based" machines) has a machine-language instruction named (MOVE and EDIT, which permits a considerable variety of editing features at hardware speeds. This suggests that a moderate programming modification could enable the full power of the EDIT instruction to be added to the FORTRAN language for such a computer. The author has, in fact, done this for Version 4 of 1410 FORTRAN. From the user's standpoint, the extension takes the form of a new specification type in the FORMAT statement, allowing a "pictorial", reminiscent of the COBOL "picture." This pictorial (which is practically identical with the corresponding 1410 machine language specification) is effective only with the "F" and "I" output modes.

As an example, we can write in a 1410 FORTRAN program:

```
WRITE OUTPUT TAPE 6, 100, A, B, K
100 FORMAT (F(XX$0.XX), F6.2, I(XX.XX))
```

This results in the following 1410 treatment:

	Internal	*Specification*	*1410 Edit Word*	*Output*
A	100.0	F(XX$0.XX)	bb$0.bb	$100.00
B	100.0	F6.2	None	100.00
K	100	I(XX.XX)	bb.bb	1.00

where the b's represent blanks. We have substituted X's for the blanks in the source format as a documentation aid. For further examples of 1410 hardware editing see page 38 of the IBM 1410 Reference Manual, Form No. A22-1407-2.

To create this feature in the 1410 FORTRAN all formats are examined at execution time for edit-type specifications. When an edit-type specification is encountered, it is converted to a standard edit word and placed in the I/O buffer. After the data word has been arranged for printing it is edited into this standard edit word. All checking features of the 1410 IO COMMON package are utilized. Fixed-point variables are edited directly from location 0500. Converted *f*-fields are moved to a new area by the standard IO COMMON package from which they are edited. The standard features of the original 1410 IO Common package are not disturbed.

This editing ability in 1410 FORTRAN allows business data processing problems to be compiled without losing either the facilities of the hardware edit feature or the programming advantages of FORTRAN. The author will be glad to supply the listing of this modification upon request.

Thanks are due F. J. Balint and E. B. Weinberger for suggestions leading to this work and to Dr. Weinberger for a number of programming suggestions.

JOHN E. FEDAKO
Gulf Research & Development Company
Pittsburgh, Pennsylvania

LETTER TO THE EDITOR

On the Communications Index

Dear Editor:

I have found the "Index to the Communications of the ACM, Volumes 1–5" the March issue of *Communications* extremely useful. I am sure that other members have found the index equally useful and I would like to add my appreciation to Mr. Youden of NBS for a job well done.

Perhaps all ACM members have not noticed that the index may be easily detached from the issue by simply removing the binding staples. Because of the method of binding the index into the issue this results in a conveniently folded section which carries its own staples.

C. L. MCCARTY, JR.
Editor, Techniques Department
Communications of the ACM

J. WEGSTEIN, Editor

ALGORITHM 173
ASSIGN

OTOMAR HÁJEK
Research Institute of Mathematical Machines, Prague, Czechoslovakia

```
procedure assign (a) the value of : (b) with dimension : (dim)
  indices : (ind) bounds : (low, up) tracer : (j);
value dim; integer dim, ind, low, up, j;
comment This procedure uses Jensen's device (cf. ALGOL
  Report, procedure Innerproduct) twice: the a, b may depend on
  ind and also ind, low, up may depend on j;
begin
  j := dim;
  for ind := low step 1 until up do
    if dim > 1
    then
      begin
        assign (a, b, dim−1, ind, low, up, j);
        j := dim
      end
    else a := b
end assign;
comment The obvious use of "assign" is in assigning the value
  of one array to another. The point here is that one procedure
  declaration serves for all the dimensions used. In fact, the
  dimension may even be a variable: thus a procedure essentially
  identical with "assign" was used by the author in implementing
  the recursive own process in an ALGOL compiler.
    However, in addition to this, "assign" can have further
  functions, as illustrated below. The activation assign (a, (if
  i=1 then false else a) ∨ b[i,i], 1, i, 1, n, j) will calculate the
  join-trace of a Boolean 2-dimensional array b.
  assign (a[i1,i2], (if i3=1 then 0 else a[i1,i2]) + b[i1,i3] × c[i3,i2],
    3, ij, 1, if j = 1 then n else if j = 2 then m else p, j)
  will assign to a the matrix product of b, c. It may be noticed that,
  more generally, "assign" will perform all the tensor operations,
  e.g. tensor multiplication, alternation, etc.
```

ALGORITHM 174
A POSTERIORI BOUNDS ON A ZERO OF A POLYNOMIAL*

ALLAN GIBB
University of Alberta, Calgary, Alberta, Canada

```
comment The procedures below make use of Algorithm 61,
  Procedures for Range Arithmetic [Comm. ACM 4 (1961)]. It is
  assumed that the procedures below and the range arithmetic
  procedures are contained in an outer block and, therefore, that
  the procedures are available as required. Together the proce-
  dures make possible an attempt to determine absolute bounds
```

* These procedures were developed under Office of Naval Research Contract Nonr-225(37) at Stanford University. The author wishes to thank Professor George E. Forsythe for assistance with this work.

```
on a zero of a polynomial given an initial estimate of the zero.
The procedures below are given for the complex case but may
readily be adapted for the real case;
procedure RngPlyC (N, A, Z, P);
comment RngPlyC finds bounds [P1, P2] + i[P3, P4] on the
value of an nth degree polynomial Σ_{k=0}^{n} {[a_{4k+1}, a_{4k+2}]
+ i[a_{4k+3}, a_{4k+4}]}z^k with complex range coefficients for a com-
plex range argument z = [Z1, Z2] + i[Z3, Z4];
integer N; array A, Z, P;
begin integer K, J; array X, Y[1 : 4];
P[1] := P[2] := P[3] := P[4] := 0;
for K := 4 × N step −4 until 0 do
  begin for J := 1 step 1 until 4 do X[J] := A[K+J];
  RNGMPYC (P[1], P[2], P[3], P[4], Z[1], Z[2], Z[3], Z[4], Y[1],
           Y[2], Y[3], Y[4]);
  RNGSUMC (Y[1], Y[2], Y[3], Y[4], X[1], X[2], X[3], X[4],
           P[1], P[2], P[3], P[4])
  end
end;
procedure RngAbsC (A, C);
comment RngAbsC produces the range absolute value [C1, C2]
of the complex range number [A1, A2] + i[A3, A4];
array A, C;
begin array B[1 : 4];
RANGESQR (A[1], A[2], B[1], B[2]);
RANGESQR (A[3], A[4], B[3], B[4]);
RANGESUM (B[1], B[2], B[3], B[4], C[1], C[2]);
C[1] := sqrt(C[1]);
C[2] := sqrt(C[2]);
comment It is assumed that the accuracy of the sqrt routine
used is known and that the maximum error in sqrt(C) is ± K
× CORRECTION(C). K is to be replaced below by its appro-
priate numerical value;
C[1] := C[1] − K × CORRECTION (C[1]);
C[2] := C[2] + K × CORRECTION (C[2])
end;
procedure BndZrPlyC (N, ZOR, ZOJ, A, W,);
integer N; real ZOR, ZOJ; array A, W;
comment BndZrPlyC attempts to determine bounds [W1, W2]
+ i[W3, W4] on a zero of an N-th degree polynomial in z with
complex range coefficients. It is assumed that an estimate
ZO = ZOR + iZOJ of the zero is available. The following
theorem is used. Assume f is regular at z_0 with f'(z_0) ≠ 0. Let
h_0 = −f(z_0)/f'(z_0), let Δ be the region | z − z_0 | ≦ r | h_0 |, and
assume that f is regular in Δ. If, for some r > 0, | f'(z) | ≧ (1/r).
| f'(z_0) | for all z ∈ Δ then Δ contains a zero of f (see [1], pp. 29–31);
begin integer I, J; array B[1:4×N], E, F, FP, D[1:4], AF,
AFP, G[1:2];
real RH, RHS, NL, NR, R, RNL, RNR;
for I := 1 step 1 until N do
  begin J := 4 × I;
  RANGEMPY (I, I, A[J+1], A[J+2], B[J−3], B[J−2]);
  RANGEMPY (I, I, A[J+3], A[J+4], B[J−1], B[J])
  end;
E[1] := E[2] := ZOR; E[3] := E[4] := ZOJ;
RngPlyC(N, A, E, F);
RngAbsC(F, AF);
RngPlyC(N−1, B, E, FP);
RngAbsC(FP, AFP);
RANGEDVD(AF[1], AF[2], AFP[1], AFP[2], NL, NR);
R := 2;
1: RANGEMPY(R, R, NR, NR, RNL, RNR);
RANGESUM(ZOR, ZOR, −RNR, RNR, W[1], W[2]);
RANGESUM(ZOJ, ZOJ, −RNR, RNR, W[3], W[4]);
comment We have replaced the disk of the theorem by a square;
RngRlyC(N−1, B, W, D);
RngAbsC(D, G);
if G[1] = 0 then go to failure1;
comment failure1 and failure2 are non-local labels;
RANGEDVD(AFP[2], AFP[2], R, R, RH, RHS);
if G[1] < RHS then
  begin R := 2 × R;
  if R > 1024 then go to failure2;
  go to 1
  end
end

comment The following procedure may replace BndZrPlyC
above;
procedure BndZrPlyC2 (N, ZOR, ZOJ, A, W);
integer N; array A, W; real ZOR, ZOJ;
comment BndZrPlyC2 is similar to BndZrPlyC above. The
theorem used here follows. If, in the disk | z − z_0 | ≦ 2 | h_0 | we
have | f''(z) | ≦ | f'(z_0) |/(2 | h_0 |), then there is a unique zero in
the disk (see [2, pp. 43-50];
begin integer I, J; array B[1:4×N], C[1:4×N−4], F, D, P,
S[1:4], X, T, Q, Y[1:2]; real V, VP, R, RL;
for I := 1 step 1 until N do
  begin J := 4 × I;
  RANGEMPY(I, I, A[J+1], A[J+2], B[J−3], B[J−2]);
  RANGEMPY(I, I, A[J+3], A[J+4], B[J−1], B[J])
  end;
for I := 1 step 1 until N − 1 do
  begin J := 4 × I;
  RANGEMPY(I, I, B[J+1], B[J+2], C[J−3], C[J−2]);
  RANGEMPY(I, I, B[J+3], B[J+4], C[J−1], C[J])
  end;
D[1] := D[2] := ZOR;
D[3] := D[4] := ZOJ;
RngPlyC(N, A, D, F);
RngPlyC(N−1, B, D, P);
RngAbsC(F, T);
RngAbsC(P, X);
if X[1] = 0 then go to failure1;
comment failure1 and failure2 are non-local labels;
RANGEDVD(T[1], T[2], X[1], X[2], Q[1], Q[2]);
RANGEMPY(2, 2, Q[2], Q[2], RL, R);
RNGSUMC(−R, R, −R, R, ZOR, ZOR, ZOJ, ZOJ, W[1], W[2],
  W[3], W[4]);
RngPlyC(N − 2, C, W, S);
RngAbsC(S, Y);
RANGEDVD(X[1], X[1], R, R, V, VP);
if Y[2] > V then go to failure2
end
```


References:

1. GIBB, ALLAN. ALGOL procedures for range arithmetic. Tech. Report No. 15, Appl. Math. and Statistics Laboratories, Stanford University (1961).
2. OSTROWSKI, A. M. *Solution of equations and systems of equations*. Academic Press, New York, 1960.

ALGORITHM 175
SHUTTLE SORT

C. J. SHAW AND T. N. TRIMBLE

System Development Corporation, Santa Monica, Calif.


```
procedure shuttle sort (m, Temporary, N);
value m; integer m; array N[1:m];
comment This procedure sorts the list of numbers N[1] through
N[m] into numeric order, by exchanging out-of-order number
pairs. The procedure is simple, requires only Temporary as
extra storage, and is quite fast for short lists (say 25 numbers)
and fairly fast for slightly longer lists (say 100 numbers). For
```

still longer lists, though, other methods are much swifter. The actual parameters for *Temporary* and *N* should, of course, be similar in type;

```
begin integer i, j;
for i := 1 step 1 until m − 1 do
  begin
  for j := i step −1 until 1 do
    begin
    if N[j] ≤ N[j+1] then go to Test;
Exchange: Temporary := N[j]; N[j] := N[j+1];
          N[j+1] := Temporary; end of j loop;
Test: end of i loop
end shuttle sort
```

ALGORITHM 176
LEAST SQUARES SURFACE FIT

T. D. Arthurs
The Boeing Company, Transport Division, Renton, Wash.

```
procedure SURFIT (F, z, W, m, n) answers: (a, e, rms);
integer m, n; real rms; array F, z, W, e;
procedure Invert, sqrt;
comment Given a set of m ordinates and the corresponding
  values of n prescribed general functions, (f_i), of one or more
  linearly independent variables, this procedure fits the points,
  in the least squares sense, with a function of the form a_1f_1 + a_2f_2
  + . . . + a_nf_n where a_i are the unknown coefficients. Also com-
  puted are the vectors of residuals (e_j) and their lengths (rms).
  Provision is made for weighting the data points. Essentially, the
  matrix equation F^T WFa = F^T Wz is solved, where a is the vector
  of unknowns, W is an m × m diagonal matrix of data point
  weights, z is the vector of ordinate values and F is the
  m × n matrix of corresponding function values. The availa-
  bility of a procedure Invert, which replaces a real matrix with
  its inverse, is assumed;
begin integer i, j, k; real sqsum, g; array G[1:n, 1:n];
  comment G is working space for the inversion procedure;
  sqsum := 0;
  for i := 1 step 1 until n do
  for j := 1 step 1 until n do
  begin G[i, j] := 0;
    for k := 1 step 1 until m do
    G[i, j] := G[i, j] + F[k, i] × F[k, j] × W[k]
  end j;
  Invert (G, n);
  for i := 1 step 1 until n do
  begin a[i] := 0;
    for j := 1 step 1 until m do
    begin g := 0;
      for k := 1 step 1 until n do
      g := g + G[i, k] × F[j, k];
      a[i] := a[i] + g × z[j] × W[j]
    end j
  end i;
  for i := 1 step 1 until m do
  begin e[i] = y[i];
    for j := 1 step 1 until n do
    e[i] := e[i] − a[j] × F[i, j];
    sqsum := sqsum + e[i] ↑ 2
  end i;
  rms := sqrt (sqsum/m)
end SURFIT
```

ALGORITHM 177
LEAST SQUARES SOLUTION WITH CONSTRAINTS

M. J. Synge
The Boeing Company, Transport Division, Renton, Wash.

```
procedure CONLSQ (A, y, w, n, m, r) results: (x) residuals:
  (e, rms);
real rms; integer n, m, r; array A, y, w, x, e; procedure
  abs, SURFIT;
comment This procedure solves an overdetermined set of n
  simultaneous linear equations in m unknowns, Ax = y. The
  first r equations (r≤m) are satisfied exactly and the remaining
  n − r are satisfied as well as possible by the method of least
  squares. Each equation is assigned a weight from the vector w,
  although the first r weights have no relevance. This procedure
  may be used for curve or surface fitting when the approximating
  function or its derivatives are required to have fixed values at a
  number of points;
begin integer i, j, k, ii, ick; integer array ic[1:m];
  array B[1:n−r, 1:m−r]; real Amax;
  for i := 1 step 1 until r do
  begin k := 1; for j := 2 step 1 until m do
    begin if abs (A[i, j]) > abs (A[i, k]) then k := j; end;
    ic[i] := k; Amax := A[i, k]; for j := 1 step 1 until m do
    A[i, j] := A[i, j]/Amax; y[i] := y[i]/Amax;
    for ii := 1 step 1 until r do
    begin if ii = i then go to skip; Amax := A[ii, k];
      for j := 1 step 1 until m do
      A[ii, j] := A[ii, j] − A[i, j] × Amax;
      y[ii] := y[ii] − y[i] × Amax;
skip: end ii
  end i;
  ick := r + 1; for j := 1 step 1 until m do
  begin k := 1;
repeat: if j = ic[k] then go to next;
  k := k + 1; if r ≥ k then go to repeat;
  ic[ick] := j; ick := ick + 1;
next: end k;
  for i := r + 1 step 1 until n do
  begin for k := 1 step 1 until r do
    y[i] := y[i] − y[k] × A[i, ic[k]];
    for j := r + 1 step 1 until m do
    begin B[i, j] := A[i, ic[j]];
      for k := 1 step 1 until r do
      B[i, j] := B[i, j] − A[i, ic[k]] × A[k, ic[j]]
    end j
  end i;
  SURFIT (B, y[r+1:n], w[r+1:n], n − r, m − r, x[r+1:m],
    e[r+1:n], rms);
comment The procedure SURFIT is called to solve the reduced
  set of n − r simultaneous linear equations in m − r unknowns,
  Bx_2 = y_2', which have no constraints;
for j := r + 1 step 1 until m do x[ic[j]] := x[j];
for j := 1 step 1 until r do
begin x[ic[j]] := y[j];
  for i := r + 1 step 1 until m do
  x[ic[j]] := x[ic[j]] − A[j, ic[i]] × x[ic[i]]
end j
end CONLSQ
```

ALGORITHM 178
DIRECT SEARCH

Arthur F. Kaupe, Jr.
Westinghouse Electric Corp., Pittsburgh, Penn.

```
procedure direct search (psi, X, DELTA, rho, delta, S);
```

```
value K, DELTA, rho, delta; integer K; array psi;
  real DELTA, rho, delta; real procedure S;
comment This procedure may be used to locate the minimum
  of the function S of K variables. A discussion of the use of this
  procedure may be found in: Robert Hooke and T. A. Jeeves,
  'Direct Search' Solution of Numerical and Statistical Problems
  [J. ACM 8, 2 (1961), 212-229]. The notation is essentially that
  used in Appendix B of the cited paper. The exceptions being the
  spelling of the Greek letters and the introduction of notation to
  distinguish between the process of calculating a value of S and
  the value itself—thus S(phi) and Sphi. A modified version of this
  procedure acceptable to the BAC compiler for the Burroughs
  205 and 220 computers has been prepared and run successfully;
begin real SS, Spsi, Sphi, theta; array phi [1:K]; integer K, k;
procedure E; for k := 1 step 1 until K do
  begin phi [k] := phi [k] + DELTA; Sphi := S(phi);
  if Sphi < SS then SS := Sphi else
    begin phi [k] := phi [k] - 2 × DELTA; Sphi := S(phi):
    if Sphi < SS then SS := Sphi else phi [k] := phi [k] + DELTA
  end E;
Start: Spsi := S(psi);
1: SS := Spsi;
for k := 1 step 1 until K do phi [k] := psi [k]; E;
if SS < Spsi then begin
  2: for k := 1 step 1 until K do begin
    theta := psi [k];
    psi [k] := phi [k];
    phi [k] := 2 × phi [k] - theta end;
  Spsi := SS; SS := Sphi := S(phi); E;
  if SS < Spsi then go to 2 else go to 1 end;
3: if DELTA ≥ delta then begin DELTA := rho × DELTA;
  go to 1 end end
```

ALGORITHM 179
INCOMPLETE BETA RATIO*

OLIVER G. LUDWIG
Mathematical Laboratory and Department of Theoretical Chemistry, University of Cambridge, England

* Based in part on work done at Carnegie Institute of Technology, Pittsburgh, Pennsylvania and supported by the Petroleum Research Fund of the American Chemical Society and by the National Science Foundation.

```
real procedure incompletebeta (x, p, q, epsilon);
value x, p, q; real x, p, q, epsilon;
begin real finsum, infsum, temp, temp 1, term, term 1, qrecur, index;
  Boolean alter;
comment This procedure evaluates the ratio Bx(p, q)/B1(p, q),
  where Bx(p, q) = ∫0^x t^(p-1)(1-t)^(q-1) dt, with 0 ≦ x ≦ 1 and p, q > 0,
  but not necessarily integers. It assumes the existence of a non-
  local label, alarm, to which control is transferred upon entry to
  the procedure with invalid arguments. Also assumed is a proce-
  dure to evaluate ∫0^∞ t^p e^-t dt which is called factorial(p), (cf. e.g.
  Algorithm 80, March, 1962);
if x > 1 ∨ x < 0 ∨ p ≦ 0 ∨ q ≦ 0 then go to alarm;
if x = 0 ∨ x = 1 then begin incompletebeta := x; go to End end;
comment This part interchanges arguments if necessary to ob-
  tain better convergence in the power series below;
if x ≦ 0.5 then alter := false else
  begin alter := true; temp := p; p := q; q := temp; x :=
    1 - x end;
comment This part recurs on the (effective) q until the power
  series below does not alternate;
finsum := 0; term := 1; temp := 1 - x; qrecur := index := q;
for index := index - 1 while index > 0 do
  begin qrecur := index;
    term := term × (qrecur+1)/(temp×(p+qrecur));
    finsum := finsum + term
  end;
comment This part sums a power series for non-integral effec-
  tive q and yields unity for integer q;
infsum := term := 1; index := 0;
comment In the following statement the convergence criterion
  might well be altered to term > epsilon, since infsum > 1 al-
  ways, thus saving one divide per cycle at the cost, perhaps, of a
  few more cycles;
for index := index + 1 while (term/infsum) > epsilon do
  begin term := term × x × (index-qrecur) × (p+index-1)/
    (index×(x+index)); infsum := infsum + term
  end;
comment This part evaluates most of the necessary factorial
  functions, minimizing the number of entries into the factorial
  procedure;
temp := temp 1 := factorial (qrecur-1);
term := term 1 := factorial (qrecur+p-1);
for index := qrecur step 1 until (q-0.5) do
  begin temp 1 := temp 1 × index;
    term 1 := term 1 × (index+p)
  end;
comment This part combines the partial results into the final
  one;
temp := x ↑ p × (infsum×term/(p×temp)+finsum×term 1×
  (1-x) ↑ q/(q×temp 1))/factorial (p-1);
incompletebeta := if alter then 1-temp else temp;
end: end incompletebeta
```

ALGORITHM 180
ERROR FUNCTION—LARGE X

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

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

```
real procedure erfL(x); value x; real x;
comment This procedure evaluates the error function of real
  argument, erf(x) = (2/√π) ∫0^x e^-u^2 du by the Laplace continued
  fraction for the complementary error function: erf(x) = 1 -
  (1/(1+v/(1+2v/(1+3v/(1+···)))))/(√π x e^x^2) where v = 1/
  (2x^2). Successive even convergents of the continued fraction are
  evaluated, using an algorithm suggested by Maehly, until the
  full accuracy of the arithmetic being used is attained.
    The continued fraction converges for all x > 0. For small x,
  however, convergence may be excessively slow, and overflow
  may occur. In this region, the Taylor series converges satis-
  factorily, and algorithms such as No. 123 are suitable.
    For x ≦ 0, the procedure calls the global procedure alarm.
    The body of this procedure has been checked on the LGP-30
  computer, using the Dartmouth Self Contained Algol Processor.
  The program was used to tabulate erf(x) from 0.9(.1)5.0. The
  maximum error was 2 × 10^-6, which is explainable by roundoff
  errors. The number of convergents calculated ranged from 36
  for x = 0.9 to 2 for x ≧ 3.3. Overflow occurred for x = 0.87;
begin integer m; real B min 2, B min 3, P, R, T, v, v2;
if x ≤ 0 then alarm;
v := x × x;
T := -0.56418958/x/exp(v);
comment The constant 0.56418958 ··· = π^-1/2, and should
  be given to the full accuracy required of the procedure;
v := 0.5/v;
```

```
P = v × T;
v2 := v × v;
T := T + 1;
m := 0;
R := B min 3 := B min 2 := 1;
for m := m + 1 while T ≠ R do
  begin R := T;
  B min 3 := v × (m−1) × B min 3 + B min 2
  T := B min 2;
  B min 2 := v × m × B min 2 + B min 3;
  T := R − P/B min 2/T;
  P := m × (m+1) × v2 × P
  end while;
  erfL := T
end
```

ALGORITHM 181
COMPLEMENTARY ERROR FUNCTION—LARGE X

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

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

```
real procedure erfcL(x); value x; real x;
comment This procedure evaluates the complementary error
  function, erfc(x) = 1 − erf(x) = (2/√π) ∫_x^∞ exp(−u²)du by
  the Laplace continued fraction:
    erfc(x) = (1/(1+v/(1+2v/(1+3v/(1+···)))))/(√π x e^{x²})
  where v = 1/(2x²). Successive even convergents of the continued
  fraction are evaluated, using an algorithm suggested by Maehly,
  until the full accuracy of the arithmetic being used is attained.
    The continued fraction converges for all x > 0. For small x,
  however, convergence may be excessively slow, and overflow
  and round-off accumulation may occur. In this region, the
  Taylor series converges satisfactorily.
    For x ≦ 0, the procedure calls the global procedure alarm.
    The body of this procedure has been checked on the LGP-30
  Computer, using the Dartmouth Self Contained Algol Processor,
  for x = 1.2(0.1)5.0. Results were generally correct to 1 in the
  6th significant digit, although a few errors were as large as 6
  in that digit. The errors are believed to be due to round-off
  only. The number of convergents calculated ranged from 46
  for x = 1.2 to 10 for x = 5.0.
    Overflow occurred for x = 1.183;
begin integer m; real B min 2, B min 3, P, R, T, v, v2;
if x ≤ 0 then alarm;
v := x × x;
T := 0.56418958/x/exp(v);
comment The constant 0.56418958 ··· = π^{−1/2}, and should be
  given to the full accuracy required of the procedure;
v := 0.5/v;
v2 := v × v;
P := v × T;
m := R := 0;
B min 3 := B min 2 := 1;
for m := m + 2 while R ≠ T do
  begin R := T;
  B min 3 := v × (m−1) × B min 3 + B min 2;
  T := B min 2;
  B min 2 := v × m × B min 2 + B min 3;
  T := R − P/B min 2/T;
  P := m × (m+1) × v2 × P
  end while;
  erfc L := T
end
```

ALGORITHM 182
NONRECURSIVE ADAPTIVE INTEGRATION

W. M. MCKEEMAN AND LARRY TESLER
Stanford University, Stanford, Calif.

```
real procedure Simpson(F) limits : (a, b) tolerance : (eps);
real procedure F; real a, b, eps; value a, b, eps;
begin comment A nonrecursive translation of Algorithm 145.
  Note that the device used here can be used to simulate recursion
  for a wide class of algorithms;
  integer lvl;
  switch return := r1, r2, r3;
  real array dx, epsp, x2, x3, F2, F3, F4, Fmp, Fbp,
  est2, est3 [1:30], pval[1:30, 1:3];
  integer array rtrn [1:30];
  real absarea, est, Fa, Fm, Fb, da, sx, est1, sum, F1;
  comment the parameter setup for the initial call;
  lvl := absarea := est := 0; da := b − a;
  Fa := F(a); Fm := 4.0 × F((a+b)/2.0); Fb := F(b);
  recur:
    lvl := lvl + 1; dx[lvl] := da/3.0;
    sx := dx[lvl]/6.0; Fl := 4.0 × F(a+dx[lvl]/2.0);
    x2[lvl] := a + dx[lvl]; F2[lvl] := F(x2[lvl]);
    x3[lvl] := x2[lvl] + dx[lvl]; F3[lvl] := F(x3[lvl]);
    epsp[lvl] := eps; F4[lvl] := 4.0 × F(x3[lvl]+dx[lvl]);
    Fmp[lvl] := Fm; est1 := (Fa+F1+F2[lvl]) × sx;
    Fbp[lvl] := Fb; est2[lvl] := (F2[lvl]+F3[lvl]+Fm) × sx;
                    est3[lvl] := (F3[lvl]+F4[lvl]+Fb) × sx;
    sum := est1 + est2[lvl] + est3[lvl];
    absarea := absarea − abs(est) + abs(est1) + abs(est2[lvl]) +
      abs (est3[lvl]);
    if (abs(est−sum) ≤ epsp[lvl] × absarea) ∨ (lvl≥30) then
    begin comment done on this level;
      up : lvl := lvl − 1;
      pval[lvl, rtrn[lvl]] := sum;
      go to return [rtrn[lvl]]
    end;
    rtrn[lvl] := 1; da := dx[lvl]; Fm := F1;
    Fb := F2[lvl]; eps := epsp[lvl]/1.7; est := est1;
    go to recur; r1:
    rtrn[lvl] := 2; da := dx[lvl]; Fa := F2[lvl];
    Fm := Fmp[lvl]; Fb := F3[lvl]; eps := epsp[lvl]/1.7;
    est := est2[lvl]; a := x2[lvl]; go to recur; r2:
    rtrn[lvl] := 3; da := dx[lvl]; Fa := F3[lvl];
    Fm := F4[lvl]; Fb := Fbp[lvl]; eps := epsp[lvl]/1.7;
    est := est3[lvl]; a := x3[lvl]; go to recur; r3:
    sum := pval[lvl, 1] + pval[lvl, 2] + pval[lvl, 3];
    if lvl > 1 then go to up;
  Simpson := sum
end Simpson
```

ALGORITHM 183
REDUCTION OF A SYMMETRIC BANDMATRIX TO TRIPLE DIAGONAL FORM

H. R. SCHWARZ
Swiss Federal Institute of Technology, Zürich, Switzerland

```
procedure bandred(a, n, m);
  value n, m; integer n, m; array a;
comment bandred reduces a real and symmetric matrix of band
  type (order n, a[i, k]=0 for |i−k|>m) by a sequence of orthog-
  onal similarity transformations to triple diagonal form. The
  procedure represents a generalization of the algorithm m21 by
  H. Rutishauser. Due to symmetry only the upper part of the
  band matrix must be given and these elements are denoted for
```

convenience in the following way: $a[i, 0]$ $(i=1, 2, \cdots, n)$ represents the diagonal element in the ith row, and $a[i, k]$ $(i=1, 2, \cdots, n-k$ and $k=1, 2, \cdots, m)$ represents the generally nonzero element in the ith row and the kth position to the right of the diagonal. After completion of the reduction, the elements of the symmetric triple diagonal matrix are given by $a[i, 0]$ $(i=1, 2, \cdots, n)$ and $a[i, 1]$ $(i=1, 2, \cdots, n-1)$;

```
begin integer r, k, i, j, p, rr; real b, g, c, s, c2, s2, cs, u, v;
  for r := m step -1 until 2 do
  begin
    for k := 1 step 1 until n-r do
    begin
      for j := k step r until n-r do
      begin
        comment This compound statement describes the rotation involving the ith and (i+1)st rows and columns
        in order to reduce either a[j, r] or the off-band element
        g to zero, respectively. This rotation produces a new
        off-band element g (in general different from zero) provided i + r < n;
        if j = k then
        begin if a[j, r] = 0 then go to endk;
          b := -a[j, r-1]/a[j, r]
        end
        else
        begin if g = 0 then go to endk;
          b := -a[j-1, r]/g
        end;
        s := 1/sqrt(1 + b×b); c := b × s;
        c2 := c × c; s2 := s × s; cs := c × s;
        i := j + r - 1;
cross elements:
        u := c2 × a[i, 0] - 2 × cs × a[i, 1] + s2 × a[i + 1, 0];
        v := s2 × a[i, 0] + 2 × cs × a[i, 1] + c2 × a[i+1, 0];
        a[i, 1] := cs × (a[i, 0] - a[i+1, 0]) + (c2-s2) ×a[i, 1];
        a[i, 0] := u; a[i+1, 0] := v;
column rotation:
        for p := j step 1 until i - 1 do
        begin
          u := c × a[p, i-p] - s × a[p, i-p+1];
          a[p, i-p+1] := s × a[p, i-p] + c × a[p, i-p+1];
          a[p, i-p] := u
        end p;
        if j ≠ k then
          a[j-1, r] := c × a[j-1, r] - s × g;
row rotation:
        rr := if r ≤ n - i then r else n - i;
        for p := 2 step 1 until rr do
        begin
          u := c × a[i, p] - s × a[i+1, p-1];
          a[i+1, p-1] := s × a[i, p] + c × a[i+1, p-1];
          a[i, p] := u
        end p;
        if i + r < n then
new g:  begin g := -s × a[i+1, r];
          a[i+1, r] := c × a[i+1, r]
        end
      end j;
endk: end k
  end r
end bandred
```

CERTIFICATION OF ALGORITHM 74
CURVE FITTING WITH CONSTRAINTS [J. E. Peck, *Comm. ACM*, Jan. 62]

KAZUO ISODA
Japan Atomic Energy Research Institute, Tokai, Ibaraki, Japan

Algorithm 74 was hand-compiled into SOAP IIa for the IBM 650 and run successfully with no corrections except the case in which the origin (0, 0) are given as both a constraint and a sample.

CERTIFICATION OF ALGORITHM 123
REAL ERROR FUNCTION, ERF (x) [Martin Crawford and Robert Techo, *Comm. ACM*, Sept. 1962]

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

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

The body of $Erf(x)$ was tested using the Dartmouth SCALP compiler for the LGP-30. For $x = 0(0.01)0.3$, the results agreed with tabulated values to 8 in the 7th decimal place, and for $x = 0.4(0.2)1.6$ the error was less than 1 in the 6th decimal. These results are compatible with the roundoff error in the arithmetic used. The computing time increased rapidly (by a factor of more than 10) as x increased from 0.01 to 1.6.

The following comments should be considered by users of the algorithm:

1. The parameter x should be called by value, both to allow the use of expressions, and also to avoid destruction of the actual parameter.
2. The constant $_{10}-10$ in statement 2 determines the accuracy of the computation. Its value should be adjusted to the arithmetic being used, and the accuracy required. A machine-independent test could be made by substituting **if** $Y - T = Y$ **then** $\cdots$.
3. For large x, the error function is more efficiently calculated from the Laplace continued fraction for $erfc(x)$. Algorithm 180 is based on this method.