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

$$\sum_{j=1}^{n} a_{i,j}\, x_j = b_i \text{ for } i = 1, 2, \ldots, m,\ m \geq n,$$

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

$$\sum_{j=1}^{n} \phi_j(t_i)\alpha_j = y_i \text{ for } i = 1, 2, \ldots, m.$$

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,TOLER,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=10**(-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             0 - 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
```

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 $(K-1)*DIMEN +$ Ith 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.

Funds for computer time used in development of this algorithm were provided by National Science Foundation Grant GJ561.

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
C THE ARRAYS EDGE ST, EDGE PT, AVE, SD, 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 ARRAY  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 ARRAY DATA.
      INTEGER DIM, N, MST(800), 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, NBR, 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 (NBR)
C FIELD 3: WEIGHT OF THIS BRANCH (WT)
C FIELD 4: POINTER TO NEXT NEIGHBOR OR END MARK=0 (NXT)
C EACH ADDITIONAL ELEMENT OF THE LIST CONSISTS OF THREE
C FIELDS.  FIELD 1 ABOVE IS OMITTED.
C THE LENGTH OF THE ARRAY MST MUST BE AT LEAST 6*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 NEAREST
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 ARRAY UI AND
C THE NUMBER OF COLUMNS OF NIT CANNOT BE LESS THAN N.
      INTEGER NIT(2,100)
      REAL UI(100)
      DIM = DIMEN
      N = NUMPTS
C COMPUTE MINIMAL SPANNING TREE USING ALGORITHM OF WHITNEY
C INITIALIZE NODE LABEL ARRAYS AND SET UP LIST FOR NODE N=KP
      NITP = N - 1
      KP = N
      KPDATA = (KP-1)*DIM + 1
      DO 10 I=1,NITP
        IDATA = (I-1)*DIM + 1
        NIT(1,I) = I
        UI(I) = DIST(DATA(IDATA),DATA(KPDATA),DIM)
        NIT(2,I) = KP
   10 CONTINUE
      FREE = N + 1
      MST(KP) = FREE
      LOC(FREE) = (KP-1)*DIM + 1
      FREE = FREE + 1
      NXT(FREE+2) = 0
C UPDATE LABEL OF NODES NOT YET IN TREE.
   20 KPDATA = (KP-1)*DIM + 1
      DO 30 I=1,NITP
        IDATA = (NIT(1,I)-1)*DIM + 1
        D = DIST(DATA(IDATA),DATA(KPDATA),DIM)
        IF (UI(I).LE.D) GO TO 30
        UI(I) = D
        NIT(2,I) = KP
   30 CONTINUE
C FIND NODE OUTSIDE TREE NEAREST TO TREE
      UK = UI(1)
      DO 40 I=1,NITP
        IF (UI(I).GT.UK) GO TO 40
        UK = UI(I)
        K = I
   40 CONTINUE
C ADD NEW EDGE TO MST
C ADD NEIGHBOR TO LIST OF NODE NIT(2,K)
C CHANGE END OF LIST MARK TO POINT TO NEXT NEIGHBOR
      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(FREE+2) = 0
      FREE = FREE + 3
C NEW NODE--CREATE ITS NEIGHBOR LIST
C SET UP HEAD POINTER
      NODE = NIT(1,K)
      MST(NODE) = FREE
C ENTER LOCATION OF THIS NODE IN DATA
      LOC(FREE) = (NODE-1)*DIM + 1
C ENTER NAME OF NEIGHBORING NODE (OFFSET PICKS UP NBR 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) = 0
      FREE = FREE + 4
      KP = NIT(1,K)
C DELETE NEW  TREE NODE FROM ARRAY 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.0) GO TO 20
      IF (PRINT) CALL PRTREE
      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 CLUSTERS 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. AVERAGE +  SPREAD  * STANDARD 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(101), EDGELN, EDGEPT(101)
      REAL AVE(100), SQ(100), SUPPWT, W
      INTEGER NUMNEI(100)
      INTEGER NEIGST(20), NEIGLN, NEIGPT(20)
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 20 NODE=1,N
        NUMNEI(NODE) = 1
        K = MST(NODE)
        AVE(NODE) = WT(K+2)
        SQ(NODE) = WT(K+2)**2
        K = NXT(K+3)
   10   IF (K.EQ.0) GO TO 20
        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 10
   20 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 TREE 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
   30 CALL CLIMB(EDGEPT, EDGEST, EDGELN, N)
      IF (EDGELN.LE.2) GO TO 70
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 = 0
      AV = 0.
      STDDEV = 0.
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 50 TO OTHEND
C GO OUT TO DEPTH D-1 ALONG BRANCHES NOT YET ADDED
   40 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, (50,60)
      NEIGLN = NEIGLN - 1
      GO TO 40
C INITIALZE NEIG ST TO SUM WEIGHTS HEADING OFF NODE A
   50 NEIGLN = 2
```

```
      NEIGST(1) = B
      NEIGPT(1) = FINDCN(B,A)
      NEIGST(2) = A
      NEIGPT(2) = -1
      ASSIGN 60 TO OTHEND
      GO TO 40
C TEST BRANCH A--B FOR INCONSISTENCY.
   60 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 30
C BRANCH A--B IS INCONSISTENT.  DELETE IT.
      NBR(K) = -IABS(NBR(K))
      K = NEIGPT(1)
      NBR(K) = -IABS(NBR(K))
      GO TO 30
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.
   70 NUMIN = 0
      CLS = 0
      CP = 1
      K = MST(1)
      NXTCLS = N
      INCLS(NXTCLS) = LOC(K)/DIM + 1
      PARENT(NXTCLS) = 0
      BAKWRD = 0
C START CLUSTER WITH NEXT AVAILABLE UNUSED NODE
   80 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
   90 K = MST(NODE) + 1
C ADD NEIGHBOR TO CLUSTER AND RECORD IT ANCESTRY
  100 NXTNBR = NBR(K)
      IF (NXTNBR.LT.0) GO TO 110
      IF (NXTNBR.EQ.BAKWRD) GO TO 120
      NUMIN = NUMIN + 1
      INCLS(NUMIN) = NXTNBR
      PARENT(NUMIN) = NODE
      GO TO 120
C THIS NEIGHBOR IS IN A DIFFERENT CLUSTER--ADD TO UNUSED
  110 NXTNBR = -NXTNBR
      IF (NXTNBR.EQ.INLIST) GO TO 120
      NXTCLS = NXTCLS - 1
      INCLS(NXTCLS) = NXTNBR
      PARENT(NXTCLS) = NODE
C GET NEXT NEIGHBOR
  120 K = NXT(K+2)
      IF (K.NE.0) GO TO 100
C ADD LIST OF NEIGHBORS OF NEXT ELEMENT OF THIS CLUSTER
      NXTCN = NXTCN + 1
      IF (NXTCN.GT.NUMIN) GO TO 130
      NODE = INCLS(NXTCN)
      BAKWRD = PARENT(NXTCN)
      GO TO 90
C END OF CLUSTER--DO OUTPUT
  130 CALL STORE(NUMIN-BEGCLS+1, C, CP, CLEN)
      IF (PRINT) WRITE (6,99999) CLS
      DO 140 I=BEGCLS,NUMIN
        IF (PRINT) WRITE (6,99997) INCLS(I)
        CALL STORE(INCLS(I), C, CP, CLEN)
  140 CONTINUE
      IF (NUMIN.LT.N) GO TO 80
      CP = 0
      CALL STORE(CLS, C, CP, CLEN)
      CALL FIXMST
      RETURN
99999 FORMAT(1H0/8H0CLUSTER, 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, 10HAND A SPRE,
     * 6HAD OF , G11.4, 21H STANDARD DEVIATIONS.)
99997 FORMAT(10X, 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 10 I=2,N
        DIST = DIST + (A(I)-B(I))**2
   10 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
   10 IF (LN.EQ.D+2) RETURN
      K = POINTR(LN)
      IF (K) 20, 30, 40
C SET POINTER TO FIRST NEIGHBOR OF TOP NODE
   20 NODE = STACK(LN)
      POINTR(LN) = MST(NODE) + 1
      GO TO 50
C BACK DOWN FROM TERMINAL NODE
   30 LN = LN - 1
C CLIMB OUT ON NEXT NEIGHBOR IF POSSIBLE
   40 POINTR(LN) = NXT(K+2)
      IF (POINTR(LN).EQ.0) RETURN
C CHECK DIRECTION
   50 K = POINTR(LN)
      NEIGHB = IABS(NBR(K))
      IF (NEIGHB.EQ.STACK(LN-1)) GO TO 40
C CLIMB OUT ON NEIGHBORING NODE
      LN = LN + 1
      STACK(LN) = NEIGHB
      POINTR(LN) = -1
      GO TO 10
      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
   10 IF (NXT(LASTPT).EQ.0) RETURN
      LASTPT = NXT(LASTPT) + 2
      GO TO 10
      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
   10 IF (IABS(NBR(FINDCN)).EQ.B) RETURN
      FINDCN = NXT(FINDCN+2)
      IF (FINDCN.NE.0) GO TO 10
      WRITE (6,99999) B, A
99999 FORMAT(5H0NODE, I3, 26H IS NOT A NEIGHBOR 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.0) RETURN
      LOC = LOC + 1
      IF (LOC.GT.N) GO TO 10
      ARRAY(LOC) = VALUE
      RETURN
   10 WRITE (6,99999) VALUE
99999 FORMAT(41H THE ARRAY USED TO STORE A DESCRIPTION OF/3H TH,
     * 30HE CLUSTERS IS NOT LONG ENOUGH /15H ITS NEXT VALUE,
     * 11H SHOULD BE , I10)
      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 20 NODE=1,N
        WRITE (6,99999) NODE
        K = MST(NODE) + 1
   10   WRITE (6,99998) NBR(K), WT(K+1)
        K = NXT(K+2)
        IF (K.NE.0) GO TO 10
   20 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 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.

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)
```

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

Fig. 1.

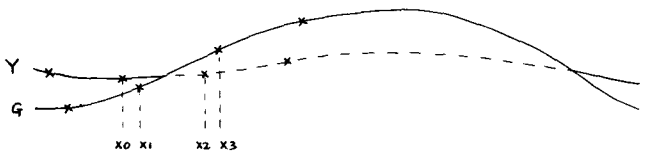


Fig. 2.

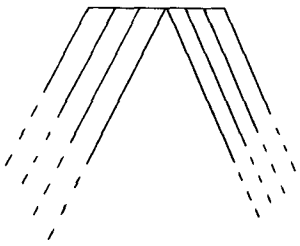


Fig. 3.

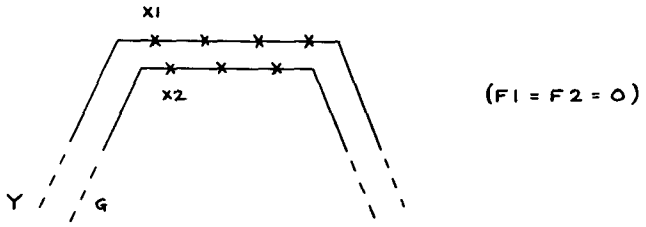
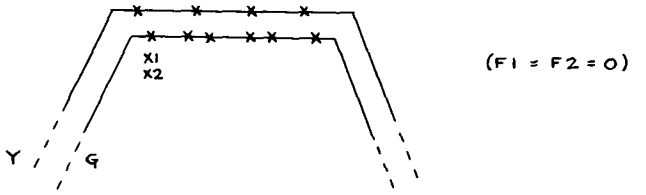


Fig. 4.



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.

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 $F1$ and $F2$ 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 $F1$ and $F2$ being zero is for the abscissa $(X1)$ 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 $X1$ corresponds to the first, and $X2$ 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 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).

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.

Scientific Applications — R.J. Hanson Editor

An Evaluation of Statistical Software in the Social Sciences

William D. Slysz
University of Connecticut

Several hundred college and university computer installations now offer various types of statistical packages for general use. Among those most widely available are OSIRIS, SPSS, BMD, DATA-TEXT, and TSAR. In order to provide users with a basis for selection and use, tests were made for each of these systems, and the results are summarized as to cost and performance.

Key Words and Phrases: statistical computation, statistical software, descriptive statistics, bivariate tables, Pearson correlation, regression, factor analysis, one-way analysis of variance

CR Categories: 1.3, 3.30, 4.19, 4.22, 4.49, 5.5

1. Introduction

There is little doubt that researchers, educators, and students have begun to make extensive use of general purpose computer software of the type recently developed in the social sciences for the management and analysis of research data. A cursory census can currently identify literally several hundred university and college computer installations making this software available. Schucany, Shannon, and Minton [1] have recently classified 37 software "packages" of this type, and Anderson [2] has assessed a number of these systems and libraries in terms of their value to undergraduate instruction. Allerbeck [3] has developed a comparative

This work was supported by the University of Connecticut Research Foundation and carried out at the Social Science Data Center and the University Computer Center, University of Connecticut. Author's address: Social Science Data Center, University of Connecticut, Storrs, CT 06268.