Algorithms

J. H. WEGSTEIN, Editor

Contributions to this department must be in the form stated in the Algorithms Department policy statement (*Communications*, February, 1960) except that ALGOL 60 notation should be used (see *Communications*, May 1960). Contributions should be sent in duplicate to J. H. Wegstein, Computation Laboratory, National Bureau of Standards, Washington 25, D. C. Algorithms should be in the Reference form of ALGOL 60 and written in a style patterned after the most recent algorithms appearing in this department. For the convenience of the printer, please underline words that are delimiters to appear in boldface type.

Although each algorithm has been tested by its contributor, no warranty, express or implied, is made by the contributor, the editor, or the Association for Computing Machinery as to the accuracy and functioning of the algorithm and related algorithm material, and no responsibility is assumed by the contributor, the editor, or the Association for Computing Machinery in connection therewith.

The reproduction of algorithms appearing in this department is explicitly permitted without any charge. When reproduction is for publication purposes, reference must be made to the algorithm author and to the *Communications* issue bearing the algorithm.

ALGORITHM 129
MINIFUN

V. W. WHITLEY
Signal Missile Support Agency, White Sands Missile Range, N. Mex.

```
procedure MINIFUN (t1, b1, eps, n, ncnt, fmin, xmin, k1, GFUN);
  value t1, b1, eps, n, ncnt; integer n, ncnt, k1; real fmin;
  real procedure GFUN; array t1, b1, eps, xmin;
comment MINIFUN is a subroutine to find the minimum of a function of n variables, using the method of steepest descent.
  Input is:
  1. t1(i), i = 1,2, ··· , n, the upper limits of the search region
  2. b1(i), i = 1,2, ··· , n, the lower limits of the search region
  3. eps(i), i = 1,2, ··· , n, the convergence criteria. The function must be a minimum in the region | x(i) − xmin(i) | ≦ eps(i)
  4. n, the number of variables (the dimension of the arrays)
  5. ncnt, the maximum number of iterations. The routine searches for a minimum until | x(i) − xmin(i) | ≦ eps(i) for all i, or until icnt = ncnt, whichever happens first.
  Output is:
  1. fmin, the minimum value of the function
  2. xmin(i), i = 1, ··· , n, the point at which the minimum occurs
  3. k1, an error code
     If k1 = 1, a minimum has been found within the specified number of iterations and the minimum is less than all values of the function at the centers of the planes forming the boundary of the epsilon-cube
     If k1 = 2, Δx(i) ≦ eps(i) but a new minimum has been found
     If k1 = 3, ncnt has been exceeded without Δx(i) ≦ eps(i). In this case, a test is made to see if the current minimum is a minimum in the epsilon-cube.
  MINIFUN has been written as a FORTRAN II subroutine and is available from the SMSA Computation Center. It should be noted that the FORTRAN II deck has been tested only on some relatively simple functions of two variables, such as GFUN (x,y) = cos(xy). The writer does not claim that the algorithm has been thoroughly tested;
begin integer j, i, icnt, k; real w, dmax, alamb, ft;
    array wnew [1:n], xt[1:n], x1b[1:n], xub [1:n],
      de1x[1:n], d12x[1:n], xmin[1:n], x[1:n, 1:4], g[1:n, 1:4],
      dxmin[1:n], d2xmn[1:n];
  comment start looking for a minimum at midpoint of region;
  for j := 1 step 1 until n do
    begin wnew[j] := (t1[j] + b1[j])/2; xt[j] := wnew[j];
      xub[j] := t1[j]; x1b[j] := b1[j]; de1x[j] := (xub[j]
        − x1b[j])/5;
      d12x[j] := de1x[j]↑2; xmin[j] := xt[j]
    end;
  fmin := GFUN (xmin);
  for j := 1 step 1 until n do
    begin w := xt[j]; for i := 1 step 1 until 4 do
      begin x[j, i] := x1b[j] + i × de1x[j];
        xt[j] := x[j,i]; g[j,i] := GFUN(xt);
      end;
      xt[j] := w;
      dxmin[j] := (g[j,3] − g[j,2])/de1x[j];
      d2xmn[j] := (g[j,4] − g[j,3] − g[j,2] + g[j,1])/d12x[j]
    end;
comment first and second difference quotients have been computed;
  icnt := 0; dmax := dxmin[1]; k := 1;
nustep: for j := 2 step 1 until n do
    begin if abs(dmax) < abs(dxmn[j]) then
        begin dmax := dxmin[j]; k := j
        end;
    end;
  alamb := dxmin[k]/d2xmn[k]; w := xt[k] − alamb;
comment a new coordinate has been computed for the variable having the largest first partial derivative. It will be checked to see if the new point still lies within the region and search will continue;
  if w < b1[k] then w := b1[k] else if w > t1[k] then w := t1[k];
  xt[k] := w; ft := GFUN(xt);
  if ft < fmin then go to check else
restart: if xt[k] < wnew[k] then go to 1bdchk
      else if xt[k] = wnew[k] then go to stnubds
      else if t1[k] > xt[k] then go to nupbds
      else xt[k] := 1.5 × wnew[k];
nupbds: xub[k] := t1[k]; x1b[k] := 2 × xt[k] − t1[k]; go to newde1;
stnubds: x1b[k] := xt[k] − 0.5 × wnew[k]; xub[k] := xt[k] + 0.5 × wnew[k];
newde1: de1x[k] := 0.2 ×(xub[k] − x1b[k]); d12x[k] := de1x[k]↑2;
    for i := 1 step 1 until 4 do
  begin x[k,i] := x1b[k] + i × de1x[k]; w := xt[k];
      xt[k] := x[k,i]; g[k,i] := GFUN(xt); xt[k] := w
  end;
  dxmin[k] := (g[k,3] − g[k,2])/de1x[k];
  d2xmn[k] := (g[k,4] − g[k,3] − g[k,2] + g[k,1])/d12x[k];
  icnt := icnt + 1;
  if icnt > ncnt then go to outcd else go to nustep;
1bdchk: if xt[k] ≦ b1[k] then xt[k] := 0.5 × wnew[k]
```

```
        else x1b[k] := b1[k];  xub[k] := 2.0 × xt[k] − b1[k];
    go to newde1;
check:  fmin := ft;  xmin[k] := xt[k];
  for j := 1 step 1 until n do if de1x[j] > eps[j] then go to restart;
recheck:  for j := 1 step 1 until n do
  begin w := xmin[j];  xmin[j] := w + eps[j];  ft := GFUN
        (xmin);
    if ft < fmin then go to set2;  xmin[j] := w − eps[j];
    ft := GFUN(xmin);  if ft < fmin then go to set2;  xmin[j]
        := w
  end;
if k1 < 3 then k1 := 1; go to bgend;
set2:  k1 := 2;  go to bgend;
outcd:  k1: =3; go to recheck;
bgend:  end MINIFUN;
```

ALGORITHM 130
PERMUTE

Lt. B. C. Eaves
U.S.A. Signal Center and School, Fort Monmouth, N. J.

```
procedure PERMUTE (A, n, x)
array  A;  integer n, x;
comment  Each entry into PERMUTE generates the next per-
  mutation of the first n elements of A. If A is read as a number
  (A[1]A[2] ··· A[n]), each generation is larger than the last:
  n := 4, x := 1
    A[1]   1  1  1  8  8  8
    A[2]   1  8  8  1  1  8          Permutations = 4!/2!2!
    A[3]   8  1  8  1  8  1
    A[4]   8  8  1  8  1  1  end
  Identical elements in A reduce the number of permutations. The
  array should be ordered before the first call on PERMUTE.
  Integer x specifies the first elements whose order should be pre-
  served:  n := 4, x := 3
    A[1]   1  1  1  4
    A[2]   2  2  4  1          Permutations = 4!/3!
    A[3]   3  4  2  2
    A[4]   4  3  3  3  end
  Before the first call on PERMUTE for a given array, first
  should be made true. If more is true, then PERMUTE was able
  to give another permutation;
begin array B[1:n];  integer f, i, k, m, p;  real r;  own real t;
  if first then t := A[x]; first := false;
  for i := 1 step 1 until n do B[i] := 0;
  for i := n step −1 until 2 do
    begin if A[i] > t∧A[i] > A[i − 1] then go to find;  end;
  more := false; go to exit;
find:  for k := n step −1 until i do
        begin if A[k] > t∧A[k] > A[i −1] then
          begin B[k] := A[k];  m := k;  end;  end;
      for k := n step −1 until i do
        begin if B[k] > 0 ∧B[k] < B[m] then
          begin B[m] := B[k];  f := k;  end;  end;
      r := A[i − 1];  A[i − 1] := B[m];  A[f] := r;
schell:  p := i − 1; m := n − p;
        for m := m/2 − .4 while m > 0 do
          begin k := n − m;
          for f := p + 1 step 1 until k do
            begin i := f;
comp:       if A[i] > A[i + m] then
              begin r := A[i + m];  A[i + m] := A[i];
              A[i] := r; i := i − m;
              if i ≧ p + 1 then go to comp;
              end end end schell;
exit:  end PERMUTE
```

ALGORITHM 131
COEFFICIENT DETERMINATION*

V. H. Smith and M. L. Allen
Georgia Institute of Technology, Atlanta 13, Ga.

* This procedure pertains to research work sponsored in part by NSF Grant G-7361.

```
procedure  DET (n, G, H);
array      G, H;  integer n;
comment  Given the first n coefficients of the power series
  G(z) = g1 + g2z + g3z² + ··· + gnz^(n−1) + ··· , and H(z) = h1 +
  h2z + h3z² + ··· + hnz^(n−1) + ··· , this procedure determines the
  coefficients di , i = 1, ··· , n, of the power series which is the
  expansion of the quotient H(z)/G(z). It is assumed that g1 ≠ 0.
  The arrays G and H initially contain the coefficients of G(z) and
  H(z), respectively. The integer n is the number of known coeffi-
  cients in the expansion of G(z) and H(z). At the conclusion, Hi
  contains the coefficient di . The procedure may also be useful in
  calculating residues for certain complex functions. Suppose
  F(z) = H(z)/G(z) is a complex valued function of a complex
  variable and that F has a pole of order m at z = b, where H(z) =
  Σ_{k=1}^{∞} hk(z − b)^(k−1), G(z) = Σ_{k=1}^{∞} gk(z − b)^(k+m−1), and g1 ≠ 0,
  h1 ≠ 0. The required residue at z = b is dm where

        D(z) = [Σ_{k=1}^{∞} hk(z − b)^(k−1)] / [Σ_{k=1}^{∞} gk(z − b)^(k−1)]

             = Σ_{j=1}^{∞} dj(z − b)^(j−1).

  For more on this, one is referred to Einar Hille, "Analytic Func-
  tion Theory, Vol. I, " Ginn and Co., 1959, pages 242–244;
begin integer  i, j, n;  real alpha, beta;
               alpha := 1/G[1];
          for  j := 1 step 1 until n do
        begin  beta := alpha × H[j];
          for  i := j + 1 step 1 until n do
               H[i] := H[i] − (beta × G[i − j + 1]) end;
          for  j := 1 step 1 until n do
               H[j] := H[j] × alpha;
end            DET
```

ALGORITHM 132
QUANTUM MECHANICAL INTEGRALS OVER ALL SLATER-TYPE INTEGRALS

J. C. Browne
The University of Texas, Austin, Tex.

```
real procedure:  allslater (p,q,pe,qe,np,nq,lp,lq,mp,mq,na,nb)
                 internuclear distance: (r);
                 real pe,qe,r;  integer p,q,np,nq,lp,lq,mp,mq,
                   na,nb;
comment  The Slater-type orbitals frequently used in quantum
  mechanical calculations on atoms and molecules are defined as
  p = k(np,pe) r^(np−1)e^(−(pe)r) Y_lp^mp(θ, φ), where k(np,pe) is a normal-
  ization constant, Y1^m(θ,φ) is a spherical harmonic with the
  phase convention [Y1^m(θ,φ)]* = (−1)^m Y1^(−m)(θ,φ), np is a positive
  integer, lp is an integer, lp < np, mp is an integer, − lp ≦ mp
  ≦ lp; and pe is a real positive constant. Algorithm 110, Y. A.
  Kruglyak and D. R. Whitman (Comm. ACM, July 1962) serves
  to compute integrals over certain operators of a quite restricted
  class of Slater-type orbitals, np ≧ 4, lp = 1, mp = 0. The algo-
  rithm given here will compute all integrals of the form
                    ∫ pc(r_c^nc)qc dτ
  which can be expressed in terms of the simple An(b) and Bn(a)
  functions. The subscript c denotes either of the two nuclei of
```

a diatomic molecule. These integrals include all those one-electron integrals necessary for a conventional energy calculation on a diatomic molecule. In the arguments of *allslater* p and q are numerical designations for the respective orbitals. p and q are even or odd as they respectively are associated with the "left," a, nucleus or "right," b, nucleus of a diatomic molecule. Global arrays, *fact* 1, of factorials and *binom*, of binomial coefficients are assumed. We first define some procedures utilized by *allslater*. The main program begins at the label *set*;

```
begin real norm, r2, alpha, beta, s, clp, clq, bpci;
  integer nsum, lsum, peven, qeven, podd, godd, limitp, limitq,
  g, h, i, j, n1p, n1q, lmp, lmq, gama, gamb, aidaa, aidab, gam,
  aida, num2; real array avalues [0:21], bvalues[0:21]; real pro-
  cedure c1, bpc, modulus;
real procedure c1(l,m,j); value l,m,j, integer l,m,j;
  begin c1 := ((-1)↑j)× fact1[2×(l - j)]/((2↑l) × fact1
    [l-2×j-m]×
    fact1[l - j]× fact1[j])
  end c1;
real procedure bpc(i, j, k); value i,j,k, integer i,j,k;
  begin real t; integer m; t := 0;
    for m := 0 step 1 until k do
    begin t := t + ((-1) ↑ (k - m)
      × binom [i, m] × binom [j, k - m]
    end
  end bpc;
real procedure modulus (i, j); value i, j; integer i, j;
  begin modulus := 1 -abs(i ÷ j) × j
  end modulus;
procedure avector (b, nmax, avalues); value b, nmax;
    real b; integer nmax; real array avalues;
  begin integer m;
    avalues[0] = exp(-b)/b;
    if nmax = 0 then go to exit;
    for m = 1 step 1 until nmax do
    begin avalues[m] = avalues[0] + (m/b) × avalues[m - 1]
    end;
exit: end avector;
procedure bvector(a nmax, bvalues); value a, nmax; real a;
  integer nmax; real array bvalues; real procedure modulus;
comment This procedure computes a sequence of values for the
  integral, B_n(a) = ∫_{-1}^{1} x^n e^{-ax} dx, for n = 0 to n = nmax. If a ≧
  alim then B_0(a) is computed and upward recursion is used to
  generate the higher n values. If a < alim then B_nmax(a) is com-
  puted by series expansion and downward recursion is used to
  generate the smaller n values. alim is determined within the
  program by a simplification of a result of Gautschi (J. ACM 8,
  21 (1961)). Gautschi has made an analysis of the recursive pro-
  cedures for the B_n(a) which could be taken as a model for workers
  in molecular quantum mechanics;
  begin real fxx, fxy, numerator, denom, sum, factor1, tsum
    factor2, t, aa; integer m,mn;
    begin if abs(a) ≧ ((nmax+nmax/6+3)/2.3) then
      up: begin fxx := exp(a);
            fxy := 1/fxx;
            bvalues [0] := (fxx-fxy)/a;
            for m := 1 step 1 until nmax do
          begin fxx := - fxx;
            bvalues[m] := (fxx-fxy + m ×
            bvalues[m-1])/a
          end;
          go to exit;
        end up;
      down: begin aa := axa;
            if modulus (nmax, 2)≠0 then
          setodd: begin numerator := nmax + 2;
            sum := a/numerator;
            factor1 := -2;
            factor2 := 3;
            go to compute;
          end setodd;
        seteven: begin numerator := nmax + 1;
            sum := 1/numerator;
            factor1 := factor2 := 2;
          end seteven;
        compute: begin denom := numerator + 2;
            t := sum;
            t := ((((t/factor2)×aa)
              /(factor2-1))×numerator)
              /denom;
            tsum := t + sum;
            if (sum-tsum)=0 then
            begin bvalues[nmax] := sum × factor1;
              go to recur;
            end;
          begin factor2 := factor2 + 2;
              numerator := denom;
              sum := tsum;
            go to compute;
          end compute;
        recur: begin fxx := exp(a);
              fxy := 1/fxx;
              mn := nmax -1;
              if modulus(nmax, 2) ≠ 0 then
              fxx := -fxx;
              for m := mn step -1 until 0 do
              begin fxx := -fxx;
                bvalues[m] := (fxx+fxy + a ×
                bvalues[m+1])/(m+1);
              end
          end recur;
        end down;
    end;
exit: end bvector;
set: begin if (mp + mq) ≠ 0 then
      begin allslater := 0.0; go to exit end;
    set: begin norm := sqrt (((2×pe)↑
        (2×np+1) × (2×lp+1) × fact1[lp-mp] × (2×qe)↑
        (2×nq+1) × (2×lq+1) × fact1[lq-mq])/(fact1[2×
        np] × fact1[lp+mp] × fact1[2×nq] × fact1[lq+mq] ×
        4));
      nsum := np+nq;
      lsum := lp+lq;
      r2 := r/2;
      norm := norm × (r2↑(nsum+1+na+nb));
      alpha := r2 ×(pe+qe);
      beta := r2 ×(((-1)↑p)×pe + ((-1)↑q) × qe);
      num2 := 2;
      avector (alpha, nsum, avalues);
      bvector (beta, nsum, bvalues);
      peven := modulus (p+1,2);
      qeven := modulus (q+1,2);
      podd := modulus (p,2);
      qodd := modulus (q,2);
      limitp := (lp-mp) ÷num2;
      limitq := (lq-mq) ÷num2;
      s := 0;
    end set;
sum: begin for g := 0 step 1 until limitp do
      begin c1p := c1(lp,mp,g);
      for h := 0 step 1 until limitq do
      begin c1q := c1(lq,mq,h);
      n1p := np-lp+2×g-1;
      n1p := nq-lq+2×h-1;
      lmp := lp-mp-2×g;
      lmp := lq-mq-2×h;
```

```
        gama := n1p × peven + n1q × qeven +1 +na;
        gamb := n1p × podd + n1q × godd +1 +nb;
        aidaa := lmp × peven + lmq × qeven;
        aidab := lmp × podd + lmq × qodd;
        gam = gama + gamb;
        aida = aidaa + aidab;
        for i := 0 step 1 until gam do
        begin bpci := bpc(gama, gamb, i);
        for j := 0 step 1 until aida do
        begin
        s := s+ c1p × c1q × bpci ×
        bpc(aidaa, aidab, j) ×
        avalues[nsum+na+nb−i−j]
        × bvalues[lsum −2 × (g+h) +i−j];
        end
        end
        end
        end;
      allslater := s × norm;
      end sum;
exit: end;
end allslater;
```

ALGORITHM 133
RANDOM

PETER G. BEHRENZ
Mathematikmaskinnämnden, Stockholm, Sweden

```
real procedure RANDOM (A, B, X0);
value A, B, X0;
real A, B;
integer X0;
comment RANDOM generates a rectangular distributed
  pseudo-random number in the interval A < B. X0 is an integer
  starting-value. The first time RANDOM is used in a program
  X0 should be a positive odd integer with 11 digits, X0 < 2^35 =
  34 359 738 368. The following times RANDOM is used, X0 should
  be X0 = 0. The mathematical method used is X_{n+1} = 5 X_n
  (mod 2^35). This sequence has period 2^33). RANDOM was suc-
  cessfully run on FACIT EDB using FACIT-ALGOL 1, which
  is a realization of ALGOL 60 for FACIT EDB, except for the
  declarator own, which is not included in FACIT-ALGOL 1.
  To test RANDOM, we computed 1/N ∑ X_n and 1/N ∑ X_n^2
  in the interval 0,1 for N = 500, 1000, 5000. The starting-
  value was X0 = 28 395 423 107. The results were 0.50625,
  0.48632, 0.50304 and 0.34304, 0.31681, 0.33469. Theoretically
  one expects 0.50000 and 0.33333;
begin
integer M35, M36, M37;
own integer X;
if X0 ≠ 0 then begin
X := X0; M35 := 34 359 738 368; M36 := 68 719 476 736;
M37 := 137 438 953 472 end; X := 5 × X;
if X ≧ M37 then X := X − M37;
if X ≧ M36 then X := X − M36;
if X ≧ M35 then X := X − M35;
RANDOM := X/M35 × (B − A) + A end
```

ALGORITHM 134
EXPONENTIATION OF SERIES

HENRY E. FETTIS
Aeronautical Research Laboratories, Wright-Patterson Air Force Base, Ohio

```
procedure SERIESPWR(A, B, P, N);
comment This procedure calculates the coefficients B[i] for
  the series (f(x))^P ≡ g(x) ≐ 1 + ∑ B[i] × x ↑ i, (i = 1, 2, ··· , N)
  given the coefficients of the series f(x) = 1 + ∑A[i] × x ↑ i.
  P may be any real number;
value A, P, N;
array A, B;
integer N;
begin integer i, k;
real p, s;
  B[1] := P × A[1];
  for i := 2 step 1 until N do
  begin s := 0;
    for k := 1 step 1 until i−1 do
    S := s + (P × [i−k] − k) × B[k] × A[i−k];
    B[i] := P × A[i] + (s/i)
    end for i;
  end SERIESPWR
```

ALGORITHM 135
CROUT WITH EQUILIBRATION AND ITERATION

WILLIAM MARSHALL MCKEEMAN*
Stanford University, Stanford, Calif.

* This work was supported in part by the Office of Naval Research under contract Nonr 225(37).

```
procedure LINEARSYSTEM (A) order:(n) right-hand sides:(B)
  number of right-hand sides:(m) answers:(X) determinant:(det,
  ex) condition of A:(cnr);
integer n, m, ex; real det, cnr; real array A, B, X;
comment, LINEAR SYSTEM uses Crout's method with row
  equilibration, row interchanges and iterative improvement
  for solving the matrix equation AX = B where A is n × n and
  X and B are n × m. As special cases one sees that: for m ≦ 0,
  only the determinant of A is evaluated, for m = 1, the algo-
  rithm solves a system of n equations in n unknowns, for m = n
  and B = the identity matrix, the algorithm inverts A.
    If the algorithm breaks down for a singular or nearly singular
  matrix A, exit to a non-local label "singular" is provided. Five
  auxiliary procedures: EQUILIBRATE, CROUT, PRODUCT,
  RESIDUALS and SOLVE are declared with appropriate com-
  ments after the end of this procedure. This code is the result of
  the joint efforts of G. Guthrie, W. McKeeman, Cleve Moler,
  Margaret Salmon, Alan Shaw and R. Van Wyk. It was written
  following ideas presented by J. H. Wilkinson as a visiting lec-
  turer in Professor George E. Forsythe's class in Advanced Nu-
  merical Analysis at Stanford, 1962;
begin integer array pivot [1:n]; integer i, j, k; real mx;
  real array LU[1:n, 1:n], y, res, mult[1:n];
  comment, remove appropriate factors from the rows of A... ;
  EQUILIBRATE(A, n, mult);
  comment ... and save the result for the eventual computation
  of residuals during iteration;
  for i := 1 step 1 until n do
    for j := 1 step 1 until n do LU[i,j] := A[i,j];
  comment, decompose the matrix into triangular factors;
  CROUT(LU, n, pivot, det);
  comment, assuming that there was no exit to "singular",
  evaluate the determinant in the form det × (10.0 ↑ ex);
  for i := 1 step 1 until n do y[i] := LU[i,i] × mult[i];
  det := det × PRODUCT(y,1,n,ex);
  comment, now begin to process right-hand sides;
  for k := 1 step 1 until m do
  begin integer i, count, limit; real normy, kr;
    kr := k;
    comment, scale the right-hand side;
    for i := 1 step 1 until n do res[i] := B[i,k] := B[i,k]/mult[i];
    comment, store the first approximation and its L(1) norm;
    normy := 0;
```

```
    SOLVE(LU, n, res, pivot, y);
    for i := 1 step 1 until n do
    begin
      normy := normy + abs(y[i]);
      X[i,k] := y[i]
    end;
    comment, enter the iterating loop. The iteration is terminated on the integer "limit" which itself is determined on the basis of the success of the first iteration and a machine-dependent real number designated here by "eps". For "eps", the programmer must insert the largest real number such that eps + 1.0 = 1.0 ;
    for count := 1, 2 step 1 until limit do
    begin integer i; real t;
      comment, compute the residuals of the solution y;
      RESIDUALS(A,n,B,k,X,res);
      comment ... and find the next increment to the solution;
      SOLVE(LU,n,res,pivot,y);
      comment, set up termination conditions;
      if count = 1 then
      begin real normdy;
        normdy := 0;
        for i := 1 step 1 until n do normdy := normdy+abs(y[i]);
        if normdy = 0 then begin cnr := 1.0; go to enditer end;
        t := normy/normdy;
        comment, The quantity ‖ A ‖·‖ A⁻¹ ‖ (spectral norm) is called the condition number of the matrix A. It is a measure of the difficulty in solving the input equation and appears naturally in error bounds for the solution (see Wilkinson [3]). cnr is a direct measure of the error and experimentally approximates the condition number;
        cnr := ((kr − 1.0) × cnr + 1.0/(eps × t))/kr;
        if t < 2.0 then go to singular;
        limit := ln(eps)/ln(1.0/t);
      end;
      comment, store the new approximation;
      for i := 1 step 1 until n do X[i,k] := X[i,k] := X[i,k] + y[i];
    end iteration;
    enditer:
  end right-hand sides
end LINEAR SYSTEM;
procedure EQUILIBRATE (A) order:(n) multipliers:(mult);
integer n; real array A, mult;
comment, scaling the rows of the matrix A to roughly the same maximum magnitude (here, dividing by the largest element) allows the procedure CROUT to select effective pivotal elements for the Gaussian decomposition of the matrix. The iterating procedure will converge to the solution for the equilibrated matrix rather than the input matrix. If the matrix is badly conditioned then the solution is sensitive to perturbations in the input and the scaling division must be done not by the largest element but rather by the power of the machine number base (2 and 10 for binary and decimal machines, respectively) nearest the largest element so as to avoid rounding errors. Equilibration is discussed in reference [3] p. 284;
begin integer i; real mx;
  for i := 1 step 1 until n do
  begin integer j;
    mx := 0.0; comment, find the largest element;
    for j := 1 step 1 until n do
      if abs(A[i,k]) > mx then mx := abs(A[i,k]);
    if mx = 0.0 then go to singular;
    comment, now store the multiplier and scale the row;
    mult[i] := mx; comment := base ↑ ex for exact scaling;
    if mx ≠ 1.0 then
      for j := 1 step 1 until n do A[i,j] := A[i,j]/mx
  end
end EQUILIBRATE;
procedure CROUT (A) order:(n) pivots:(pivot) interchanges:(sg).
integer n; integer array pivot; real array A; real sg;
comment, this is Crout's method with row interchanges as formulated in reference [1] for transforming the matrix A into the triangular decomposition LU with all the L[k,k] = 1.0. pivot[k] stores the index of the pivotal row at the k-th stage of the elimination for use in the procedure SOLVE;
begin integer i, j, k, imax, p; real t, quot;
  real procedure IP1 (A) extra term:(t) length:(f);
  integer f; real t; real array A; comment non-local i, j, k;
  comment, IP1 forms a row by column inner product of A, namely the sum of A[i,p] × A[p,k] for p := 1, 2, ... , f, and then adds the extra term t. If f < 1, the value of IP1 is t. This procedure is the inner loop of the algorithm. The programmer can expect a substantial advantage from substituting a faster and more accurate inner product here;
  begin real sum; integer p;
    sum := t;
    for p := 1 step 1 until f do sum := sum + A[i,p] × A[p,k];
    IP1 := sum
  end IP1;
  sg := 1.0;
  comment, k is the stage of the elimination;
  for k := 1 step 1 until n do
  begin
    t := 0;
    for i := k step 1 until n do
    begin comment, compute L. Note that the first calls on IP1 are empty;
      A[i,k] := −IP1(A, −A[i,k],k−1);
      if abs(A[i,k]) > t then
        begin t := abs(A[i,k]); imax := i end
    end;
    if t = 0 then go to singular;
    comment, A[imax,k] is the largest element in the remainder of column k. Interchange rows if necessary and record the change;
    pivot[k] := imax;
    if imax ≠ k then
    begin
      sg := −sg;
      for j := 1 step 1 until n do
      begin
        t := A[k,j]; A[k,j] := A[imax, j]; A[imax, j] := t
      end
    end;
    comment, compute a column of multipliers;
    quot := 1.0/A[k,k];
    for i := k+1 step 1 until n do A[i,k] := A[i,k] × quot;
    comment, and compute a row of U;
    for j := k+1 step 1 until n do
      A[k,j] := −IP1(A,−A[k,j],k−1)
  end
end CROUT;
real procedure PRODUCT (factors) start:(s) finish:(f)
  exponent:(ex);
integer s,f,ex; real array factors;
comment, PRODUCT multiplies the numbers stored from index s through f inclusive in the array "factors", preventing exponent overflow. The answer is normalized so that 1.0 > abs (PRODUCT) ≧ 0.1. The exponent appears in ex;
begin integer i; real p, p1;
  ex := 0; p := 1.0;
  for i := s step 1 until f do
  begin
    p1 := factors [i];
    if abs(p1) < 0.1 then begin p1 = 10.0 × p1; ex := ex−1
      end;
    p := p × p1;
```

```
    if p = 0 then begin ex := 0; go to fin end;
 1: if abs(p) < 0.1 then
    begin p := p × 10.0;  ex : = ex−1; go to 1 end;
 2: if abs(p)≧1.0 then
    begin p := p/10.0;  ex := ex + 1;  go to 2 end;
  end;
  fin: PRODUCT := p
end PRODUCT;
procedure RESIDUALS (A) order:(n) right-hand sides:(B)
  column of B:(k) approximate solution:(X) residuals:(res);
  integer n, k; real array A, B, X, res;
comment, RESIDUALS computes b − Ay where b is the kth
  column of the right-hand side matrix B and y is the kth column
  of X;
    real procedure IP2 (A) row: (i) order:(n) approximate
      solution:(X)
    column:(k) extra therm:(t);
    integer i, k, n;  real t real array A, X;
    comment, IP2 forms the inner product of row i of the matrix
      A and column k of the solution matrix X, then adds the
      single term t. It is essential that IP2 be an "accumulating"
      or double precision inner product as discussed in reference
      [3] p. 296. The value of IP2 is the rounded single precision
      result of the double precision arithmetic. The body of the
      procedure is left undefined;
begin integer i;
for i := 1 step 1 until n do
  res[i] := −IP2(A,i,n,X,k,−B[i,k])
end RESIDUALS;
procedure SOLVE (A) order:(n) right-hand side:(b) pivots:
  (pivot) answer:(y);
integer n; integer array pivot; real array A, b, y;
comment, SOLVE processes a right-hand side b and then back-
  solves for the solution y using the LU decomposition provided
  by CROUT;
begin integer k, p;  real t;
  for k := 1 step 1 until n do
  begin
    t := b[pivot[k]];  b[pivot[k]] := b[k];
    for p := 1 step 1 until k−1 do t := t − A[k,p] × b[p];
    b[k] := t
  end ...having modified b by L inverse;
  comment, now the back solution for y;
  for k := n step −1 until 1 do
  begin
    t := b[k];
    for p := k+1 step 1 until n do t := t − A[k,p] × y[p];
    y[k] := t
  end backsolution
end SOLVE
```


REFERENCES

1. GEORGE E. FORSYTHE, Crout with Pivoting. Algorithm 16. *Comm. ACM 3*, 2 (Sept. 1960), 507.
2. DEREK JOHANN ROEK, Simultaneous System of Equations and Matrix Inversion Routine. Algorithm 92. *Comm. ACM 5*, 5 (May 1962), 286.
3. J. H. WILKINSON, Error Analysis of Direct Methods of Matrix Inversion, *J. ACM 8*, 3 (July 1961), 281–330.

ALGORITHM 136
ENLARGEMENT OF A GROUP

M. WELLS*
University of Leeds, England
* Currently with Burroughs Corporation, Pasadena, California


```
procedure Enlarge group (G, n, g, Abelian);
  array G, g;  integer n;  Boolean Abelian;
comment This procedure combines the element g with the sub-
  group G, of n elements, to form a new group. The Boolean
  Abelian has the value true if the group to which G and g belong
  is Abelian. Two procedures, multiply and equal are assumed
  to be declared: multiply (G[i]) by : (G[j]) to give : (G[k]) will set
  the element G_k equal to the product of the elements G_i and G_j.
  equal (G[i], G[j]) is a Boolean procedure whose value is true
  if, and only if, the elements G_i and G_j are equal. On leaving the
  procedure the enlarged group is in G, and n is equal to the
  number of elements in the new sub-group G. The procedure
  will function correctly if g is included in G on entry. It is prob-
  able that g and the elements of G will be arrays, and the pro-
  cedure body will, in practice, need to be altered considerably.
  The procedure has been used successfully in connection with
  problems of space-group theory;
begin integer i, j, k;
  for i := 1 step 1 until n do
  if equal (G[i], g) then go to not new generator;
  n := n + 1;  G[n] := g;
  for i := n step 1 until n do
  begin for j := 1 step 1 until n do
  begin multiply (G[i], G[j], G[n+1]);
    for k := 1 step 1 until n do
    if equal (G[k], G[n+1]) then go to not new element 1;
    n := n + 1;
not new element 1: if Abelian then go to take next element;
  multiply (G[j], G[i], G[n+1]);
  for k := 1 step 1 until n do
  if equal (G[k], G[n+1]) then go to not new element 2;
  n := n + 1;
not new element 2: take next element:
  end of j-loop;
  end of i-loop;
not new generator: end of group enlargement
```

ALGORITHM 137
NESTING OF FOR STATEMENT I

DAVID M. DAHM & M. WELLS*
Burroughs Corp., Pasadena, Calif.
* On leave of absence from the University of Leeds, England.


```
procedure Fors 1 (n, P);
  value n;  integer n;  procedure P;
  comment Fors 1 generates a nest of n for statements with the
    procedure P at their center. Two non-local arrays I and U,
    which give the value of the controlled variable and its upper
    bound for each level are assumed to be declared;
  begin integer j;
    if n = 0 then P
    else for j := 1 step 1 until U[n] do
    begin I[n] :=j;  Fors 1 (n−1,P) end end Fors 1
```

ALGORITHM 138
NESTING OF FOR STATEMENT II

DAVID M. DAHM & M. WELLS*
Burroughs Corp., Pasadena, Calif.
* On leave of absence from the University of Leeds, England.


```
procedure Fors 2 (P);
  procedure P;
  comment Fors 2 performs the same function as Fors 1, but is
    more economic of storage space. It is expected, however,
    that Fors 1 would be more economic of time. The formal
    parameter n is now replaced by the non-local integer n;
```

```
begin if n = 0 then P
  else for I[n] := 1 step 1 until U[n] do
  begin n := n−1; Fors 2 (P) end;
  n := n + 1 end Fors 2
```

ALGORITHM 139
SOLUTIONS OF THE DIOPHANTINE EQUATION

J. E. L. PECK
University of Alberta, Calgary, Alberta, Canada

```
procedure Diophantus (a,b,c); integer a,b,c;
comment This procedure seeks the integer solutions of the
  equation ax + by = c, where the integers a,b,c are given. It
  assumes a non-local integer M, which should be as large as
  storage will allow, two nonlocal labels INDETERMINATE
  and NO SOLUTION and two non-local Boolean variables
  'general solution' and 'time permits' which are self explanatory.
  It also assumes the procedures abs, sign and print;
begin integer n,r,s,d,i; integer array q[1:M];
n := i := 0; d := s := abs(a); r := abs(b);
comment d will become the greatest common divisor of a and b.
  If b = 0 then d = | a |. The vector q will retain the successive
  quotients in the Euclidean algorithm r_{i-1} = r_{iqi} + r_{i+1},
  i = 1, 2, ··· , n, where 0 ≦ r_{i+1} < r_i , r_0 = | a |, r_1 = | b |,
  and r_{n+1} = 0;
for i := i + 1 while r ≠ 0 do
  begin n := i; d := r; q[i] := s ÷ d;
  r := s − d × q[i]; s := d end This records the quotients and
    the number n of divisions for use below;
if d = 0 then go to if c = 0 then INDETERMINATE
else NO SOLUTION; comment The case d = 0 occurs when
  a² + b² = 0. If d now does not divide c then the equation can-
  not be solved so;
if (c ÷ d) × d ≠ c then go to NO SOLUTION;
if d ≠ 1 then
  begin a := a/d, b := b/d; c := c/d end, which removes
    the common factor and reduces the equation to the case
    where a and b are relatively prime;
  begin comment We shall now find u_1 and v_1 in order to
    express
    1 = au_1 + bv_1 , using the relations r_n = r_iv_i + r_{i-1}u_i ,
    i = n, n−1, ··· , 1, v_n = 1, u_n = 0, and r_{i+1} = −r_iq_i + r_{i-1} ,
    i = n−1, n−2, ··· , 1; integer u,v;
  if n = 0 then
    begin v := 0; u := 1 end, which takes care of the case
      b = 0
  else
    begin v := 1; u := 0;
    for i := n−1 step −1 until 1 do
      begin integer t;
      t := v; v := u − v × q[i]; u := t
      end i
    end the case n ≠ 0. It remains now to multiply the equality
      1 = au_1 + bv_1 through by c;
    begin integer x0, y0;
    x0 := c × u × sign(a); y0 := c × v × sign(b); print (x0,y0);
    comment If x_0 ,y_0 is a particular solution then x_0 ± ib,
      y_0 ∓ ia, i=1,2, ... gives the general solution. Therefore;
    if general solution then
      begin u := b; v := a;
      A: print(x0 + u, y0 − v); print(x0−u, y0 + v);
      u := u + b; v := v + a;
      if time permits then go to A
      end general solution and
    end solution.
  end u,v
end Diophantus.
```

ALGORITHM 140
MATRIX INVERSION

P. Z. INGERMAN
University of Pennsylvania, Philadelphia, Penn.

```
procedure invert (a) of order:(n) with tolerance:(eps) and
  error exit:(oops);
value n, eps; array a; integer n; real eps; label oops;
comment This procedure inverts a matrix by using elementary
  row operations. Although the method is not particularly good
  for ill-conditioned matrices, the simplicity of the algorithm
  and the fact that the inversion occurs in place make it useful
  on occasion;
begin integer i;
  for i := 1 step 1 until n do
  begin integer j, k; real q;
    q := a[i,i];
    if abs(q)≦abs(eps) then go to oops;
    a[i,i] := 1;
    if q≠1 then for k := 1 step 1 until n do a[i,k] := a[i,k]/q;
    for j := 1 step 1 until n do
      if i≠j then
        begin q := a[j,i]; a[j,i] := 0;
          for k := 1 step 1 until n do
            a[j,k] := a[j,k]−q×a[i,k] end end end
```

ALGORITHM 141
PATH MATRIX

P. Z. INGERMAN
University of Pennsylvania, Philadelphia, Penn.

```
procedure find path (a, n);
value n; Boolean array a; integer n;
comment This procedure is merely an Algol implementation
  of the method of Warshall (JACM 9(1962), 11–12). Some ad-
  vantage is taken of the characteristics of the problem to in-
  crease the efficiency;
begin integer i, j, k;
  for j := 1 step 1 until n do
  for i := 1 step 1 until n do
  if a[i,j] ∧ i≠j then
  for k := 1 step 1 until n do
  a[i,k] := a[i,k]∨a[j,k] end findpath
```

CERTIFICATION OF THE CALCULATION OF EASTER...[Donald Knuth, *Comm. A.C.M.*, Apr. 1962]

M. R. WILLIAMS
University of Alberta, Calgary, Alberta, Canada

The two programs, written to demonstrate ALGOL and COBOL, were translated into FORTRAN for the IBM 1620. Both programs correctly determined the month and day of the "Western Easter" for the years 1901 to 1999. No furher checking was done because a more comprehensive reference list of the dates of the "Western Easter" was not available.

If the statement:

epact := mod (11 × *golden number* + 20 + *Clavian correction* − *Gregorian correction*, 30);

is changed to:

epact := *mod* (11 × *golden number* + 19 + *Clavian correction* − *Gregorian correction*, 30) + 1;

it eliminates the statement:

if *epact* ≦ 0 **then** *epact* := *epact* + 30;

CERTIFICATION OF ALGORITHM 84
SIMPSON'S INTEGRATION [P. E. Hennion, *Comm. ACM*, Apr. 62]

PETER G. BEHRENZ
Matematikmaskinnämnden, Stockholm, Sweden

SIM was successfully run on FACIT EDB using FACIT-ALGOL 1, which is a realization of ALGOL 60 for FACIT EDB. No changes in the program were necessary. To test SIM some polynomials were integrated.

CERTIFICATION OF ALGORITHM 94
COMBINATION [J. Kurtzberg, *Comm. ACM*, June 1962]

RONALD W. MAY
University of Alberta, Calgary, Alberta, Canada

Algorithm 94 was translated into FORTRAN for the IBM 1620 and run successfully with no corrections. The variable A, however, has not been declared.

REMARK ON ALGORITHM 99
EVALUATION OF JACOBI SYMBOL [S. J. Garland and A. W. Knapp, *Comm. ACM 6*, June 1962]

RONALD W. MAY
University of Alberta, Calgary, Alberta, Canada

One syntactical error was found in this procedure. It occurs in the second **if** statement following the **label** even. The statement

```
if q then if parity ((m↑2−1)÷8) then
    p := ¬ p;
```

might be changed as follows.

```
    if q then go to CHECK;
next 1: if n = 1 then go to done;
CHECK: if parity ((m ↑ 2 − 1) ÷ 8) then
    p := ¬ p;
    go to next 1;
```

The two statements beginning with *CHECK* could be inserted before the **label** done and after the statement **go to** loop;.

REMARK ON ALGORITHM 106
COMPLEX NUMBER TO A REAL POWER [Margaret L. Johnson and Ward Sangren, *Comm. ACM 5*, Jul. 1962]

GRANT W. ERWIN, JR.
The Boeing Co., Renton, Wash.

The comment "if W is a reciprocal integer it does not follow that the desired power (a root) will be calculated" might better read "if W is the reciprocal of an integer N, the procedure will calculate an nth root, but possibly not the particular nth root desired. E.g. $w = \frac{1}{3}, x = -1, y = 0$ uields $A = \frac{1}{2}, B = \frac{1}{2}\sqrt{3}$ rather than the simpler $A = -1, B = 0$."

The comment should be made that it is assumed that the arctan function yields a result between $-\pi/2$ and $\pi/2$.

The following four corrections should be made:

(1) **if** $x<0 \wedge y < 0$ **then begin** THETA: = 3.1415927;
should read:
$\cdots$ THETA: = −3.1415927;

(2) go **to** RETURN **end**:
should read:
go to RETURN **end**;

(3) **if** $x = 0 \wedge y<0 \cdots$
should read:
if x = 0 $\wedge$ y>0 $\cdots$

(4) **if** $x = 0 \wedge y > 0$
should read:
if $x = 0 \wedge y < 0 \cdots$

CERTIFICATION OF ALGORITHM 135
CROUT WITH EQUILIBRATION AND ITERATION [William Marshall McKeeman,* *Comm. ACM*, Nov. 1962]

WILLIAM MARSHALL MCKEEMAN,
Stanford University, Stanford, Calif.

* This work was supported in part by the Office of Naval Research under contract Nonr 225(37).

A BALGOL translation of the algorithm was tested for accuracy, proper termination and running time on the Burroughs 220. The exact inverse of the Hilbert segment of order 6 can be stored in the 8-decimal-digit floating word of the B220 and was used in the accuracy and termination tests. The Hilbert segment H_6 is very ill-conditioned (for the spectral norm, $\| H_6 \| \cdot \| H_6^{-1} \| = 1.3 \times 10^7$). Hence the number of iterations required should not be taken as typical.

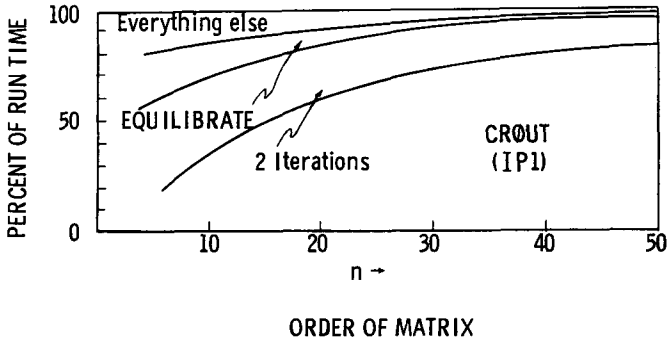
The $[n,n]$ element (mathematically $\frac{1}{11} = .090909 \cdots$) is representative of the behavior of the rest:

	"exact" equilibration (by powers of 10)	*equilibration by largest element in row*
initial solution	.092587535	.094091506
first iteration	.090877240	.091498265
second iteration	.090909695	.091570311
third iteration	.090909080	.091568310
fourth iteration	.090909091	.091568365
fifth iteration	terminated	.091568364
		terminated

Conclusions: The iterating procedure terminated correctly, or performed one extra iteration in each case. If the equilibration procedure alters the data, the iteration will converge to the solution for the altered matrix. If the matrix is ill-conditioned, as in the case above, the equilibration may cost a great deal more than it gains. As a practical matter, a machine language substitute for EQUILIBRATE which will not cause rounding of the data is probably the best course of action.

The running time is approximately proportional to n^3 as expected. If for a given machine, μ is the floating multiply time in seconds, one can expect that run time will be given by $rt := 1.3 \times \mu \times (n + 7) \uparrow 3$ seconds for a call on LINEARSYSTEM with one right-hand side.

The division of run time between the various phases of the algorithm is as follows:



REFERENCE:
1. SAVAGE AND LUKACS, Tables of inverses of finite segment of the Hilbert matrix. In Olga Taussky (Ed.), Contributions to the Solution of Systems of Linear Equations and the Determination of Eigenvalues, pp. 105–108, Nat. Bur. Standards Appl. Math. Series no. 39, U. S. Government Printing Office, Wash., D.C., 1954.