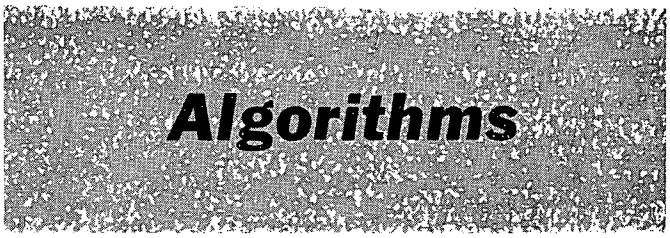

J. H. WEGSTEIN, Editor

ALGORITHM 194
ZERSOL

CARLOS DOMINGO
Universidad Central, Caracas, Venezuela

```
procedure ZERSOL (h, YI, m, epsi, F, f, Z); real h, epsi, f;
  array YI, Z; integer m; procedure F;
comment ZERSOL finds the simple zeros of the solution Y1(Y0)
  of the set of m first order differential equations  Yj = Fj(Y0,
  Y1, ···, Ym). h is the step of integration, epsi the error with
  which the zeros are to be determined (assuming no error in the
  process of integration). F(YS, j, v) is a procedure which calcu-
  lates the functions Fj, taking the arguments from the array
  YS and leaving the results in v. The search for zeros stops
  when Y0 > f. The zeros are stored as elements of the array Z.
  MR is a 4 × 4 matrix with the coefficients of a Runge-Kutta
  method. For example MR may be row-wise  0.5, 1, 0.5, 0, 1 − a,
  1 − a, 1 − a, 0.5, 1 + a, 1 + a, 1 + a, 0, 1/6, 1/3, 0.5, 0.5, where
  a = sqrt(2);
begin real v, r, d; integer j, s, n, k; array Q[1:m], YS[0:m],
  YAL [0:m], YT[1:m], MR[1:4,1:4]; switch S := NOZ, ZER;
  n := 1;
for d := h while YI[0] ≦ f do
  begin s := 1;
    R1: for j := 1 step 1 until m do
        begin Q[j] := 0.0; YS[j] := YI[j]; YT[j] := YI[j] end;
        YS[0] := YI[0];
    R2: for k := 1 step 1 until 4 do
        begin YS[0] := YS[0] + MR[k, 4] × d;
          for j := 1 step 1 until m do
         begin F(YS,j,v); v := v × d;
           r := MR[k,1] × v − MR[k,2]× Q[j];
           YT[j] := YT[j] + r;
           Q[j] := Q[j] + 3.0 × r − MR[k,3] × v
         end;
         for j := 1 step 1 until m do YS[j] := YT[j]
         end;
        go to S(s);
  NOZ: if sign(YI[1]) ≠ sign(YS[1]) then go to IT;
    TR: for j := 0 step 1 until m do YI[j] := YS[j]; go to R2;
    IT: s := 2;
        for j := 0 step 1 until m do YAL[j] := YS[j];
  ZER: d := d/2;
    if d ≦ epsi then go to STZ;
    if sign(YI[1]) = sign(YS[1]) then go to TR else go to R1;
  STZ: Z[n] := YI[0] := YI[0] + d; n := n + 1;
       for j := 0 step 1 until m do YI[j] := YAL[j]
  end;
end
```

ALGORITHM 195
BANDSOLVE

DONALD H. THURNAU
Marathon Oil Co., Littleton, Colo.

```
procedure BANDSOLVE (C,N,M,V); value N,M; integer
  N,M; real array C,V;
comment BANDSOLVE is effective in solving the matrix equa-
  tion AX = B when the matrix A is of large order and sparse
  such that a narrow band centered on the main diagonal includes
  all the non-zero elements. Parameter N is the order of A, and M
  is the width of the band, necessarily an odd number of elements.
  BANDSOLVE is very efficient because it operates only on the
  band portion of the matrix A, given in the N by M array C. The
  band elements of a given row of A appear in the same row of C
  but shifted such that element A[i,j] becomes C[i,j−i+(M+1)/2].
  All band elements whether zero or non-zero must be given. The
  values of undefined elements of C, such as C[1,1] or C[N,M], are
  irrelevant. The array V initially contains the vector B. After
  solution, the array V contains the answer vector X. The con-
  tents of array C are destroyed during solution which is done by
  Gauss elimination with row interchanges, followed by back sub-
  stitution;
begin integer JM,LR,I,PIV,R,J; real T;
  LR := (M+1) ÷ 2;
  for R := 1 step 1 until LR − 1 do
    for I := 1 step 1 until LR − R do
      begin for J := 2 step 1 until M do
        C[R,J−1] := C[R,J];
        C[R,M] := C[N+1−R,M+1−I] := 0
      end of row shifting and zero placement;
  for I := 1 step 1 until N − 1 do
    begin PIV := I;
      for R := I + 1 step 1 until LR do
        if abs(C[R,1])>abs(C[PIV,1]) then
          PIV := R;
      if PIV ≠ I then
        begin T := V[I];
          V[I] := V[PIV];
          V[PIV] := T;
          for J := 1 step 1 until M do
            begin T := C[I,J];
              C[I,J] := C[PIV,J];
              C[PIV,J] := T
            end J
        end of row interchange;
      V[I] := V[I]/C[I,1];
      for J := 2 step 1 until M do
        C[I,J] := C[I,J]/C[I,1];
      for R := I + 1 step 1 until LR do
        begin T := C[R,1]
          V[R] := V[R] − T × V[I];
          for J := 2 step 1 until M do
            C[R,J−1] := C[R,J] − T × C[I,J];
          C[R,M] := 0
        end R;
      if LR ≠ N then LR := LR + 1
    end of triangularization;
  V[N] := V[N]/C[N,1];
  JM := 2;
    for R := N − 1 step −1 until 1 do
      begin for J := 2 step 1 until JM do
        V[R] := V[R] − C[R, J] × V[R−1+J];
      if JM ≠ M then JM := JM + 1
      end of back solution
end BANDSOLVE
```

ALGORITHM 196
MULLER'S METHOD FOR FINDING ROOTS OF AN ARBITRARY FUNCTION

ROBERT D. RODMAN
Burroughs Corp., Pasadena, Calif.

```
procedure MULLER (p1, p2, p3, mxm, nrts, ep1, ep2, sw1, sw2,
  sw3, swr, rrt, irt);
value p1, p2, p3, mxm, nrts, ep1, ep2, sw1, sw2, sw3, swr;
integer mxm, nrts; boolean sw1, sw2, sw3, swr;
real p1, p2, p3, ep1, ep2; array rrt, irt;
begin comment procedure MULLER finds real and complex
  roots of an arbitrary function. p1, p2, and p3 are starting values.
  Roots nearest these points are found first. mxm is the maximum
  number of iterations to be made in finding any one root. ep1 and
  ep2 are specified as tolerance parameters. If ABS((X[i+1]-X[i])/
  X[i+1]) < ep1 or if the function value and modified function value
  are both less than ep2, a root has been found. If sw1 is true,
  then each iterant of each root is printed. If sw2 is true, the
  value of each root found is printed. If sw3 is true, then, when
  applicable, the complex conjugate of each root found is admitted
  as a root. If swr is true, only real roots are found. rrt and irt
  contain the real, and imaginary parts of each root found. Proce-
  dure function is the function generator and procedure com-
  plex performs necessary complex operations;
boolean bool; integer c1, rtc, i, itc; real rx1, rx2, rx3, ix1,
  ix2, ix3, rroot, iroot, rdnr, idnr, t1, it1, frroot, firoot, rfx1, rfx2,
  rfx3, ifx1, ifx2, ifx3, rh, ih, rlam, ilam, rdel, idel, t2, it2, t3, it3, t4,
  it4, rg, ig, rden, iden, rfunc, ifunc;
switch j := m2, m3, m4, m7, m11;
procedure function (reale, imag, reval, ieval);
value reale, imag; real reale, imag, reval, ieval;
begin comment Coding for this procedure must be inserted at
  compile time. reale and imag are the real and imaginary parts
  of the dependent variable. reval and ieval, the real and imaginary
  parts of the function;
end function;
procedure complex (a, ia, b, ib, k, c, ic);
value a, ia, b, ib, k; integer k;
real a, ia, b, ib, c, ic;
begin real temp; switch j := mpy, dvd, sqt;
      go to j[k];
mpy:  c := a × b − ia × ib; ic := a × ib + ia × b; go to exit;
dvd:  if (ib=0) ∧ (b=0) then begin ic := 0; c := 1;
      go to exit end; temp := b ↑ 2 + ib ↑ 2;
      c := (a×b+ia×ib)/temp; ic := (ia×b−a×ib)/temp;
      go to exit;
sqt:  if (ia=0) ∧ (a<0) then
      begin c := 0; ic := sqrt (−a) end
      else if ia = 0 then
      begin c := sqrt(a); ic := 0 end
      else begin temp := sqrt (a↑2+ia↑2);
        c := sqrt ((temp + a)/2);
        ic := if (temp − a) < 0 then 0
          else sqrt ((temp − a)/2) end;
      if ((b+c) ↑ 2 + (ib+ic) ↑ 2) < ((b−c) ↑ 2 + (ib−ic) ↑ 2)
      then begin c := b − c; ic := ib − ic end
      else begin c := b + c; ic := ib + ic end;
exit: end of complex;
start: for i := 1 step 1 until nrts do rrt [i] := irt [i] := 0; rtc := 0;
m0:   ix1 := ix2 := ix3 := c1, := iroot := itc := 0;
      rroot := p1; bool := false;
m1:   c1; := c1 + 1; rdnr := 1; idnr := 0;
      for i := 1 step 1 until rtc do
        begin
          complex (rdnr, idnr, rroot-rrt [i], iroot-irt [i], 1, t1, it1);
            rdnr := t1; idnr := it1
        end;
      function (rroot, iroot, t1, it1);
      complex (t1, it1, rdnr, idnr, 2, frroot, firoot);
      go to j[c1];
m2:   rfx1 := frroot; ifx1 := firoot; rroot := p2;
      go to m1;
m3:   rfx2 := frroot; ifx2 := firoot; rroot := p3;
      go to m1;
m4:   rfx3 := frroot; ifx3 := firoot; rx1 := p1;
      rx2 := p2; rx3 := p3; rh := rx3 − rx2;
      ih := ix3 − ix2;
      complex (rh, ih, rx2− rx1, ix2− ix1, 2, rlam, ilam);
      rdel := rlam + 1; idel := ilam;
m9:   if (rfx1=rfx2) ∧ (rfx2=rfx3) ∧ (ifx1=ifx2) ∧ (ifx2=ifx3)
      then begin rlam := 1; ilam := 0; go to m8 end;
      complex (rfx1, ifx1, rlam, ilam, 1, t1, it1);
      complex (rfx2, ifx2, rdel, idel, 1, t2, it2);
      t1 := t1 − t2 + rfx3; it1 := it1 − it2 + ifx3;
      complex (rdel, idel, rlam, ilam, 1, t2, it2);
      complex (t1, it1 t2, it2, 1, t3, it3);
      complex (rfx3, ifx3, t3, it3, 1, t1, it1);
      t1 := −4 × t1; it1 := −4 × it1;
      complex (rfx3, ifx3, rlam+rdel, ilam+idel, 1, t2, it2);
      complex (rdel ↑ 2−idel ↑ 2, 2×rdel×idel, rfx2, ifx2, 1, t3, it3);
      complex (rlam ↑ 2−ilam ↑ 2, 2×rlam×ilam, rfx1, ifx1, 1,
        t4, it4);
      rg := t4 − t3 + t2; ig := it4 − it3 + it2;
      if swr ∧ ((rg ↑ 2+t1)<0) then
        begin rden := rg; iden := ig := 0 end
      else complex (rg ↑ 2−ig ↑ 2+t1, 2×rg×ig+it1, rg, ig, 3,
        rden, iden);
      complex (−2×rfx3, −2×ifx3, rdel, idel, 1, t1, it1);
      complex (t1, it1, rden, iden, 2, rlam, ilam);
m8:   itc := itc + 1;
      rx1 := rx2; rx2 := rx3; rfx1 := rfx2; rfx2 := rfx3;
      ix1 := ix2; ix2 := ix3; ifx1 := ifx2; ifx2 := ifx3;
      complex (rlam, ilam, rh, ih, 1, t1, it1);
      rh := t1; ih := it1;
m6:   rdel := rlam + 1; idel := ilam; rx3 := rx2 + rh;
      ix3 := ix2 + ih; c1 := 3; rroot := rx3;
      iroot := ix3; go to m1;
m7:   rfx3 := frroot; ifx3 := firoot;
      function (rx3, ix3, rfunc, ifunc);
      complex (rfx3, ifx3, rfx2, ifx2, 2, t1, it1);
      if (t1 ↑ 2+it1 ↑ 2) > 100 then
        begin rlam := rlam/2; rh := rh/2; ilam := ilam/2;
          ih := ih/2; go to m6 end;
      if sw1 then . . .
comment option to output iterant and associated function
  values;
  t1 := rx3 − rx2; it1 := ix3 − ix2;
  complex (t1, it1, rx2, ix2, 2, t2, it2);
  if sqrt (t2 ↑ 2+it2 ↑ 2) ≦ ep1 then go to fin1;
  if (sqrt (rfx3 ↑ 2+ifx3 ↑ 2)≦ep2) ∧
  (sqrt (rfunc ↑ 2+ifunc ↑ 2)≦ep2) then go to fin 2;
  go to if itc ≧ mxm then fin3 else m9;
fin1: if sw2 then . . .
comment option to output root; go to m12;
fin2: if sw2 then . . .
comment option to output root; go to m12;
fin3: if sw2 then . . .
comment no convergence, option to output last iterant;
  bool := true;
m12:  rtc := rtc + 1; rrt[rtc] := rx3; irt[rtc] := ix3;
      if rtc ≧ nrts then go to exit;
      if (ABS(ix3)>ep1) ∧ sw3 ∧ ¬ bool then
        begin ix3 := −ix3; function (rx3, ix3, rfunc, ifunc);
          rroot := rx3; iroot := ix3; c1 := 4;
          go to m1;
m11:  if sw2 then . . .
```

```
comment the complex conjugate of the last root found is acceptable. Option to output this root;
    rtc := rtc + 1; rrt[rtc] := rx3; irt[rtc] := ix3
      end else go to m0;
    if rtc < nrts then go to m0;
exit: end of procedure MULLER
```

ALGORITHM 197
MATRIX DIVISION

M. WELLS
University of Leeds, Leeds, England

```
procedure Pos Div (b, c, m, n, solve);
value m, n, solve; array b, c; integer m, n; Boolean solve;
comment The matrix c, with m rows and n columns, is divided
  by the positive definite matrix b, of order m, by the square root
  method (see Fadeeva, V. N., Computational Methods of Linear
  Algebra, Chap 2, §10). The upper triangle of b is replaced by
  an upper triangular matrix N such that N^tN = b. The other
  elements of b are undisturbed. The matrix c is replaced by b^-1c.
  The Boolean solve is used as a switch. If its value is true, then
  it is assumed that an earlier entry to Pos Div has left the matrix
  N in place, and a further division of c by b takes place;
begin integer i, j, k;
  real procedure dot (a, b, p, q);
  value q; real a, b; integer p, q;
  comment This is innerproduct, modified to define a function
    designator;
  begin real s; s := 0;
  for p := 1 step 1 until q do s := s + a × b;
  dot := s end dot;
  Start of program: if solve then go to back substitution;
  for i := step 1 until m do
  begin b [i, i] := sqrt (b[i, i] − dot (b[j, i]↑2, 1, j, i − 1));
    for j := i + 1 step 1 until m do
    b[i,j] := (b[i,j] − dot (b[k,i], b[k,j], k, i−1))/b[i,i]
  end formation of upper triangular matrix;
back substituton: for i := 1 step 1 until n do
  begin for j := 1 step 1 until m do
    c[i,j] := (c[i,j] − dot (b[k,j], c[i,k], k, j−1))/b[j,j];
      for j := m step −1 until 1 do
    c[i,j] := (c[i,j] − dot (b[j,m+1−k], c[i,m+1−k], k, m−j))/
      b[j,j]
  end of double back substitution
end of Pos Div
```

ALGORITHM 198
ADAPTIVE INTEGRATION AND MULTIPLE INTEGRATION

WILLIAM MARSHALL MCKEEMAN
Stanford University, Stanford, Calif.

```
begin comment This program illustrates the declaration and
  call of a procedure used to numerically approximate definite
  integrals and multiple integrals. The integrand is an expression
  substituted for the first formal parameter and must be a func-
  tion of the simple variable replacing the second formal pa-
  rameter. Multiple integration is accomplished by substituting
  a complete call of Integral for the first formal parameter. Note
  that in this case that the limits of integration on the inside calls
  may be functions of the variable of integration on the outer
  call. The parameter rule selects a Newton-Cotes formula which
  matches a polynomial of degree = rule to the function in the
  interval of integration. (See Hamming, Numerical Methods for
  Scientists and Engineers, Sec. 12.2). In any case, the procedure
  integral adapts its step size to the function in seeking to mini-
  mize the number of function evaluations. The program has been
  tested and run on a variety of functions using the ALGOL com-
  piler on the Burroughs B-5000.;
real procedure Integral (F) a function of the real variables: (x)
  between limits: (a,b) polynomial degree: (rule) tolerance: (eps);
value a, b, rule, eps; integer rule;
real F, x, a, b, eps;
begin comment set up the parameters for the recursion before
  calling the procedure NC;
switch nct := R1, R2, R3, R4, R5, R6, R7;
real array cf, fn [1:rule+1];
integer k; real da, ab;
real procedure NC(F,x,a,da,fn,k,cf,rule,eps,es,ab,lvl);
value a, da, rule, eps, es, lvl; real array cf;
integer k, rule, lvl; real F, x, a, da, fn, eps, es, ab;
begin comment NC is the adaptive heart of Integral;
  real array fc[1:rule+1,1:rule+1], est, xx[1:rule+1];
  integer i, j; real dx, int, ep;
  real procedure SUM(term, index, upperlimit);
  real term; integer index, upperlimit;
  begin real t; t := 0;
    for index := 1 step 1 until upperlimit do
      t := t + term;
    SUM := t
  end of SUM;
  comment begin the integration by evaluating F on the mesh
    points;
  for k := 1 step 1 until rule + 1 do fc[k,k] := fn;
  dx := da/(rule×(rule+1));
  x := a;
  for i := 1 step 1 until rule + 1 do
    for j := 1 step 1 until rule do
    begin
      if j = 1 then xx[i] := x;
      if i ≠ j then fc[i,j] := cf[j] × F;
      x := x + dx;
    end having done all necessary function evaluations;
  for i := 1 step 1 until rule do
    fc[i, rule+1] := fc[i+1,1];
  ep := eps/sqrt(rule+1);
  comment eps/(rule + 1) is the value to give an absolute
    error bound of eps in the final answer. It proves too strict in
    practice;
  dx := dx × rule;
  comment compute the integrals of the subintervals;
  for i := 1 step 1 until rule + 1 do
    est[i] := SUM(fc[i,j],j,rule+1)×dx;
  ab := ab − abs(es) + SUM(abs(est[i]),i,rule+1);
  comment ab is the area under abs(F). It is used in computing
    the relative error upon which to terminate;
  int := SUM(est[i],i,rule+1);
  if lvl ≧ 100/(rule+1) then go to error;
  NC := if abs(es−int) ≦ eps × ab ∧ es ≠ 1.0 then int
    else SUM(NC(F,x,xx[i],dx,fc[i,j],j,cf,rule,ep,est[i], ab,lvl+1),
      i,rule+1);
  go to return;
  error; NC := int;
    comment abs(es − int) is the approximate error caused by
      terminating the recursion. In most cases, termination at
      this level will not adversely affect the accuracy of the result;
  return:
end of NC;
comment now initialize the Newton-Cotes coefficients;
go to nct [rule];
R1: cf[1] := cf[2] := 1.0/2.0; go to compute;
R2: cf[1] := cf[3] := 1.0/6.0; cf[2] := 4.0/6.0;
```

```
comment R1 is trapezoidal rule, R2 is Simpson's rule;
go to compute;
R3: cf[1] := cf[4] := 1.0/8.0;
    cf[2] := cf[3] := 3.0/8.0; go to compute;
R4: cf[1] := cf[5] := 7.0/90.0;
    cf[2] := cf[4] := 32.0/90.0;
    cf[3] := 12.0/90.0; go to compute;
R5: cf[1] := cf[6] := 19.0/288.0;
    cf[2] := cf[5] := 75.0/288.0;
    cf[3] := cf[4] := 50.0/288.0; go to compute;
R6: cf[1] := cf[7] := 41.0/840.0;
    cf[2] := cf[6] := 216.0/840.0;
    cf[3] := cf[5] := 27.0/840.0;
    cf[4] := 272.0/840.0; go to compute;
R7: cf[1] := cf[8] := 75.1/1728.0;
    cf[2] := cf[7] := 357.7/1728.0;
    cf[3] = cf[6] := 134.3/1728.0;
    cf[4] := cf[5] := 298.9/1728.0;
compute: da := b - a;
for k := 0 step 1 until rule do
begin
  x := a + k × da/rule;
  fn[k+1] := F × cf[k+1];
end;
ab := 1.0;
    Integral := NC(F,x,a,da,fn[k],k,cf,rule,eps,1.0,ab,0);
  end of Integral;
  comment Now evaluate the integral of 1.0/sqrt(abs(x+y))
    on the unit disk in the x,y-plane;
  real x, y, answer;
  answer := Integral(Integral(1.0/sqrt(abs(x+y)), x,
    -sqrt(1.0-y↑2), sqrt(1.0-y↑2), 7, 0.001),y,-1.0,1.0,3,0.001);
end of program;
```

ALGORITHM 199
CONVERSIONS BETWEEN CALENDAR DATE AND JULIAN DAY NUMBER

ROBERT G. TANTZEN
Air Force Missile Development Center, Holloman AFB, New Mex.

```
procedure JDAY (d,m,y,j);
integer d,m,y,j;
comment JDAY converts a calendar date, Gregorian calendar,
  to the corresponding Julian day number j. From the given day
  d, month m, and year y, the Julian day number j is computed
  without using tables. The procedure is valid for any valid
  Gregorian calendar date. When transcribing JDAY for other
  compilers, be sure that integers of size 3 × 10⁶ can be handled;
begin integer c, ya;
  if m > 2 then m := m - 3
    else begin m := m + 9; y := y - 1 end;
  c :=y ÷ 100; ya := y -100 × c;
  j := (146097×c)÷4+(1461×ya)÷4+(153×m+2)÷5+d+1721119
end JDAY
```

```
procedure JDATE (j,d,m,y);
integer j,d,m,y;
comment JDATE converts a Julian day number j to the corre-
  sponding calendar date, Gregorian calendar. Since j is an integer
  for this procedure, it is correct astronomically for noon of the
  day. JDATE computes the day d, month m, and year y, without
  using tables. The procedure is valid for any valid Gregorian
  calendar date. When transcribing JDATE for other compilers,
  be sure that integers of size 3 × 10⁶ can be handled;
begin j := j - 1721119;
  y := (4×j-1) ÷ 146097; j := 4×j - 1 - 146097 × y;
    d := j÷ 4;
  j := (4×d+3) ÷ 1461; d := 4×d + 3 - 1461 × j;
    d := (d + 4) ÷ 4;
  m := (5×d-3) ÷ 153; d := 5×d - 3 - 153 × m;
    d :=(d+5) ÷ 5;
  y := 100 × y + j; if m < 10 then m := m + 3
    else begin m := m - 9; y := y + 1 end;
end JDATE
```

```
procedure KDAY (d,m,ya,k);
integer d,m,ya,k;
comment KDAY converts a calendar date, Gregorian calendar,
  to the corresponding serial day number k. From the given day
  d, month m, and the last two decimals of the year, ya, the serial
  day number k is computed without using tables. The procedure
  is valid from 1 March 1900 (k=1) to 31 December 1999
  (k = 36465). To obtain the Julian day number j (valid at noon)
  use j = k + 2415079;
begin if m > 2 then m := m - 3
    else begin m := m + 9; ya := ya - 1 end;
  k := (1461×ya) ÷ 4+(153×m+2) ÷ 5 + d
end
```

```
procedure KDATE (k,d,m,ya);
integer k,d,m,ya;
comment KDATE converts a serial day number k to the corre-
  sponding calendar date, Gregorian calendar. It computes day d,
  month m, and the last two decimals of the year, ya, without
  using tables. The procedure is valid from k = 1 (1 March 00) to
  k = 36465 (31 December 99) for any one century. For the 20th
  Century the relation between k and theulian day number j
  (at noon) is j = k + 2415079;
begin ya := (4×k-1) ÷ 1461; d := 4×k - 1 - 1461 × ya;
  d := (d+4) ÷ 4; m := (5×d-3) ÷ 153;
    d := 5×d - 3 - 153×m;
  d := (d+5) ÷ 5;
  if m < 10 then m := m + 3
    else begin m := m - 9; ya := ya + 1 end;
end KDATE
```

ALGORITHM 200
NORMAL RANDOM

RICHARD GEORGE*
Argonne National Laboratory, Argonne, Ill.

* Work supported by United States Atomic Energy Commission.

```
real procedure NORMAL RANDOM (Mean, Sigma n);
  procedure Random;
  real Mean, Sigma;
  integer n;
  comment Random is assumed to be a real procedure which
    generates a random number uniform on the interval (-1, +1).
    The value of n should be greater than 10, in order to approxi-
    mate the normal distribution with accuracy. However, very
    large values of n will increase the running time. The use of
    Mean and Sigma should be obvious. Reference: R. W. Ham-
    ming, Numerical Methods for Scientists and Engineers;
  begin
    integer i; real sum;
    sum := 0;
    for i := step 1 until n do
      sum := sum + Random;
    NORMAL RANDOM := Mean + Sigma × sum × sqrt (3.0/n)
  end NORMAL RANDOM
```

ALGORITHM 201
SHELLSORT

J. BOOTHROYD
English Electric-Leo Computers, Kidsgrove, Staffs, England

```
procedure Shellsort (a, n); value n; real array a; integer n;
comment a[1] through a[n] of a[1:n] are rearranged in ascending
  order. The method is that of D. A. Shell, (A high-speed sorting
  procedure, Comm. ACM 2 (1959), 30-32) with subsequences
  chosen as suggested by T. N. Hibberd (An empirical study of
  minimal storage sorting, SDC Report SP-982). Subsequences
  depend on m1 the first operative value of m. Here m1 = 2^k - 1
  for 2^k ≤ n < 2^(k+1). To implement Shell's original choice of m1 =
  [n/2] change the first statement to m := n;
begin integer i, j, k, m; real w;
  for i := 1 step i until n do m := 2 × i - 1;
  for m := m ÷ 2 while m ≠ 0 do
    begin k := n - m;
      for j := 1 step 1 until k do
        begin for i := j step -m until 1 do
          begin if a[i+m] ≧ a[i] then go to 1;
            w := a[i]; a[i] := a[i+m]; a[i+m] := w;
          end i;
        1: end j
    end m
end Shellsort;
```

CERTIFICATION OF ALGORITHM 37
TELESCOPE 1 [K. A. Brons, *Comm. ACM*, Mar. 1961]

JAMES F. BRIDGES
Michigan State University, East Lansing, Mich.

This procedure was tested on the CDC 160A, using 160A FORTRAN. The 10th degree polynomial obtained by truncating the series exp $(-x)$ was telescoped using $L = 1$ and lim = 0.001. The result was $N = 3$, eps = $0.21061862_{10} - 3$ and coefficients $+0.99978965$, -0.99307236, $+0.46364955$, -0.10267767. The error curve was computed for $x = 0(0.02)1.0$ and no error exceeded eps, the worst error being 2% of eps less than eps.

This result is in close agreement with that of Henry C. Thatcher, Jr. in his Certification (*Comm. ACM*, Aug. 1962). Mr. Thatcher has pointed out that he inadvertantly referred to the series for exp $(-x)$ as the "exponential series" thereby inferring the positive series exp $(+x)$. There is also a typographical error in his eps. It should be $+0.2103505_{10} - 3$.

CERTIFICATION OF ALGORITHM 38
TELESCOPE 2 [K. A. Brons, *Comm. ACM*, Mar., 1961]

JAMES F. BRIDGES
Michigan State University, East Lansing, Mich.

This procedure was tested on the CDC 160A using 160A FORTRAN. The 10th degree polynomial obtained by truncating the series expansion of exp $(+x)$ was telescoped using $L = 1.0$ and lim = 0.001. The result was $N = 4$, eps = $0.59159949_{10} - 3$ and coefficients $+1.0000447$, $+0.99730758$, $+0.49919675$, $+0.17734729$, $+0.043793910$. Errors were calculated for $x = -1.0(0.02)1.0$. The only error to exceed eps was at $x = 1.0$ and was within 0.6% of eps.

CORRECTION TO EARLIER REMARKS ON ALGORITHM 42 INVERT, ALG. 107 GAUSS'S METHOD, ALG. 120 INVERSION II, AND gjr [P. Naur, *Comm. ACM*, Jan. 1963, 38–40.]

P. NAUR
Regnecentralen, Copenhagen, Denmark

George Forsythe, Stanford University, in a private communication has informed me of two major weaknesses in my remarks on the above algorithms:

1) The computed inverses of rounded Hilbert matrices are compared with the exact inverses of unrounded Hilbert matrices, instead of with very accurate inverses of the rounded Hilbert matrices.

2) In criticizing matrix inversion procedures for not searching for pivot, the errors in inverting positive definite matrices cannot be used since pivot searching seems to make little difference with such matrices.

It is therefore clear that although the figures quoted in the earlier certification are correct as they stand, they do not substantiate the claims I have made for them.

To obtain a more valid criterion, without going into the considerable trouble of obtaining the very accurate inverses of the rounded Hilbert matrices, I have multiplied the calculated inverses by the original rounded matrices and compared the results with the unit matrix. The largest deviation was found as follows:

	Maximum deviation from elements of the unit matrix		
Order	*INVERSION II*	*gjr*	*Ratio*
2	$-1.49_{10}-8$	$-1.49_{10}-8$	1.0
3	$-4.77_{10}-7$	$-8.34_{10}-7$	0.57
4	$-9.54_{10}-6$	$-3.43_{10}-5$	0.28
5	$-7.32_{10}-4$	$-4.58_{10}-4$	1.6
6	$-1.61_{10}-2$	$-1.42_{10}-2$	1.1
7	$-5.78_{10}-1$	$-5.47_{10}-1$	1.1
8	$-1.20_{10}-2$	$-1.38_{10}1$	8.7
9	$-4.91_{10}1$	$-2.22_{10}1$	2.2

This criterion supports Forsythe's criticism. In fact, on the basis of this criterion no preference of INVERSION II or *gjr* can be made.

The calculations were made in the GIER ALGOL system, which has floating numbers of 29 significant bits.

CERTIFICATION OF ALGORITHM 43
CROUT II [Henry Thacher, Jr., *Comm. ACM* (1960), 176]

C. DOMINGO AND F. RODRIGUEZ-GIL
Universidad Central, Caracas, Venezuela

CROUT II was coded in PUC-R2 and tested in the IBM-1620. Two types of INNERPRODUCT subroutines were used. The first one finds the scalar product in fixed-point arithmetic to increase accuracy, using an accumulator of 32 digits. The second one uses ordinary floating-point with eight significative figures.

Using a unit matrix as right-hand side, a 6×6 segment of Hilbert matrix was inverted. The inverse was inverted again.

The maximum difference between this result and the original segment of Hilbert matrix was:

Using fixed-point INNERPRODUC..... 8.2426×10^{-4}
(Value of determinant..... $4.7737088 \times 10^{-18}$)
Using floating-point INNERPRODUC..... 3.014016×10^{-2}
(Value of determinant..... $4.4950721 \times 10^{-18}$)

Two typographical errors were observed in the algorithm:

The statement:

$b[k] := g[k] -$ INNERPRODUCT $(A[k,p], b[p], p,i,k-1)$

should be:

$b[k] := b[k] -$ INNERPRODUCT $(A[k,p], b[p], 1,k-1)$

The statement:

$y[k] :=$ (b[k] − INNERPRODUCT $(A[k,p], y[p], p,k+1,n)/A[k,k]$

should be:

$y[k] :=$ (b[k] − INNERPRODUCT $(A[k,p], y[p], p,k+1,n))/A[k,k]$

Storage may be saved eliminating the array y and using instead the array b, in which the solution is formed.

A previous certification of this algorithm [*Comm. ACM 4*, 4 (Apr. 1961), 182] was tested again with the same results. Two errors were detected in the certification: The row that must replace the last row of A in order to obtain a singular matrix must be:

19,1927 33.4409 −251298 −5.2811

CERTIFICATION OF ALGORITHM 47 ASSOCIATED LEGENDRE FUNCTIONS OF THE FIRST KIND FOR REAL OR IMAGINARY ARGUMENTS [John R. Herndon, *Comm. ACM*, Apr. 1961]

RICHARD GEORGE*
Argonne National Laboratory, Argonne, Ill.

* Work supported by United States Atomic Energy Commission.

This procedure was programmed in FORTRAM for the IBM 1620 and was tested with a number of real arguments. A few errors were detected:

1. In the following sequence the *end* must be removed:

begin if $(n - m + 2)/2 < i$ **then go to** last **end**;

2. In these, the lower bound of 1 is needed:

for $i :=$ **step** 1 **until** n **do**
for $i :=$ **step** 1 **until** j **do**

3. There are four places where integer arithmetic is clearly intended and we must substitute the symbol ÷ for the symbol /.

In addition, it might be mentioned that the statement

if $n = m$ **then go to** main;

could be omitted from the ALGOL program without harm, though the FORTRAN version requires it. Here, and elsewhere in the procedure, one might make an equivalent but more succinct statement. With change in style, the variable j could be eliminated.

ADDITIONAL REMARKS ON ALGORITHM 52 A SET OF TEST MATRICES [J. R. Herndon, *Comm. ACM* (Apr. 1961), 180]

P. NAUR
Regnecentralen, Copenhagen, Denmark

From an inspection of the results of eigenvalue-finding algorithms I conclude that all but two of the eigenvalues of TESTMATRIX are unity while the two remaining are given by the expressions $6/(p\times(n+1))$ and $p/(n\times(5-2\times n))$ where

$p = 3 +$ sqrt $((4\times n-3) \times (n-1) \times 3/(n+1))$.

These expressions have been used for the determination of absolute errors of the eigenvalues calculated by JACOBI, Algorithm 85, and Householder Tridiagonalisation, etc. as reported below. They were also used to calculate the following table (using GIER ALGOL, with 29 significant bits):

n	*Determinant*	*Eigenvalues Differing from unity*	
3	−.500 000 00	.224 744 87	−2.224 744 9
4	−.100 000 00	.153 112 89	−.653 112 89
5	−.040 000 000	.113 238 08	−.353 238 08
6	−.020 408 163	.088 290 570	−.231 147 71
7	−.011 904 762	.071 428 571	−.166 666 67
8	−.007 575 757 6	.059 386 081	−.127 567 90
9	−.005 128 205 2	.050 422 549	−.101 704 60
10	−.003 636 363 6	.043 532 383	−.083 532 383
11	−.002 673 796 8	.038 097 478	−.070 183 039
12	−.002 024 291 5	.033 718 770	−.060 034 559
13	−.001 569 858 7	.030 128 103	−.052 106 125
14	−.001 242 236 0	.027 139 206	−.045 772 747
15	−.001 000 000 0	.024 619 013	−.040 619 013
16	−.000 816 993 47	.022 470 157	−.036 359 046
17	−.000 676 132 52	.020 619 902	−.032 790 288
18	−.000 565 930 96	.019 012 916	−.029 765 605
19	−.000 478 468 90	.017 606 429	−.027 175 807
20	−.000 408 163 27	.016 366 903	−.024 938 332

The figures for $n = 20$ agree very well with the results quoted by H. E. Gilbert in his certification [*Comm. ACM 4* (Aug. 1961), 339].

CERTIFICATION OF ALGORITHMS 63, 64 AND 65, PARTITION, QUICKSORT, AND FIND, [*Comm. ACM*, July 1961]

B. RANDELL AND L. J. RUSSELL
The English Electric Company Ltd., Whetstone, England

Algorithms 63, 64, and 65 have been tested using the Pegasus ALGOL 60 Compiler developed at the De Havilland Aircraft Company Ltd., Hatfield, England.

No changes were necessary to Algorithms 63 and 64 (Partition and Quicksort) which worked satisfactorily. However, the comment that Quicksort will sort an array without the need for any extra storage space is incorrect, as space is needed for the organization of the sequence of recursive procedure activations, or, if implemented without using recursive procedures, for storing information which records the progress of the partitioning and sorting.

A misprint ('if' for '**if**' on the line starting '**else** if $J \leqq K$ **then** ⋯') was corrected in Algorithm 65 (Find), but it was found that in certain cases the sequence of recursive activations of *Find* would not terminate successfully. Since *Partition* produces as output two integers J and I such that elements of the array $A[M:N]$ which lie between $A[J]$ and $A[I]$ are in the positions that they will occupy when the sorting of the array is completed, *Find* should cease to make further recursive activations of itself if K fulfills the condition $J < K < I$.

Therefore the conditional statement in the body of *Find* was changed to read

if $K \leqq J$ **then** find (A,M,J,K)
else if $I \leqq K$ **then** find (A,I,N,K)

With this change the procedure worked satisfactorily.

REMARK ON ALGORITHM 77 INTERPOLATION, DIFFERENTIATION, AND INTEGRATION [P. E. Hennion, *Comm. ACM*, Feb., 1962]

P. E. HENNION
Giannini Controls Corp., Berwyn, Penn.

It was brought to my attention through the CERTIFICATION OF ALGORITHM 77 AVINT [V. E. Whittier, *Comm. ACM*, June,

1962] that restrictions on the upper and lower limits of integration existed, i.e., (1) $x10 \leqq xa$ (1), (2) $xup \geqq xa(nop)$. To remove these restrictions the following two changes should be made.

1. Before line $L16$: and after the statement $ib := 2$; place the following code:

for $ia := 1$ **step** 1 **until** nop **do begin**
if $xa(ia) \geqq x10$ **then go to** $L17$; $ib := ib + 1$; **end**;
$L17$: $ju\ 1 := nop + 1$; **for** $ia := 1$ **step** 1 **until** nop **do begin**
$ju\ 1 := ju\ 1 - 1$; **if** $xa(ju\ 1) > xup$ **end**; $ju\ 1 := ju\ 1 - 1$;

2. Change line $L13$: to read:

$L13$: **if** $jm \neq ib$ **then go to** $L14$;

CERTIFICATION OF ALGORITHM 85
JACOBI [Thomas G. Evans, *Comm. ACM* (Apr. 1962), 208]

P. NAUR
Regnecentralen, Copenhagen, Denmark

We have first run this algorithm in the GIER ALGOL system with the following corrections included:

1. The change given by J. S. Hillmore [*Comm. ACM 5* (Aug. 1962), 440] with capital V changed to v.

TABLE 1
HBH TESTMATRIX

Range of true errors of eigenvalues					Range of deviations from relation Av − lambda $v = 0$						Range of deviations from relation A − (ST) LAMBDA $S = 0$					
Order	*j*	*error[j]*	*j*	*error[j]*	*Element*	*Vector*	*Error*	*Element*	*Vector*	*Error*	*Element*	*Vector*	*Error*	*Element*	*Vector*	*Error*
					rho = $1.0_{10}-3$											
5	1	$-1.1_{10}-6$	3	$5.2_{10}-8$	1	1	$-1.7_{10}-4$	1	3	$2.0_{10}-4$	1	1	$-2.5_{10}-4$	5	5	$1.0_{10}-4$
10	9	$-7.9_{10}-5$	8	$3.5_{10}-5$	7	2	$-3.3_{10}-3$	6	6	$3.0_{10}-3$	1	1	$-4.2_{10}-3$	6	7	$3.2_{10}-3$
15	15	$-9.2_{10}-5$	12	$3.7_{10}-5$	6	3	$-1.7_{10}-3$	11	13	$1.7_{10}-3$	9	15	$-1.5_{10}-3$	8	9	$1.8_{10}-3$
					rho = $1.0_{10}-5$											
5	1	$-1.1_{10}-6$	3	$6.0_{10}-8$	2	5	$-1.3_{10}-7$	5	2	$4.1_{10}-8$	1	2	$-1.6_{10}-7$	4	5	$4.5_{10}-8$
10	1	$-1.2_{10}-5$	2	$2.2_{10}-7$	7	3	$-2.7_{10}-5$	2	8	$2.2_{10}-5$	7	7	$-2.4_{10}-5$	2	8	$2.3_{10}-5$
15	1	$-3.5_{10}-5$	4	$3.9_{10}-7$	11	9	$-6.4_{10}-6$	7	2	$4.8_{10}-6$	11	12	$-5.3_{10}-6$	12	12	$4.7_{10}-6$
					rho = $1.0_{10}-8$											
5	1	$-1.1_{10}-6$	3	$6.0_{10}-8$	2	5	$-1.3_{10}-7$	4	2	$6.5_{10}-9$	2	2	$-1.3_{10}-7$	4	4	$3.0_{10}-8$
10	1	$-1.2_{10}-5$	2	$2.2_{10}-7$	1	10	$-1.1_{10}-6$	4	2	$6.4_{10}-8$	1	2	$-5.7_{10}-7$	9	9	$8.2_{10}-8$
15	1	$-3.5_{10}-5$	4	$3.9_{10}-7$	1	14	$-3.4_{10}-6$	4	2	$3.9_{10}-7$	2	2	$-1.3_{10}-6$	15	15	$8.9_{10}-8$

TESTMATRIX, Algorithm 52

Range of true errors of eigenvalues					Range of deviations from relation Av − lambda $v = 0$						Range of deviations from relation A − (ST) LAMBDA $S = 0$					
Order	*j*	*error[j]*	*j*	*error[j]*	*Element*	*Vector*	*Error*	*Element*	*Vector*	*Error*	*Element*	*Vector*	*Error*	*Element*	*Vector*	*Error*
					rho = $1.0_{10}-5$											
5	4	$-1.0_{10}-8$	1	.0	5	5	$-3.3_{10}-8$	5	4	$4.3_{10}-8$	5	5	$-5.1_{10}-8$	4	4	$3.9_{10}-8$
10	8	$-1.1_{10}-8$	4	.0	7	7	$-1.2_{10}-8$	9	6	$1.3_{10}-8$	7	8	$-5.1_{10}-9$	6	6	$2.0_{10}-8$
15	13	$-1.1_{10}-8$	6	.0	14	14	$-9.3_{10}-9$	10	10	$9.4_{10}-9$	8	9	$-1.9_{10}-9$	10	10	$1.3_{10}-8$
					rho = $1.0_{10}-8$											
3	3	$-7.5_{10}-9$	1	$3.7_{10}-9$	3	1	$-2.8_{10}-9$	2	2	$9.3_{10}-9$	1	3	.0	1	2	$1.9_{10}-8$
4	4	$-5.6_{10}-9$	3	.0	2	2	$-4.5_{10}-9$	3	4	$3.3_{10}-9$	2	2	.0	2	3	$9.3_{10}-9$
5	4	$-1.0_{10}-8$	1	.0	5	4	$-4.9_{10}-9$	4	4	$5.8_{10}-9$	1	1	$-7.5_{10}-9$	3	4	$7.5_{10}-9$
6	4	$-4.7_{10}-9$	4	.0	4	3	$-2.8_{10}-9$	5	4	$3.6_{10}-9$	1	6	$-2.3_{10}-10$	4	5	$9.3_{10}-9$
7	4	$-5.1_{10}-9$	5	.0	6	6	$-2.8_{10}-9$	4	4	$3.4_{10}-9$	5	7	$-1.2_{10}-10$	5	6	$7.5_{10}-9$
8	7	$-7.5_{10}-9$	5	.0	5	5	$-6.0_{10}-9$	5	6	$3.2_{10}-9$	8	8	$-1.2_{10}-10$	7	7	$9.3_{10}-9$
9	6	$-4.4_{10}-9$	7	.0	6	5	$-5.1_{10}-9$	7	6	$3.2_{10}-9$	5	5	$-7.5_{10}-9$	8	8	$1.5_{10}-8$
10	8	$-1.5_{10}-8$	8	.0	8	9	$-9.3_{10}-9$	9	7	$7.2_{10}-9$	6	7	$-2.3_{10}-9$	9	9	$2.0_{10}-8$
11	10	$-7.5_{10}-9$	1	.0	9	10	$-6.5_{10}-9$	8	11	$3.0_{10}-9$	1	1	$-3.1_{10}-9$	8	8	$7.5_{10}-9$
12	8	$-5.0_{10}-9$	11	.0	10	6	$-7.6_{10}-9$	10	8	$2.4_{10}-9$	6	6	$-1.7_{10}-8$	4	4	$1.3_{10}-8$
13	12	$-1.1_{10}-8$	10	.0	10	11	$-6.9_{10}-9$	12	10	$9.1_{10}-9$	7	7	$-3.0_{10}-8$	12	12	$3.2_{10}-8$
14	10	$-1.5_{10}-8$	4	.0	13	13	$-1.1_{10}-8$	10	10	$6.7_{10}-9$	9	10	$-3.5_{10}-9$	6	6	$1.7_{10}-8$
15	13	$-1.1_{10}-8$	6	.0	14	14	$-1.1_{10}-8$	11	10	$3.5_{10}-9$	8	9	$-3.0_{10}-9$	6	11	$7.5_{10}-9$

2. The 4th **for** clause corrected to read:

for j := 1 **step** 1 **until** $i - 1$ **do**

3. The last **for** clause corrected to read:

for i := 1 **step** 1 **until** n **do**

On closer examination we have found, however, that a significant number of superfluous operations could be eliminated in the innermost loop by rewriting the two **for** statements at the center of the algorithm as a single **for** statement, to read as follows:

```
.....
cost := sqrt (1-sint↑2);
for i := 1 step 1 until n do
  begin if i ≠ p ∧ i ≠ q then
    begin intl := A[i,p]; mu := A[i,q];
    A[q,i] := A[i,q] := intl × sint + mu × cost;
    A[p,i] := A[i,p] := intl × cost - mu + sint
    end;
  intl := S[i,p]; mu := S[i,q];
  S[i,q] := intl × sint + mu × cost;
  S[i,p] := intl × cost - mu × sint
  end;
A[p,p] := v1 × cost ↑ 2 + v3 × sint ↑ 2 - 2 × v2 × sint × cost;
.....
```

This revision is particularly advantageous in systems having a comparatively slow subscript mechanism, such as GIER ALGOL, because it eliminates more than 3 out of 8 references to subscripted variables.

JACOBI has been tried with two different sets of matrices having known eigenvalues. In both cases a test program was set up to find the range of errors of the eigenvalues computed by JACOBI. In addition, the relations $Av - \lambda v = 0$ (A is the given matrix, v an eigenvector, and λ the corresponding eigenvalue) and $A - (ST)$ LAMBDA $S = 0$ (S is the matrix having the eigenvectors as columns and ST its transpose, and LAMBDA is the diagonal matrix of the eigenvalues) were used as checks. The test matrices were TESTMATRIX calculated by the revised algorithm 52 given in *Comm. ACM 6* (Jan. 1963), 39, and the following matrix suggested by Mr. H. B. Hansen:

$$\text{HBH TESTMATRIX}[j,i] = \text{HBH TESTMATRIX}[i,j] = n + 1 - j \qquad j \geqq i$$

having the eigenvalues $0.5/(1 - \cos((2\times i-1)\times pi/(2\times n+1)))$.

The results were as shown in Table 1 (GIER ALGOL works with floating numbers of 29 significant bits).

The compile time for the program which produced one of these tables was about 40 seconds. Run times were as follows:

Rho	n	*Original algorithm* TESTMATRIX ALG. 52 HBH TESTMATRIX (seconds)		*Revised algorithm* HBH TESTMATRIX (seconds)
10^{-3}	5			3
	10			22
	15			70
10^{-5}	5	3		5
	10	5	41	29
	15	13	148	99
10^{-8}	5	4	7	6
	6	5	12	
	7	5	18	
	8	5	25	
	10	13		38
	15	22		116

From these figures it looks as if TESTMATRIX, Algorithm 52, is atypical as far as solution by means of JACOBI is concerned. The much higher accuracy obtained for this matrix as compared with the HBH matrix points in the same direction.

For further comparison it may be mentioned that the algorithms published by J. H. Wilkinson [*Num. Math. 4* (1962), 354–376] also have been tested successfully with GIER ALGOL. Wilkinson's algorithms reduce the matrix to tridiagonal form by means of Householder's method and use Sturm sequences to find the eigenvalues and inverse iteration to find the eigenvectors. In GIER ALGOL this method is about 1.3 times as fast as JACOBI for the range of matrices considered here. JACOBI has the advantage that the eigenvectors are properly orthogonal, even in the case of multiple eigenvalues, and also has a much simpler logic. On the other hand if only some of the eigenvalues and/or eigenvectors are sought Wilkinson's algorithms will often offer much higher speed than JACOBI, which always finds them all.

CERTIFICATION OF ALGORITHM 140 MATRIX INVERSION [P. Z. Ingerman, *Comm. ACM*, Nov. 1962]

RICHARD GEORGE*
Argonne National Laboratory, Argonne, Ill.

* Work supported by the United States Atomic Energy Commission.

Algorithm 140 was tested on the LGP-30, using SCALP, a load-and-go compiler from the Dartmouth College Computation Center, and it was shown to be syntactically correct.

It is indeed a simple procedure. It is so simple because the author has eliminated the very necessary search for largest elements and the row interchanges. As a result, this procedure will fail to invert many non-singular matrices. To be invertable by this procedure, a matrix must be such that all of its leading diagonal submatrices will have non-zero determinants.

One would do well to avoid this algorithm and use one (such as 120) which employs the pivoting process.

CERTIFICATION OF ALGORITHM 153
GOMORY [F. L. Bauer, *Comm. ACM 6*, Feb. 1963]

B. LEFKOWITZ AND D. A. D'ESOPO*
Stanford Research Institute, Menlo Park, California

* Work supported by Office of Naval Research.

GOMORY was hand-coded in BALGOL for the Burroughs 220 and in FORTRAN for the CDC 1604. The following corrections should be made:

The statement

lambda := *abs*(a[r,1]/t[1]);

should read

lambda := *abs*(a[r,l]/t[l]);

The statement

for *j* := 2 **step** 1 **until** *n*−1 **do if** *a*[*r*,*j*] < 0 **then**

should read

for *j* := 1 **step** 1 **until** *n*−1 **do if** *a*[*r*,*j*] < 0 **then**

The following changes to Bauer's program were made to increase its efficiency and reduce storage requirements.

Change the statement

begin integer *i*, *k*, *j*, 1, *r*;

to read

begin integer *i*, *k*, *j*, 1, *r*, *c*, *t*;

Change the statement

real lambda;

to read

real lambda, lambd;

Delete the statement

integer array *t*[1: *n*−1], *c*[1: *n*];

Before the statement

for *j* := 1 **step** 1 **until** *n*−1 **do if** *a*[*r*,*j*] < 0 **then**

insert the statement

lambda := 1.0;

Change the statement

begin if *a*[0,*l*] ≠ 0 **then** *t*[*j*] := *entier*(*a*[0,*j*]/*a*[0,*l*])

to read

begin if *a*[0,*l*] ≠ 0 **then** *t* := *entier*(*a*[0,*j*]/*a*[0,*l*])

Change the statement

else *t*[*j*] := 1

to read

else *t* := 1

After the statement

else *t*[*j*] := 1

insert the statements

lambd := −*a*[*r*,*j*]/*t*;
lambda := **if** *lambda* < *lambd* **then** *lambd* **else** *lambda*;

Delete the statements starting with

lambda := *abs*(*a*[*r*,1]/*t*[1]);

up to and including

lambda := *abs*(*a*[*r*,*j*]/*t*[*j*]) **end**;

Change the statement

begin *c*[*j*] := *entier* (*a*[*r*, *j*]/*lambda*);

to read

begin *c* := *entier*(*a*[*r*,*j*]/*lambda*);

Change the statement

if *c*[*j*] ≠ 0 **then**

to read

if *c* ≠ 0 **then**

Change the statement

for *i* := 0 **step** 1 **until** *m* **do** *a*[*i*,*j*] := *a*[*i*,*j*] + *c*[*j*] ×

to read

for *i* := 0 **step** 1 **until** *m* **do** *a*[*i*,*j*] := *a*[*i*,*j*] + *c* ×

The "tie-breaking" procedure embodied in the three statements beginning at

3: **if** *a*[*i*,*j*] < *a*[*i*,*l*] **then** *l* := *j* **else**

will fail if the two columns being compared are identical. Although this cannot happen on the first iteration, it may occur later. To test for this condition change the two statements beginning with

begin *i* := *i* + 1; **go to** 3 **end**

to read

begin *i* := *i* + 1; **if** *i* > *m* **then go to** 31 **else go to** 3 **end**;
31: **end**;

The revised algorithm yielded satisfactory answers on a ten equation-seven variable problem in 159 iterations and a 35-equation 14-variable problem in 447 iterations.

The following comments may be helpful for preparing a problem for GOMORY. The problem constraints must be stated in the form:

$$\sum_j a_{ij}x_j + s_i = b_i$$

where the s_i are slack variables. The columns representing these slack variables need not appear in the initial tableau-matrix a.

Since the only variables in the solution that will necessarily be non-negative are the s_i, any non-negativity constraints on the other variables must be among the above equations (e.g. the constraint $x_j \geqq 0$ is represented by $-x_j + s_k = 0$).

The size of the integers in the b vector substantially affects the number of iterations.

The requirement that all but the last tableau-columns be lexicographically positive means that the first nonzero element in these columns must be positive.

EDITOR'S NOTE: Prof. Bauer wishes to indicate that for the Algorithm 153, GOMORY, credit is due to Ch. Witzgall, who wrote the draft.

CERTIFICATION OF ALGORITHM 154
COMBINATION IN LEXICOGRAPHICAL ORDER [Charles J. Mifsund, *Comm. ACM*, Mar. 1963]

K. M. BOSWORTH
I.C.T. Ltd., Hayes, Middlesex, England

This procedure was tested

for *r* := 1 **step** 1 **until** *n* with *n* = 6

with correct results.

CERTIFICATION OF ALGORITHM 155
COMBINATION IN ANY ORDER [Charles J. Mifsud, *Comm. ACM*, Mar. 1963]

K. M. BOSWORTH
I.C.T. Ltd., Hayes, Middlesex, England

This procedure was tested using

$m[1] = 4 \quad m[2] = 3 \quad m[3] = 2 \quad m[4] = 2$

$M[1] = 4 \quad M[2] = 7 \quad M[3] = 9 \quad M[4] = 16$

and **for** *r* := 1 **step** 1 **until** *s*

It is correctly generated for $r = 1$ the four combinations 4, 7, 9, 16, as well as the ten combinations for $r = 2$, the eighteen combinations for $r = 3$, and the twenty-six combinations for $r = 4$.

Changes made due to compiler limitations were (i) systematic changes of upper case letters where there was conflict due to having only one case of letters, (ii) transfer of **own** declared variables to non-local variables, and (iii) integer labels to identifiers.

CERTIFICATION OF ALGORITHM 156
ALGEBRA OF SETS [Charles J. Mifsud, *Comm. ACM*, Mar. 1963]

K. M. Bosworth
I.C.T. Ltd., Hayes, Middlesex, England

One correction required in this procedure is the systematic change of label A to avoid conflict with the formal parameter array A.

The procedure was then tested for $n = 9$ and $Ai = i$, $i = 1, \cdots, n$, producing the correct answer $SUM = 1$.

Two other tests with arbitrary values of Ai and $n = 4$ were also correct.

CERTIFICATION OF ALGORITHM 160
COMBINATORIAL OF M THINGS TAKEN N AT A TIME [M. L. Wolfson and H. V. Wright, *Comm. ACM*, Apr. 1963]

Dmitri Thoro
San Jose State College, San Jose, Calif.

Algorithm 160 was translated into Fortran II and Forgo for the IBM 1620. Correct results were obtained for values of m up to 20.

CERTIFICATION OF ALGORITHM 161
COMBINATORIAL OF M THINGS TAKEN ONE AT A TIME, TWO AT A TIME, UP TO N AT A TIME [H. V. Wright and M. L. Wolfson, *Comm. ACM*, Apr. 1963]

Dmitri Thoro
San Jose State College, San Jose, Calif.

Algorithm 161 was translated into Fortran II and Forgo for the IBM 1620. Correct results were obtained for values of m up to 20.

CERTIFICATION OF ALGORITHM 162
XYMOVE PLOTTING [Fred G. Stockton, *Comm. ACM*, Apr. 1963]

William E. Fletcher
Bolt, Beranek and Newman Inc., Los Angeles, Calif.

The line in the body of the procedure which read:

if $D \geqq$ **then** $I := I + 2$;

was corrected to read:

if $D \geqq 0$ **then** $I := I + 2$;

With this one change the body of the procedure was transliterated into DECAL-BBN and successfully run on a PDP-1 computer utilizing the cathode ray tube output to display the path of a simulated digital incremental plotter.

CERTIFICATION OF ALGORITHM 164
ORTHOGONAL POLYNOMIAL LEAST SQUARES SURFACE FIT [R. E. Clark, R. N. Kubik, L. P. Phillips, *Comm. ACM*, April 1963]

C. V. Bitterli
Johns Hopkins Univ. Applied Physics Lab., Silver Spring, Md.

The SURFACEFIT algorithm was translated into Fortran and successfully run on an IBM 7094. It was necessary to make the following corrections:

(a) 12th line after

comment evaluate orthogonal polynomials;

should read

$numa := numa + u[i] \times x[i] \times p[n-1,i] \uparrow 2$;

(b) 2nd line after

comment evaluation of orthogonal polynomial coefficients;

should read

$pc[n,n] := 1.0$;

(c) 12th line after

comment evaluation of orthogonal polynomial coefficients;

should read

if $t \neq 1$ **then** $qc[m,t] := qc[m,t] + qc[m-1,t-1]$;

(d) 8th line after

comment evaluation of dependent variables using the approximating polynominal

should read

for $t := mmax - 1$ **step** -1 **until** 1 **do**

The following function was used to generate data for checking this algorithm:

$$z = 1 - x + y - xy + x^2 - y^2$$

for $x = 0, 1, 2, 3, 4$

and $y = 0, 1, 2, 3, 4$

The resulting polynomial was:

$$z = x - 5y - xy + x^2 - y^2$$

which is correct for the normalized variables.

It should be pointed out in the **comment** for this procedure that the resulting polynomial is in the normalized variables and not the original variables.

REMARK ON ALGORITHM 170
REDUCTION OF A MATRIX CONTAINING POLYNOMIAL ELEMENTS [P. E. Hennion, *Comm. ACM*, Apr. 1963]

P. E. Hennion
Giannini Controls Corp., Berwyn, Penn.

Four typographical errors were found upon reviewing the procedure. The following corrections should be made:

(1) The increment for the **for** statement of line start:, should be 1.
(2) The colon at the end of the third line after line start:, should be replaced by a semicolon.
(3) The semicolon at the end of the first line after line LO:, may be removed.
(4) The last statement of the first column should read:

$MAT[i,j] := k$; **end end**;