often described—for example by Gotlieb (1963)—the procedure has the disadvantage of using a great deal of storage space, and also of taking a lot of time on a collection phase, when items are assembled into a single stack after each pass. In the present procedure, this is overcome and the storage used (apart from the program itself and work space) is $2n + 2b$ words, where n is the number of items to be sorted, and b is the radix. b has been included as a parameter of the procedure. Its value, which may be very large, can be chosen to optimise the procedure on any particular computer and for any given word length of items to be sorted. b would normally be chosen of the form 2^N, so as to make use of the computer's ability to select sequences of bits from a word.

In order to optimise the sort it should be noticed that the time taken is $C_1nm + C_2mb + C_3n$ where the constants C_1, C_2 and C_3 are best found empirically. m is the number of passes, and depends on b and the word length of items to be sorted. The speed is unaffected by the initial order of the data.

Reference

Gotlieb, C. C. (1963). Sorting on Computers, *Communications of the ACM*, Vol. 6, p. 195.

```
procedure sort (a, n, b, m, digit, use); value b, n, m;
integer array a; integer b, n, m; integer procedure digit;
procedure use;
comment the n items to be sorted are held as a[1] to a[n]. b is
the chosen base of enumeration, or radix. digit(x, k, b) is the
kth digit to the right in the representation of x in the scale of b: in
other words it has the value x ÷ b ↑ (k − 1) − (x ÷ b ↑ k) × b.
If this Algol expression is used to evaluate digit, it would be
more efficient to incorporate it in the program, rather than
calling a procedure (digit appears in only one statement).
Further it would be better to arrange to compute b ↑ (k − 1)
and b ↑ k only once for each value of k, also to avoid evaluating
x ÷ 1 when k = 1 on the first pass. However, for most
compilers, the program will be much better if digit is realised in
the machine instructions used to unpack sequences of bits from
a storage location. The parameter m is the number of digits
in the representation of the items a[i] (m corresponds to the
number of passes in the mechanical punched card sort). If the
greatest a[i] has value max, then m must not be less than the
logarithm of max to base b. For example with items all less
than 1,000,000 it might be convenient to use b = 1,024, m = 2.
The procedure sort will activate the procedure use(x) with x
taking the values of the n items of the array a sorted into order.
As an example, if use(x) is a statement causing the value of x
to be printed on a new line, the procedure sort will print out
the values of the items in order;

 begin integer array list[0: n], start, last[0: b − 1];
 comment lists corresponding to the stacks of cards in the
 various stages of a punched card sort are described by the
 array list. The successor of any item a[i] of a list will be
 a[list[i]]. The end of a list will sometimes be a[k] where
 list[k] = 0, and sometimes a[k] where k is held as the last
 item of that list;

 integer i, k, w, j, x, b1;
 b1 := b − 1;
 for i := n − 1 step − 1 until 0 do list[i] := i + 1;
 list[n] := 0;
 comment the initial listing, starting from a[list[0]], corre-
 sponds to a stack of cards in the order they have arrived;
 for k := 1 step 1 until m do
   begin
   comment the kth digit from the right is to be used for
   sorting;
   for i := b − 1 step − 1 until 0 do start[i] := 0;
   comment the list starting from start[i] will correspond to a
   stack of cards whose kth digit is i. The index of the end of
   this list will be held as last[i];
   w := 0;
   for w := list[w] while w ≠ 0 do
     begin
     j := digit(a[w], k, b);
     if start[j] = 0 then start[j] := w else list[last[j]] := w;
     last[j] := w
     end of the compound statement describing the placing of
     a[w] in its stack for this pass;
   x := 0;
   comment now the lists will be joined to form a single list,
   starting at a[list[0]] and ending at a[x] where list[x] = 0;
   for j := 0 step 1 until b1 do
   if start[j] ≠ 0 then
     begin
     list[x] := start[j]; x := last[j]
   end;
   list[x] := 0
     end. The items are now in a single list;
 j := 0;
 for j := list[j] while j ≠ 0 do use(a[j])
 end
```

Algorithm 44

SOLUTION OF NONLINEAR SIMULTANEOUS EQUATIONS

C. G. Broyden
University of Essex

Author's Note:

The two algorithms give alternative methods of solving nonlinear simultaneous equations using a particular form of Broyden's method used in conjunction with a particular form of Davidenko's method. The method is described in Broyden (1969). *nonlinb* is a longer, more complicated procedure and is intended to be used in cases where *nonlina* does not work.

Reference

Broyden, C. G. (1969). A New Method of Solving Nonlinear Simultaneous Equations, *The Computer Journal*, Vol. 12, No. 1, pp. 94–99.

```
procedure nonlina(equs, order, tol, maxf, FAIL, type);
value equs, order, tol, maxf; real tol;
integer equs, order, maxf, type; label FAIL;
```

comment *this procedure solves the nonlinear simultaneous equations $f[i](x[1], x[2], \dots, x[n]) = 0$, $i = 1, 2, \dots, n$. It assumes that two $n \times 1$ arrays, x and f, and a procedure computef(equs, FAIL) have already been declared. Using the contents of array x as data the procedure computef should calculate the appropriate residuals and assign these to the array f. Before calling nonlina initial values of the elements of x should have been assigned. Of the two formal parameters of computef the first, an integer, indicates which set of nonlinear equations is to be solved, i.e. which particular mapping of x onto f should be carried out by computef. This enables any one of an arbitrary number of sets of nonlinear equations to be solved. The second formal parameter, FAIL, is a label to which control should be transferred in the event of failure during execution of computef. The formal parameters of nonlina are as follows:*

(1) *equs (integer), fulfils the same function as the first formal parameter of computef.*
(2) *order (integer), the number of equations and unknowns in the set of equations selected by equs.*
(3) *tol (real), the largest acceptable value of $||f|| \uparrow 2$.*
(4) *maxf (integer), the maximum acceptable number of evaluations of f.*
(5) *FAIL (label), the label of the failure exit.*
(6) *type (integer). See below.*

The integer type is a guide to the kind of failure that may have occurred. It assumes the following values:

0. *No failure.*
1. *Maximum number of function evaluations exceeded. Possible causes: tol or maxf too small, problem too nonlinear, initial matrix too inaccurate. Possible action: inspect $||f||$ and if small increase either maxf or tol. If $||f||$ large, use nonlinb.*
2. *Division by zero while updating h, store elements of f in different order and if this fails use nonlinb.*
3. *Failure in computef. Use nonlinb.*
4. *Division by zero while initialising h. Compute elements of f in different order.*
5. *Failure in computef while initialising h. Choose improved initial estimate of solution;*

```
  begin real sa, sb; integer i, j, k, fcount;
  array y, p, v[1 : order], h[1 : order, 1 : order];
  procedure step(F1, F2); label F1, F2;
    begin
    for i := 1 step 1 until order do
      begin
      x[i] := x[i] + p[i]; v[i] := f[i];
      end;
    computef(equs, F1); fcount := fcount + 1;
    for i := 1 step 1 until order do y[i] := f[i] − v[i];
    sa := 0;
    for i := 1 step 1 until order do
      begin
      sb := 0;
      for j := 1 step 1 until order do sb := sb + h[i, j] * y[j];
      v[i] := sb − p[i]; sa := sa + sb * p[i]
      end calculation of hy − p and phy;
    if sa = 0 then goto F2;
    for j := 1 step 1 until order do
      begin
      sb := 0;
      for i := 1 step 1 until order do sb := sb + p[i] * h[i, j];
      sb := sb/sa;
      for i := 1 step 1 until order do h[i, j] := h[i, j] − sb * v[i]
      end of modification of h
    end of procedure step;
  type := 0;
  computef(equs, F5); fcount := 1;
  for i := 1 step 1 until order do
    begin
    p[i] := 0; h[i, i] := 1·0;
    for j := i + 1 step 1 until order do h[i, j] := h[j, i] := 0
    end of initialisation;
  for k := 1 step 1 until order do
    begin
    p[k] := 0·001; step(F5, F4);
    p[k] := 0
    end of calculation of initial iteration matrix;
REPEAT: for i := 1 step 1 until order do
    begin
    sa := 0;
    for j := 1 step 1 until order do sa := sa − h[i, j] * f[j];
    p[i] := sa
    end calculation of step vector p;
  step(F3, F2); sa := 0;
  for i := 1 step 1 until order do sa := sa + f[i] * f[i];
  if sa < tol then goto EXIT;
  if fcount ⩾ maxf then goto F1;
  goto REPEAT;
F5: type := 5; goto FAIL;
F4: type := 4; goto FAIL;
F3: type := 3; goto FAIL;
F2: type := 2; goto FAIL;
F1: type := 1; goto FAIL
EXIT: end of procedure nonlina;
```

```
procedure nonlinb(equs, order, tol, maxf, maxint, lamda, FAIL,
   type);
value equs, order, tol, maxf, maxint, lamda; real tol, lamda;
integer equs, order, maxf, maxint, type; label FAIL;
```

comment *this procedure is used in an identical manner to nonlina. The formal parameters of nonlinb are the same as the corresponding ones of nonlina with the following exceptions:*

(1) *maxf (integer), the maximum number of function evaluations permitted for solving each intermediate problem excluding those required to establish the initial approximation to the iteration matrix.*
(2) *maxint (integer), the maximum permitted number of intermediate problems excluding the first two.*
(3) *lamda (real). See Broyden (1969). A good value to start with is 0·5.*

The integer type assumes the following values:

0. *No failure.*
1. *Permitted number of intermediate problems exceeded.*
2. *maxf exceeded. Possible causes: tol or maxf too small or lamda too large. If $||f||$ small either increase tol or increase maxf. If $||f||$ large reduce lamda.*
3. *Division by zero when updating h. Reduce lamda.*
4. *Failure in computef. No satisfactory automatic remedy. A new set of initial values could be tried or lamda could be reduced.*
5. *As 2 but occurrence during first two intermediate problems. Possible cause, initial iteration matrix inaccurate. Possible remedy, store elements of f in a different order.*
6. *As 3 but during first two problems. Remedy as 5.*
7. *As 4 but during first two problems. Choose a new initial solution;*

```
  begin real s, sa, sb, sc, s1, s2, theta, theta1, theta2;
  array g, y, p, v, v1, x1, x2[1 : order], h[1 : order, 1 : order];
  integer i, j, k, fcount, intcount;

  procedure step(F1, F2); label F1, F2;
    begin
    for i := 1 step 1 until order do
      begin
```

```
        x[i] := x[i] + p[i]; v[i] := f[i]
      end;
    computef(equs, F1); fcount := fcount + 1;
    for i := 1 step 1 until order do y[i] := f[i] - v[i];
    sa := 0;
    for i := 1 step 1 until order do
      begin
        sb := 0;
        for j := 1 step 1 until order do sb := sb + h[i, j] * y[j];
        v[i] := sb - p[i]; sa := sa + sb * p[i]
      end calculation of hy - p and phy;
    if sa = 0 then goto F2;
    for j := 1 step 1 until order do
      begin
        sb := 0;
        for i := 1 step 1 until order do sb := sb + p[i] * h[i, j];
        sb := sb / sa;
        for i := 1 step 1 until order do h[i, j] := h[i, j] - sb * v[i]
      end of modification of h
    end of procedure step;

procedure inival;
    begin
    theta2 := theta1; theta1 := theta;
    theta := sa;
    for i := 1 step 1 until order do
      begin
        sa := s * x[i] + s1 * x1[i] + s2 * x2[i]; x2[i] := x1[i];
        x1[i] := x[i]; x[i] := sa;
        g[i] := f[i] * theta / theta1
      end of calculation of new initial values
    end of procedure inival;

procedure jacobian(F1, F2); label F1, F2;
    begin
    computef(equs, F1);
    for k := 1 step 1 until order do
      begin
        p[k] := 0; v1[k] := f[k]
      end;
    for k := 1 step 1 until order do
      begin
        p[k] := 0·001; step(F1, F2);
        p[k] := 0
      end of main loop;
    for k := 1 step 1 until order do
      begin
        x[k] := x[k] - 0·001; f[k] := v1[k]
      end
    end of procedure jacobian;

procedure solve (F1, F2, F3); label F1, F2, F3;
    begin
    fcount := 0;
 REPEAT: for i := 1 step 1 until order do
      begin
        sa := 0;
        for j := 1 step 1 until order do
        sa := sa - h[i, j] * (f[j] - g[j]); p[i] := sa
      end of calculation of step vector p;
    step(F1, F2); sa := 0;
    for i := 1 step 1 until order do
      begin
        sb := f[i] - g[i]; sa := sa + sb * sb
      end of calculation of norm;
    if sa < tol then goto EXIT;
    if fcount ⩾ maxf then goto F3;
    goto REPEAT
 EXIT: end of procedure solve;

 type := 0;
 for i := 1 step 1 until order do
      begin
      h[i, i] := 1·0;
      for j := i + 1 step 1 until order do
      h[i, j] := h[j, i] := 0
      end of setting up unit matrix;
 jacobian(F1, F2); theta := 1·0;
 sa := 0·99; s := 1·0;
 s1 := s2 := 0;
 comment set up data for inival;
 inival; solve(F1, F2, F3);
 sa := 0·98; s := 2·0;
 s1 := - 1·0;
 comment set up data for inival;
 inival; computef(equs, F1);
 solve(F1, F2, F3); intcount := 0;
 REPEAT: s := (theta2 - theta) * (theta - theta1);
 s1 := (theta - theta1) * (theta1 - theta2);
 s2 := (theta1 - theta2) * (theta2 - theta);
 sa := sb := 0;
 for i := 1 step 1 until order do
      begin
      sc := x[i] / s + x1[i] / s1 + x2[i] / s2; sa := sa + sc * sc;
      sc := (theta - theta1)* x2[i] / s2 + (theta - theta 2)* x1[i]
        / s1 + (2·0 * theta - theta1 - theta2) * x[i] / s;
      sb := sb + sc * sc
      end of calculation of vector norms;
 sa := theta - lamda * sqrt(sb) / sqrt(sa);
 comment next value of theta;
 if sa < 0 then sa := 0;
 s := (theta2 - sa) * (sa - theta1) / s;
 s1 := (theta - sa) * (sa - theta2) / s1;
 s2 := (theta1 - sa) * (sa - theta) / s2;
 inival; jacobian(F4, F5);
 solve(F4, F5, F6);
 if theta = 0 then goto EXIT;
 intcount := intcount + 1;
 if intcount ⩾ maxint then goto F7;
 goto REPEAT;
 F1: type := type + 1;
 F2: type := type + 1;
 F3: type := type + 1;
 F4: type := type + 1;
 F5: type := type + 1;
 F6: type := type + 1;
 F7: type := type + 1; goto FAIL
 EXIT: end of procedure nonlinb;
```

Notes on Algorithms 25, 26

25 SORT A SECTION OF THE ELEMENTS OF AN ARRAY BY DETERMINING THE RANK OF EACH ELEMENT

26 ORDER THE SUBSCRIPTS OF AN ARRAY SECTION ACCORDING TO THE MAGNITUDES OF THE ELEMENTS

(1) The strategy used in these sorting algorithms has the property that they become slower as more of the elements to be sorted are equal. In the limiting case when all the elements are equal, the time needed to sort is proportional to the square of the size. This criticism applies also to the algorithm 'Quickersort' (Scowen, 1965) and has been cured by R. C. Singleton (1969). The accompanying table illustrates this effect for 'Quickersort'.

(2) When the procedure 'keysort' is asked to sort an array with only one element, no assignment is made to the result array '*r*'. This error is easily cured by reordering the statements at the beginning of the procedure body; i.e.:

```
begin integer size, i, k;
  size := n - m + 1;
  comment initialize rank index vector;
```

```
for i := m step 1 until n do r[i] := i;
if  size ⩾ 2 then
    begin
    comment compute size of address arrays;
    k := 0;
    for i := 1, i + 1 while i < size do k := k + 1;
      begin integer j, p, ri, rj, rm, rn; real d;
      integer array f, g[1 : k];
      k := 1;
      comment deal with subsets of order 2 separately;
```

Table

n	t_1	r	t_2	t_3
100	0·29	23	0·47	1·39
200	0·71	29	1·32	5·26
500	1·94	83	5·87	31·9
1,000	4·33	188	12·8	128

n	number of elements in the sorted array.
t_1	time to sort array when all elements are different.
r, t_2	t_2 is the time to sort an array of size n with only r different elements.
t_3	time to sort an array of size n when all elements have the same value.

References

Boothroyd, J. (1967). Algorithms 25, 26, *The Computer Journal*, Vol. 10, pp. 308–310.

Scowen, R. S. (1965). Algorithm 271, Quickersort, *Communications of the ACM*, Vol. 8, pp. 669–670.

Singleton, R. C. (1969). Algorithm 347, An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, Vol. 12, pp. 185–187.

R. S. Scowen
National Physical Laboratory
Teddington

Note on Algorithm 40

SPLINE INTERPOLATION OF DEGREE THREE

This algorithm includes a jump to formal label *EXIT* if $n < 3$. But if $n < 3$ many Algol compilers will report a failure and terminate the program before the test is reached, since the array bounds of e are $[1 : n - 2]$ and the upper bound will be less than the lower bound.

The difficulty can be avoided by rewriting the first three lines after the comment as:

```
if n < 3 then goto EXIT else
begin integer i, j, n1, n2, k; real z, h1, h2, h3, h4;
array h, dy[1 : n], s[1 : n − 1], e[1 : n − 2];
```

I. D. Hill
MRC Computer Unit (London)
242 Pentonville Road
London N.1

Contributions for the Algorithms Supplement should be sent to

Mrs. M. O. Mutch
University Mathematical Laboratory
Corn Exchange Street
Cambridge