Algorithms supplement

Algorithm 96

A LINEAR EQUATION SOLVER FOR GALERKIN AND LEAST SQUARES METHODS

L. M. Delves
Department of Computational and Statistical Science
University of Liverpool
Liverpool
England

Author's Notes

The asymmetric or modified Galerkin method for the solution of a linear operator equation

$$\mathscr{L}f = g \tag{1}$$

in an inner product space, proceeds by setting

$$f \approx f_N = \sum_{i=1}^{N} a_i h_i \tag{2}$$

and computing $\mathbf{a} = (a_i)$ from the Galerkin equation

$$\mathbf{La} = \mathbf{g} \tag{3}$$

where

$$L_{ij} = (h_i', \mathscr{L}h_j); \; g_i = (h_i', g)$$

and $\{h_i\}$, $\{h_i'\}$ are 'suitably chosen' sequences of functions. The symmetric Galerkin technique chooses $\{h_i'\} = \{h_i\}$; the method of least squares follows from the choice $\{h_i'\} = \{\mathscr{L}h_i\}$. Such calculations are common in approximation theory problems ($\mathscr{L} = I$) and in the fields of differential and integral equations; it is therefore desirable to have efficient solution methods for the resulting linear equations. For a wide class of one-dimensional differential and integral equation problems, it has been shown (Freeman, Delves and Reid, 1974; Bain and Delves, 1977; Delves, 1974) that the use of orthogonal basis functions leads to matrices with a structure referred to as 'Asymptotic diagonality' (Delves, 1977). In Delves (1977), a block iterative algorithm was described for matrices with such a structure; the convergence rate of this algorithm is independent of N, giving a solution time of $\mathcal{O}(N^2)$; if all such problems led to such matrices, this algorithm would therefore be preferred to a direct solution. Unfortunately, this is not so; expansion methods with non-orthogonalised bases may lead to very ill-conditioned matrices. For example, the ill-conditioning of the least squares method with basis $\{1, x, x^2 \ldots\}$ is notorious, and for this reason least squares problems are usually discretised and solved as a set of overdetermined equations, the (normal) equations (3) never being formed. Recently, however, Barrodale and Stuart (1977) have introduced a new version of Gauss elimination with particular advantages for the solution of ill-conditioned expansion problems. In this variant, which uses both row and column pivoting, the unknowns yielding the largest residual are eliminated first, and a sensible solution to the approximation problem can be returned even if the matrix proves to be numerically singular.

We give here an implementation of the iterative scheme of Delves (1977) which maintains the advantages of the Barrodale algorithm and which can therefore be used for both well-conditioned and ill-conditioned problems.

The algorithm

The iterative scheme of (2) requires that the first M equations be partitioned off and solved directly with the current values of the remaining unknowns on the right hand side; a straightforward Gauss-Seidel scheme is used for the N-M additional equations. The value of M required depends on the matrix to be solved. In the algorithm given below, an initial sweep of the matrix is performed to choose an appropriate value of M; the $M \times M$ leading submatrix is triangulated using Gauss elimination, and the convergence of the iterations is then monitored. If convergence is too slow, or non-existent, the value of M is increased. Eventual convergence is therefore guaranteed for any matrix; if the problem proves well-conditioned, the time taken will be $\mathcal{O}(N^2)$, while if it is sufficiently ill-conditioned, the algorithm will eventually set $M = N$ and perform a direct solution of the problem.

The Gauss elimination routine used for the direct solution, employs the pivoting strategy of Barrodale and Stuart (1977), and is arranged to process additional rows and columns (if M is increased) and right hand sides (as the iterations proceed). For a well-conditioned approximation problem of significant size, M will be much less than N; experience to date indicates that the initial estimate of M is increased only rarely. If the matrix proves ill-conditioned, the routine may set $M = N$. If in this case the matrix is found after Q elimination stages to be numerically singular, the remaining N-Q unknowns are set to zero and a solution of the Q-term approximation problem returned, with a warning flag; note that the zero elements may not be the last few, due to the column pivoting employed. A singular matrix with $M < N$ merely results in M being increased and the triangulation restarted. Full iterative refinement with double length accumulation of the residuals is provided as an option for users with ill-conditioned problems of a type for which this is appropriate.

The routine returns a structure containing:

(*a*) The solution vector

(*b*) An estimate of the accuracy achieved

(*c*) The value of M used by the routine

(*d*) An error flag set to **false** if with $M = N$ the matrix was found to be numerically singular.

In this case the value of M returned is not N, but Q, the number of significant, nonzero entries in the solution vector and the numerical rank of the input matrix.

If on output $M = N$ and no iterative refinement has been called for, the error estimate will not be meaningful.

One other aspect of the algorithm is perhaps worthy of comment; this is the use of matrix operators *, +, − to multiply a matrix by a vector, and to add or subtract two vectors. In practice these will normally be available through a standard installation library prelude, and would then be machine coded. In the timings which follow we consider three versions of the algorithms:

Version A:	As given here, with matrix operators coded in ALGOL 68
Version B:	As given here, but using machine coded matrix operators
Version C:	A rewritten version with the operator calls removed and replaced by in-line ALGOL 68 code.

Code for the matrix operators is not included here.

Performance

Table 1 shows some timings obtained from the algorithm on an ICL 1906S computer. The equations solved are those arising from a Galerkin solution of the Fredholm integral equation (see, e.g. Delves, 1974)

$$f(x) = g(x) + \lambda \int_0^1 e^{\beta xy} f(y)\, dy \tag{4}$$

with

$$g(x) = e^{\alpha x} - \lambda[e^{\alpha+\beta x} - 1]/[\alpha + \beta x] \tag{5}$$

and exact solution $f(x) = e^{\alpha x}$. The expansion set used are the translated Chebychev polynomials; these lead to an asymptotically diagonal matrix L for all values of the parameter β, although the condition number of the matrix increases with $|\beta|$. In Table 1, values of $\beta = 1, 5, 20$ are taken, with $\lambda = \alpha = 1$. For each value of β, four sets of times are given. The first is that for a standard Gauss elimination routine (NAG library routine FO4AJB) using row interchanges, displayed for comparison purposes; the other three

Table 1 Solution of the linear equations arising from the integral equation (4)

All times are in msec and were obtained on an ICL 1906S using the RRE ALGOL 68R compiler
The versions of the routine are as follows:
    Version A As given here; matrix operators coded in ALGOL 68
    Version B As given here; matrix operators machine coded (partially)
    Version C Routine rewritten to replace matrix operator calls by inline (ALGOL 68) code

	$N =$	5	9	17	33	65
Routine FO4AJB (Gauss Elimination)	$T =$	15	36	125	616	3,780
$\beta = 1$;	$M =$	3	3	3	3	3
Version A	$T =$	63	56	79	189	509
Version B	$T =$	59	51	68	137	320
Version C	$T =$	35	40	66	184	621
$\beta = 5$;	$M =$	5	8	10	15	15
Version A	$T =$	19	48	117	195	726
Version B	$T =$	18	44	101	144	443
Version C	$T =$	16	36	95	191	871
$\beta = 20$;	$M =$	5	9	17	22	22
Version A	$T =$	19	58	170	783	1,094
Version B	$T =$	19	53	128	630	745
Version C	$T =$	17	52	156	720	1,202

relate to the three versions A, B, C of the algorithm as described above.

These timings show, first, the $\mathcal{O}(N^2)$ cost of the iterative algorithm, in contrast to the $\mathcal{O}(N^3)$ dependence of the standard Gauss elimination routine. The differences between versions A, B, and C show that the extra overheads of calling the matrix operators are negligible; the easy availability of the machine coded versions of these operators, and the substantial improvement which these yield, for the larger matrices, indicates that their use increases the efficiency as well as the clarity of the code.

Acknowledgement
I am grateful to Dr A. Hinxman for providing the machine code version of the matrix operators used to obtain timings for version B, and to D. Yates, J. M. Watt and M. A. Hennell for discussions of this work in general, and of operator packages in particular.

References

Barrodale, I. and Stuart, G. F. (1977). A New Variant of Gaussian Elimination, *JIMA*, Vol. 19, p. 39

Delves, L. M. (1977). On the Solution of the Linear Equations resulting from Ritz Galerkin Calculations, *JIMA*, to be published

Freeman, T. L., Delves, L. M., and Reid, J. K. (1974). On the Convergence of Variational methods II. Asymptotically Diagonal Systems of type B,C. *JIMA*, Vol. 14, pp. 145-157

Bain, M. and Delves, L. M. (1977). On the Optimum Choice of Weight Function in a class of Variational Calculations, *Num. Mathematik*, Vol. 27, pp. 209-218

Delves, L. M. (1974). A Black Box Galerkin procedure for the solution of linear Fredholm equations of the Second kind, Conference on Numerical Software, Purdue

Delves, L. M. (1977). A Fast Method for the solution of Fredholm integral equations, *JIMA*, to be published

```
   MODE MATRESULT =
      STRUCT ( REF [, ] REAL ans, REAL ep, INT evals, BOOL fin);
   PROC leqwad =
      ( REF [, ] REAL a, b, x, REAL acc, BOOL refine) MATRESULT :
      BEGIN
#solves the linear equations a•x=b for multiple righthand sides b[,i]#
#by a block iterative method.For a suitable value of m,the matrix a  #
#is partitioned.The first m equations are solved directly,and the    #
#remainder solved using a Gauss-Seidel scheme.The value of m is      #
#chosen to ensure rapid convergence of the iterations.               #
#   For wellconditioned approximation problems,the solution time for #
#n equations is o(n↑2).For sufficiently illconditioned problems,the  #
#method reduces to a direct solution using the pivotting strategy of #
#Barrodale (see list of references).                                 #
#                                                                    #
#    Parameters                                                      #
#       a - a [1:n,1:n] REAL matrix of coefficients.                 #
#           Left unchanged by the procedure.                         #
```

```
#       b - a [1:n,1:nrhs] matrix containing as its columns the        #
#           nrhs righthand sides for which solutions are required.     #
#           Left unchanged by the procedure.                           #
#       x - a [1:n,1:nrhs]matrix containing on entry an initial        #
#           approximation to the solution vectors.If no information    #
#           is available,this approximation may be zero.               #
#           On exit, x will contain the computed solution.             #
#     acc - a user -provided estimate of the accuracy required.        #
#           The procedure will attempt to return an approximation      #
#           xcalc to the exact solution x such that                    #
#                 //x-xcalc// < acc.                                   #
#   refine- if set to TRUE on entry,the procedure carries out          #
#           double-length evaluation of the residuals,and iterative    #
#           refinement of the solution of the first m equations.       #
#                                                                      #
#    result                                                            #
#           The routine returns the calculated solution together       #
#           with an error estimate,in a MATRESULT structure.           #
#           The fields of this structure have the following            #
#           significance:                                              #
#     ans - a [1:n,1:nrhs] REAL array containing the solution xcalc    #
#      ep - an estimate of the infinity norm //xcalc-x//               #
#   evals - normally contains the value of m,the size used for the     #
#           first block of equations.A large value of m indicates      #
#           illconditioning in the matrix.                             #
#              If the matrix a was found to be numerically singular,   #
#           then evals contains instead the numerical rank r of a.     #
#           In this case,n-r of the rows of xcalc will be found to be#
#           identically zero(these may not be the last rows),and the #
#           remaining r rows solve the reduced r•r problem.            #
#     fin - a boolean value normally set to TRUE ;                     #
#           set to FALSE if a is found to be numerically singular.     #
#                                                                      #
#   operators.                                                         #
#           The procedure assumes that the following matrix operators#
#           have been defined:                                         #
#   NORM a -gives the infinity norm of a                               #
#   NORM1 a-gives the one norm of a                                    #
#    +:=,-:=,/:=,•  : with their usual meanings.                       #
#    D• , D-:=   :double length versions of •,- :=                     #
#----------------------------------------------------------------------#
       INT n := 1 UPB a, m := 0, limit, rank, nrhs := 2 UPB b;
       BOOL rowexchange = TRUE , colexchange = FALSE ;
       BOOL notconverged, converging;
       REF [, ] REAL null = NIL ;
       REAL cutoff := 0.1, del1 := 1.1# used by setm #;
       PROC interchange =
          ( REF [, ] REAL a, INT r, s, BOOL row) VOID :
          BEGIN # interchanges two columns or rows of a #
             INT l;
             IF r/=s
             THEN
                REF [] REAL b, c;
                IF row
                THEN b := a[s, ]; c := a[r, ]; l := 2
                ELSE b := a[, s]; c := a[, r]; l := 1
                FI ;
                [l LWB a:l UPB a] REAL d;
                d := c;
                c[] := b[];
                b[] := d
             FI
          END # interchange #;
       PROC maxind = ( REF [] REAL a) INT :
          BEGIN
             # returns a pointer to the element of maximum size in a #
             INT m;
             REAL temp := 0.0, tomp;
             m := LWB a;
             FOR i FROM LWB a To UPB a
             DO
```

```
                  tomp := ABS a[i];
                  IF tomp>temp THEN temp := tomp; m := i FI
               OD ;
               m
            END # maxind #;
         PROC leqdirect =
            ( REF [, ] REAL a, b, REF [] INT row, col, REF INT mpreve
            ) BOOL
               :
# Gives a direct solution of the m equations a•x=b,using Gauss        #
# elimination,given that mpreve rows and columns of a have already    #
# been processed.The vectors row,col hold on entry pivotting          #
# information for these first mpreve equations,and on exit similar    #
# information for the remaining equations also.The vector b is        #
# overwritten by the solution x.                                      #
#     If a proves to have numerical rank r<m,the last(before          #
# reordering)m-r rows of b are set zero,and a solution of the         #
# remaining r equations returned,together with the value TRUE         #
# for leqdirect.                                                      #
               BEGIN
                  INT rowi, colj, m := UPB a;
                  REAL delta := 10•small real, temp, mult;
                  BOOL sing := FALSE , banded := mpreve<m;
                  INT piv;
                  rank := m# initial assumption #;
                  # treat first mpreve rhsides and new columns #
                  FOR newrow To mpreve
                  DO
                     interchange(b, newrow, row[newrow], rowexchange);
                     IF banded
                     THEN
                        interchange(a[, mpreve+1:m], newrow,
                           row[newrow], rowexchange);
                        interchange(a[mpreve+1:m, ], newrow,
                           col[newrow], colexchange)
                     FI ;
                     temp := a[newrow, newrow];
                     FOR i FROM newrow+1 TO m
                     DO
                        mult := a[i, newrow]/temp;
                        IF ABS mult>smallreal
                        THEN
                           b[i, ]-:=mult•b[newrow, ];
                           IF banded
                           THEN
                              limit :=
                                 IF i>mpreve
                                 THEN newrow
                                 ELSE mpreve
                                 FI +1;
                              a[i, limit:m]-:=mult•a[newrow, limit:m]
                           FI
                        FI
                     OD # i #
                  OD # newrow #;
                  # now complete the processing of the new rows #
                  FOR newrow
                     FROM IF mpreve=0 THEN 1 ELSE mpreve FI
                     TO m
                  DO
                   # pick a row pivot #
                     row[newrow] := piv :=
                        maxind(b[newrow:m AT newrow, 1]);
                     interchange(b, newrow, piv, rowexchange);
                     interchange(a[, newrow:m], newrow, piv, rowexchange
                     );
                     # pick a column pivot #
                     col[newrow] := piv :=
                        maxind(a[newrow, newrow:m AT newrow]);
                     interchange(a, newrow, piv, colexchange);
                     temp := a[newrow, newrow];
                     IF ABS temp<delta
                     THEN
                        sing := TRUE ;
                        rank := newrow-1;
                        CLEAR b[rank+1:m, ];
                        GOTO backsub
                     FI ;
                     FOR i FROM newrow+1 TO m
                     DO
                        mult := a[i, newrow]/temp;
                        a[i, newrow+1:m]-:=mult•a[newrow, newrow+1:m];
                        b[i, ]-:=mult•b[newrow, ]
                     OD # i #
                  OD # newrow #;
                  # now back substitute #
backsub:          b[rank, ]/:=a[rank, rank];
                  FOR i FROM rank-1 BY -1 TO 1
                  DO (b[i, ]-:=a[i, i+1:rank]•b[i+1:rank, ])/:=a[i, i]
                  OD ;
                  # now resort #
                  FOR i FROM m-1 BY -1 TO 1
                  DO interchange(b, i, col[i], rowexchange)
                  OD ;
                  mpreve := m;
                  sing
               END # leqdirect #;
         PROC setm = ( INT mold) INT :
            # computes a suitable value for m,the point at which the
               matrix a is partitioned #
               BEGIN
                  REAL temp, sum;
                  INT m := mold;
                  IF m=0
                  THEN
                     # check the bottom rows #
                     m := n;
                     FOR i FROM n BY -1 TO 2
                     WHILE
                        ((temp := ABS a[i, i])/=0.0) AND
                           ((sum := NORM1 a[i, ])<del1•temp)
                     DO m-:=1
                     OD
                  FI ;
                  # now check the sides #
                  IF m NE n
                  THEN
                     REAL anorm := 0.0;
                     FOR i TO m DO anorm+:= NORM1 a[i, ] OD ;
                     anorm•:=cutoff;
                     FOR i
                     WHILE (i<m AND m<n)
                     DO
                        sum := 0.0;
                        FOR j FROM n BY -1 To m+1
                        WHILE
                           (
                              IF sum>anorm THEN m := j FI ;
                              sum<=anorm
                           )
                        DO sum+:= ABS a[i, j]
                        OD
                     OD ;
                     SKIP
                  FI ;
                  # if we are close to n,the hell with iterating #
                  IF (m/n)>0.8 THEN m := n FI ;
                  # force m>mold by a reasonable margin #
                  m := ( ROUND (1.2•mold)) MAX m;
                  # check that this is not too much #
                  m := m MIN n;
                  m
               END # setm #;
         PROC leqiter =
            ( REF [, ] REAL adold, REF [] INT rowold, colold)
               MATRESULT
            :
# This is the main routine of leqwad, and implements the block        #
# iterative scheme described in the text.An estimate of the current   #
# accuracy is obtained from e(n)=//x(n)-x(n-1)//.If the iterations    #
# diverge,the blocksize m is increased and leqiter is called          #
# recursively to generate increased temporary storage.This should     #
# happen only rarely.Otherwise,the iterations are terminated when     #
# either the predicted current error falls below the requested acc,   #
# or e(n) begins to oscillate.Such oscillations cannot occur,for a    #
# linear problem,in the absence of roundoff errors;they are taken to  #
# be an indication that the maximum achievable accuracy has been      #
# attained.                                                           #
               BEGIN
                  INT mnew;
                  REAL eps := acc+1, epsprev := acc+2,
                     convergencefactor := 1;
                  REAL epp;
                  BOOL singular, oscillating := FALSE ;
                  # first update the value of m and get appropriate
                  workspace #
                  mnew := setm(m);
                  [1:mnew, 1:mnew] REAL ad;
                  [1:mnew, 1:nrhs] REAL xold;
                  [1:nrhs] REAL temp;
                  [1:mnew] INT row, col;
                  REF [] REAL xx;
                  # copy appropriate part of the matrix #
                  IF adold ISNT null
                  THEN
                     ad[1:m, 1:m] := adold;
                     row[1:m] := rowold;
                     col[1:m] := colold;
                     ad[1:m, m+1:mnew] := a[1:m, m+1:mnew]
                  ELSE m := 0
                  FI ;
                  ad[m+1:mnew, 1:mnew] := a[m+1:mnew, 1:mnew];
                  REF [, ] REAL x0 = x[1:mnew, ], b0 = b[1:mnew, ];
                  notconverged := converging := TRUE ;
                  FOR iter
                  WHILE notconverged AND converging
                  DO
                     xold := x0;
                     x0 := b0;
                     IF refine
                     THEN x0 D-:= a[1:mnew, 1:n] D• x
                     ELSE
                        IF mnew<n
                        THEN x0-:=a[1:mnew, mnew+1:n]•x[mnew+1:n, ]
                        FI
                     FI # refine #;
                     singular := leqdirect(ad, x0, row, col, m);
                     # note that leqdirect sets m:=mnew #
                     # check the change in x0 #
                     epp := epsprev;
                     epsprev := eps;
                     IF refine
                     THEN eps := NORM x0; x0+:=xold
                     ELSE eps := NORM (x0-xold)
                     FI ;
                     IF singular
                     THEN converging := FALSE
                     ELSE
                        IF m<n# usual case # OR refine
                        THEN
                           # now Gauss Seidel on the remaining
                           equations #
                           FOR i FROM m+1 TO n
                           DO
                              xx := x[i, ];
                              temp := xx;
                              (x[i, ] := b[i, ])-:=a[i, 1:i-1]•
                                 x[1:i-1, ];
                              IF i<n
                              THEN xx-:=a[i, i+1:n]•x[i+1:n, ]
                              FI ;
                              xx/:=a[i, i];
                              eps := eps MAX ( NORM1 (xx-temp))
                           OD # i #;
                           IF iter>1
                           THEN
                              convergencefactor :=
                                 (
                                    converging :=
                                       (epsprev-eps)>10•smallreal
                                 ! eps/(epsprev-eps)-
                                 ! 1.0
                                 )
                           FI ;
                           convergencefactor :=
                              convergencefactor MIN 100.0;
                           IF iter>2
                           THEN
                              oscillating :=
                                 (eps>=epsprev) AND (epsprev<=epp)
                           FI ;
                           notconverged :=
                        (eps•convergencefactor>acc) AND
                           ( NOT oscillating)
                  ELSE
                     # m=n and no iterative refinement.dont trust
                        the error estimate #
                     eps := 0.0;
                     notconverged := converging := FALSE
                  FI # main loop #
               FI # singularity test #
            OD # iteration loop #;
            IF (notconverged) AND m<n
            THEN
               cutoff/:=100;
               IF singular
               THEN
                  leqiter( NIL , NIL , NIL )
                     # restarts the triangulation #
               ELSE leqiter(ad, row, col)
               FI
            ELSE
               (x, eps•convergencefactor,
                  IF singular THEN rank ELSE m FI ,
                  NOT singular)
            FI
         END # leqiter #;
   # now the main call #
   leqiter( NIL , NIL , NIL )
END # leqwad #;
```

Algorithm 97

THE FAST GALERKIN ALGORITHM FOR THE SOLUTION OF LINEAR FREDHOLM EQUATIONS

L. M. Delves
Department of Computational and Statistical Science
University of Liverpool
and
L. F. Abd. El Al
Department of Mathematics
University of Cairo

Authors' notes

In a recent paper, Delves (1977) gave a 'Fast Galerkin' Algorithm for the solution of the linear Fredholm equation of the second kind:

$$f(x) = g(x) + \int_a^b K(x, y)\, f(y)dy \qquad (1)$$

The algorithm described produces an approximate solution $f_N(x)$ of the form

$$f_N(x) = \sum_{i=0}^{N=1} A_i T_i(\alpha x - \beta) \qquad (2)$$

$$\alpha = (2/(b - a),\ \beta = (a + b)/(b - a)$$

where $T_i(z)$, $-1 \leqslant z \leqslant 1$, is a Chebyshev polynomial of the first kind; and in addition an error estimate which takes account of both truncation errors (finite N) and quadrature errors (in the approximation of the integrals involved). The computational cost involved is $\mathcal{O}(N^2 \ln N)$, compared with the $\mathcal{O}(N^3)$ cost of previous Chebyshev schemes for (1) (Elliott, 1963; Scraton, 1969; El-Gendi, 1969; Watson, 1973; Miller and Symm, 1975).

We give here an implementation of the algorithm, referring to Delves (1977) for a description of the method and of the error estimate; and for timings obtained with the algorithm. The language used is ALGOL 68 as defined in the Revised Report (van Wijngaarden *et al.*, 1974) and the program makes use of an FFT procedure (**proc** FFT) of which only a dummy version is included here.

References

Delves, L. M. (1977). A Fast Method for the Solution of Fredholm Integral Equations, *JIMA* to be published
Elliott, D. (1963). *The Computer Journal*, Vol. 6, pp. 102-111
Scraton, L. E. (1969). *Math. Comp.*, Vol. 23, pp. 837-845
El-Gendi, S. E. (1969). *The Computer Journal*, Vol. 12, pp. 282-287
Watson, G. A. (1973). *The Computer Journal*, Vol. 16, pp. 77-84
Miller, G. F. and Symm, G. T. (1975). Procedure Fred 2b, National Physical Laboratory preprint DS/06/1/A/7/75
van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., Koster, C. H. A., Sintzoff, M., Lindsey, C. H., Meertens, G. L. T. and Fisher, R. G. (1974). Revised Report on the Algorithmic Language Algol 68, supplement to *Algol Bulletin*, 36, University of Alberta

```
PROC postconvert = ( REF [] REAL a, b) VOID :
   # Forms the vector b:=a•c,where c is the transformation matrix #
   # referred to in the text of ref (1),equation (26)              #
   BEGIN
      INT n := UPB a, m2, sufix;
      REAL temp, t1;
      FOR s FROM LWB b TO UPB b
      DO
         temp := 0.0;
         b[s] := a[s];
         FOR m TO n
         DO
            m2 := 2•m;
            t1 := IF s+m2<=n THEN a[s+m2] ELSE 0.0 FI ;
            sufix := ABS (s-m2-1);
            IF sufix<n THEN t1+:=a[sufix+1] FI ;
            IF t1/=0.0 THEN temp+:=t1/(m2•m2-1) FI
         OD ;
         b[s]-:=temp
      OD
   END # postconvert #;
PROC preconvert = ( REF [] REAL a, b) VOID :
   # Forms the vector b:=c•a,where c is the same matrix as in
      postconvert #
      (a[1]•:=2.0; postconvert(a, b); a[1]/:=2.0; b[1]/:=2.0);
PROC leqfred =
   ( REF [, ] REAL a, REF [] REAL b, x, REAL acc) VECRESULT :
   #Solves the equations (iprime+a•c)•x=b by block iteration#
   # where iprime(i)=1,i>1;=0.5,i=1                          #
   # This procedure is adapted from proc leqwad,ref(2).     #
   # The iterative scheme(21) of ref (1) is used,and        #
   #The product a•c is not formed explicitly.               #
   BEGIN
      INT n := 1 UPB a, m := 0, limit, rank;
      BOOL rowexchange = TRUE , colexchange = FALSE ;
      BOOL notconverged,converging;
      REF [, ] REAL null = NIL ;
      REAL cutoff := 0.02, del1 := 0.02# used by setm #;
      [1:n] REAL y # used as temporary storage to hold c•x #;
      PROC interchange =
         ( REF [, ] REAL a, INT r, s, BOOL row) VOID :
         BEGIN # interchanges two rows or columns of the matrix a #
            INT l;
            IF r/=s
            THEN
               REF [] REAL b, c;
               IF row
               THEN b := a[s, ]; c := a[r, ]; l := 2
               ELSE b := a[, s]; c := a[, r]; l := 1
               FI ;
               [l LWB a:l UPB a] REAL d;
               d := c;
               c[] := b[];
               b[] := d
            FI
         END # interchange #;
      PROC maxind = ( REF [] REAL a) INT :
         BEGIN # returns a pointer to the largest element in a #
            INT m := LWB a;
            REAL temp := 0.0, tomp;
            FOR i FROM LWB a TO UPB a
            DO
               tomp := ABS a[i];
               IF tomp>temp THEN temp := tomp; m := i FI
            OD ;
            m
         END # maxind #;
      PROC leqdirect =
         (
            REF [, ] REAL a, REF [] REAL b, REF [] INT row, col,
               REF INT mpreve
         ) BOOL :
      # Gives a direct solution of the equations a•x=b.          #
      # The solution x is returned in b.                         #
      # This procedure is a modification of proc leqdirect given#
      # in ref(2),and differs in treating only a single rh side #
            BEGIN
               INT rowi, colj, piv, m := UPB a;
               REAL delta := 10•small real, temp, mult;
               BOOL sing := FALSE , banded := mpreve<m;
               rank := m # initial assumption #;
               # treat first mpreve rhsides and new columns #
               FOR newrow TO mpreve
               DO
                  interchange(b, newrow, row[newrow], colexchange);
                  # note that algol68 treats b as a row vector,hence
                  colexchange #
                  IF banded
                  THEN
                     interchange(a[, mpreve+1:m], newrow,
                        row[newrow], rowexchange);
                     interchange(a[mpreve+1:m, ], newrow,
                        col[newrow], colexchange)
                  FI ;
                  temp := a[newrow, newrow];
                  FOR i FROM newrow+1 TO m
                  DO
                     mult := a[i, newrow]/temp;
                     IF ABS mult>smallreal
                     THEN
                        b[i]-:=mult•b[newrow];
                        IF banded
                        THEN
                           limit :=
                              IF i>mpreve
                              THEN newrow
                              ELSE mpreve
                              FI +1;
                           a[i, limit:m]-:=mult•a[newrow, limit:m]
                        FI
                     FI
                  OD # i #
               OD # newrow #;
               # now complete the processing of the new rows #
               FOR newrow
                  FROM IF mpreve=0 THEN 1 ELSE mpreve FI
                  TO m
               DO
                  # pick a row pivot #
                  row[newrow] := piv :=
                     maxind(b[newrow:m AT newrow]);
                     interchange(b, newrow, piv, colexchange);
                     interchange(a[, newrow:m], newrow, piv, rowexchange
                     );
                     # pick a column pivot #
                     col[newrow] := piv :=
                        maxind(a[newrow, newrow:m AT newrow]);
                     interchange(a, newrow, piv, colexchange);
                     temp := a[newrow, newrow];
                     IF ABS temp<delta
                     THEN
                        sing := TRUE ;
                        rank := newrow-1;
                        CLEAR b[rank+1:m];
                        GOTO backsub
                     FI ;
                     FOR i FROM newrow+1 To m
                     DO
                        mult := a[i, newrow]/temp;
                        a[i, newrow+1:m]-:=mult•a[newrow, newrow+1:m];
                        b[i]-:=mult•b[newrow]
                     OD # i #
                  OD # newrow #;
                  # now back substitute #
backsub:          b[rank]/:=a[rank, rank];
                  FOR i FROM rank-1 BY -1 To 1
                  DO (b[i]-:=a[i, i+1:rank]•b[i+1:rank])/:=a[i, i]
                  OD ;
                  # now resort #
                  FOR i FROM m-1 BY -1 TO 1
                  DO interchange(b, i, col[i], colexchange)
                  OD ;
                  mpreve := m;
                  sing
               END # leqdirect #;
         PROC setm = ( INT mold) INT :
           # computes a suitable value for the size of the first block #
               BEGIN
                  REAL temp, sum;
                  INT m := mold;
                  IF m=0
                  THEN
                     # check the bottom rows #
                     m := n;
                     FOR i FROM n BY -1 To 2
                     WHILE
                        ((temp := ABS (1.0+a[i, i]))/=0.0) AND
                           ((sum := NORM1 a[i, ])<del1•temp)
                     DO m-:=1
                     OD
                  FI ;
                  # now check the sides #
                  IF m NE n
                  THEN
                     REAL anorm := m+1;
                     FOR i TO n DO anorm+:= NORM1 a[i, ] OD ;
                     anorm•:=cutoff;
                     FOR i
                     WHILE (i<m AND m<n)
                     DO
                        sum := 0.0;
                        FOR j FROM n BY -1 To m+1
                        WHILE
                           IF sum>anorm THEN m := j FI ;
                           sum<=anorm
                        DO sum+:= ABS a[i, j]
                        OD
                     OD ;
                     SKIP
                  FI ;
                  # if we are close to n,the hell with iterating #
                  IF (m/n)>0.8 THEN m := n FI ;
                  # force m>mold by a reasonable margin #
                  m := ( ROUND (1.2•mold)) MAX m;
                  # check that this is not too much #
                  m := m MIN n
               END # setm #;
         PROC leqiter =
           ( REF [, ] REAL adold, REF [] INT rowold, colold)
               VECRESULT
          : # a modified version of proc leqiter,ref(2) #
           BEGIN
              INT mnew;
              REAL eps := acc+1, epsprev := acc+2,
                 convergencefactor := 1,epp,temp;
              BOOL singular, oscillating := FALSE ;
              # first update the value of m and get appropriate
              workspace #
              mnew := setm(m);
              [1:mnew, 1:mnew] REAL ad;
              [1:mnew] REAL delt;
              [1:mnew] INT row, col;
              # form appropriate part of the matrix ip +k•c and place
                 in ad #
              IF adold ISNT null
              THEN
                 ad[1:m, 1:m] := adold;
                 row[1:m] := rowold;
                 col[1:m] := colold;
                 FOR i TO m
                 DO postconvert(a[i, ], ad[i, m+1:mnew AT m+1])
                 OD
              ELSE m := 0
              FI ;
              FOR i FROM m+1 To mnew
              DO
                 postconvert(a[i, ], ad[i, ]);
                 ad[i, i]+:= IF i=1 THEN 2.0 ELSE 1.0 FI
              OD ;
              REF [] REAL x0 = x[1:mnew], b0 = b[1:mnew];
              notconverged := converging := TRUE ;
              FOR iter
              WHILE notconverged AND converging
              DO
                 # form y=c•x #
                 preconvert(x, y);
                 # compute change in first m components of x #
                 (delt := b0)-:=x0;
                 delt[1]-:=x[1];
                 delt-:=a[1:mnew, ]•y;
                 singular := leqdirect(ad, delt, row, col, m);
                 # note that leqdirect sets m:=mnew #
```

```
                     # check the change in x0 and update #
                     epp := epsprev;
                     epsprev := eps;
                     eps := NORM delt;
                     x0+:=delt;
                     IF singular
                     THEN converging := FALSE
                     ELSE
                        IF m<n # usual case #
                        THEN
                           # now iterate on the remaining equations #
                           FOR i FROM m+1 To n
                           DO
                              temp := b[i]-a[i, ]•y;
                              eps := eps MAX ( ABS (temp-x[i]));
                              x[i] := temp
                           OD # i #;
                           IF iter>1
                           THEN
                              convergencefactor :=.
                                 (converging := (epsprev-eps)>10•smallreal
                                 ! eps/(epsprev-eps)
                                 ! 1.0
                                 )
                           FI ;
                           convergencefactor :=
                              convergencefactor MIN 100.0;
                           IF iter>2
                           THEN
                              oscillating :=
                                 (eps>=epsprev) AND (epsprev<=epp)
                           FI ;
                           notconverged :=
                              (eps•convergencefactor>acc) AND
                                 ( NOT oscillating)
                        ELSE
                           # m=n.dont trust the error estimate #
                           eps := 0.0;
                           notconverged := converging := FALSE
                        FI # main loop #
                     FI # singularity test #
                  OD # iteration loop #;
                  IF (notconverged) AND m<n
                  THEN
                     cutoff/:=100;
                     IF singular
                     THEN
                        leqiter( NIL , NIL , NIL )
                           # restarts the triangulation #
                     ELSE leqiter(ad, row, col)
                     FI
                  ELSE
                     (x, eps•convergencefactor,
                        IF m=n THEN rank ELSE n FI , NOT singular)
                  FI
               END # leqiter #;
            # now the main call #
            leqiter( NIL , NIL , NIL )
         END # leqfred #;
      PROC cosines = ( REF [] REAL xx) VOID :
         BEGIN
            REF [] REAL x = xx[ AT 0];
            INT n = UPB x, np = UPB x%2;
            # computes cos(i•pi/n),i=0,1,...n,using the stable recurrence #
            # relation of Hopgood and Litherland(ref(3)). #
            INT spacing := np, ispaced, nup := 1,
               m := ROUND (ln(n)/ln(2));
            x[0] := 1.0;
            x[np] := 0.0;
            REAL phi := pi/2.0, x1;
            FOR l FROM 2 TO m
            DO
               phi/:=2.0;
               ispaced := spacing;
               spacing%:=2;
               x[spacing] := x1 := cos(phi);
               x1:=0.5/x1;
               ispaced-:=spacing;
               nup•:=2;
               FOR i FROM 3 BY 2 TO nup
               DO
                  ispaced+:=spacing•2;
                  x[ispaced] := x1•(x[ispaced-spacing]+x[ispaced+spacing])
               OD # i #
            OD # l #;
            FOR j FROM 0 TO np-1 DO x[n-j] := -x[j] OD
         END # cosines #;
      PROC check = ([] REAL b, REF REAL norm) BOOL :
#this procedure checks the vector b to see whether its elements #
#are smoothly converging to zero,allowing for a possible         #
#odd-even effect,and returns in parameter norm either b[n] or    #
#b[n-1] as an estimate of the truncation error associated with b#
         BEGIN
            INT n := UPB b;
            BOOL converged := FALSE # exceptional case #;
            norm := ABS b[n];
            IF n>2
            THEN-
               # normal case #
               REAL z1 := ABS b[n-1], z2 := ABS b[n-2];
               converged :=norm<z2;
               IF (z1>z2 AND z1>norm) THEN norm := z1 FI
            FI # normal case #;
            converged
         END # check #;
      PROC fft = ( REF [] REAL aa) VOID :
         BEGIN
            REF [] REAL a = aa[ AT 1];
#  evaluates the cosine transform                                        #
#          n-1                                                           #
# aa(k) = SIGMA aa(j)•cos(j•pi•k/(n-1)),k=0,1...n-1                      #
#          j=0                                                           #
#       where the first and last terms are halved.                       #
#       This dummy version should be replaced by one using the           #
#       Fast Fourier Transform.                                          #
       INT n:=1 UPB a;[1:n] REAL b; REAL temp,fac;
       FOR k TO n DO
          temp:=0.0;fac:=pi•(k-1)/(n-1);
          FOR j TO n DO
             temp+:=
             a[j]•cos(fac•(j-1))•( IF j=1 OR j=n THEN 0.5 ELSE 1.0 FI )
                     OD ;
```

```
              b[k]:=temp•2/(n-1)
                   OD ;
            a:=b
   END #fft#;
  PROC fredsol =
     (
        REF [] REAL x, REAL a, b,
           PROC ( REAL , REAL ) REAL kernel, PROC ( REAL ) REAL gg
     ) VECRESULT
        :
#----------------------------------------------------------------#
#  Solves the linear Fredholm equation                           #
#                   b                                            #
#  f(r) = g(r) + INT k(r,s)•f(s)•ds                              #
#                   a                                            #
#  using the Fast Galerkin method described in ref(1).           #
#  f(r) is approximated by a truncated Chebyshev polynomial      #
#  expansion of the form:                                        #
#                   n                                            #
#       f(r) = SIGMA x(i)•t(i-1)((2•x-a-b)/(b-a))                #
#                  i=1                                           #
#  Note that the first term is NOT halved.                       #
#  The time taken by the routine is O(n↑2•ln(n)).                #
#     leqfred is an adaptation of procedure leqwad of ref(2).#
#     It iterates the linear equations to an accuracy            #
#     determined from the a priori estimates given in ref(1)  #
#                                                                #
# Parameters                                                     #
# ----------                                                     #
# x - a real vector of dimensions [1:n].on input,should          #
#     contain an approximation to the expansion coefficients.#
#     This approximation may be zero if no information is        #
#     available,but if fredsol has been called previously        #
#     with a smaller value of n,the previous coefficients        #
#     should be saved and augmented with zeroes to yield some#
#     saving of time in the solution of the linear equations.#
#     On output,x will contain the computed expansion            #
#     coefficients.                                              #
# a,b- The lower and upper limits of integration in the          #
#     integral operator.                                         #
# kernel- a user provided function yielding the kernel k(r,s)#
#     of the integral equation.                                  #
# gg - a user provided function yielding the driving function#
#     g(r) of the integral equation.                             #
# results.                                                       #
# -------                                                        #
#     The procedure returns a result of mode vecresult,a         #
#     structure whose fields have the following significance #
# ans - a pointer to x,the vector of expansion coefficients.  #
#  ep - an estimate of the absolute accuracy:                    #
#          b                                                     #
#        max abs(f(s)-fapprox(s))                                #
#        s=a                                                     #
#      achieved by the routine.For a discussion of the error#
#      estimate, see ref(1).                                     #
# evals-the number of significant components in the vector x.#
#      This will normally be n.If however the problem was        #
#      found to be numerically singular,then the routine         #
#      will return with evals=m<n,and n-m components of x        #
#      will be set to zero.These will be the components          #
#      found to be the least important numerically.In this       #
#      case,the solution x returned still satisfies the          #
#      approximate Galerkin equations,but the error estimate#
#      ep may be unreliable.                                     #
# fin - a boolean field usually set to true.Set false if         #
#      there is reason to suppose that the error estimate ep#
#      may be unreliable.This may happen for any of the          #
#      following reasons:                                        #
#            symptom                           internal flag#
#            -------                           -------- ----#
#      Matrix numerically singular             fin OF result#
#      Matrix not singular,but too ill                           #
#      conditioned to solve accurately,                          #
#      given the quadrature errors present.     convergence #
#      Ragged convergence in the matrix k           conk     #
#      Ragged convergence in the rhs vector g       cong     #
#      Ragged convergence in the solution vector    cona     #

#----------------------------------------------------------------#
#  References                                                    #
#  ----------                                                    #
#   (1) L.M.Delves,a Fast Galerkin method for the solution       #
#       of Linear Fredholm Equations,J.I.M.A to be published.#
#   (2) L.M.Delves,a Linear Equation solver for Galerkin and #
#       Least Squares problems,submitted to Computer Journal.#
#   (3) F.R.A.Hopgood and C.Litherland,CACM 9(1966)270           #
#          See also J.Oliver, Stable Methods of evaluating the#
#          Points cos(i•pi/n),J.I.M.A 16(1975)247                #
#----------------------------------------------------------------#
        BEGIN
           VECRESULT result;
           INT n := UPB x;
           [1:n] REAL g,ss;
           [1:n, 1:n] REAL k;
           REAL normdg, normg, normdl, normlinv, norma, temp, rq, rt,
              fac,error,accy;
           REAL c1 := (b-a)•0.5, c2 := (a+b)•0.5, c3 := -c1;
           BOOL cong, cona, conk,convergence;
           # first compute the (mapped) cosine values #
           cosines(ss);
           FOR i TO n DO ss[i] := c1•ss[i]+c2 OD ;
           # now compute the kernel and driving term function values #
           FOR i TO n
           DO
              temp := ss[i];
              g[i] := gg(temp);
              FOR j TO n DO k[i, j] :=c3•kernel(temp, ss[j]) OD
           OD # i #;
           # now transform these #
           fft(g);
           FOR i TO n DO fft(k[i, ]) OD ;
           FOR i TO n DO fft(k[, i]) OD ;
           normg := NORM g;
           conk :=
              check(( NORM1 k[n-2, ], NORM1 k[n-1, ], NORM1 k[n, ]),
                 normdl);
           normdl•:= ABS c1;
           cong := check(g, normdg);
           accy := normdg MAX normdl;
           result := leqfred(k, g, x, 0.1•accy);
           norma := NORM x;
           normlinv := norma/normg;
           cona := check(x, rt);
           fac := 1-normlinv•normdl;
           convergence :=
              IF fac<10•smallreal
              THEN fac := 10•sqrt(smallreal); FALSE
              ELSE TRUE
              FI ;
           rq := normlinv•(normdg+normdl•norma)/fac;
           ep OF result+:=rq+n•rt;
           fin OF result :=
              convergence AND cona AND cong AND conk AND
                 (fin OF result);
           result
        END # fredsol #;
```

Algorithm 98

A NOTE ON THE SOLUTION OF CERTAIN TRIDIAGONAL SYSTEMS OF LINEAR EQUATIONS

W. D. Hoskins
Computer Science Department
The University of Manitoba
Winnipeg, Manitoba
G. E. McMaster
Department of Mathematics
and Computer Science
Brandon University
Brandon, Manitoba

Authors' notes

The problem of solving a tridiagonal system of linear equations

$$A\mathbf{x} = \mathbf{d} \tag{1.1}$$

occurs frequently in the solution of partial and ordinary differential equations (Fox, 1962), and in the solutions of cubic spline problems when prescribed derivatives are specified at the boundaries (Spath, 1974).

In the following discussion we consider the case when the matrix A has the form:

$$A = \begin{bmatrix} z & b & & & & \\ b & a & b & & & \\ & \cdot & \cdot & \cdot & & \\ & & \cdot & \cdot & \cdot & \\ & & & \cdot & \cdot & \cdot \\ & & & b & a & b \\ & & & & b & w \end{bmatrix}_{n,n} \tag{1.2}$$

Since the matrix A is symmetric, a modification of the coupled algorithm of Andres, Hoskins and McMaster (1974), McMaster (1976) or the Evans and Hatzopoulos (1976) algorithm could be used to solve the equation system (1.1). In this paper, an algorithm that decouples the equation system at, or adjacent to, the centre is presented. This decoupling enables the algorithm to minimise the effect of the boundary conditions on the solution values and in a parallel processing environment, the algorithm is very efficient.

1. Description of the procedure

The Malcolm-Palmer (1974) form for the LU decomposition of A can be shown to converge for most boundary equations; however, for each set of boundary conditions, a value for the convergence factor c must be determined. Additionally, any uncertainties in the boundary conditions are propagated in both the elimination and in the solution steps.

The algorithm proposed here, which we will call Madison, exploits several of the economies of the Malcolm-Palmer technique and for sufficiently large n [greater than the convergence factor (which is subsequently determined)], the convergence of the method is independent of the boundary conditions. Algorithm Madison utilises

Table 1 Upper bound C and observed values of c on the IBM 360

	Short precision	
a	C	c
3·0	19	17
3·5	16	14
4·0	14	12
5·0	12	12
7·0	10	10
8·0	9	9
9·0	8	8
20·0	6	6
25·0	5	5

the symmetry and near centrosymmetry to decouple the equation system at the value, $m = [(n + 1)/2]$, the integer part of $n + 1$ divided by 2, and to separate the work into two nearly parallel processes.

A multiple of the mth row of the equation system with coefficient matrix as defined in (1.1) may be added to the remaining rows to attempt to simultaneously move the coefficient matrix A to upper and lower triangular form. This elimination step propagates a non-zero entry in column $m + 1$ in the upper half and column $m - 1$ in the lower half of the coefficient matrix resulting in A having the form:

$$\begin{bmatrix}
u^* \\
1 & u \\
& \cdot & \cdot \\
&& \cdot & \cdot \\
&&& 1 & u \\
&&&& 1 & u_c &&&&&& s_c \\
&&&&& 1 & \cdot &&&&& \cdot \\
&&&&&& \cdot & \cdot &&&& \cdot \\
&&&&&&& 1 & u_2 &&& s_2 \\
&&&&&&&& 1 & u_1 & 0 & s_1 \\
&&&&&&&&& 1 & s & 1 \\
&&&&&&&&& s_1 & 0 & u_1 & 1 \\
&&&&&&&&& s_2 &&& u_2 & 1 \\
&&&&&&&&& \cdot &&&& \cdot & \cdot \\
&&&&&&&&& \cdot &&&&& \cdot & \cdot \\
&&&&&&&&& s_c &&&&&& u_c & 1 \\
&&&&&&&&&&&&&&&& u & 1 \\
&&&&&&&&&&&&&&&&& \cdot & \cdot \\
&&&&&&&&&&&&&&&&&& \cdot & \cdot \\
&&&&&&&&&&&&&&&&&&& u & 1 \\
&&&&&&&&&&&&&&&&&&&& w^*
\end{bmatrix} \tag{2.1}$$

Notice that the u_i are shown as converging to u (Malcolm and Palmer, 1974) and simultaneously the s_i given by

$$s_i = (-1)^i \, s_{i-1}/u_{i-1}$$

are shown as converging to 0 to machine precision after some predetermined value $i = c$. The solution values $x_1, x_2, \ldots, x_{m-c-1}$; $x_n, x_{n-1}, \ldots, x_{m+c+1}$ may be obtained immediately, in parallel, in the back-substitution process. The solution values x_{m-c-1} and x_{m+c+1} may be subtracted from equations $m - c$ and $m + c$ respectively to leave a smaller equation system with coefficient matrix in the form

$$\begin{bmatrix}
u_c &&&&&&& s_c \\
1 & u_{c-1} &&&&&& \cdot \\
& \cdot & \cdot &&&&& \cdot \\
&& \cdot & \cdot &&&& \cdot \\
&&& 1 & u_2 &&& s_2 \\
&&&&& u_1 & 0 & s_1 \\
&&&&& 1 & a & 1 \\
&&&&& s_1 & 0 & u_1 & 1 \\
&&&&& s_2 &&& u_2 & 1 \\
&&&&& \cdot &&&& \cdot & \cdot \\
&&&&& \cdot &&&&& \cdot & \cdot \\
&&&&& \cdot &&&&&& u_{c-1} & 1 \\
&&&&& s_c &&&&&&& u_c
\end{bmatrix} \tag{2.2}$$

Using an additional elimination phase, we have

$$\begin{bmatrix}
u_c &&&&&& s_c \\
& u^*_{c-1} &&&&& s^*_{c-1} \\
&& \cdot &&&& \cdot \\
&&& \cdot &&& \cdot \\
&&&& u^*_1 & 0 & s^*_1 \\
&&&& 1 & a & 1 \\
&&&& s^*_1 & 0 & u^*_1 \\
&&&& \cdot &&& \cdot \\
&&&& \cdot &&&& \cdot \\
&&&& s^*_{c-1} &&&&& u_{c-1}^* \\
&&&& s_c &&&&&& u_c
\end{bmatrix} \tag{2.3}$$

The two by two equation system with coefficient matrix

$$\begin{bmatrix} u^*_1 & s^*_1 \\ s^*_1 & u^*_1 \end{bmatrix}$$

at the centre is solved first, the remainder of the solution values being obtained readily by back substitution.

In a manner analogous to that of Malcolm and Palmer (1974), it is possible to determine an upper bound C after which both the u_i have converged and the s_i may be assumed to be zero to machine precision.

For seven figures of accuracy, an upper bound is given by

$$C = 8 \ln (10)/(\ln (a^2 - 4)^{\frac{1}{2}}) - \ln (2)) \; .$$

2. Discussion

It is clear that algorithm Madison as described in Section 1 requires only $n + 2c + 1$ words of storage for the entire solution.

It is interesting to note that the elimination process for the Madison algorithm is performed from the centre values of the matrix outwards, thus reducing the growth of the error due to the effect of the boundary conditions and round-off error in the elimination stage. The final solution is then obtained from the outer edges inwards thus reducing the growth of rounding error in the backward substitution (Wilkinson, 1965).

In the forward elimination, a little over half the work need be done if a parallel processing machine is available, and the two back substitution phases can be arranged to take place simultaneously.

3. Program

3.1. *Formal parameter list*

(*a*) Input to the procedure Madison

- n the order of the equation system to be solved. It must be $> 2c + 1$
- c the value after which the u_i in the LU decomposition will converge and the s_i may be assumed to be zero (from Table 1)
- d the vector of constant values on the righthand side of the equation system
- a the value on the main diagonal of the coefficient matrix (apart from the positions (1, 1) and (n, n))
- z the value in position (1, 1) of the coefficient matrix
- w the value in position (n, n) of the coefficient matrix.

(*b*) Output from procedure Madison

- d the vector of solution values. The original contents of d are lost.

3.2. *ALGOL 60 procedure for algorithm Madison*

```
begin
procedure madison(c, n, d, a, z, w);
comment madison solves a tridiagonal linear system of equations
  ax = d where a has ones on the off diagonals and the values (z, a, a,
  . . . w) on the main diagonal. The effect of the boundary values z, w
  is minimised in the solution since these values are not used in the
  program until the final stage of the elimination process;
begin
value c, n, a, z, w; integer c, n; array d; real a, z, w;
begin
real array s[0: c]; real array u[0: c];
real u1, alpha, temp, t; integer i, j, m;
s[0] := 1. ;u[0] := a; m := (n + 1) div 2;
for i := 1 step 1 until c do
 begin s[i] := −s[i − 1]/u[i − 1];
   u[i] := a − 1./u[i − 1];
   d[m − i] := d[m − i] − d[m − i + 1]/u[i − 1];
   d[m + i] := d[m + i] − d[m + i − 1]/u[i − 1]
 end;
u1 := u[c];
for i := m − c − 1 step −1 until 2 do
  d[i] := d[i] − d[i + 1]/u1;
for i := m + c + 1 step 1 until n − 1 do
  d[i] := d[i] − d[i − 1]/u1;
d[1] := (d[1] − d[2]/u1)/z − 1./u1);
d[n] := (d[n] − d[n − 1]/u1)/(w − 1./u1);
for i := 2 step 1 until m − c − 1 do
  d[i] := (d[i] − d[i − 1])/u1;
for i := n − 1 step −1 until m + c + 1 do
  d[i] := (d[i] − d[i + 1])/u1;
d[m − c] := d[m − c] − d[m − c − 1];
d[m + c] := d[m + c] − d[m + c + 1];
for i := c − 1 step −1 until 1 do
 begin
   s[i] := s[i] − s[i + 1]/u[i + 1];
   d[m − i] := d[m − i] − d[m − i − 1]/u[i + 1];
```

```
    d[m + i] := d[m + i] - d[m + i + 1]/u[i + 1]
  end;
alpha := -1./(u[1] + s[1]); temp := (u[1]**2 - s[1]**2);
d[m] := (d[m] + alpha*(d[m - 1] + d[m + 1]))/a;
t := (d[m - 1]*u[1] - s[1]*d[m + 1])/temp;
d[m + 1] := (u[1]*d[m + 1] - s[1]*d[m - 1])/temp;
d[m - 1] := t; temp := d[m + 1];
for i := 2 step 1 until c do
  begin
    d[m - i] := (d[m - i] - s[i]*temp)/u[i];
    d[m + i] := (d[m + i] - s[i]*t)/u[i]
  end
end madison;
```

References

Andres, T., Hoskins, W. D., and McMaster, G. E. (1974). A Coupled Algorithm for the Solution of Certain Tridiagonal Systems of Linear Equations, *The Computer Journal*, Vol. 17, Algorithm 84, pp. 378-379

Evans, D. J. and Hatzopoulos, M. (1976). The Solution of Certain Banded Systems of Linear Equations Using the Folding Algorithm, *The Computer Journal*, Vol. 19, No. 2, pp. 184-187

Fox, L. (Editor), (1962). *Numerical Solution of Ordinary and Partial Differential Equations*, Pergamon Press, Oxford: Addison-Wesley, Reading, Mass.

Malcolm, M. A. and Palmer, J. (1974). A Fast Method for Solving a Class of Tridiagonal Linear Equations, *CACM*, Vol. 17, No. 1, pp. 14-17

McMaster, G. E. (1976). *The Computation of Splines and the Solution of Related Equation Systems*, Ph.D. Thesis, The University of Manitoba, Winnipeg, Manitoba, Canada.

Spath, H. (1974). *Spline Algorithms for Curves and Surfaces*, Translation by W. D. Hoskins and H. W. Sager, Utilitas Mathematics Publishing Inc., Winnipeg.

Algorithm 99

ON THE CONSTRUCTION OF BALANCED BINARY TREES FOR PARALLEL PROCESSING

D. J. Evans and S. A. Smith
Department of Computer Studies
Loughborough University of Technology
Loughborough, Leics.

Authors' note

Over the past few years new computer architectures have emerged which possess a number of arithmetic units or processors. To utilise these machines efficiently statements should be compiled so as to indicate where several parts of an arithmetic expression (or indeed several sequential expressions) can be computed in parallel.

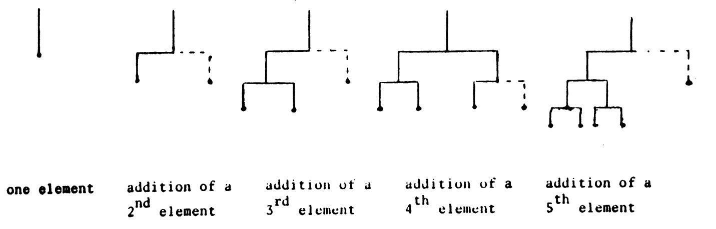
If an arithmetic expression is represented by a binary tree any operations that appear at the same level within the tree can be assigned to separate arithmetic units or processors and computed in parallel.

In this paper we present a method which produces a binary tree representation of an arithmetic expression which is of minimum height wherever possible. Throughout, emphasis will be directed to an algorithmic process to be used for creating such trees, which will be incorporated within the compiling process.

1. Formation of a balanced tree

Methods of forming tree representations of arithmetic expressions are well known and given in Knuth (1968). We consider the formation of a binary tree of minimum height for its composite elements and subtrees. Such a tree will be called a 'balanced binary tree' or a 'balanced tree'. The remaining terminology will be taken as that given by Knuth (1968; 1973).

We now consider the specific case where a balanced binary tree is systematically constructed from single element components (see Fig. 1). Assume that the first element is attached to the null node, then a second element can be added by forming a new node whose left hand son is the original element and whose right hand son is the new element (this is called 'inserting one place above'). Further elements can then be added, the third by inserting a new node two places above the previous element whilst a fourth can be added by inserting a new node one place above the third element. Finally, the fifth element can be added by inserting a new node three places above the fourth element, etc. This tree construction process can be enumerated by using a numeric code which is generated in the following manner. At any point on the tree the number 1 is used to represent the placing of an element one place above the previous entry in the tree and the number 2 to represent its placing two positions above. So the four insertions shown in Fig. 1 can be represented by a code given by the series of numbers 1, 2, 1, 2.



one element
addition of a 2nd element
addition of a 3rd element
addition of a 4th element
addition of a 5th element

Fig. 1

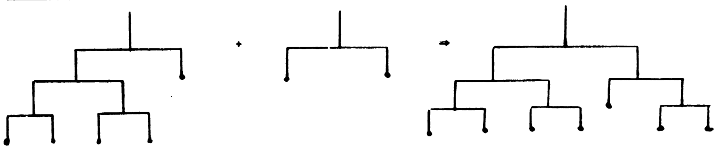


Fig. 2(*a*)

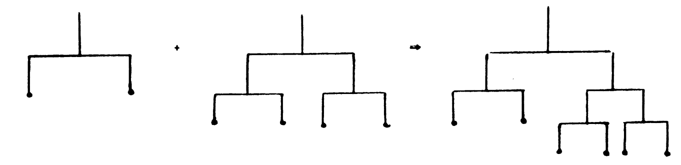


Fig. 2(*b*)

To extend this process the requirements of balancing dictates that the following three insertions will be in the same manner as the first 3 (i.e., 1, 2, 1). The eighth, that is the 2^3, insertion will be situated at the fourth (3 + 1) level above all the other nodes. In general then the next $2^i - 1$ insertions will be in the same manner as the first $(2^i - 1)$ insertions and the $2^{(i+1)\text{th}}$ insertion will be at the $(i + 2)$th level above all other nodes.

2. Addition of subtrees to an existing tree

The process outlined in the previous section will allow single elements to be inserted into a tree but it is also useful to be able to add subtrees to an existing tree where the subtrees will retain their structure within the overall tree structure. To do this the following criteria are defined:

1. Any increase in the overall height of the tree caused by the insertion process should be kept to an absolute minimum. This is so that the number of levels in the tree will continue to be minimised.
2. An insertion at the top of the tree is preferable to extending the tree below the lowest existing level. This provides for possible, future extensions to the tree. If we extend the tree below the lowest existing level then the next insertion *must* also extend the height of the tree. If the tree is extended above all existing levels the next insertion *may not* extend the overall height of the tree.
3. The subtrees should be placed in the first available position in the tree, providing the previous conditions are met. This again is done to provide for future extensions to the tree, so that the maximum number of vacant nodes are available for successive insertions.

Previously the position for insertion of a new node has been defined in terms of the previous element inserted. For the insertion of a subtree a dummy pointer will be required so that the next insertion can be implemented at the correct position in the tree. The value of this dummy pointer will depend on the relative heights of the subtree and the tree into which it is to be inserted. If the subtree is shorter than the tree into which it is being inserted then it is assumed that the maximum number of elements that could be held by a subtree of that number of levels has been added, and the dummy pointer is theoretically obtained from the last element added. If the subtree height is greater than or equal to that of the tree into which it is being inserted then a different approach is necessary. Since the next insertion will be required to be above the join of the tree and the subtree (the reasoning being the same as that for criterion 2), then the dummy pointer will need to be theoretically taken from the last element inserted in a tree (containing the maximum number of elements) of one level greater than the subtree actually inserted. Details of this procedure for the additions of subtrees can be seen in **Fig. 2**(*a*) and (*b*).

3. Conclusion

A great deal of research has been carried out into the handling of binary trees containing data, see for instance Day (1976). Some work has also been done in producing syntactic trees with a minimum number of levels, see for instance Baer and Bovet (1968). In comparison the algorithm described here will produce a binary tree representation of the elements and subtrees as given, which will have minimum height possible for such a tree without having to reorder the expression. Any operations that appear at the same level within the binary tree can be assigned to separate arithmetic units or processors and computed in parallel.

4. ALGOL 68 procedure

A procedure that balances trees in the manner described hitherto has been written and tested in ALGOL 68.

Throughout the algorithm the existence of a recursive ALGOL 68 MODE TREE is assumed with at least the following five fields:

LEFT	this is a reference to the tree that is the left hand son of the present tree. This can be null if there is no left hand son.
OPERATOR	this is the operator (e.g. +) that joins the left and right hand son, or if a leaf node, the variable name.
RIGHT	this is a reference to the tree that is the right hand son of the present tree. This tree can be null if there is no right hand son.
LEVEL	this is the level of the operator in the overall tree, usually one for a variable.
FATHER	this is a reference to the tree that is the father of the present tree. This tree can be null if this tree has no father.

Other fields may be added to meet the needs of a given situation. The following ALGOL 68 declaration will define the structure explained.
MODE TREE = STRUCT(REF TREE left, STRING operator, REF TREE right, INT level, REF TREE father)

References

Knuth, D. E. (1968). *The Art of Computer Programming*, Vol. 1, Fundamental Algorithms, Addison Wesley

Knuth, D. E. (1973). *The Art of Computer Programming*, Vol. 3, Sorting and Searching, Addison Wesley

Day, A. C. (1976). Algorithm 91: Balancing a Binary Tree, *The Computer Journal*, Vol. 19, No. 4, pp. 360-362

Baer, J. L. and Bovet, D. P. (1968). Compilation of Arithmetic Expressions for Parallel Computations, *Proc. IFIP Congr.* 1968, pp. 340-346

```
C************************************************************************
    <make balanced tree> adds an item or a sub-tree to an existing tree,
     at the most suitable point for an entry of its size
-***********************************************************************C
     PROC make balanced tree=(REF INT next,randtop,optop,
                              INT last REF [] TREE this,
                              REF REF TREE orig,[] REF TREE randstack,
                              [] CHAR operators)
                              VOID:
C-----------------------------------------------------------------------
next      indicates the next free position in the array of trees<this>,
              originally 0
randtop   indicates the top item of<randstack>
optop     indicates the number of entries in<operators>
last      indicates where in<this>references to the current set of sub-trees
              begins, initially 1
this      a stack of trees used to hold all subtrees formed
orig      contains the current sub-tree
randstack a stack containing sub-trees and operands
operators a stack of operators, which correspond to the operands
     this procedure will form a balanced tree of operands and operators,
     as long as the operator remains the same
-----------------------------------------------------------------------C
         (INT temp;
         INT count,prev,pcount;
         CHAR oper←operators[optop];
         pcount←0;
         INT nooflevels←12;
         [1:2↑nooflevels-1] INT predefined;
         INT value←1;
         predefined[1]←1;
         FOR i FROM 2 TO nooflevels DO
              (value TIMES 2;predefined[value]←i;
              predefined[value+1:2*value-1]←predefined[1:value-1]);
         OP '>'=(TREE expra,exprb) INT:
              (INT lev←(level OF expra>level OF exprb!level OF expra!
                        level OF exprb)+1;lev);
C************************************************************************
    <pa>adjusts<point>so that instead of pointing to a node it points to
     to its father
-***********************************************************************C
         PROC pa=(REF INT point) VOID:
              (INT temp←point;
              FOR i1 FROM last TO next WHILE temp=point DO

                 WHILE this[i1] IS father OF this[point] DO point←i1);
C-----------------------------------------------------------------------
     attach first element or sub-tree to null node
-----------------------------------------------------------------------C
              next PLUS 1;
              level OF this[next]←level OF randstack[randtop];
              left OF this[next]←randstack[randtop];
              operator OF this[next]←"@";
              orig←left OF this[next];
              randtop MINUS 1;
              prev←next;
              count←2↑(level OF this[next]-1);
C-----------------------------------------------------------------------
     the loop that builds up the tree
-----------------------------------------------------------------------C
          WHILE (optop>= LWB operators AND randtop>=LWB randstack
              oper=operators[optop] ! FALSE) DO
              (optop MINUS 1;
              next PLUS 1;
              temp←0;
              IF level OF randstack[randtop]>=level OF orig THEN
                  count←2↑(level OF orig-1)
              ELSE WHILE level OF randstack[randtop]>predefined[count] DO
                  count PLUS 1
              FI;
              left OF this[next]←randstack[randtop];
              operator OF this[next]←oper;
              IF predefined[count]=1
                  OR pcount=0
              THEN
C-----------------------------------------------------------------------
     place one place above the previous entry
-----------------------------------------------------------------------C
                  father OF this[next]←this[prev];
                  right OF this[next]←left OF this[prev];
                  left OF this[prev]←this[next]
              ELSE
C-----------------------------------------------------------------------
     place<temp>+ 1 places above the previous entry
-----------------------------------------------------------------------C
                  temp←predefined[count]-level OF this[prev];
                  FOR i TO temp WHILE operator OF (father OF this[prev])#"@"
                                                    DO pa(prev);
                  father OF this[next]←father OF this[prev];
                  IF operator OF (father OF this [prev])="@" THEN
                     operator OF (father OF this [next])←"@"
                  ELSE this [prev] IS right OF (father OF this [prev]) THEN
                     right OF (father OF this [prev])←this[next]
                  ELSE left OF (father OF this [prev])←this[next]
                  FI;
                  father OF this [prev]←this[next];
                  right OF this[next]←this[prev]
              FI;
C-----------------------------------------------------------------------
    if a sub-tree of level greater than one has been added update count
    to allow for this
-----------------------------------------------------------------------C
              IF level OF randstack[randtop]=1 THEN prev←next
              ELSE level OF randstack[randtop]>=level OF orig THEN
                  prev←next;
                  count←2↑(level OF randstack[randtop])-1
              ELSE count PLUS 2↑(level OF randstack[randtop]-1)-1;
                  FOR i1 FROM last TO next-1 DO
                       IF this[i1] IS left OF this[next] THEN
                            prev←i1;
                            father OF this[prev]←this[next];
                            GOTO lz
                       FI;
    lz:           SKIP
              FI;
              IF operator OF (father OF this[next])="@" OR pcount=0 THEN
                  orig←this[next]; pcount←1
         FI;
         count PLUS 1:
         randtop MINUS 1:
         temp←next:
C-----------------------------------------------------------------------
     update levels of all trees affected by this insertion
-----------------------------------------------------------------------
         IF operator OF this[temp]= "@" THEN
         level OF this[temp]←left OF this[temp]'>'right OF this[temp]
         ELSE
         WHILE operator OF this[temp]#"@" DO
         (level OF this [temp]←left OF this[temp]' >'
              right OF this[temp];pa(temp))
         FI));
```