Algorithms

J. G. HERRIOT, Editor

Revised Algorithms Policy • August, 1966

A contribution to the Algorithms Department should be in the form of an algorithm, a certification, or a remark. Contributions should be sent in duplicate to the editor, typewritten double spaced. Authors should carefully follow the style of this department with especial attention to indentation and completeness of references.

An algorithm must normally be written in the ALGOL 60 Reference Language [*Comm. ACM 6* (Jan. 1963), 1-17] or in ASA Standard FORTRAN or Basic FORTRAN [*Comm.* ACM 7 (Oct. 1964), 590-625]. Consideration will be given to algorithms written in other languages provided the language has been fully documented in the open literature and provided the author presents convincing arguments that his algorithm is best described in the chosen language and cannot be adequately described in either ALGOL 60 or FORTRAN.

An algorithm written in ALGOL 60 normally consists of a commented procedure declaration. It should be typewritten double spaced in capital and lower-case letters. Material to appear in **boldface** type should be underlined in black. Blue underlining may be used to indicate italic type, but this is usually best left to the Editor. An algorithm written in FORTRAN normally consists of a commented subprogram. It should be typewritten double spaced in the form normally used for FORTRAN or it should be in the form of a listing of a FORTRAN card deck together with a copy of the card deck. Each algorithm must be accompanied by a complete driver program in its language which generates test data, calls the procedure, and produces test answers. Moreover, selected previously obtained test answers should be given in comments in either the driver program or the algorithm. The driver program may be published with the algorithm if it would be of major assistance to a user.

For ALGOL 60 programs, input and output should be achieved by procedure statements, using any of the following eleven procedures (whose body is not specified in ALGOL) [See "Report on Input-Output Procedures for ALGOL 60," *Comm. ACM 7* (Oct. 1964), 628-629]:

insymbol	*inreal*	*outarray*	*ininteger*
outsymbol	*outreal*	*outboolean*	*outinteger*
length	*inarray*	*outstring*	

If only one channel is used by the program for output, it should be designated by 1 and similarly a single input channel should be designated by 2. Examples:

```
outstring (1, 'x='); outreal (1,x);
for i := 1 step 1 until n do outreal (1,A[i]);
ininteger (2, digit [17]);
```

For FORTRAN programs, input and output should be achieved as described in the ASA preliminary report on FORTRAN and Basic FORTRAN.

It is intended that each published algorithm be well organized, clearly commented, syntactically correct, and a substantial contribution to the literature of Algorithms. It is necessary but not sufficient that a published algorithm operate on some machine and give correct answers. It must also communicate a method to the reader in a clear and unambiguous manner. All contributions will be refereed both by human beings and by an appropriate compiler. Authors should pay considerable attention to the correctness of their programs, since referees cannot be expected to debug them.

Certifications and remarks should add new information to that already published. Readers are especially encouraged to test and certify previously uncertified algorithms. Rewritten versions of previously published algorithms will be refereed as new contributions and should not be imbedded in certifications or remarks.

Galley proofs will be sent to authors; obviously rapid and careful proofreading is of paramount importance.

Although each algorithm has been tested by its author, no liability 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.—J.G.H.

ALGORITHM 290
LINEAR EQUATIONS, EXACT SOLUTIONS [F4]

J. Boothroyd* (Recd. 7 Sept. 1965 and 21 Mar. 1966)
U. of Tasmania, Hobart, Tas., Australia

* Thanks are due to the referee for useful criticism and awkward test cases.

```
procedure exactle(a, b, n, det);  value n;  integer n, det;
  integer array a, b;
comment solves the matrix equation Ax = b for A = a [1:n,
  1:n] and x, b[1:n] where the elements of A, b are small integers
  and the results are required as ratios of integers. The solution
  vector overwrites b and has values given by det A × x where det
  A is the determinant of A and x is the true solution vector. The
  user is warned that this procedure, of limited though useful
  application, is not a substitute for other well-established
  methods of solving general sets of linear equations owing to the
  inherent danger of integer overflow. This may occur in the
  reduction if the elements of the matrix are large or in the back
  substitution if the determinant and/or the elements of the right-
  hand side are large and may even occur with small elements and
  determinant if the order of the matrix and the nature of the
  equations combine to produce large solution values. Four
  devices intended to avoid integer overflow are incorporated.
  These are, (1) choice of column pivots having the smallest non-
  zero absolute value, (2) division by previous pivots (both after
  Fox, L., An Introduction to Numerical Linear Algebra, Oxford
  U. Press, New York, 1965, p. 82), and (3) the local procedures
  crossmpy and abdivc which respectively evaluate integer expres-
  sions of the form (a×b−c×d) ÷ e and a × b ÷ c by performing
  the divisions before the multiplications. The output parameter
  det yields the determinant of A. If A is singular det := 0;
 begin integer piv, pivot, sum, arii, aki, i, j, k, pivi, ri, rk, m;
   integer array r [1:n]; boolean zpiv;
   integer procedure iabs (it); value it; integer it;
     iabs := if it < 0 then − it else it;
   integer procedure crossmpy(a)times:(b)minus:(c)times:(d)all
     over:(e);
     value a,b,c,d,e;  integer a,b,c,d,e;
   begin integer qab,qcd,r,res;
     if iabs(a) > iabs(b) then
     begin
       qab := a ÷ e;  r := a − qab × e;
       qab := qab × b;  res := r × b
     end
     else
     begin
       qab := b ÷ e;  r := b − qab × e;
       qab := qab × a; res := r × a
     end;
     if iabs(c) > iabs(d) then
     begin
       qcd := c ÷ e;  r := c − qcd × e;
       qcd := qcd × d;  res := res − r × d
     end
     else
```

```
    begin
      qcd := d ÷ e;  r := d − qcd × e;
      qcd := qcd × c;  res := res − r × c
    end;
    crossmpy := qab − qcd + res ÷ e
  end crossmpy;
  integer procedure abdivc(a,b,c,sum);  value a,b,c;  integer
      a,b,c,sum;
  comment evaluates expressions of the form a × b ÷ c by
      performing divisions before multiplications, assigning the
      quotient to abdivc and accumulating the remainder in sum;
  begin integer q,r,temp;
    if iabs(a) > iabs(b) then
    begin q := a ÷ c;  temp := q × b;
      r := a − c × q;
      q := b ÷ c;
      abdivc := temp + q × r;
      sum := sum + (b−q×c) × r
    end
    else
    begin q := b ÷ c;  temp := q × a;
      r := b − c × q;
      q := a ÷ c;
      abdivc := temp + q × r;
      sum := sum + (a−q×c) × r
    end
  end abdivc;
  procedure permb(b,r,n);  value n;  integer array b,r;  inte-
      ger n;
  comment rearranges the elements of b[1:n] so that b[i] :=
      b[r[i]], i = 1, 2, ···, n;
  begin integer i,k,w;
    for i := n step −1 until 2 do
    begin k := r[i];
L:
      if k ≠ i then
      begin
        if k > i then begin k := r [k];  go to L end;
        w := b[i];  b[i] := b[k];  b[k] := w
      end
    end
  end permb;
  m := 1;
  for i := 1 step 1 until n do r[i] := i;
  for i := 1 step 1 until n do
  begin pivot := 0;  zpiv := true;
    for k := i step 1 until n do
    begin aki := iabs(a[r[k],i]);
      if zpiv ∧ aki > 0 ∨ aki ≠ 0 ∧ aki < iabs(pivot) then
      begin zpiv := false; pivi := k;  pivot := a[r[k],i] end
    end;
    if pivot = 0 then begin det := 0;  go to out end;
    ri := r[pivi];  r[pivi] := r[i];  r[i] := ri;  if pivi ≠ i then
        m := − m;
    for k := i + 1 step 1 until n do
    begin rk := r[k];  aki := a[rk,i];
      for j := i + 1 step 1 until n do
        a[rk,j] := if i = 1 then a[rk,j] × pivot − aki × a[ri,j]
              else crossmpy(a[rk,j],pivot,aki,a[ri,j],piv);
      b[rk] := if i = 1 then b[rk] × pivot − aki × b[ri]
                    else crossmpy(b[rk],pivot,aki,b[ri],piv)
    end;
    piv := pivot
  end;
  ri := r[n];
  if m ≠ 1 then
  begin det := aki := − a[ri,n];  b[ri] := − b[ri] end
  else det := aki := a[ri,n];
  for i := n − 1 step −1 until 1 do
  begin ri := r[i];  arii := a[ri,i];
    sum := 0;  piv := abdivc(b[ri],aki,arii,sum);
    sum := − sum;
    for j := i + 1 step 1 until n do
      piv := piv − abdivc(b[r[j]],a[ri,j],arii,sum);
    b[ri] := piv − sum ÷ arii
  end;
  permb(b,r,n);
out:
end exactle
```

ALGORITHM 291
LOGARITHM OF GAMMA FUNCTION [S14]

M. C. Pike and I. D. Hill (Recd. 8 Oct. 1965 and 12 Jan. 1966)
Medical Research Council's Statistical Research Unit, University College Hospital Medical School, London, England

```
real procedure loggamma (x);
  value x;  real x;
comment This procedure evaluates the natural logarithm of
  gamma(x) for all x > 0, accurate to 10 decimal places. Stirling's
  formula is used for the central polynomial part of the procedure.;
begin
  real f, z;
  if x < 7.0 then
  begin f := 1.0;  z := x − 1.0;
    for z := z + 1.0 while z < 7.0 do
    begin x := z;  f := f × z
    end;
    x := x + 1.0;  f := − ln(f)
  end
  else f := 0;
  z := 1.0/x ↑ 2;
  loggamma := f + (x−0.5) × ln(x) − x + .91893 85332 04673 +
    (((−.00059 52380 95238×z+.00079 36507 93651) × z −.00277
    77777 77778)×z+.08333 33333 33333)/x
end loggamma
```

REMARK ON ALGORITHM 178 [E4]
DIRECT SEARCH [Arthur F. Kaupe, Jr., *Comm. ACM* 6 (June 1963), 313]

M. Bell and M. C. Pike (Recd. 15 Nov. 1965 and 22 Apr. 1966)
Institute of Computer Science, University of London, London, England, and Medical Research Council's Statistical Research Unit, London, England

Algorithm 178 has the following syntactical errors:
(1) The parameter list should read
    $(psi, K, DELTA, rho, delta, S)$.
(2) The declaration
    integer K,k;
should read
    integer k;
(3) An extra **end** bracket is required immediately before **end** E;.

The algorithm compiled and ran after these modifications had been made but for a number of problems took a prodigious amount of computing owing to a flaw in the algorithm caused by rounding error. This flaw is in **procedure** E and may be illustrated by the one-dimensional case. Let $S(x) = 1.5 - x\ (x \leq 1.5),\ 3x - 4.5\ (x>$

1.5), and start at 0 with a step of 1. The first move puts *psi* [1] = 1, *phi* [1] = 2. The second move should then put *phi* [1] = 1 = *psi*[1] resulting in a jump to label 1. On many machines, however, *E* will put *phi* [1] = 1 + *e* (*e*>0 and very small) so that direct search begins to move away from 1 in very small steps. This is clearly not desirable and may be avoided by altering the line

```
    if SS < Spsi then go to 2 else go to 1 end;
```

to

```
    if SS ≥ Spsi then go to 1;
    for k := 1 step 1 until K do
    if abs (phi[k]-psi[k]) > 0.5 × DELTA then go to 2
    end;
```

To accelerate the procedure, direct search should take advantage of its knowledge of the sign of its previous move in each of the *K* directions. Take, for example, the one-dimensional case with starting point zero and the minimum far out and negative; the pattern moves will arrive there quite efficiently but each first move of *E* on the way will be positive whereas the previous experience of the search should lead it to suspect the minimum to be in the opposite direction.

Finally, two changes which we have found very useful are (i) some escape clause in the procedure to enable an exit to be made if the procedure has not terminated after some given number of function evaluations *maxeval*, with a Boolean *converge* taking the value **true** in general but **false** if the procedure has terminated through exceeding this number of function evaluations; and (ii) taking *Spsi* into the parameter list where it is called by name so that on exit *Spsi* contains the minimum value of the function.

With these modifications the procedure now reads:

```
procedure direct search (psi,K,Spsi,DELTA,rho,delta,S,converge,
    maxeval);
  value K,DELTA,rho,delta,maxeval; integer K,maxeval;
    array psi;
  real DELTA,rho,delta,Spsi; real procedure S; Boolean
    converge;
comment This procedure locates the minimum of the function S of
  K variables. The method used is that of R. Hooke and T. A.
  Jeeves ["Direct search" solution of numerical and statistical
  problems, J. ACM. 8 (1961), 212-229] and the notation used is
  theirs except for the obvious changes required by ALGOL. On
  entry: psi[1:K] = starting point of the search, DELTA =
  initial step-length, rho = reduction factor for step-length,
  delta = minimum permitted step-length (i.e. procedure is termi-
  nated when step-length < delta), maxeval = maximum per-
  mitted number of function evaluations. On exit: psi[1:K] =
  minimum point found and Spsi = value of S at this point,
  converge = true if exit has been made from the procedure be-
  cause a minimum has been found (i.e., step-length < delta)
  otherwise converge = false (i.e. maximum number of function
  evaluations has been reached);
begin integer k,eval; array phi,s[1:K]; real Sphi,SS,theta;
  procedure E;
  for k := 1 step 1 until K do
  begin phi[k] := phi[k] + s[k]; Sphi := S(phi); eval := eval
      + 1;
    if Sphi < SS then SS := Sphi else
    begin s[k] := − s[k]; phi[k] := phi[k] + 2.0 × s[k];
      Sphi := S(phi); eval := eval + 1;
      if Sphi < SS then SS := Sphi else
      phi[k] := phi[k] − s[k]
    end
  end E;
Start: for k := 1 step 1 until K do s[k] := DELTA;
  Spsi := S(psi); eval := 1; converge := true;
1: SS := Spsi;
  for k := 1 step 1 until K do phi[k] := psi[k]; E;
  if SS < Spsi then
  begin
2: if eval ≥ maxeval then
    begin converge := false;
      go to EXIT
    end;
    for k := 1 step 1 until K do
    begin if phi[k] > psi[k] ≡ s[k] < 0 then s[k] := −s[k];
      theta := psi[k]; psi[k] := phi[k]; phi[k] := 2.0 × phi[k] −
        theta
    end;
    Spsi := SS; SS := Sphi := S(phi); eval := eval + 1; E;
    if SS ≥ Spsi then go to 1;
    for k := 1 step 1 until K do
      if abs(phi[k]−psi[k]) > 0.5 × abs(s[k]) then go to 2
  end;
3: if DELTA ≥ delta then
  begin DELTA := rho × DELTA;
    for k := 1 step 1 until K do s[k] := rho × s[k]; go to 1
  end;
EXIT:
end direct search
```

REMARKS ON:
ALGORITHM 34 [S14]
GAMMA FUNCTION
[M. F. Lipp, *Comm. ACM 4* (Feb. 1961), 106]
ALGORITHM 54 [S14]
GAMMA FUNCTION FOR RANGE 1 TO 2
[John R. Herndon, *Comm. ACM 4* (Apr. 1961), 180]
ALGORITHM 80 [S14]
RECIPROCAL GAMMA FUNCTION OF REAL ARGUMENT
[William Holsten, *Comm. ACM 5* (Mar. 1962), 166]
ALGORITHM 221 [S14]
GAMMA FUNCTION
[Walter Gautschi, *Comm. ACM 7* (Mar. 1964), 143]
ALGORITHM 291 [S14]
LOGARITHM OF GAMMA FUNCTION
[M. C. Pike and I. D. Hill, *Comm. ACM 9* (Sept. 1966), 684]

M. C. Pike and I. D. Hill (Recd. 12 Jan. 1966)
Medical Research Council's Statistical Research Unit,
University College Hospital Medical School,
London, England

Algorithms 34 and 54 both use the same Hastings approximation, accurate to about 7 decimal places. Of these two, Algorithm 54 is to be preferred on grounds of speed.

Algorithm 80 has the following errors:

(1) *RGAM* should be in the parameter list of *RGR*.

(2) The lines

```
  if x = 0 then begin RGR := 0; go to EXIT end
```

and

```
  if x = 1 then begin RGR := 1; go to EXIT end
```

should each be followed either by a semicolon or preferably by an **else**.

(3) The lines

```
  if x = 1 then begin RGR := 1/y; go to EXIT end
```

and

```
  if x < − 1 then begin y := y × x; go to CC end
```

should each be followed by a semicolon.

(4) The lines

```
  BB: if x = −1 then begin RGR := 0; go to EXIT end
```

and

```
  if x > −1 then begin RGR := RGAM(x); go to EXIT end
```

should be separated either by **else** or by a semicolon and this second line needs terminating with a semicolon.

(5) The declarations of **integer** *i* and **real array** *B*[0:13] in *RGAM* are in the wrong place; they should come immediately after
begin real *z*;

With these modifications (and the replacement of the array *B* in *RGAM* by the obvious nested multiplication) Algorithm 80 ran successfully on the ICT Atlas computer with the ICT Atlas ALGOL compiler and gave answers correct to 10 significant digits.

Algorithms 80, 221 and 291 all work to an accuracy of about 10 decimal places and to evaluate the gamma function it is therefore on grounds of speed that a choice should be made between them. Algorithms 80 and 221 take virtually the same amount of computing time, being twice as fast as 291 at $x = 1$, but this advantage decreases steadily with increasing x so that at $x = 7$ the speeds are about equal and then from this point on 291 is faster—taking only about a third of the time at $x = 25$ and about a tenth of the time at $x = 78$. These timings include taking the exponential of *loggamma.*

For many applications a ratio of gamma functions is required (e.g. binomial coefficients, incomplete beta function ratio) and the use of algorithm 291 allows such a ratio to be calculated for much larger arguments without overflow difficulties.

CERTIFICATION OF: ALGORITHM 41 [F3] EVALUATION OF DETERMINANT [Josef G. Solomon, *Comm. ACM 4* (Apr. 1961), 171] ALGORITHM 269 [F3] DETERMINANT EVALUATION [Jaroslav Pfann and Josef Straka, *Comm. ACM 8* (Nov. 1965), 668]

A. Bergson (Recd. 4 Jan. 1966 and 4 Apr. 1966)
Computing Lab., Sunderland Technical College, Sunderland, Co. Durham, England

Algorithms 41 and 269 were coded in 803 ALGOL and run on a National-Elliott 803 (with automatic floating-point unit).

The following changes were made:
(i) **value** *n*; was added to both Algorithms;
(ii) In Algorithm 269, since procedure *EQUILIBRATE* is only called once, it was not written as a procedure, but actually written into the **procedure** *determinant* body.

The following times were recorded for determinants of order *N* (excluding input and output), using the same driver program and data.

N	T_1 *Algorithm 41*	T_2 *Algorithm 269*
	(minutes)	
10	0.87	0.78
15	2.77	2.18
20	6.47	4.78
25	12.47	8.99
30	21.37	14.98

From a plot of $\ln(T_1)$ against $\ln(N)$ it was found that

$$T_1 = 0.00104N^{2.92}.$$

Similarly,

$$T_2 = 0.00153N^{2.70}.$$

From a plot of T_1 against T_2, it was found that Algorithm 269 was 30.8 percent faster than Algorithm 41, but Algorithm 41 required less storage.

CERTIFICATION OF ALGORITHM 251 [E4] FUNCTION MINIMISATION [M. Wells, *Comm. ACM 8* (Mar. 1965), 169]

R. Fletcher (Recd. 9 Aug. 1965 and 24 Mar. 1966)
Electronic Computing Lab., U. of Leeds, England

Two points need correcting concerning the procedure *FLEPOMIN.*

(i) When the method has converged, either or both of the vectors **s** and **g** can become zero, hence also the scalars *sg* and *ghg*, causing division by zero when updating the matrix *h*.

(ii) The part of the procedure connected with the linear search along **s** does not make use of the fact that the identifier *h* (η in the Appendix to the source paper Fletcher and Powell [1]) tends to 1 as the process converges. This knowledge must be included to achieve the rapid convergence obtained by Fletcher and Powell. However, the particular choice of η given there can also be insufficient when its optimum value would be much greater than 1 (as happens for example in the minimization of $f(\mathbf{x}) = [\mathbf{H}(\mathbf{x}-\mathbf{1})]^2$ where **1** is the vector $(1, 1, \cdots, 1)$ and **H** is a segment of the Hilbert matrix, from an initial approximation $\mathbf{x} = (0, 0, \cdots, 0)$).

An alternative approach is to estimate η by using its value at the previous iteration, increasing or decreasing its value by some constant factor when appropriate (I have arbitrarily used 4). This approach removes the need for the estimate *est* of the minimum value of $f(x)$.

The appropriate changes to be made are thus:
(i) omit *est* as a formal parameter,
(ii) include amongst the **real** identifiers at the head of the procedure body the following:
step, ita, fa, fb, ga, gb, w, z, lambda
(iii) replace the statements from the label
start of minimisation
to the end of the program by the following:

```
start of minimisation:
  conv := true;  step := 1;
  funct(n,x,f,g);
  for count := 1, count +1 while oldf > f do
  begin
    for i := 1 step 1 until n do
    begin sigma[i] := x[i];  gamma[i] := g[i];
      s[i] := -up dot(h,g,i)
end preservation of x,g and
    formation of s;
search along s:
    fb := f;  gb := dot (g,s);
    if gb ≥ 0 then go to exit;
    oldf := f;  ita := step;
    comment a change of ita × s is made in x and the function
      is examined. ita is determined from its value at the previous
      iteration (step) and is increased or decreased by 4 where
      necessary. It should tend to 1 at the minimum;
extrapolate:  fa := fb;  ga := gb;
    for i := 1 step 1 until n do x[i] := x[i] +ita × s[i];
    funct (n,x,f,g);
    fb := f;  gb := dot(g,s);
    if gb <0 ∧ fb < fa then
    begin ita := 4 × ita;  step := 4 × step;  go to extrapolate
end;
interpolate:  z := 3 × (fa-fb)/ita + ga + gb;
    w := sqrt (z↑2-ga×gb);
    lambda := ita × (gb+w-z)/(gb-ga+2×w);
    for i := 1 step 1 until n do x[i] := x[i] - lambda × s[i];
    funct (n,x,f,g);
    if f > fa ∨ f > fb then
    begin step := step/4;
      if fb < fa then
```

```
        begin for i := 1 step 1 until n do x[i] := x[i] + lambda ×
          s[i];  f := fb
        end else
        begin gb := dot(g,s);
          if gb < 0 ∧ count > n ∧ step <₁₀-6 then go to exit;
          fb := f;  ita := ita − lambda;
          go to interpolate
        end;
skip: end of search along s;
      for i := 1 step 1 until n do
      begin sigma [i] := x [i] − sigma [i];
        gamma[i] := g[i] − gamma[i]
      end;
      sg := dot(sigma,gamma);
      if count ≥ n then
      begin if sqrt (dot(s,s)) < eps ∧ sqrt(dot(sigma,sigma)) <eps
        then go to finish
      end;
      for i := 1 step 1 until n do s[i] := up dot (h,gamma,i);
      ghg := dot(s, gamma);
      k := 1;
      if sg = 0 ∨ ghg = 0 then go to test;
      for i := 1 step 1 until n do for j := i step 1 until n do
      begin h[k] := h[k] + sigma[i] × sigma[j]/sg − s[i] × s[j]/ghg;
        k := k + 1
      end updating of h;
test: if count > limit then go to exit;
    end of loop controlled by count;  go to finish;
exit:conv := false;
finish:
end of FLEPOMIN
```

With these changes the procedure was run successfully on a KDF 9 computer on the first of the test functions used by Fletcher and Powell, and the appropriate rate of convergence was achieved. (The corresponding values in [1, Table 1, col. 4] being 24.200, 3.507, 2.466, 1.223, 0.043, 0.008, 4×10^{-5}). It could well be, however, that these changes may still not prove satisfactory on some functions. In such cases it will most likely be the search for the linear minimum along **s** which will be at fault, and not the method of generating **s**. It should not be necessary to evaluate the function and gradient more than 5 or 6 times per iteration in order to estimate the minimum along **s**, except possibly at the first few iterations.

I am indebted to William N. Nawatani of Dynalectron Corporation, Calif., for pointing out the discrepancies in the rates of convergence, and to the referee for his calculations and comments with regard to the Hilbert Matrix function.

REFERENCE

1. Fletcher, R., and Powell M. J. D. A rapidly convergent descent method for minimization. *Comput. J. 6* (July 1963), 163.

REMARK ON ALGORITHM 256 [C2] MODIFIED GRAEFFE METHOD [A. A. Grau, *Comm. ACM 8* (June 1965), 379]

G. Stern (Recd. 8 Mar. 1965 and 24 Mar. 1965)
University of Bristol Computer Unit, Bristol 8, England

This procedure was tested on an Elliott 503 using the two simplifications noted in the comments on page 380. When the 16th line from the bottom of page 380, first column, was changed to read

$$h1 := aa \uparrow (1/(k-s+1));$$

(as suggested in a private communication from the author) correct results were obtained.

REMARK ON ALGORITHM 266 [G5] PSEUDO-RANDOM NUMBERS [M. C. Pike and I. D. Hill, *Comm. ACM 8* (Oct. 1965), 605]

L. Hansson (Recd. 25 Jan. 1966)
DAEC, Riso, Denmark

As stated in Algorithm 266, that algorithm assumes that integer arithmetic up to 3125 × 67108863 = 209715196875 is available. Since this is frequently not the case, the same algorithm with the constants 125 and 2796203 may be useful. In this case the procedure should read

```
real procedure random (a, b, y);
  real a, b;  integer y;
begin
  y = 125 × y; y := y − (y÷2796203) × 2796203;
  random := y/2796203 × (b−a) + a
end
```

The necessary available integer arithmetic is 125 × 2796203 = 348525375 < 2 ↑ 29. With this procedure body, any start value within the limits 1 to 2796202 inclusive will do.

Seven typical runs of the poker-test gave the results:

start value	*all different*	*1 pair*	*2 pairs*	*3*	*3 + pair*	*4*	*5*
100001	129	199	39	31	2	0	0
1082857	115	206	45	31	2	1	0
724768	120	195	49	32	3	1	0
78363	130	198	36	31	5	0	0
1074985	127	189	44	34	4	2	0
2567517	124	193	50	28	3	2	0
2245723	119	202	49	24	4	1	1

Totals for 7 runs:

	864	1382	312	211	23	7	1

Totals for 100 consecutive runs with first start value 100001:

	12023	20297	4301	2837	358	181	3

REMARK ON ALGORITHM 266 [G5] PSEUDO-RANDOM NUMBERS [M. C. Pike and I. D. Hill, *Comm. ACM 8* (Oct. 1965), 605]

M. C. Pike and I. D. Hill (Recd. 9 Sept. 1965)
Medical Research Council, London, England

Algorithm 266 assumes that integer arithmetic up to 3125 × 67108863 = 209715196875 is available, which is not so on many computers. The difficulty arises in the statements

```
  y := 3125 × y;  y := y − (y÷67108864) × 67108864;
```

They may be replaced by

```
  integer k;
  for k := ⟨for list⟩ do
  begin
    y := k × y;
    y := y − (y÷67108864) × 67108864
  end;
```

where the ⟨for list⟩ may be

125, 25 (requiring integer arithmetic up to less than 2^{33})

25, 25, 5 (requiring integer arithmetic up to less than 2^{31})

or

5, 5, 5, 5, 5 (requiring integer arithmetic up to less than 2^{29})

according to the maximum integer allowable. The first is appropriate for the ICT Atlas. [And also for the IBM 7090, the second for the IBM System/360 . . . Ref.]

Note. There are frequently machine-dependent instructions available which will give the same values as the above statements much more quickly, if speed is of much importance.