# Algorithms Supplement

## Previously published algorithms

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

(*a*) **Communications of the ACM** (July–September 1969)

352 CHARACTERISTIC VALUES AND ASSOCIATED SOLUTIONS OF MATHIEU'S DIFFERENTIAL EQUATION

*Consists of three primary and several secondary routines which compute characteristic values of Mathieu's differential equation, compute the associated solutions of this equation, and evaluate Bessel functions.*

353 FILON QUADRATURE

*Evaluates the integrals*

$$C = \int_0^1 F(X) \cos (M\pi X) dX$$

*and* $$S = \int_0^1 F(X) \sin (M\pi X) dX$$

*using the Filon quadrature algorithm.*

354 GENERATOR OF SPANNING TREES

*Finds all trees that span a nondirected graph on n nodes.*

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

(*a*) **Communications of the ACM** (July 1969)

RECOVERY OF REENTRANT LIST STRUCTURES IN SLIP (Vol. 12, No. 7, pp. 370–372)

## New algorithms

**Algorithm 45**

*AN INTERNAL SORTING PROCEDURE USING A TWO-WAY MERGE*

A. D. Woodall
Reading College of Technology

**Author's note:**

This program, which is very fast when used with data in random order, improves when there is some structure in the data, and is at its best when the data is already nearly in order (or reverse order).

The procedure uses a chaining technique to sort $n$ keys held as the elements of an array $a[1]$ to $a[n]$. The basis of the procedure is to form ordered lists, and then successively merge them into longer lists, until all the elements form a single ordered list.

At any stage, if the first item of the $j$th list is $a[i]$ then its index $i$ will be held as the value of $hds[j]$ where $hds$ is an integer array. The successor to any element $a[k]$ of a list is $a[s]$ where $s = list[k]$ — $list$ is another integer array. If the last element of a list is $a[r]$, then $list[r]$ will be zero.

Among the advantages of this method of defining lists is that two may be very simply linked: during the merging of two lists, if one is exhausted, the merge is at once complete.

A further advantage is that during the first pass, when any ordered sequences already present in the data are joined to form the first lists, these may be simply linked up in either direction. We start each list in the direction of its first two items and proceed as far as it continues to run.

Thus every list (except possibly the last) has at least two members. It follows that the array $hds$ must go from $hds[1]$ to $hds[n \div 2 + 1]$ for the worst possible case, and that the storage needed for the program (apart from the program itself and workspace) is for $2{\cdot}5n$ words.

To give an indication of its speed, the procedure was timed against Algorithm 26 'keysort' (Boothroyd, 1967), using the same data for both. The tests were run on an Elliott 803 computer, and the following times, in seconds, are typical:

| | random data | | part-ordered data |
|---|---|---|---|
| | $n = 100$ | $n = 200$ | $n = 100$ |
| Keysort | 35 | 77 | 58 |
| Mergesort | 31 | 70 | 21 |

In keysort, the first item of each segment was always taken. The time for part-ordered data would have been less if the middle item had been taken. However, it would not have been better than the time shown for random data.

**Reference**

Boothroyd, J. (1967). Algorithm 26, *The Computer Journal* Vol. 10, p. 309.

```
procedure mergesort (n, a, list, start); value n; integer n, start;
array a; integer array list;
comment n is the number of keys to be sorted. The actual
parameter corresponding to list should have bounds 1 to n. a[1]
to a[n] should contain the n keys to be sorted. After mergesort
has been called, a will be unchanged, start will have the index
of the start of the ordered list, so that a[start] is the smallest
key, and the remaining order is defined by list. To pick up the
items in order after calling mergesort is simple. For example
if print(x) is a procedure which prints the value of x on a new
line, then all of the items of a[i] would be printed in order by
the statement for i := start, list[i] while i ≠ 0 do print(a[i]);
  begin real try, next, at1, at2; integer j, k, t, nol, try1, try2;
  integer array hds[1 : n ÷ 2 + 1];
  comment from here to the label MERGE the first pass links
  adjacent items into ordered lists, using existing runs in the
  data. A list may be either forward, if the run is in order, or
  backward if it is in reverse order;
  j := t := 1; k := 2;
  try := a[1];
L2:next := a[k];
  if try > next then goto BACKWARD;
  hds[j] := t;
FORWARD:list[t] := k;
  if k = n then
    begin
    list[k] := 0; goto MERGE
    end;
  try := next; t := k;
  k := k + 1; next := a[k];
  if try ⩽ next then goto FORWARD;
  list[t] := 0;
L1:t := k; k := k + 1;
  j := j + 1;
  if t = n then
    begin
    hds[j] := n; list[n] := 0;
    comment the last list has one member;
    goto MERGE
    end;
  try := next; goto L2;
BACKWARD:list[t] := 0;
BW:list[k] := t;
  if k = n then
    begin
    hds[j] := k; goto MERGE
    end;
  try := next; t := k;
  k := k + 1; next := a[k];
  if try ⩾ next then goto BW;
  hds[j] := t; goto L1;
MERGE:if j = 1 then goto FIN;
  comment this would imply that the data was already in order
  or reverse order;
  nol := j;
  comment nol is the number of lists after each pass;
  for t := 1, t + t while t < nol do k := t;
  j := t := k + k + 1 − nol; goto LB;
  comment merging starts part-way through, at a point chosen
  to reduce the number of lists to a power of 2 after the first
  merging pass;
LA:t := 1; j := 1;
  comment lists starting from hds[t] and hds[t + 1] are merged
  into a list starting from hds[j];
LB:try1 := hds[t]; try2 := hds[t + 1];
  at1 := a[try1]; at2 := a[try2];
  if at1 ⩽ at2 then
    begin
    hds[j]:= try1; goto LL1
    end;
  hds[j]:= try2;
LL2:k := list[try2];
  if k = 0 then
    begin
    list[try2] := try1; goto EXIT
    end;
  at2 := a[k];
  if at1 < at2 then
    begin
    list[try2] := try1; try2 := k
    end
  else
    begin
    try2 := k; goto LL2
    end;
LL1:k := list[try1];
  if k = 0 then
    begin
    list[try1] := try2; goto EXIT
    end;
  at1 := a[k];
  if at2 < at1 then
    begin
    list[try1] := try2; try1 := k;
    goto LL2
    end;
  try1 := k; goto LL1;
EXIT:j := j + 1; t := t + 2;
  if nol > t then goto LB;
  nol := j − 1;
  if nol > 1 then goto LA;
FIN: start := hds[1]
  end
```

## Algorithm 46

*A MODIFIED DAVIDON METHOD FOR FINDING THE MINIMUM OF A FUNCTION, USING DIFFERENCE APPROXIMATION FOR DERIVATIVES*

Shirley A. Lill
Dept. of Computational Science
University of Leeds

**Author's note:**

Davidon's method minimises a function $f(x_1, x_2, \ldots x_n)$ by successive linear minimisations along chosen search directions, see Davidon (1959) and Fletcher & Powell (1963). The method, which is quadratically convergent, is very powerful, but the gradient vector $g(x)$ of $f(x)$ is required, and many functions exist for which the exact calculation of $g$ is either difficult or lengthy. In order to minimise such functions using Davidon's Method, Stewart (1967) proposed calculating $g$ by differences. Intervals $d_i$ for differencing $f(x)$ along each of the co-ordinate directions are recalculated at each iteration, using available information on the function and its derivatives. The formula for the $i$th component of the gradient is

$$g_i = \frac{f(x + d_i e_i) - f(x)}{d_i}$$

where $e_i$ is the vector whose $i$th component is unity and remaining components are zero. Following Stewart, if the 'simple difference formula' is predicted to be of low accuracy an alternative formula based on central differences is used. In this way only $n$ extra function evaluations are generally necessary to compute the gradient. To reduce the number of calculations of the gradient a linear minimisation is used in which function values are needed only at points away from the start of the search.

Results obtained with this method for the usual test functions compare very favourably with those of other methods not requiring derivatives. The ratio of the total number of function evaluations needed in Powell's (1964) method to the number required in this method increases with $n$, so that when $n = 10$ it is approximately 2.

In his paper Stewart gives the results of a FORTRAN IV coding of the method, using a linear minimisation on function evaluations only, which is similar to that used by Powell. However, in this algorithm use is also made of the gradient at the starting point of each search. On balance this linear minimisation is rather faster than that used by Stewart.

### References

Davidon, W. C. (1959). Variable metric method for minimization, A.E.C. Research and Development Report, ANL-5990 (Rev.).

FLETCHER, R., and POWELL, M. J. D. (1963). A rapidly convergent descent method for minimization, *The Computer Journal*, Vol. 6, p. 163.

POWELL, M. J. D. (1964). An efficient method of finding the minimum of a function of several variables without calculating derivatives, *The Computer Journal*, Vol. 7, p. 155.

STEWART III, G. W. (1967). A modification of Davidon's minimization method to accept difference approximations to derivatives, *JACM*, Vol. 14, p. 72.

```
procedure DAPODMIN(n, x, f, funct, monitor, dfirst);
value n; real f; integer n;
array x, dfirst; procedure funct, monitor;
comment DAPODMIN, function minimisation by a modification
of the Fletcher and Powell method to accept difference approxi-
mations of derivatives. On entry x[1 : n] is an estimate of the
position of the minimum, dfirst[1 : n] is the vector of step sizes
used in the initial approximation of the gradient and its com-
ponents should be set arbitrarily to, say, 1/20 of those of the
variable x. The statement funct(x, n, f) assigns to f the
function value at the point x. The output statement monitor
(n, x, f, g, count, h, evaluation) occurs once per iteration, count
is the iteration number, and evaluation is the number of function
evaluations.
A print out of all the parameters of monitor should only be
needed for diagnostic purposes if the iteration is failing to
converge. The final function value, the estimated position of
the minimum and the metric are in f, x and h. For simplicity
Jensens device is used in procedure up dot;
   begin
   real oldf, sg, yhy, sy, gx, fy, fz, a, b, c, min, fm, Ef;
   integer i, j, k, count, evaluation; Boolean check;
   array g, s, oldg, oldx, y, z, H[1 :n], h[1 :n × (n + 1) ÷ 2];
   real procedure dot(a, b); array a, b;
   comment inner product of a and b;
     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 i-th 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;
     for j := 1 step 1 until n do s := s + a[k + j − i] × b[j];
     up dot := s
     end of up dot;

   procedure grad (x, first, g); Boolean first; array x, g;
   comment calculate the gradient vector g by differences at the
   point x. If first is true the supplied intervals for differencing,
   dfirst, are used, otherwise intervals d are calculated;
     begin real E, d, estd, bd, fpd, fmd;
     integer i, j; array xplusd, xminusd[1 :n];
     for j := 1 step 1 until n do
       begin
       if first then
         begin
         d := dfirst[j];
         goto SIMPLE DIFFERENCES
         end;
 CALCULATE D:bd := abs (oldg[j] × x[j] / f)
       × 0·5 × 10 ↑ (− 12);
       E := if Ef ⩾ bd then Ef else bd;
       if oldg[j] ↑ 2 ⩾ abs (H[j] × f) × E then
         begin
         estd := 2 × sqrt(abs (f / H[j]) × E);
         d := estd × (1 − abs (H[j]) × estd /
         (3 × abs (H[j]) × estd + 4 × abs (oldg[j])))
         end
       else
         begin
         estd := 2 × exp (ln (abs (f × oldg[j])
          × E / H[j] ↑ 2) / 3);
         d := estd × (1 − 2 × abs (oldg[j])/
         (3 × abs (H[j]) × estd + 4 × abs (oldg[j])))
         end;
       comment If the relative truncation error for simple differ-
       ences is greater than some upper bound, say, 0·01
       calculate a new d and use central differences;
       if 0·5 × abs (H[j] × d / oldg[j]) > 0·01 then
         begin
         d := − abs (oldg[j] / H[j]) +
         sqrt (oldg[j] ↑ 2 + 200 × f × E × abs (H[j])) /
         abs (H[j]);
         for i := 1 step 1 until n do
         xplusd[i] := xminusd[i] := x[i];
         xplusd[j] := xplusd[j] + d;
         xminusd[j] := xminusd[j] − d;
         evaluation := evaluation + 2;
         funct(xplusd, n, fpd);
         funct(xminusd, n, fmd);
         g[j] := (fpd − fmd) / 2 / d;
         goto OMIT
         end of central differences;

 SIMPLE DIFFERENCES: for i := 1 step 1 until n do
         xplusd[i] := x[i];
         xplusd[j] := xplusd[j] + d;
         funct(xplusd, n, fpd);
         evaluation := evaluation + 1;
         g[j] := (fpd − f) / d;
       OMIT:end of jth component
     end of grad;

   procedure set unit h and H;
   comment form the unit matrix in h and the diagonal elements
   of its inverse in H;
     begin integer i, j, k;
     k := 1;
     for i := 1 step 1 until n do
       begin
       h[k] := 1; H[i] := 1;
       for j := 1 step 1 until n − i do h[k + j] := 0;
       k := k + h − i + 1
       end
     end of set;
   comment start of minimisation, Ef is an estimate of the error
   in evaluating the function, fm is the estimated minimum of f;
   Ef := ₁₀− 10; fm := 0·0;
   set unit h and H;
   evaluation := 1;
   funct (x, n, f);
   grad (x, true, g);
   monitor (n, x, f, g, 0, h, evaluation);
   for count := 1, count + 1 while oldf > f do
     begin
     for i := 1 step 1 until n do
       begin
       oldx[i] := x[i]; oldg[i] := g[i];
       s[i] := − up dot(h, g, i)
       end;

 SEARCH ALONG S: oldf := f;
     gx := dot(g, s);
```

```
      if count > 1 then
        begin
        check := gx ≥ 0;
        for i := 1 step 1 until n do
        check := check ∧ H[i] < 0;
        if check then
          begin
          set unit h and H;
          for i := 1 step 1 until n do s[i] := −g[i];
          gx := dot(g, s)
          end
        end of check on h and H;
      if count ≤ n then min := − 2 × (f − fm) / gx
      else if count = n + 1 then min := 1
      else if c / min > 2 then min := 2 × min
      else if c / min < 0·5 then min := min × 0·5;
      b := min; a := 0;
      for i := 1 step 1 until n do y[i] := x[i] + b × s[i];
      funct(y, n, fy);
      evaluation := evaluation + 1;
      if fy ≥ f then
        begin
        c := if f = fy then b / 2 else
        − gx × b ↑ 2 / 2 / (fy − f − gx × b);
        for i := 1 step 1 until n do
        x[i] := oldx[i] + c × s[i];
        funct(x, n, f);
        evaluation := evaluation + 1;
        if f ≥ oldf then
          begin
          if f = oldf ∧ oldf = fy then goto EXIT;
          if f = oldf then c := 0·5 × c
          else
            begin
            for i := 1 step 1 until n do
            y[i] := oldx[i] − c × s[i];
            funct(y, n, fy);
            evaluation := evaluation + 1;
            c := (fy − f) × c / 2 / (f + fy − 2 × oldf)
            end;
          for i := 1 step 1 until n do
          x[i] := oldx[i] + c × s[i];
          funct(x, n, f);
          evaluation := evaluation + 1
          end;
        goto END SEARCH
        end;

DOUBLE:c := if fy − f − b × gx ≤ 0 then 3 × b else
      − gx × b ↑ 2 / 2 / (fy − f − b × gx);
      if c > 3 × b then c := 3 × b;
      for i := 1 step 1 until n do z[i] := x [i] + c × s[i];
      funct(z, n, fz);
      evaluation := evaluation + 1;
      if fz > fy then
        begin
        if c = 3 × b ∨ a > 0 then
          begin
          c := a + (f × (b ↑ 2 − c ↑ 2) + fy × c ↑ 2
          − fz × b ↑ 2) / 2 / (f × (b − c) + fy × c
          − fz × b);
          for i := 1 step 1 until n do
          x[i] := oldx [i] + c × s[i];
          funct(x, n, f);
          evaluation := evaluation + 1
          end
        else
          begin
          f := fy; c := a + b;
          for i := 1 step 1 until n do x[i] := y[i]
          end;
          goto END SEARCH
          end;
        if c = 3 × b then
          begin
          f := fy; fy := fz;
          gx := 0;
          for i := 1 step 1 until n do
            begin
            x[i] := y[i]; y[i] := z[i];
            gx := gx + s[i] ↑ 2
            end;
          gx := (fy − f) / b × sqrt(gx);
          a := a + b; b := 2 × b;
          goto DOUBLE
          end
        else
          begin
          f := fz; c := a + c;
          for i := 1 step 1 until n do x[i] := z[i]
          end;

END SEARCH:grad(x, false, g);
        if count > n then
          begin
          check := true;
          for i := 1 step 1 until n do
          check := check ∧ abs(s[i]) < ₁₀ − 6 ∧
          abs (c × s[i]) < ₁₀ − 6;
          if check then goto EXIT
          end;
        for i := 1 step 1 until n do oldx[i] := x[i] − oldx[i];
        for i := 1 step 1 until n do y[i] := g[i] − oldg[i];
        sy := dot(oldx, y);
        for i := 1 step 1 until n do s[i] := up dot(h, y, i);
        yhy := dot(s, y);
        k := 1;
        if sy = 0 ∨ yhy = 0 then goto PRINT;
        for i := 1 step 1 until n do
        for j := i step 1 until n do
          begin
          h[k] := h[k] + oldx[i] × oldx[j] / sy
          −s[i] × s[j] / yhy;
          k := k + 1
          end of updating of h;
        sg := dot(oldg, oldx);
        for i := 1 step 1 until n do
        H[i] := H[i] + y[i] × (y[i] × (1 − sg × c / sy)
        + 2 × c × oldg[i]) / sy;
        comment end of updating of H;
   PRINT:monitor(n, x, f, g, count, h, evaluation)
        end of iteration loop;
EXIT: end of DAPODMIN;
```

**Algorithm 47**

## *A CLUSTERING ALGORITHM*

C. J. van Rijsbergen
King's College Research Centre
King's College, Cambridge

**Author's note:**

This subroutine is designed to produce an unstratified hierarchy of clusters. Given a set $U = \{1, 2, \ldots, n\}$ of $n$ objects, and an $n \times n$ dissimilarity matrix $A = [a_{ij}]$, the subroutine produces subsets $C$ of $U$ which satisfy

$$\max \{a_{ij} : i, j \in C\} < \min \{a_{lk} : l \in C\} \quad \text{for all} \quad k \notin C.$$

This will be referred to as the $L_1$-condition. Given any two clusters satisfying the $L_1$-condition, either one is a subset of the other, or they are disjoint (see Jardine, 1969).

The $L_1$-condition is a strong condition, since such clusters are neighbourhoods of all their points. They are therefore useful in classificatory problems in which highly homogeneous clusters are required.

On entry the subroutine expects to find the elements of $A = [a_{ij}]$ for which $i < j$ stored linearly by columns in the vector $Z$. A corresponding vector $IJ$ holding the indices such that $Z(k) = a_{ij}$ if and only if $IJ(k) = (i \times 1000 + j)$ must be set up. $NDAT = n(n-1)/2$ where $n$ is the order of the matrix $A$.

The vectors $IA$, $IB$ are solely concerned with the output of the clusters. After subroutine $SORT1$ has been called vector $NTIE$ becomes a binary vector such that $NTIE(k) = 1$ if and only if $Z(k) = Z(k+1)$; otherwise $NTIE(k) = 0$. The vector $Y$ identifies the clusters and their contents.

The subroutine generates a series of subsets $C_p$ of $U$ defined as follows. Let $Q(k) = \{i, j\}$ where $IJ(k) = (i \times 1000 + j)$. Define

$$C_p = \cup\{Q(k) : Y(k) = p, k = 1, \ldots, NDAT\}.$$

It does this by considering each $Q(k)$ for $k = 1, \ldots, NDAT$ in turn. For each pair it performs the following check

C1 Is there a $k$ such that $i \in C_k$?

C2 Is there an $l$ such that $j \in C_l$?

If both are true and $k = l$ then no action is taken. If both are true and $k \neq l$ then a new set $C_m = (C_k \cup C_l)$ is formed. If one is true, say C1, then $C_m = (C_k \cup \{i, j\})$. If neither is true, then $C_m = \{i, j\}$. New subsets are checked for the $L_1$-condition, which in the last case is trivially satisfied.

The calling program must contain the following.

*DIMENSION Z(NDAT), Y(NDAT), IJ(NDAT),*
*1IA(NDAT), NTIE(NDAT), IB(N)*
*EQUIVALENCE* (*Z*(1), *Y*(1))
*INTEGER Y*

where $N = n$.

The program has been used extensively on taxonomic data· In particular, it has been used to guess evolutionary trees (see Jardine, van Rijsbergen, and Jardine (1969)). For a 20 × 20 dissimilarity matrix the program takes about 30 seconds computation time on the TITAN computer at the Cambridge University Mathematical Laboratory.

## References

JARDINE, N. (1969). Towards a general theory of clustering (Abstract), *Biometrics*, Vol. 25, p. 609.

JARDINE, N., VAN RIJSBERGEN, C. J., and JARDINE, C. J. (1969). Evolutionary rates and the influence of evolutionary tree-forms, *Nature, Lond.*, Vol. 224, p. 185.

```
      SUBROUTINE CLUST(Z,Y,IJ,IA,IB,NDAT,NTIE)
      DIMENSION Z(1),Y(1),IJ(1),IA(1),IB(1),NTIE(1)
      INTEGER Y
      LOGICAL TYPE
      CALL SORT1(Z,IJ,NDAT,NTIE)
      DO 1 LB=1,NDAT
1     Y(LB)=0
      KN=0
2     KN=KN+1
      IF(KN.GT.NDAT)GOTO 4
      IF(Y(KN).NE.0)GOTO 2
      Y(KN)=KN
      CALL PREV(Y,IJ,KN,M,L)
      CALL FORW(Y,IJ,KN,M,L,TYPE,NDAT)
      IF(.NOT.TYPE)GOTO 3
      CALL OUTPUT(Y,IJ,IA,IB,NDAT,KN,NTIE)
3     IF(KN.LT.NDAT)GOTO 2
4     RETURN
      END
```

```
      SUBROUTINE PREV(Y,IJ,KN,M,L)
      DIMENSION Y(1),IJ(1)
      INTEGER Y
      LOGICAL L1,L2,L3
      IN=IJ(KN)/1000
      JN=IJ(KN)-IN*1000
      M=0
      L=0
      IF(KN.EQ.1)RETURN
      DO 4 LL=1,KN-1
      KP=KN-LL
      MT=Y(KP)
      L1=(MT.EQ.M).OR.(MT.EQ.L)
      IF(L1)GOTO 3
      IP=IJ(KP)/1000
      JP=IJ(KP)-IP*1000
      L2=(IP.EQ.IN).OR.(JP.EQ.IN).OR.(IP.EQ.JN).OR.(JP.EQ.JN)
      L3=.NOT.L2
      IF(L3)GOTO 4
      IF(M)2,1,2
1     M=MT
      GOTO 3
2     L=MT
3     Y(KP)=KN
4     CONTINUE
      DO 5 KK=1,KN-1
      IF(Y(KK).EQ.M)Y(KK)=KN
      IF(Y(KK).EQ.L)Y(KK)=KN
5     CONTINUE
      RETURN
      END
```

```
      SUBROUTINE FORW(Y,IJ,KN,M,L,TYPE,NDAT)
      DIMENSION Y(1),IJ(1)
      COMMON/MAXD/MAXD
      INTEGER Y
      LOGICAL TYPE,L1,L2,L3,L4,M1,M2
      MAXD=KN
      MIND=NDAT
      IF(M+L.EQ.0)GOTO 7
      DO 6 KF=KN+1,NDAT
      M1=.FALSE.
      M2=.FALSE.
      IF(Y(KF))5,1,5
1     IF=IJ(KF)/1000
      JF=IJ(KF)-IF*1000
      DO 2 LA=1,KN
      KP=KN-LA+1
      IF(Y(KP).NE.KN)GOTO 2
      IP=IJ(KP)/1000
      JP=IJ(KP)-IP*1000
      L1=(IF.EQ.IP).OR.(IF.EQ.JP)
      L2=(JF.EQ.IP).OR.(JF.EQ.JP)
      IF(L1)M1=.TRUE.
      IF(L2)M2=.TRUE.
2     CONTINUE
      IF(M1.AND.M2)GOTO 3
      IF(M1.OR.M2)GOTO 4
      GOTO 6
3     Y(KF)=KN
      MAXD=KF
      GOTO 6
4     IF(MIND.EQ.NDAT)MIND=KF
      GOTO 6
5     L3=(Y(KF).EQ.M).OR.(Y(KF).EQ.L)
      L4=.NOT.L3
      IF(L4)GOTO 6
      Y(KF)=KN
      MAXD=KF
6     CONTINUE
7     IF(MAXD.GT.MIND)GOTO 8
      TYPE=.TRUE.
      RETURN
8     TYPE=.FALSE.
      RETURN
      END
```

```
      SUBROUTINE SORT1(Y,IJ,NDAT,NTIE)
      DIMENSION Y(1),IJ(1),NTIE(1)
      COMMON/MAXD/MAXD
      DO 1 LL=1,NDAT
1     NTIE(LL)=0
      K=NDAT
2     DO 4 I=1,K-1
      IF(Y(I)-Y(I+1))4,4,3
3     T1=Y(I)
      I2=IJ(I)
      Y(I)=Y(I+1)
      IJ(I)=IJ(I+1)
      Y(I+1)=T1
      IJ(I+1)=I2
4     CONTINUE
      K=K-1
      IF(K-1)5,5,2
5     DO 7 J=1,NDAT
      IF(Y(J)-Y(J+1))7,6,7
6     NTIE(J)=1
7     CONTINUE
      RETURN
      END
      SUBROUTINE SORT2(IA,J)
      DIMENSION IA(1)
      K=J
      IF(K.EQ.1)RETURN
1     DO 3 I=1,K-1
      IF(IA(I)-IA(I+1))3,3,2
2     T1=IA(I)
      IA(I)=IA(I+1)
      IA(I+1)=T1
3     CONTINUE
      K=K-1
      IF(K-1)4,4,1
4     RETURN
      END

      SUBROUTINE OUTPUT(Y,IJ,IA,IB,NDAT,N,NTIE)
      DIMENSION Y(1),IJ(1),IA(1),IB(1),NTIE(1)
      COMMON/MAXD/MAXD
      INTEGER Y
      J=0
      DO 1 I=1,NDAT
      IF(Y(I).NE.N)GOTO 1
      J=J+1
      IA(J)=IJ(I)
1     CONTINUE
      CALL SORT2(IA,J)
      I1=IA(1)/1000
      L=1
      IB(1)=I1
      IF(J.EQ.1)GOTO 3
      DO 2 I=2,J
      I2=IA(I)/1000
      IF(I2.EQ.I1)GOTO 2
      L=L+1
      IB(L)=I2
      I1=I2
2     CONTINUE
3     L=L+1
      IB(L)=IA(J)-I1*1000
      NNN=MAXD
4     IBIN=NTIE(NNN)
      IF(IBIN)5,8,5
5     NNN=NNN+1
      IF(Y(NNN))4,6,4
6     ITIE=IJ(NNN)/1000
      JTIE=IJ(NNN)-ITIE*1000
      DO 7 LL=1,L
      IIB=IB(LL)
      IF((ITIE.EQ.IIB).OR.(JTIE.EQ.IIB))RETURN
7     CONTINUE
      GOTO 4
8     WRITE(0,9)
9     FORMAT(///,'A TYPE IS')
      WRITE(0,10)(IB(K),K=1,L)
10    FORMAT(1H ,10X,14I4)
      RETURN
      END
```

## Note on Algorithm 42

### *INTERPOLATION BY CERTAIN QUINTIC SPLINES*

There is a USASI FORTRAN error in Algorithm 42 as published: dimensioning the arrays, X, Y, Y1, A, B, C, D as having length 1 should force any value greater than 1 for an actual subscript expression to be rejected (see section 7.2.1.1 in the standard). In fact some compilers (IBM 1130 FORTRAN for example) even refuse to compile the routine due to the use of X(2) and Y(2) in statement number 1 and its successor.

This usage is quite widespread (at least on IBM machines) but is quite definitely non-standard. The desired effect is not obtainable in USASI Basic FORTRAN, but can be achieved in Standard FORTRAN by replacing the first four non-comment statements by:

```
    SUBROUTINE QUINT (N, N1, X, Y, Y1, Y21,
   1Y2N, A, B, C, D)
    DIMENSION X(N), Y(N), Y1(N), A(N1), C(N1),
   1D(N), B(N1)
    IF (N - N1 - 1) 13, 14, 13
14  D(1) = 0·5 * Y21
```

A check that in fact N1 = N − 1 has been inserted to prevent hidden errors from occurring. An alternative method is to agree that all the formal arrays are dimensioned to N, even though only the first (N − 1) elements may be needed for some. In this case only the second statement of Algorithm 42 needs to change, and this becomes:

```
DIMENSION  X(N), Y(N), Y1(N), A(N), B(N), C(N), D(N)
```

A. H. J. Sale
Computer Centre
University of Natal
Durban

Contributions for the Algorithms Supplement should be sent to

**Mrs. M. O. Mutch**
**University Engineering Department**
**Control Engineering Group**
**Mill Lane, Cambridge**