5. HASTINGS, C., JR. Approximation for digital computers. R-264, Rand Corp., Nov. 1954.
6. BELAGA, E. G. *Dok. Akad. Nauk 123*, 5 (1958).

S. H. EISMAN
Frankford Arsenal
Philadelphia 37, Pa.

REALIZING BOOLEAN CONNECTIVES ON THE IBM 1620

The IBM 1620 (Mod 1) performs its addition by automatic table lookup to a table stored in core storage. Since the contents of the table may be changed under program control, several interesting and powerful operations may be obtained in a simple manner [1].

One such class of operations is the 16 Boolean connectives of two variables. Assume the simplest representation of operand bits—each operand bit is stored as one IBM 1620 digit (extension of the principles discussed here to more dense packing of up to three bits per digit is straightforward). Any of the 16 Boolean connectives of two variables may be represented as a four-bit string which gives the truth table for the connective. For example, the connective NOR can be represented by the truth table.

Operand Bit A	*Operand Bit B*	*Function*
0	0	1
0	1	0
1	0	0
1	1	0

The coding of connective NOR would then be 1000. Table 1 gives encodings for all 16 Boolean connectives.

TABLE 1. ENCODING OF BOOLEAN CONNECTIVES

Description of Connective	*Symbolic*	*Encoded Connective*
0	0	0 0 0 0
A AND B	$A \wedge B$	0 0 0 1
A AND (NOT B)	$A \wedge \bar{B}$	0 0 1 0
A	A	0 0 1 1
(NOT A) AND B	$\bar{A} \wedge B$	0 1 0 0
B	B	0 1 0 1
A EXCLUSIVE OR B	$(A \neq B)$	0 1 1 0
A OR B	$A \vee B$	0 1 1 1
A NOR B	$\overline{A \vee B}$	1 0 0 0
A EQUAL B	$(A = B)$	1 0 0 1
NOT B	$\bar{B}$	1 0 1 0
A OR (NOT B)	$A \vee \bar{B}$	1 0 1 1
NOT A	$\bar{A}$	1 1 0 0
(NOT A) OR B	$\bar{A} \vee B$	1 1 0 1
NOT (A AND B)	$\overline{A \wedge B}$	1 1 1 0
1	1	1 1 1 1

A general subroutine may be written which contains as linkage parameters (1) the address of operand A, (2) the address of operand B, (3) the connective coded as shown in Table 1, and (4) the return location from subroutine.

The subroutine itself consists of the following parts:
1. Access of linkage parameters (this may be done by the linkage itself)
2. Replacement of a portion of the addition table (e.g., four digits) by the encoded connective
3. ADD instruction, which now performs the connective operation on the addressed fields
4. Restoration of the original addition table

Algorithms

J. WEGSTEIN, Editor

EDITOR'S NOTE: Algorithm 152 is printed here as it should have appeared.

ALGORITHM 152
NEXCOM

JOHN HOPLEY
Peat, Marwick, Mitchell & Co., London, England

```
procedure nexcom (char, N, setcomplete, nullvector);
array char; integer N;
label setcomplete, nullvector;
comment char is a column vector containing N elements each of
   which is either 1 or 0. Nexcom transforms char into another
   vector containing the same number of 1's and 0's, but in a differ-
   ent sequence. Starting with char in the state of having 1 in each of
   the element positions 1, ..., r and zeros elsewhere then repeated
   application of nexcom generates all nCr patterns of char. The
   procedure terminates if the presented vector char has 1 in each
   of the positions N, N-1, ... N-r+1 and zeros elsewhere. Termi-
   nation is indicated by exit through the formal label 'setcomplete'.
   If char is the null vector then procedure exists through the
   formal label 'nullvector';
begin integer n, p, m;
comment find the first 1 in char;
for n := 1 step 1 until N do if
char [n] = 1 then go to A;
go to nullvector;
comment how many adjacent 1's;
A:  p := 0;
for m := n + 1 step 1 until N do
if char [m] = 1 then p := p + 1 else go to B;
comment Have all combinations been generated;
B:  if p + n = N then go to setcomplete;
comment Set up next combination; char[n+p+1] := 1;
for m := n + p step - 1 until n do char [m] := 0;
for m := 1 step 1 until p do char [m] := 1;
end nexcom;
```

5. Branch out of the subroutine

A routine of this type has been written for the IBM 1620.

The particular functions OR or EXCLUSIVE OR may also be obtained by replacing only a single digit in the ADD table rather than the four digits required by the above generalized routine. This will be faster than realizing these functions using the general subroutine.

The function NOT may also be obtained by subtracting the argument field from a field of all 1's. The result replaces the field of 1's. Since the two connectives NOT and OR suffice to find all others, for short field lengths it may sometimes be faster to use these routines rather than the general routine.

REFERENCE:

1. GERSON, G. On modifying the 1620 add table. *IBM Systems J.* (Sept. 1962).

H. HELLERMAN AND D. N. SENZIG
International Business Machines Corp.
Yorktown, N. Y.

ALGORITHM 184
ERLANG PROBABILITY FOR CURVE FITTING

A. Colker
U. S. Steel Applied Research Laboratory
Monroeville, Penn.

```
procedure ERLANG (X, XO, M, VARS, C, FACTORIAL, P);
value XO, M, VARS, C;  integer C;  real array X, P;
integer procedure FACTORIAL;
comment  Computes the Erlang probability for the ith interval
  by ∫_0^{x_i} f(x)dx − ∫_0^{x_{i−1}} f(x)dx where f(x) = + [(Kμ)^K/(K−1)!]
  ·(x−x_0)^{K−1} e^{−Kμ(x−x_0)} where μ = 1/M, K = (M−X_0)^2 VARS is
  the upper boundary for the class intervals, X_0 is the lower
  boundary of the first class interval, M is the mean of the Er-
  lang, VARS is the variance corrected by Sheppard's correction,
  C is the number of class intervals and P_i is the calculated
  probability;
begin
  integer I, J, K, F;  real array XE[0 : C];
    for I := 1 step 1 until C do
      XE[I] := X[I] − XO;
    XE[0] := 0;
    ME := M − XO;
    K := 0.5 + (ME↑2)/VARS;
    U := K/ME;
    SP := 0;
    for I := 1 step 1 until C do
    begin
        SUM1 := 0;
        SUM2 := 0;
        for J := 0 step 1 until K − 1 do
        begin
          F := FACTORIAL (J);
          Z1 := U × XE[I−1];
      SUM1 := SUM1 + (Z1↑J)/F;
        Z2 := U × XE[I];
      SUM2 := SUM2 + (Z2↑J)/F;
    end J;
    P[I] := SUM1 × (EXP(−U×XE[I−1])) − SUM2
      × (EXP(−U×XE[I]));
    SP := SP + P[I];
  end I;
    P[C+1] := 1.0 − SP;
end Erlang
```

ALGORITHM 185
NORMAL PROBABILITY FOR CURVE FITTING

A. Colker
U. S. Steel Applied Research Laboratory
Monroeville, Penn.

procedure *NORMAL* (*X*, *M*, *VARS*, *C*, *HASTINGS*, *P*);
value *M*, *VARS*, *C*; **integer** *C*; **real array** *X*, *P*;
real procedure *HASTINGS*;
comment Computes the normal probabilities for the ith interval by $\int_0^{x_i} f(x)dx - \int_0^{x_{i-1}} f(x)dx$ where $f(x)$ is Hastings' approximation to the normal interval. Hastings' formula is

$$\phi(X_{ni}) = \tfrac{1}{2}[1-(1+a_1X_{ni}+a_2X_{ni}^2+a_3X_{ni}^3+a_4X_{ni}^4+a_5X_{ni}^5)^{-8}]$$

where $a_1 = 0.09979268$, $a_2 = 0.04432014$, $a_3 = 0.00969920$, $a_4 = -0.00009862$, and $a_5 = 0.00058155$. The X_{ni} are normalized boundary values of X_i where $X_{ni} = (X_i - M)/\sqrt{VARS}$, where M is the mean and $VARS$ is the variance corrected by Sheppard's correction, C is the number of class intervals and P_i the calculated probability;

```
begin
  integer I;  real array XN[1 : C];
  for I := 1 step 1 until C do XN[I] := (X[I]−M)/SQRT(VARS);
  P[1] := 0.5 − HASTINGS (ABS(XN[1]));
  for I := 2 step 1 until C do
  begin
    if
  XN[I] < 0 then
  P[I] := HASTINGS (ABS(XN[I−1])) − HASTINGS
    (ABS(XN[I]));  else
    begin
      if (XN[I]>0) ∧ (XN[I−1]<0)
      then P[I] := HASTINGS (XN[I]) + HASTINGS
        (ABS(XN[I−1]));  else
      P[I] := HASTINGS (XN[I]) − HASTINGS (XN[I−1]);
    end;
  end I;
  P[C+1] := 0.5 − HASTINGS (XN[C]);
end NORMAL
```

ALGORITHM 186
COMPLEX ARITHMETIC

R. P. van de Riet
Mathematical Centre, Amsterdam, Holland

```
procedure Complex arithmetic (a, b, R, r);  value a, b;  array
  a, b, R, r;
comment  This procedure assigns the value a² + b² to R and the
  value (a+ib)/(a−ib) to r, where a, b, R and r are complex
  numbers. These two arithmetic expressions are of course fully
  arbitrary. They serve only to demonstrate the use of the pro-
  cedures P, Q, S, T, J and U. With them one can build up any
  arithmetic expression with complex variables, as easily as one
  can form them with real variables in ALGOL 60 (As one sees
  immediately these procedures can easily be extended for use in
  quaternion arithmetic or general vector and tensor calculus).
  We focus attention to the value call of the procedure-parameters,
  which is essential. Furthermore, we notice that the depth or
  height of the accumulator H is the number of right-handed
  brackets placed one after another not counting the brackets
which occur in parameter-delimeters. It is perhaps superfluous to
mention that this procedure was tested on the X1 computer of the
Mathematical Centre.;
begin  integer i, k;  array H[1:4,1:2];
  integer procedure P(i, j);  value i, j;  integer i, j;
  comment  P forms the product of the ith and jth element of H;
  begin real a;  k := k − 1;  a := H[i, 1] × H[j, 1] − H[i, 2]
        × H[j, 2];  H[k, 2] := H[i, 1] × H[j, 2] + H[i, 2] ×
        H[j, 1];  H[k, 1] := a;  P := k
  end;
  integer procedure Q(i, j);  value i, j;  integer i, j;
  comment  Q forms the quotient of the ith and jth element of H;
  begin real a, b;  k := k − 1;  b := H[j, 1] ↑ 2 + H[j, 2] ↑ 2;
        a := (H[i, 1]×H[j, 1]+H[i, 2]×H[j, 2])/b;
        H[k, 2] := (H[i, 2]×H[j, 1]−H[i, 1]×H[j, 2])/b;
        H[k, 1] := a;  Q := k
  end;
  integer procedure S(i, j);  value i, j;  integer i, j;
  comment  S forms the sum of the ith and jth element of H;
  begin k := k − 1;  H[k, 1] := H[i, 1] + H[j, 1];
        H[k, 2] := H[i, 2] + H[j, 2];  S := k
  end;
  integer procedure T(a);  array a;
  comment  T assigns to the k+1th element of H the complex
    variable a;
  begin k := k + 1;  H[k, 1] := a[1];  H[k, 2] := a[2];  T := k
    end;
```

```
  integer procedure J(i, expi);  integer i;  real expi;
  comment  J assigns to the (k+1)th element of H a complex
    variable which is decomposed in real and imaginary part;
  begin k := k + 1;  i := 1;  H[k, 1] := expi;  i := 2;
        H[k, 2] := expi;
       J := k
  end;
  procedure U(i, R);  value i;  integer i;  array R;
  comment  U assigns to R the ith element of H;
  begin R[1] := H[i, 1];  R[2] := H[i, 2];  k := 0 end;
  k := 0;  U(S(P(T(a))times:(T(a)))plus:(P(T(b))times:
    (T(b))), R);
  comment  (a×a) + (b×b) =: R;  U(Q(S(T(a)) plus:
    (P(J(i, i−1)) times: (T(b)))) divided by: (S(T(a))
    plus: (P(J(i, 1−i)) times: (T(b)))), r);
  comment  (a+(i×b))/(a+(−i×b)) =: r;
end Complex Arithmetic;
```

The contents of this Algorithm are published in the Technical Note TN 27, Mathematical Centre, Nov. 1962.

ALGORITHM 187
DIFFERENCES AND DERIVATIVES

R. P. van de Riet
Mathematical Centre, Amsterdam, Holland

```
begin  real h;  integer i, k;  array A[1 : 50];
comment  This program calculates, only to demonstrate the
  procedures DELTA and DER, the third derivative of the expo-
  nential function with a sixth order difference scheme. We do
  not propose to use these procedures in actual calculations, for
  as we observed with the X1 computer of the Math. Centre, they
  work, but very slowly as a consequence of the strong recursive-
  ness of the procedures. In actual programming one has to take
  the trouble to write out the well-known formula of Gregory, or
  for higher derivatives to multiply this formula a number of
  times by itself, then one has to collect the same function-values.
  All this trouble is taken over by the computer if one uses the
  procedures described below. My purpose, however, in publishing
  these procedures lies not in the numerical use but in a demon-
  stration of the flexibility of ALGOL 60, if one uses the recursive-
  ness property of procedures.;
  real procedure SUM(i, h, k, ti);  value k;  integer i, k, h;
    real ti;
  begin  real s;  s := 0;  for i := h step 1 until k do s := s + ti;
         SUM := s
  end;
  real procedure DELTA (N, k, k0, fk);  value N, k0;  real fk;
  integer N, k, k0;
  comment  N is the order of the forward difference which is
    calculated from a set of function-values with equidistant
    parameter-values;
  begin  integer i;
         DELTA := if N = 1
         then SUM (k, k0, k0+1, (−1)↑(k+1−k0)×fk)
         else DELTA (1, i, k0, DELTA (N−1, k, i, fk))
  end;
  real procedure DER (OR, N, h, k, k0, fk);  value OR, N, h, k0;
    real fk, h;
  integer OR, N, k, k0;
  comment  OR is the order of the derivative, calculated from a
    given set of function-values f(k), with equidistant parameter-
    values, the error is of the order h ↑ (N+1−OR), where h is the
    steplength. k0 is the point where the derivative is calculated;
  begin  integer i;
         DER := if OR = 1
         then SUM(i, 1, N, DELTA(i, k, k0, fk)
           ×(−1)↑(i+1)/i)/h
         else DER(1, N+1−OR, h, i, k0, DER(OR−1, N−1, h,
           k, i, fk))
  end;
  for i := 1 step 1 until 50 do A[i] := exp(i/50);
  for i := 1 step 1 until 25 do A[i] := DER(3, 6, .02, k, i, A[k])
end
```

The contents of this Algorithm are published in the Technical Note TN 27, Mathematical Centre, Nov. 1962.

ALGORITHM 188
SMOOTHING 1.

F. Rodriguez-Gil
Central University, Caracas, Venezuela

```
procedure Smooth 13(n, x);
integer n;
real array x;
comment  This procedure uses Gram's first-degree three-point
  formulas, as described in Hildebrand's "Introduction to Nu-
  merical Analysis," Ch. 7, to smooth a series of n equally spaced
  values. If the procedure is entered with less than three points,
  control is transferred to a nonlocal label error;
begin real array xp[1 : n];  integer i;
  if n < 3 then go to error;
  for i := 1 step 1 until n do xp[i] := x[i];
  x[1] := 0.83333333 × xp[1] + 0.33333333 × xp[2] − 0.16666667
    × xp[3];
  for i := 2 step 1 until n − 1 do x[i] := (xp[i−1]+xp[i]
    + xp[i+1]) × 0.33333333;
  x[n] := − 0.16666667 × xp[n−2] + 0.33333333 × xp[n−1]
    + 0.83333333 × xp[n]
end  Smooth 13
```

ALGORITHM 189
SMOOTHING 2

F. Rodriguez Gil
Central University, Caracas, Venezuela

```
procedure Smooth 35(n, x);
integer n;
real array x;
comment  This procedure is similar to Smooth 13, except that
  Gram's third-degree five-point formulas are used, and that a
  minimum of five points is needed for a successful application;
begin real array xp[1 : n];  integer i;
  if n < 5 then go to error;
  for i := 1 step 1 until n do xp[i]:= x[i];
  x[1] := 0.98571429 × xp[1] + 0.05714286 × (xp[2]+xp[4])
    − 0.08571429 × xp[3] − 0.01428571 × xp[5];
  x[2] := 0.05714286 × (xp[1]+xp[5]) + 0.77142857 × xp[2]
    + 0.34285714 × xp[3] − 0.22857143 × xp[4];
  for i := 3 step 1 until n − 2 do x[i] := − 0.08571429 × (xp[i−2]
    +xp[i+2]) + 0.34285714 × (xp[i−1]+xp[i+1]) + 0.48571429
    × xp[i];
  x[n−1] := 0.05714286 × (xp[n−4]+xp[n]) − 0.22857143
    × xp[n−3] + 0.34285714 × xp[n−2] + 0.77142857 × xp[n−1];
  x[n] := − 0.01428571 × xp[n−4] + 0.05714286 × (xp[n−3]
    +xp[n−1]) − 0.08571429 × xp[n−2] + 0.98571429 × xp[n]
end  Smooth 35
```

ALGORITHM 190
COMPLEX POWER

A. P. Relph
The English Electric Co. Ltd., Whetstone, England

```
procedure Complex power (a, b, c, d, n, x, y);   value a, b, c, d, n;
  real a, b, c, d, x, y;   integer n;
comment  This procedure calculates (x+iy) = (a+ib) ↑ (c+id)
  where i is the root of −1. In the complex plane, with a cut along
  the real axis from 0 to −∞, p is the sum of the principal value
  of the argument of (a+ib) and 2nπ (n is positive, negative or
  zero depending on the solution required). arctan is assumed to
  be in the range −π/2 to π/2. The case n = 0, d = 0 is given by
  Algorithm 106;
begin  real p, r, v, w;
       if a = 0 then begin if b = 0 then begin x := y := 0;
                                         go to L end
                                   else p := 1.570796633 ×
                                          (sign(b)+4×n)
                     end
              else begin p := 6.28318532 × n + arctan(b/a);
                         if a < 0 then begin if b ≧ 0 then
                                                p := p + 3.14159265
                                             else
                                                p := p − 3.14159265
                                       end
                   end;
       r := .5 × ln(a↑2+b↑2);   v := c × p + d × r;
       w := exp(c×r−d×p);
       x := w × cos(v);   y := w × sin(v);
L:  end
```

ALGORITHM 191
HYPERGEOMETRIC

A. P. Relph
The English Electric Co. Ltd., Whetstone, England

```
procedure Hypergeometric (a1, a2, b1, b2, c1, c2, z1, z2) Results:
  (s1, s2);   value a1, a2, b1, b2, c1, c2, z1, z2;   real a1, a2, b1, b2,
  c1, c2, z1, z2, s1, s2;
begin  comment  calculates the hypergeometric function
          1F2(a, b, c, z) with complex parameters (a=a1+ia2,
          etc);
       real d, y1, y2;   integer n;
       procedure comp mult (a1, a2, b1, b2, c1, c2);   value a1,
         a2, b1, b2;   real a1, a2, b1, b2, c1, c2;
         begin  comment calculates the product of the two
                  complex numbers (a1+ia2) and (b1+ib2)
                  where i is the root of −1;
                c1 := a1 × b1 − a2 × b2;   c2 := a2 × b1 +
                  a1 × b2
         end;
       s1 := y1 := 1;   s2 := y2 := 0;
       for n := 1 step 1 until 100 do
       begin d := n × ((c1+n−1)↑2+c2↑2);
             comp mult (a1+n−1, a2, y1/d, y2/d, y1, y2);
             comp mult (y1, y2, b1+n−1, b2, y1, y2);
             comp mult (y1, y2, c1+n−1, −c2, y1, y2);
             comp mult (y1, y2, z1, z2, y1, y2);
             if s1 = s1 + y1 ∧ s2 = s2 + y2 then go to L;
             s1 := s1 + y1;   s2 := s2 + y2
       end;
L:  end
```

ALGORITHM 192
CONFLUENT HYPERGEOMETRIC

A. P. Relph
The English Electric Co. Inc., Whetstone, England

```
procedure Confluent hypergeometric (a1, a2, c1, c2, z1, z2)
  Result : (s1, s2);   value a1, a2, c1, c2, z1, z2;
  real a1, a2, c1, c2, z1, z2, s1, s2;
begin  comment calculates the confluent hypergeometric func-
          tion 1F1(a, c, z) with complex parameters
          (a=a1+ia2, etc);
       real d, y1, y2;   integer n;
       procedure comp mult (a1, a2, b1, b2, c1, c2);
         value a1, a2, b1, b2;   real a1, a2, b1, b2, c1, c2;
         begin  comment calculates the product of the two
                  complex numbers (a1+ia2) and (b1+ib2)
                  where i is the root of −1;
                c1 := a1 × b1 − a2 × b2;
                c2 := a2 × b1 + a1 × b2
         end;
       s1 := y1 := 1;   s2 := y2 := 0;
       for n := step 1 until 100 do
       begin  d := n × ((c1+n−1)↑2+c2↑2);
                comp mult (a1+n−1, a2, y1/d, y2/d, y1, y2);
                comp mult (y1, y2, c1+n−1, −c2, y1, y2);
                comp mult (y1, y2, z1, z2, y1, y2);
              if s1 = s1 + y1 ∧ s2 = s2 + y2 then go to L;
              s1 := s1 + y1;   s2 := s2 + y2
       end;
L:  end
```

ALGORITHM 193
REVERSION OF SERIES

Henry E. Fettis
Aeronautical Research Laboratories, Wright-Patterson Air
Force Base, Ohio

```
procedure SERIESRVRT (A, B, N);
value A, N;   array A, B;   integer N;
comment This procedure gives the coefficients B[i] for the series
  x = y + ΣB[i] × y ↑ i (i=2, 3, ··· , n) when the coefficients
  A[i] of the series y = x + ΣA[i] × x ↑ i are given. The procedure
  uses successive approximations after writing y_{L+1} = x − ΣB[i] ×
  y_L ↑ i (i=2, 3, ··· , L+2 and L=0, 1, ··· , N−2) starting with
  y_0 = x;
begin integer i, j, k, m;
  array Q, R [0 : N];
  real s;
  A[1] := B[0] := 0;
  B[1] := 1;
  for k := 1 step 1 until N − 1 do
  begin B[k+1] := 0;
    for i := 0 step 1 until k + 1 do
    R[i] := 0;
    for j := k + 1 step − 1 until 1 do
    begin Q[0] := R[0] − A[j];
      for i := 1 step 1 until k + 1 do
      Q[i] := R[i];
      for i := 0 step 1 until k + 1 do
      begin s := 0;
        for m := 0 step 1 until i do
        s := s + B[m] × Q[i−m];
        R[i] := s
      end for i;
    end for j;
    for i := 2 step 1 until k + 1 do B[i] := R[i]
  end for k;
end SERIESRVRT
```

REMARK ON CERTIFICATION OF MATRIX INVERSION PROCEDURES

CLEVE MOLER
Stanford University, Stanford, California

(Work supported, in part, by National Science Foundation, and by Office of Naval Research under Contract No. 225(37).)

In a recent certification [1], two matrix inversion procedures were tested by inverting machine-generated Hilbert matrices and comparing the results with the theoretical inverses. As has been pointed out elsewhere [2], this is an inappropriate and deceptive test. We give here a further discussion of the difficulties involved.

Hilbert matrices, even of low orders, are so poorly conditioned that the small errors created by truncating or rounding their elements to fit a computer word cause severe changes in their inverses. The results of an inversion procedure should therefore be compared with the true inverses of these modified input matrices, rather than with the inverses of the unaltered Hilbert matrices.

To be more specific, let H_n denote the $n \times n$ Hilbert matrix defined by $(H_n)_{i,j} = 1/(i+j-1)$, $i, j = 1, \cdots, n$. Let $H_n^{(b)}$ denote the matrix of normalized floating-point numbers with b-bit fractions obtained by truncating the binary expansions of the elements of H_n. That is, $(H_n^{(b)})_{i,j} = 2^{-b-k}[2^{b+k}(H_n)_{i,j}]$ where $[x]$ is the greatest integer not exceeding x, and k is the integer for which $\frac{1}{2} \leqq 2^k(H_n)_{i,j} < 1$. Let T_n and $T_n^{(b)}$ denote the true inverses of H_n and $H_n^{(b)}$ respectively.

Table 1 gives the maximum elements of T_n and $T_n^{(b)}$, for several values of n and b. It should be noted that the differences between T_n and $T_n^{(29)}$ are about the same size as the "errors" found in [1], where a 29-bit fraction was used. This is typical: the changes caused by the truncation of input data are often as large as the errors caused by numerical inversion.

With the true inverses of $H_n^{(b)}$ available, it is possible to use these matrices to test the procedures *InversionII* and *gjr* discussed in [1]. *InversionII* is a partial pivoting routine; i.e. at each step of the reduction a single column is searched for the pivot of greatest absolute value. *gjr* uses full pivoting; i.e. the entire unreduced matrix is searched for pivot at each stage. In addition, two experimental modifications were made to *gjr* to cause partial pivoting and no pivoting, respectively. The four routines were run on an IBM 7090 using SUBALGOL (Stanford-Burroughs ALGOL) and FORTRAN. For a 7090 one has $k = 27$, so the results were compared with $T_n^{(27)}$. The maximum errors are tabulated in Table 2. For $n \geqq 8$, the results are dominated by errors. Table 2 also gives the maximum element of $T_n^{(27)} - T_n$ for comparison with the actual errors.

TABLE 1. $T_n^{(b)}$ is the true inverse of the b-bit approximation to the Hilbert matrix.

	Maximum Element				
n	$T_n^{(25)}$	$T_n^{(27)}$	$T_n^{(29)}$	$T_n^{(36)}$	T_n
4	6480.224	6480.046	6480.015	6480.000	6480
5	179273.60	179246.94	179207.51	179200.08	179200
6	4492480.8	4434355.1	4414530.4	4410074.9	4410000
7	$1985.829_{10}5$	$1340.502_{10}5$	$1347.137_{10}5$	$1334.385_{10}5$	$1334.025_{10}5$
8	$58864.37_{10}5$	$27268.01_{10}5$	$54260.62_{10}5$	$42527.95_{10}5$	$42499.42_{10}5$
10	—	—	—	$-5.07_{10}12$	$3.48_{10}12$

TABLE 2. Error matrices are the differences between $T_n^{(27)}$ and the output of the procedures.

	Maximum Element of Error Matrix				Maximum Element
n	No Pivoting *gjr*	Partial Pivoting *ghr*	Full Pivoting *ghr*	*InversionII*	$T_n^{(27)} - T_n$
3	$4.2_{10}-5$	$5.0_{10}-5$	$4.2_{10}-5$	—	$8.6_{10}-5$
4	$2.2_{10}-2$	$3.6_{10}-2$	$3.9_{10}-2$	$1.9_{10}-2$	$4.6_{10}-2$
5	$2.0_{10}1$	$0.31_{10}1$	$1.2_{10}1$	$0.51_{10}1$	$4.5_{10}1$
6	$0.27_{10}4$	$1.1_{10}4$	$0.35_{10}4$	$1.7_{10}4$	$0.065_{10}4$
7	$1.9_{10}7$	$2.8_{10}7$	$1.2_{10}7$	$2.3_{10}7$	$0.065_{10}7$

Note that for $n = 4$, *InversionII* gives the least maximum error, while for $n = 5, 6, 7$ the best routines are partial pivoting *gjr*, no pivoting *gjr* and full pivoting *gjr*, respectively. Thus, these results do not indicate the superiority of either a full or a partial pivoting strategy. An explanation is supplied by the fact that $H_n^{(b)}$ is positive definite for the values of n and b considered. Wilkinson's matrix inversion error bounds are not altered by the omission of pivoting for positive definite matrices [3]. The need for at least partial pivoting for general matrices can, of course, be clearly demonstrated by simple examples.

The matrices $T_n^{(b)}$ were calculated with an iterative improvement technique described and analyzed in [4]. They are correct to the number of figures given. The routine used is similar to that given by McKeeman [5], except that multiple precision arithmetic is used.

REFERENCES:

1. NAUR, P. Certification of algorithms 120 and matrix inversion by Gauss-Jordan. *Comm ACM 6*, 1 (Jan. 1963), 40.
2. WILKINSON, J. H. Error analysis of direct methods of matrix inversion. *J. ACM 8*, 3 (July 1961), 319–322.
3. ——, op. cit., 285–287.
4. MOLER, C. Numerical matrix inversion with iterative improvement. Tech. Report 32–394, Jet Propulsion Lab., Pasadena, Calif., Mar. 1963.
5. MCKEEMAN, W. M. Crout with equilibration and iteration. Algorithm 135, *Comm ACM 5*, 11 (Nov. 1962), 553.

CERTIFICATION OF ALGORITHM 105 NEWTON MAEHLY [F. L. Bauer and J. Stoer, COMM. ACM, July 1962]

JOANNE KONDO
Burroughs Corp., Pasadena, Calif.

Algorithm 105 was successfully run on Burroughs 220 computer after the following correction had been made:

for i := 0 **step** 1 **until** $n - 1$ **do** $b[i] := (n-1) \times a[i]$

changed to

for i := 0 **step** 1 **until** $n-1$ **do** $b[i] := (n-i) \times a[i]$.

The following polynomials were tested for real roots using this algorithm:

polynomial	*epsilon*	*accuracy*
(1) $x^3 - 2x^2 - 5x + 6$	0.0000001	10^{-8}
(2) $x^5 - 15x^4 + 85x^3 - 225x^2 + 274x - 120$	0.000001	10^{-5}

A contribution to this department must be in the form of an Algorithm, a Certification, or a Remark. Contributions should be sent in duplicate to the Editor and should be written in a style patterned after recent contributions appearing in this department. An algorithm must be written in ALGOL 60 (see *Communications of the ACM*, January 1963) and accompanied by a statement to the Editor indicating that it has been tested and indicating which computer and programming language was used. For the convenience of the printer, contributors are requested to double space material and underline delimiters and logical values that are to appear in boldface type. Whenever feasible, Certifications should include numerical values.

CERTIFICATION OF ALGORITHMS 134 AND 158 EXPONENTIATION OF SERIES [Henry E. Fettis, Comm. ACM, Oct. 1962 and Mar. 1963]

Henry C. Thacher, Jr.
Reactor Engineering Div., Argonne National Laboratory Argonne, Ill.

Work supported by the U.S. Atomic Energy Commission.

The bodies of SERIESPWR were transcribed for the Dartmouth Scalp processor for the LGP-30 computer. In addition to the modifications required by the limitations of this translator, the following corrections were necessary:

1. Add "**real** P;" to the specifications.
2. Delete "p," from the declarations in the procedure body.
3. (134 only) Replace "S" by "s" and $[i-k]$ by "$(i-k)$" in the statement $S := s + \cdots$.
4. (158 only) Changes last sentence of comment to "Setting $P := 0$ gives the coefficients for $ln(f(x))$. In this series, the constant term is 0, instead of 1 as elsewhere;"
5. (158 only) Add the identifier $P2$ to the declared real variables.
6. (158 only) Make the first statements read:
   "**if** $P = 0$ **then** $P2 := 1$ **else** $P2 := P$;
   $B[1] := P2 \times A[1]$;
7. (158 only) Make the statement of the **for** k loop read

   "$S := S+(P\times(i-k)-k) \times B[k] \times A[i-k]$;"

8. Change the last statement to

   "$B[i] := P2 \times A[i] + S/i$ **end** for i;

In addition, the following modifications would improve the efficiency of the program:

1. Remove A from the value list.
2. Omit the statement $B[1] := P \times A[1]$; ($P2\times A[1]$ in 158 according to correction 6) and change the initial value of i in the statement following from 2 to 1.

When these changes were made, both procedures produced the first ten coefficients of the series for $(exp(x)) \uparrow 2.5$ from the first ten coefficients of the exponential series. The procedures were also used to generate the binomial coefficients by applying them to $(1+x)^P$, for $P = 2.0$, and 0.5000000. Algorithm 158 was also tested with $P := 0$ for $1+x$ and for the series expansions for $(sin\ x)/x$, $cos\ x$, and $exp\ x$. In all cases, the coefficients agreed with known values within roundoff.

REMARK ON ALGORITHM 150 SYMINV2 [H. Rutishauser, Comm. ACM, Feb. 1963]

Arthur Evans, Jr.
Carnegie Institute of Technology, Pittsburgh, Pennsylvania

The identifier "a" appears twice in the procedure heading as a formal parameter. It is not clear that this situation has any meaning in Algol. Indeed, it is not at all obvious how one might translate the procedure. If the actual parameters corresponding to the two formal parameters with the same identifier are different there is no way for the translator (or for the reader) to distinguish which 'a' is to be used. Further, it would take a detailed examination of the published algorithm to determine how this situation might be corrected. It is certainly not clear that it would be safe merely to delete one occurrence of the formal parameter 'a', since the operation of the algorithm might requre that two separate matrices be available.

REMARK ON ALGORITHM 150 SYMINV2 [H. Rutishauser, Comm. ACM, Feb. 1963]

H. Rutishauser
Eidg. Technische Hochschule, Zurich, Switzerland

procedure *syminv* 2 (a, n) result : (a) exit : $(fail)$; $\cdots$ indicates that the value of parameter "a" is changed by the computing process (the matrix a is changed into its inverse, whereby the given matrix is destroyed). In any procedure call, the two actual parameters corresponding to the two a's must be identical, otherwise the action of the procedure will be undefined (by virtue of the substitution rule). The user may also change the procedure heading into *syminv* 2 (a, n) exit : $(fail)$; $\cdots$ without changing the effect of the procedure.

Editor's Note: The ALCOR group has adopted the rule that if the value of a parameter is changed by the execution of the procedure, then the parameter should be listed twice. Although the Algol 60 Report does not forbid listing a formal parameter twice, it would appear that a compiler which thus restricts the language could not accept some of the examples given in the Algol 60 Report.

REMARK ON ALGORITHM 177 LEAST SQUARES SOLUTION WITH CONSTRAINTS [Michael J. Synge, Comm. ACM, June 63]

Michael J. Synge
The Boeing Co., Transport Division, Renton, Wash.

In row-reducing the constraint equations, *CONLSQ* does not use full pivoting nor does it detect redundancy or inconsistency of the constraints; it was felt that the constraints were likely to be few in number and well-conditioned. However, these omissions may be made good by replacing the statement

$$ick := ick + 1;$$

by

$$done: ick := ick + 1;$$

and substituting the lines below for the first seven lines of the first compound statement of *CONLSQ*. If inconsistency is found, the procedure exits to the nonlocal label *inconsistent*. A roundoff tolerance, *eps*, is used in checking consistency, and some numerical value (e.g. 10^{-6}) should be substituted for it.

```
begin integer i, j, k, ii, ick, mr; integer array ic[1 : m];
   array B[1 : n-r, 1 : m-r];
  real Amax, Atemp;
  for i := 1 step 1 until r do
  begin k := 1;  mr := i;  Amax := A[i, 1];
    for ii := i step 1 until m do
    begin for j := 1 step 1 until m do
      begin if abs(Amax ≧ abs(A[ii, j]) then go to nogo;
        mr := ii;  k := j;  Amax := A[ii, j];
nogo: end j
    end ii;
    if abs(Amax) ≧ eps then go to allswell;  mr := i;
test: if abs(y[mr]) ≧ eps then go to inconsistent else mr := mr + 1;
    if r ≧ mr then go to test else r := i - 1;
    go to done;
allswell: for j := 1 step 1 until r do
    begin Atemp := A[mr, j];  A[mr, j] := A[i, j];
      A[i, j] := Atemp/Amax
    end j;
    Atemp := y[mr];  y[mr] := y[i];  y[i] := Atemp/Amax:
```

The Algorithm then continues with the line:

```
    for ii := 1 step 1 until r do
```