The actual choice of n and M for a given precision is left for the individual programmer. Programmed for the GE-225, accurate answers of $\tan^{-1} 1 = \pi/4$ were obtained for double precision in comparison with the known values. The semiiterative method can be programmed in a very short time for any multiple precision and is efficient in comparison with taking more terms of the Taylor's series, taking into account the divisions required for the square root process. Telescoped Taylor's series may be used if desired, but a shorter telescoped Taylor's series can be used if the semiiterative scheme is employed. Telescoping, however, is expected to take more storage for the coefficients than a simple truncated series. Telescoping also discards the advantage of flexibility and applicability to n-tuple precision programs.

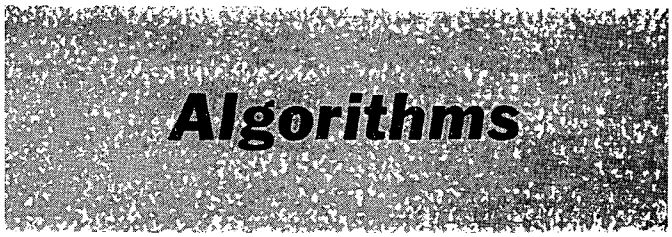
REFERENCES:

1. NATIONAL BUREAU OF STANDARDS. *Table of Circular and Hyperbolic Tangents and Cotangents for Radian Arguments.* Columbia Univ. Press, New York, 1947.
2. NATIONAL BUREAU OF STANDARDS. *Table of Arctan X*, U. S. Government Printing Office, Washington 25, D. C.
3. LANCE, G. N. *Numerical Methods for High-Speed Computers.* Iliffe & Sons, Ltd., London, 1960.
4. BEMER, ROBERT W. Techniques Department: Editor's note. *Comm. ACM, 1*, 9 (Sept. 1958).

WEN-HWA CHU
DONALD R. SAATHOFF
Southwest Research Institute
San Antonio, Texas

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.

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



J. WEGSTEIN, Editor

ALGORITHM 202
GENERATION OF PERMUTATIONS IN LEXICOGRAPHICAL ORDER

MOK-KONG SHEN
Postfach 74, München 34, Germany


```
procedure PERLE (S, N, I, E);
integer array S; integer N; Boolean I; label E;
comment If the array S contains a certain permutation of the
  N digits 1, 2, ··· , N before call, the procedure will replace
  this with the lexicographically next permutation. If initializa-
  tion is required set the Boolean variable I equal true, which
  will be changed automatically to false through the first call,
  otherwise set I equal false. If no further permutation can be
  generated, exit will be made to E. For reference see BIT 2
  (1962), 228-231;
begin integer j, u, w;
if I then begin for j = 1 step 1 until N do S[j] := j;
            I := false; go to Rose
          end;
w := N;
Lilie: if S[w] < S[w-1] then
       begin if w = 2 then go to E;
         w := w - 1; go to Lilie
       end;
u := S[w-1];
for j := N step -1 until w do
begin if S[j] > u then
      begin S[w-1] := S[j];
        S[j] := u; go to Tulpe
      end
end;
Tulpe: for j := 0 step 1 until (N-w-1)/2 + 0.1 do
       begin u := S[N-j];
         S[N-j] := S[w+j]; S[w+j] := u
       end;
Rose:
end PERLE
```

ALGORITHM 203
STEEP1

E. J. WASSCHER
Philips Research Laboratories
N. V. Philips' Gloeilampenfabrieken
Eindhoven-Netherlands


```
procedure STEEP1 (lb, xs, ub, dx, xmin, fmin, n, eps, relax, dxmax,
  eta, psi, pmax, zeta, FUNK);
value dx, n, eps, relax, dxmax, eta, psi, pmax, zeta;
integer n;
real fmin, eps, relax, dxmax, eta, psi, pmax, zeta;
array lb, xs, ub, dx, xmin; real procedure FUNK;
comment STEEP1 is a subroutine to find the minimum of a
  differentiable function of n variables, using the method of
  steepest descent. It mainly consists of three parts: (1) a sub-
```

routine *ATIVE*, for computing the partial derivatives, (2) a subroutine *STEP*, for computing the components of an array *xstep*[1:*n*], which is a new approximation of *xmin*[1:*n*], (3) the compound tail of the procedure body. Both subroutines are only called for once, but by writing the program in this way it is quite easy to change the flow of the program.

Significance of the parameters: *lb*(*i*), *ub*(*i*) are lower and upper bounds for the independent variables. *xs*(*i*) is the starting value for *xmin*(*i*). *xmin*(*i*) is the computed *i*th component of the minimum, *fmin* the value of the function in *xmin*. *n* is the number of variables. *eps* is a small number which is a measure of the desired accuracy—rather of *fmin* than of *xmin*(*i*). *FUNK* (*x*) is the function to be minimized. The other parameters are described in the comments on the three parts mentioned;

```
begin integer j; real alpha, p; array xstep, dfdx, dfpr[1:n];
procedure ATIVE;
begin real beta, gamma, lambda; Boolean A, B;
comment 1. A useful estimate for the derivative is
  (f(x+dx)−f(x−dx))/2dx, where dx should be small, but not so small
  that roundoff noise dominates. This may be achieved by taking
  dx such that eta < |(f(x+dx)−f(x−dx))/f(x)| < 100 eta, where eta is
  a measure for the relative roundoff error. When |f(x)| < 1 it is
  better to replace the denominator by a constant. In the program
  the parameter psi is used for this purpose. The components
  dx(i) are used as a first guess. When the derivative is 0, the
  program enlarges dx until dx > dxmax.
ATIVE computes dfdx[1:n] in xmin. The previously computed
  partial derivatives dfpr[1:n] as well as relax are used for relaxa-
  tion purposes. See comment 3. The Boolean A is used when
  x+dx or x−dx crosses the boundary ub or lb. In that case fmin
  has to be recomputed afterwards. The Boolean B is of a some-
  what complicated nature. It may be seen that dx has the char-
  acter of an own array for ATIVE. In the neighborhood of the
  minimum this may have the following effect: A step in one
  variable is taken such that f(x+dx) becomes equal to f(x−dx).
  Then in the next call for ATIVE dx has to be doubled, etc. By
  using the Boolean B it is possible to keep dx constant near the
  minimum.
  A similar effect may occur in the large. When f(x) tends to a
  constant for x tending to +∞ and −∞, then for |x| large dx
  has to be taken large. It is only possible to make dx smaller in
  the neighborhood of the minimum by reducing dx after each
  call of ATIVE.
  From the last two remarks one may deduce that the first
  guess for dx(i) should be made with considerable care. Tabulat-
  ing the function near the starting point may be very helpful;
begin ATIVE: lambda := 0;
for j := 1 step 1 until n do
begin
large: A := B := false; if xmin[j] + dx[j] > ub[j]
       then begin xmin[j] := ub[j] − dx[j]; A := true end
         else if xmin[j] − dx[j] < lb[j]
       then begin xmin[j] := lb[j] + dx[j]; A := true end;
small: xmin[j] := xmin[j] + dx[j]; alpha := FUNK (xmin);
       xmin[j] := xmin[j] − 2 × dx[j]; beta := FUNK (xmin);
       xmin[j] := xmin[j] + dx[j]; if A then fmin := FUNK
         (xmin);
       A := false;
       if alpha − fmin > 0 ∧ beta − fmin > 0
       then begin B := true; go to comp end;
       gamma := abs((alpha−beta)/(if abs(fmin) < psi then
         psi else fmin));
       if gamma > 100 × eta then
         begin dx[j] := .2 × dx[j]; go to small end;
       if gamma < eta then
         begin dx[j] := 2 × dx[j]; if dx[j] < dxmax then
           go to large else dx[j] := dxmax end
       comp: dfdx[j] := (alpha−beta)/(2 × dx[j]);
         lambda := lambda + dfdx[j] ↑ 2;
         if ¬ B then dx[j] := .5 × dx[j]
       end for; lambda := sqrt (lambda);
       for j := 1 step 1 until n do
         dfdx[j] := dfdx[j]/lambda
       end procedure ATIVE;

procedure STEP;
comment 2. A step is taken in all variables at the same time.
  The order of magnitude of the step in one variable should be
  of the order of magnitude of this variable. To accomplish this
  three weighting factors are given to the partial derivatives:
  1) λ = (Σ_{i=1}^{n} (∂f/∂x_i)^2)^{−½} (see subroutine ATIVE),
  2) |x_i|, or when small, zeta,
  3) a number p, which is put equal to 1 at the beginning of the
     program and which tends to 0 at the minimum.
  After a decrease of the function the step is accepted and p is
  multiplied by 1.5. After an increase p is divided by 2. pmax
  replaces p when p becomes greater than pmax;
begin for j := 1 step 1 until n do
begin alpha := (1−relax) × dfdx[j] + relax × dfpr[j];
  xstep[j] := xmin[j] − p × alpha ×
  (if abs(xmin[j]) < zeta then zeta else abs(xmin[j]));
  dfpr[j] := alpha;
  if xstep[j] > ub[j] then xstep[j] := ub[j]
else if xstep[j] < lb[j] then xstep[j] := lb[j]
end for
end STEP;
comment 3. In the next part—the compound tail—the calls for
  ATIVE and STEP are organized. The values 1.5 and .5 of the
  factors of p are not very important. During the iteration p gets
  an optimal value, which slowly varies. Only at the end p rapidly
  tends to 0. The programme was tested on the functions (y²+1)/(x²+1)
  and ((x−y)²−2)/((x+y)²+2), the latter being the first one except for a rotation
  of the xy-plane over π/4 radians. In the first case a "gutter"
  coincides with the x-axis, while for x > 0 and |y| ≷ 1 ∂f/∂x ≶ 0.
  In the second case, where the gutter is along the line x=y, the
  relaxation is especially interesting, because with relax = 0
  (and pmax=100) the iteration follows the gutter in an unstable
  way. With starting values x=−14 and y=21 from x=y=26 about
  300 steps were taken along the gutter with p about .01. With
  relax = .35 and pmax = .5 we had about 150 steps from x=y=23.
  In the gutter itself relax = .85 gave the best results, but in that
  case the gutter was reached at x=y=63.
    Other parameter values were: zeta = psi = 1, dxmax=100,
  eta = 10⁻⁷ with eps = 10⁻⁸ gave fmin in 10 figures correctly and
  xmin[i] in 4 to 6 figures for various starting values of xs[i];
p := 1;
for j := 1 step 1 until n do
begin xmin[j] := xs[j]; dfpr[j] := 0 end; fmin := FUNK
  (xmin);
deriv: ATIVE;
next: STEP;
      alpha := FUNK (xstep);
if alpha < fmin then
  begin fmin := alpha; p := 1.5 × p;
        if p > pmax then p := pmax;
        for j := 1 step 1 until n do xmin[j] := xstep[j];
        go to deriv end;
p := .5 × p;
if p > eps then go to next;
comment As p has become smaller than eps this is the end of
```

*STEEP*1. The program *ATIVE* takes up rather a lot of computer time by the way it chooses a value for $dx(i)$. A thorough simplification is obtained by taking $dx(i)$ as $10\uparrow -3 \times abs(xmin[i])$, where again $xmin[i]$ may be replaced by *zeta*. Further, at the cost of some loss of accuracy, computing time is saved by taking $\frac{f(x+h)-f(x)}{h}$ as an estimate for the derivative. This program, as far as it differs from *STEEP*1, is described in algorithm 204, *STEEP*2. An interesting compromise between the two methods is obtained by interchanging the computation of *dx* and *dfdx* in *ATIVE* of *STEEP*1 and omitting the iteration on *dx*. This routine *ATIVE*, which has to be used in *STEEP*1, is given by J. G. A. Haubrich in algorithm 205;
end *STEEP*1

ALGORITHM 204
STEEP2
E. J. WASSCHER
Philips Research Laboratories
N. V. Philips' Gloeilampenfabrieken
Eindhoven-Netherlands

```
procedure STEEP2 (lb, xs, ub, dx, xmin, fmin, n, eps, relax
  dxmax, pmax, zeta, FUNK);
value dx, n, eps, relax, dxmax, pmax, zeta;
integer n;
real dx, fmin, eps, relax, dxmax, pmax, zeta;
array lb, xs, ub, xmin; real procedure FUNK;
comment dx should now be taken about 10↑ − 3, dxmax could
  be taken equal to 1. As the program is equal to STEEP1 after
  the declaration of the procedure ATIVE, the ALGOL description
  is cut off there;
begin integer j; real alpha, p;
  array xstep, dfdx, dfpr [1:n];
procedure ATIVE;
begin real beta, lambda; lambda := 0;
  for j := 1 step 1 until n do
begin alpha := dx × (if abs(xmin[j]) < dxmax
  then dxmax else abs (xmin[j]));
  if xmin[j] + alpha > ub[j] then alpha := −alpha;
  xmin[j] := xmin[j] + alpha; beta := FUNK (xmin);
  xmin[j] := xmin[j] − alpha;
  dfdx[j] := (beta − fmin)/alpha;
  lambda := lambda + dfdx[j] ↑2
end for; lambda := sqrt (lambda);
  for j := 1 step 1 until n do dfdx[j] := dfdx[j]/lambda;
end procedure ATIVE
```

ALGORITHM 205
ATIVE
J. G. A. HAUBRICH
Philips Research Laboratories
N. V. Philips' Gloeilampenfabrieken
Eindhoven-Netherlands

```
procedure ATIVE;
begin real beta, lambda; Boolean A;
comment This routine may replace ATIVE in STEEP1. The
  significance of eta has slightly changed;
lambda := 0;
for j := 1 step 1 until n do
begin A := false; alpha := dx[j];
if xmin[j] + alpha > ub[j] then
begin xmin[j] := ub[j] − alpha; A := true end
else if xmin[j] − alpha < lb[j] then
begin xmin[j] := lb[j] + alpha; A := true end;
      xmin[j] := xmin[j] + dx[j]; alpha := FUNK(xmin);
      xmin[j] := xmin[j] − 2 × dx[j]; beta := FUNK(xmin);
      xmin[j] := xmin[j] + dx[j]; if A then fmin := FUNK
        (xmin);
      dfdx[j] := (alpha−beta)/(2 × dx[j]);
      lambda := lambda+dfdx[j] ↑2;
if alpha − fmin > 0 ∧ beta − fmin > 0 then go to end;
beta := abs((alpha−beta)/(if abs(fmin)<psi then psi else fmin));
if beta > eta then dx[j] := .3 × dx[j] else
begin dx[j] := × d3x[j]; if dx[j] > dxmax then dx[j]: = dxmax end;
end: end for;
lambda := sqrt (lambda);
for j := 1 step 1 until n do dfdx[j] := dfdx[j]/lambda
end procedure ATIVE
```

ALGORITHM 206
ARCCOSSIN
MISAKO KONDA
Japan Atomic Energy Research Institute, Tokai, Ibaraki, Japan

```
procedure ARCCOSSIN(x) Result:(arccos, arcsin);
value x;
real x, arccos, arcsin;
comment This procedure computes arccos(x) and arcsin(x) for
  −1≦x≦1. The constant 2^−27 depends on the word length and
  relative machine precision, and may be replaced by a variable
  identifier. Alarm is the procedure which messages that x is in-
  valid.
  The approximation formula used here was coded for MUSA-
  SINO-1 in its own language at the Electrical Communication
  Laboratory Tokyo. This algorithm was translated into FAP and
  successfully ran on an IBM 7090;
begin real A, x1, x2, a; integer r;
  if abs(x) > 1
  then go to Alarm
  else if abs(x) > 2↑ (−27)
    then go to L1
    else begin arccos := 1.5707963; go to L3
      end;
L1: if x = 1
    then begin arccos := 0; go to L3
      end
    else if x = − 1
      then begin arccos := 3.1415926; go to L3
        end
      else begin A := 0; x1 := x;
          for r := 0 step 1 until 26 do
          begin if x1<0
            then begin a := 1; x2 := 1−2 × x1 ↑2 end
            else begin a := 0; x2 := 2 × x1 ↑2 − 1 end;
            A := A + a × 2 ↑(−r−1);
            x1 := x2
          end;
        arccos := 3.1415926 × A;
        end;
L3: arcsin := 1.570963 − arccos;
    end ARCCOSSIN
```

CERTIFICATION OF ALGORITHM 41
EVALUATION OF DETERMINANT [Josef G. Solomon, RCA Digital Computation and Simulation Group, Moorestown, N. J.]

BRUCE H. FREED
Dartmouth College, Hanover, N. H.

When Algorithm 41 was translated into SCALP for running on the LGP-30, the following corrections were found necessary:

1. In the "*y*" loop after "*B[Count,y] := Temp*" and before the "**end**" insert
   "*Temp := C[Count+1,y];*
   C[Count +1,y] := C[Count,y];
   C[Count,y] := Temp"
2. "Sign" is an ALGOL word when uncapitalized. However, many systems (if not all) do not recognize the difference between small and capital letters. For this reason "*Sign*" was changed to "*ssign*" for the LGP-30 run (and in the revision which follows later).

The following addition might be made in the specification as a concession to efficiency: "**value** *A,n;*".

The following changes might be made to make the Algorithm less wordy:

1. for "*Ssign := 1; Product := 1;*"
   put "*Ssign := Product := 1;*"
2. for "**begin** *B[i,j] := A[i,j]; C[i,j] := A[i,j]* **end**;"
   put "*B[i,j] := C[i,j] := A[i,j];*"
3. for "**begin** *B[i,j] := B[i,j] − Factor × C[r,j]* **end end**;"
   put "*B[i,j] := B[i,j] − Factor × C[r,j]* **end**;"

The above corrections and changes were made and the program was run with the correct results, as follows:

$$A = \begin{pmatrix} 10.96597 & 35.10765 & 96.72356 \\ 2.35765 & -84.11256 & .87932 \\ 18.24689 & 22.13579 & 1.11123 \end{pmatrix}$$

Determinant = $.1527313_{10}06$

Hand calculation on a desk calculator gives the value of the determinant for the above matrix as 152,731.3600.

$$A = \begin{pmatrix} 1.0 & 3.0 & 3.0 & 1.0 \\ 1.0 & 4.0 & 6.0 & 4.0 \\ 1.0 & 5.0 & 10.0 & 10.0 \\ 1.0 & 6.0 & 15.0 & 20.0 \end{pmatrix} \quad \text{Determinant} = .9999999_{10}+00$$

The above matrix, being a finite segment of Pascal's triangle, has determinant equal to 1.000000000.

$$A = \begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 5.0 & 9.0 & 2.0 \\ 7.0 & 5.0 & 4.0 \end{pmatrix} \quad \text{Determinant} = .0000000_{10}+00$$

This is, of course, exactly correct.

Finally, one major change can be made which does away with several instructions and reduces variable storage requirements by n^2. This change is the complete removal of matrix *C* from the program. It is extraneous.

The revised Algorithm was translated into SCALP and run on the LGP-30 with exactly the same results as above.

The revised Algorithm 41 follows.

ALGORITHM 41, REVISION
EVALUATION OF DETERMINANT [Josef G. Solomon, RCA Digital Computation and Simulation Group, Moorestown, N. J.]

BRUCE H. FREED
Dartmouth College, Hanover, N. H.

```
real procedure determinant (a,n);
real array a; integer n; value a,n;
comment This procedure evaluates a determinant by triangularization;
begin real product, factor, temp;
array b[1:n,1:n];
integer count, ssign, i, j, r, y;
ssign := product := 1;
for i := 1 step 1 until n do
for j := 1 step 1 until n do
b[i,j] := a[i,j];
for r := 1 step 1 until n−1 do
begin count := r−1;
zerocheck: if b[r,r] ≠ 0 then go to resume;
if count < n−1 then count := count +1 else go to zero;
for y := r step 1 until n do
begin temp := b[count+1,y];
  b[count+1,y] := b[count,y];
  b[count,y] := temp end;
ssign := −ssign;
go to zerocheck;
zero: determinant := 0; go to return;
resume: for i := r+1 step 1 until n do
begin factor := b[i,r]/b[r,r];
for j := r+1 step 1 until n do
b[i,j] := b[i,j] − factor × b[r,j] end end;
for i := 1 step 1 until n do
product := product × b[i,i];
determinant := ssign × product;
return: end
```

CERTIFICATION OF ALGORITHM 45
INTEREST [Peter Z. Ingerman, *Comm. ACM* Apr. 1961 and Oct. 1960]

CARL B. WRIGHT
Dartmouth College, Hanover, N. H.

INTEREST was translated into Dartmouth College Computation Center's "Self Contained ALGOL Processor" for the Royal-McBee LGP-30. When using SCALP, memory capacity is severely limited and thus it was necessary to run this program in two blocks. Block I ended with the computation of I, and Block II started with the "*newm*" loop. After making the changes listed below, test problems using up to three interest rates and up to 18 time periods were used with the following results:

Loan	*Periods*	*Interest Rates*	*Payments*	*Final Balance**	*Tolerance*
$100.00	1	0.05	$105.00	$0.00	$0.25
1800.00	10	0.03	211.01	0.05	4.50
875.65	8	0.08 to 500.00			
		0.05 over 500.00	139.78	−1.49	2.19
14750.00	18	0.06 to 5000.00			
		0.05 to 10,000.00			
		0.04 over 10,000.00	1201.70	10.35	36.88

* Hand calculation.

It is noted that in each case the final balance is within the prescribed tolerance (0.0025 of the loan).

In the following corrections bracketed subscripts replace ordinary subscripts and exponentiation is represented by ↑ rather than superscript.

The following corrections should be made in the Note on Interest in the October, 1960, issue of *Comm. ACM*:

1. Definition of *B[n]*: Replace "*minimum*" by "*maximum*". Replace "*j[n]*" by "*j[n−1]*".
2. Define *B[k+1] ≡ L.*
3. Definition of *K[n]*: Replace "*B[n]*" by "*B[n+1]*".

The following corrections were found necessary in the procedure:

1. The upper limit of the vector B is $k+1$, not k. It is not necessary to change the upper limit of the I-vector. (See correction 4 below.)

2. D, E, F, u, v were not declared and must be declared as **real**.

3. In the **array** declaration replace "$M[1:k]$" by "$M[1:k+1]$".

4. As j approaches 0, i approaches 1 and lim $(h/S) = 1/t$. Thus for $j[k+1] = 0$, $i[k+1] = 1$, and $M[k+1] = L/t$. Thus after

$M[p] := L \times (h[p,t]/S[p,t])$ **end**;

insert

$M[k+1] := L/t$; $B[k+1] := L$;

5. In the conditional statement following computation of $b[p]$, replace "$>$" by "$\geqq$".

6. In same conditional statement, next line, "$mb := bp$" should read "$mb := b[p]$".

7. $D := 1$; $E := F := 0$;
newm: for $p := 1$ **step 1 until** k **do**

should be changed to

newm: $D := 1$; $E := F := 0$;
for $p := 1$ **step** 1 **until** k **do**

8. **begin** *get F*: $F := (D+m-E)/(1+i[q])$;
if $B[q+1] \geqq F$ **then** $D := F$ **else** $q := q + 1$;
if $D \neq F$ **go to** *get F* **end**;

should be changed to read as follows:

begin *get F*: $F := (D+m)/i[q]$;
if $B[q+1] \geqq F$ **then** $D := F$ **else**
begin if $q < k$ **then** $q := q + 1$ **else** $D := F$ **end**;
if $D \neq F$ **then go to** *get F* **end**;

Note that the "then" in the last line was omitted from the original procedure.

9. In the "*redo*" loop insert a semicolon after the statement $T[ib] := T[ib] + T[p] - b[p]$;

10. In the "*redo*" loop, next line, omit the second "*end*".

11. In the "*redo*" loop,

$p := k$ **end**;

should be changed to

$p := k$ **end end**;

REMARK ON ALGORITHM 129 MINIFUN

MINIFUN [V. W. Whitley, *Comm. ACM*, Nov. 1962]

E. J. Wasscher
Philips Research Laboratories
N. V. Philips' Gloeilampenfabrieken
Eindhoven-Netherlands

Some errors found in Algorithm 129 *MINIFUN* [*Comm. ACM*, Nov. 1962] are given below.

In addition, the way "steepest descent" is used to compute the minimum of a function of n variables is not entirely satisfactory. The method for computing first derivatives may be improved in two ways:

1. Instead of computing $\frac{f(x+h)-f(x)}{h}$ it is better to take $\frac{f(x+h)-f(x-h)}{2h}$. As $f(x-h)$ has been computed by *MINIFUN* this does not give rise to extra computations.

2. In *MINIFUN* the choice of h seems rather deliberate. Indeed, h is taken as $.2 \times (xub-x1b)$, where xub and $x1b$ are variable bounds of x. In the beginning of the program these bounds are put equal to the fixed bounds $b1$ and ub; afterwards in the iteration process they should tend towards each other, and in the limit they provide the minimum. So especially when a good approximation to the minimum is unknown, $b1$ and ub have to be taken well apart from each other, which means that h is rather large. At the limit, however, h is very small. It is better to take h in such a way that the nominator $f(x+h)-f(x-h)$ attains an appropriate value.

As the method used by *MINIFUN* is the Newton-Raphson method applied to the first derivatives, convergence is not always secured—especially since first and second partial derivatives are estimated with numerical methods.

It should be noted that the test on end of program is not correct. For a further possible decrease of the function one has not to look in the direction of the coordinate axes but in the direction of the steepest descent.

Algol descriptions of some "steepest descent" programs which were written in the symbolic code of the Philips computer Pascal [cf. H. J. Heijn and J. C. Selman, *IRE Trans. EC10* (June 1961), 175–183] are given in Algorithms 203, 204 and 205.

Corrections of *MINIFUN*:

Printing errors: The line below label *nustep* should read:

begin if *abs*(*dmax*) <*abs* (*dxmin* [*j*]) **then**

The label 1 *bdchk* should be *lbdchk*

In **comment** *MINIFUN*: $k1=2$: a new minimum has not been found.

The **label** nustep should be placed before the statement: $dmax := dxmin[j]$; The declaration of *xmin* should be removed from the blockhead of the procedure body. The 2-dimensional arrays $x[1:n, 1:4]$ and $g[1:n, 1:4]$ can be replaced by a **real** x and a 1-dimensional **array** $g[1:4]$ respectively.

An improvement could be the insertion of the statement

$k1 := 1$;

just before the **label** nustep.

I am having considerable trouble with the obviously important part played by the **array** *wnew*, although it does not change after being set in the first statement of the program. Furthermore it seems to me that *wnew* plays a double rôle: first the component $wnew[k]$ is the value of $xt[k]$ before an iteration on $xt[k]$. But then one should insert another statement after **label** *nustep*: $wnew[k] := xt[k]$; Secondly $wnew[k]$ is to be understood as half the distance between upper and lower bound $t1[k]$ and $b1[k]$, which is only true when $b1[k] = 0$.

Convergence of $de1x[j]$ to 0 is only achieved when $x1b[k]$ and $xub[k]$ are tending towards each other. This indicates that $wnew[k]$ should go to 0 too. (See statements after **label** *stnubds*.)

The following modifications could remove these objections (starting with the line above **label** *restart*):

```
          if ft < fmin then go to check else xt[k] := wnew[k];
restart:  if xt[k] < wnew[k] then go to lbdchk;
          if xt[k] = wnew[k] then go to stnubds;
          if xt[k] < t1[k] then go to nupbds;
          xt[k] := 0.5 × (wnew[k] + t1[k]);
nupbds:   xub[k] := t1[k]; x1b[k] := 2 × xt[k] - t1[k]; go to
            newde1;
stnubds:  x1b[k] := xt[k] - 0.5 × (wnew[k] - x1b[k]);
            xub[k] := xt[k] + 0.5 × (wnew[k] - x1b[k]); (etc.)
lbdchk:   if xt[k] = b1[k] then xt[k] := 0.5 × (wnew[k] + b1[k]);
            x1b[k] := b1[k]; xub[k] := 2 × xt[k] - b1[k]; go to
            newde1; (etc.)
```

REMARK ON ALGORITHM 157
FOURIER SERIES APPROXIMATION [C. J. Mifsud, *Comm ACM*, Mar. 1963]

Richard George*
Argonne National Laboratory, Argonne, Ill.

This algorithm was written in FAP language for the 32-K IBM 704. It was tested on a sawtooth curve, and the sawtooth was recreated by summing the expansion up through the $2N + 1$ constants, with excellent results.

* Work supported by the United States Atomic Energy Commission.

The arrays S, C and u are never referenced with a variable subscript. For a saving of time, I suggest that simple variables be used instead.

By declaring one additional real variable, one can bring the phrase

$$2/(2 \times N + 1)$$

outside of the **for** loops, because N does not change through the procedure. This results in a saving of $4N+2$ mult-ops.

CERTIFICATION OF ALGORITHM 158 EXPONENTIATION OF SERIES [H. E. Fettis, *Comm. ACM*, Mar. 1963]

J. Dennis Lawrence
Lawrence Radiation Laboratory, Livermore, Calif.

This procedure was translated into Fortran and run on the Remington-Rand Larc Computer. Three changes are necessary.

(1) The last line of the comment should read

for the natural logarithm of $f(x)$;

(2) The third line from the end should read

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

(This line was given correctly in algorithm 134.)

(3) The second line from the end apparently should read

$$B[i] := A[i] := (S/i);$$

for the case $P = 0$ only. Probably the best way to incorporate this is by making two changes:

(a) Change the **if** clause to read

if $P = 0$ **then** $R := 1$ **else** $R := P$; $B[1] := R \times A[1]$;

(b) Change the second line from the end to read

$$B[i] := R \times A[i] + (S/i);$$

A large number of examples were run quite successfully; the following give representative samples.

(1) $(1+2x+3x^2+0.5x^3)^2 = 1+4x+10x^2+13x^3+11x^4+3x^5+0.25x^6$ (using $A[4] := A[5] := A[6] := 0$).

(2) Setting $P := 1$ gives $B[i] := A[i]$.

(3) Let $f(x) = e^x = 1 + \sum_{i=1}^{n} \frac{1}{i!} x^i$ and let $P = ln2 = .693147181$.

Then $g(x) = 2^x = 1 + \sum_{i=1}^{n} \frac{(ln2)^i}{i!} x^i$. (See Table 1.)

(4) Let $f(x)=e^x$ and $P=-1$. Then $g(x)=e^{-x}$. For $P=0$, apparently the constant term of $g(x)$ should be zero instead of one.

TABLE 1

	$A[i]$	$B[i]$
1	1.000000000	0.693147181
2	0.500000000	0.240226507
3	0.166666667	0.055504109
4	0.041666667	0.009618129
5	0.008333333	0.001333356
6	0.001388889	0.000154035
7	0.000198413	0.000015253
8	0.000024802	0.000001322
9	0.000002756	0.000000102
10	0.000000276	0.000000007

(5) Let $f(x)=e^x$ and $P=0$. Then $g(x)=x$.

(6) Let $f(x)=\sum_{i=0}^{n} x^n$ and $P=0$. Then $g(x)=ln(1-x^n)-ln(1-x)=\sum_{i=1}^{n} \frac{1}{i} x^i$. (See Table 2.)

TABLE 2

	$A[i]$	$B[i]$
1	1.0	1.000000000
2	1.0	0.500000000
3	1.0	0.333333340
4	1.0	0.250000000
5	1.0	0.200000000
6	1.0	0.166666670
7	1.0	0.142857140
8	1.0	0.125000000
9	1.0	0.111111110
10	1.0	0.100000000
11	1.0	0.090909100
12	1.0	0.083333330
13	1.0	0.076923080
14	1.0	0.071428580
15	1.0	0.066666660

CERTIFICATION OF ALGORITHM 163 MODIFIED HANKEL FUNCTION [Henry E. Fettis, *Comm. ACM*, Apr. 1963]

Henry C. Thacher, Jr.*
Argonne National Laboratory, Argonne, Ill.

Since this algorithm is a function declaration, the procedure declaration should be:

real procedure $EXPK(D, X, E)$; $\cdots$

Otherwise, no syntactical errors were noticed.

The body of the procedure was translated and run on the LGP-30 computer, using the Dartmouth Scalp system. Results for $E = 0.0001$, $X = 0.1(0.1)1.0$, $P = 0$, 0.3333333, 0.6666667 and 1.000000 agreed with values tabulated in Jahnke-Emde-Losch to the 3-4D given in the tables, except for errors discovered in the table of $2/\pi K_{2/3}(x)$.

With $X = 0$, the program ended in floating-point overflow. The algorithm itself, or the call of the procedure, should include a test to insure that the variable is greater than *eps*, where *eps* is chosen to prevent exceeding machine capacity.

The algorithm was found to be excessively slow. Times on the LGP-30 were of the order of 6 minutes. A considerable saving in time could be realized by improving the quadrature formula, currently the simple midpoint formula, repeated completely for each iteration. A more effective method would be a modified Romberg algorithm. A procedure based on the latter approach is being developed in this division.

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

TABLE A

n	$X[n]$	$V[n]$	$B[n]$	$B[n-1]$	$B[n-2]$	$B[n-3]$	$B[n-4]$	$B[n-5]$
1	5.0	148.4132	148.4132					
2	5.0	148.4132	148.4132	148.4132				
3	6.0	403.4288	403.4287	255.0155	106.6023			
4	6.0	403.4288	403.4287	403.4287	148.4132	41.81091		
5	5.0	74.20658	148.4132	255.0155	148.4132	41.81091	9.415191	
6	6.0	201.7144	403.4287	255.0155	148.4132	53.30115	11.49023	2.075043

The forward differences lie along the top diagonal.
Use of these results with *BNEWT* and with *FNEWT* gave the following results, for $N = 6$.

z	*BNEWT*			*FNEWT*		
	P	D	E	R	D	E
5.000000	148.4132	148.4132	$.4567298 \times 10^{-4}$	148.4132	148.4132	$.7420658 \times 10^{-5}$
5.500000	244.6973	244.6924	$.4173722 \times 10^{-4}$	244.6973	244.6924	$.3078276 \times 10^{-4}$
6.000000	403.4287	403.4287	$.2017143 \times 10^{-4}$	403.4287	403.4287	$.7441404 \times 10^{-4}$

CERTIFICATION OF ALGORITHM 167
CALCULATION OF CONFLUENT DIVIDED DIFFERENCES [W. Kahan and I. Farkas, *Comm. ACM*, Apr. 1963]

CERTIFICATION OF ALGORITHM 168
NEWTON INTERPOLATION WITH BACKWARD DIVIDED DIFFERENCES [W. Kahan and I. Farkas, *Comm. ACM*, Apr. 1963]

CERTIFICATION OF ALGORITHM 169
NEWTON INTERPOLATION WITH FORWARD DIVIDED DIFFERENCES [W. Kahan and I. Farkas, *Comm. ACM*, Apr. 1963.]

HENRY C. THACHER, JR.*
Argonne National Laboratory, Argonne, Ill.

The bodies of these procedures were tested on the LGP-30 computer using the Dartmouth SCALP compiler. Compilation and execution revealed no syntactical or mathematical errors.

It is to be noted that, although with Algorithm 169, reducing the value of N from that used to generate F leads to an interpolation polynomial based on fewer points, this is not true for Algorithm 168. This flexibility could be supplied by adding an additional formal parameter, *deg*, say, to the procedure, and by making the **for** statement read:

"**for** i := N − *deg* **step** 1 **until** N **do** ··· "

The logic of the error estimate in Algorithms 168 and 169 is not entirely clear. However, it appears that the estimate can be adjusted for different precision of arithmetic by adjusting the constant $3_{10}-8$ appropriately. For the SCALP arithmetic, this constant was changed to $1_{10}-7$.

The algorithms were tested on the examples given by Milne-Thomson [*The Calculus of Finite Differences*, p. 4, Macmillan, 1951] and by Milne [*Numerical Calculus*, p. 204, Princeton, 1949]. In both examples, Algorithm 167 reproduced the divided difference table, and both Algorithms 168 and 169 reproduced the input values. As a check of the calculation of confluent divided differences, values of the exponential function of its first two derivatives at $x = 5.0$ and 6.0 were used. The difference table shown in Table A was obtained.

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

REMARK ON ALGORITHM 166
MONTECARLO INVERSE [R. D. Rodman, *Comm. ACM*, Apr. 1963]

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

The algorithm contained two errors:
(1) The line which reads
start: $p := (n-1)/n \times n$;
should read
start: $p := (n-1)/n \uparrow 2$;
(2) The line which reads
start2: walk := (**random**/p) + 1;
should read
start2: *walk*: = **entier** ((**random**/p) + 1);

After making the preceding corrections, procedure montecarlo was transliterated into EXTENDED ALGOL and run successfully on the Burroughs B-5000. Convergence occurred in all cases where the matrix satisfied the conditions set down in the comment statement of the algorithm. It was found that convergence was quickest and the routine most practical for matrices with eigenvalues small relative to one.

DATES TO REMEMBER

FJCC	Las Vegas	Nov. 12–14, 1963
SJCC	Washington	Apr. 21–23, 1964
ACM	Philadelphia	Aug. 25–28, 1964
IFIP	New York	May 22–24, 1965