



Algorithms

ALGORITHM 61
PROCEDURES FOR RANGE ARITHMETIC
ALLAN GIBB*

University of Alberta, Calgary, Alberta, Canada

begin

procedure RANGESUM (a, b, c, d, e, f);

real a, b, c, d, e, f;

comment The term "range number" was used by P. S. Dwyer, *Linear Computations* (Wiley, 1951). Machine procedures for range arithmetic were developed about 1958 by Ramon Moore, "Automatic Error Analysis in Digital Computation," LMSD Report 48421, 28 Jan. 1959, Lockheed Missiles and Space Division, Palo Alto, California, 59 pp. If $a \leq x \leq b$ and $c \leq y \leq d$, then RANGESUM yields an interval $[e, f]$ such that $e \leq (x + y) \leq f$. Because of machine operation (truncation or rounding) the machine sums $a + c$ and $b + d$ may not provide safe end-points of the output interval. Thus RANGESUM requires a non-local real procedure ADJUSTSUM which will compensate for the machine arithmetic. The body of ADJUSTSUM will be dependent upon the type of machine for which it is written and so is not given here. (An example, however, appears below.) It is assumed that ADJUSTSUM has as parameters real v and w, and integer i, and is accompanied by a non-local real procedure CORRECTION which gives an upper bound to the magnitude of the error involved in the machine representation of a number. The output ADJUSTSUM provides the left end-point of the output interval of RANGESUM when ADJUSTSUM is called with $i = -1$, and the right end-point when called with $i = 1$. The procedures RANGESUB, RANGEMPTY, and RANGEDVD provide for the remaining fundamental operations in range arithmetic. RANGESQR gives an interval within which the square of a range number must lie. RNGSUMC, RNGSUBC, RNGMPYC and RNGDVDC provide for range arithmetic with complex range arguments, i.e. the real and imaginary parts are range numbers;

begin

 e := ADJUSTSUM (a, c, -1);

 f := ADJUSTSUM (b, d, 1)

end RANGESUM;

procedure RANGESUB (a, b, c, d, e, f);

real a, b, c, d, e, f;

comment RANGESUM is a non-local procedure;

begin

 RANGESUM (a, b, -d, -c, e, f)

end RANGESUB;

procedure RANGEMPTY (a, b, c, d, e, f);

real a, b, c, d, e, f;

comment ADJUSTPROD, which appears at the end of this procedure, is analogous to ADJUSTSUM above and is a non-local real procedure. MAX and MIN find the maximum and minimum of a set of real numbers and are non-local;

begin

real v, w;

if a < 0 \wedge c \geq 0 **then**

1: **begin**

 v := c; c := a; a := v; w := d; d := b; b := w

end 1;

if a \geq 0 **then**

2: **begin**
if c \geq 0 **then**
3: **begin**
 e := a \times c; f := b \times d; **go to** 8
end 3;
 e := b \times c;
if d \geq 0 **then**
4: **begin**
 f := b \times d; **go to** 8
end 4;
 f := a \times d; **go to** 8
5: **end** 2;
if b > 0 **then**
6: **begin**
 if d > 0 **then**
begin
 e := MIN(a \times d, b \times c);
 f := MAX(a \times c, b \times d); **go to** 8
end 6;
 e := b \times c; f := a \times c; **go to** 8
end 5;
 f := a \times c;
if d \leq 0 **then**
7: **begin**
 e := b \times d; **go to** 8
end 7;
 e := a \times d;
8: e := ADJUSTPROD (e, -1);
 f := ADJUSTPROD (f, 1)
end RANGEMPTY;
procedure RANGEDVD (a, b, c, d, e, f);
 real a, b, c, d, e, f;
comment If the range divisor includes zero the program exists to a non-local label "zerodvrs". RANGEDVD assumes a non-local real procedure ADJUSTQUOT which is analogous (possibly identical) to ADJUSTPROD;
begin
 if c \leq 0 \wedge d \geq 0 **then go to** zerodvrs;
 if c < 0 **then**
1: **begin**
 if b > 0 **then**
2: **begin**
 e := b/d; **go to** 3
end 2;
 e := b/c;
3: **if** a \geq 0 **then**
4: **begin**
 f := a/c; **go to** 8
end 4;
 f := a/d; **go to** 8
end 1;
if a < 0 **then**
5: **begin**
 e := a/c; **go to** 6
end 5;
 e := a/d;
6: **if** b > 0 **then**
7: **begin**
 f := b/c; **go to** 8
end 7;
 f := b/d;
8: e := ADJUSTQUOT (e, -1); f := ADJUSTQUOT (f, 1)
end RANGEDVD;
procedure RANGESQR (a, b, c, d);
 real a, b, c, d;
comment ADJUSTPROD is a non-local procedure;
begin
 if a < 0 **then**

```

1: begin
  if b < 0 then
2:   begin
    e := b × b;  f := a × a;  go to 3
  end 2;
  e := 0;  m := MAX (-a,b);  f := m × m;  go to 3
end 1;
  e := a × a;  f := b × b;
3:  ADJUSTPROD (e, -1);
  ADJUSTPROD (f, 1)
end RANGESQR;
procedure RNGSUMC (aL, aR, bL, bU, cL, cR, dL, dU, eL, eR, fL, fU);
  real aL, aR, bL, bU, cL, cR, dL, dU, eL, eR, fL, fU;
comment Rangesum is a non-local procedure;
begin
  RANGESUM (aL, aR, cL, cR, eL, eR);
  RANGESUM (bL, bU, dL, dU, fL, fU)
end RNGSUMC;
procedure RNGSUBC (aL, aR, bL, bU, cL, cR, dL, dU, eL, eR, fL, fU);
  real aL, aR, bL, bU, cL, cR, dL, dU, eL, eR, fL, fU;
comment RNGSUMC is a non-local procedure;
begin
  RNGSUMC (aL, aR, bL, bR, -cR, -cL, -dU, -dL, eL, eR, fL, fU)
end RNGSUBC;
procedure RNGMPYC (aL, aR, bL, bU, cL, cR, dL, dU, eL, eR, fL, fU);
  real aL, aR, bL, bU, cL, cR, dL, dU, eL, eR, fL, fU;
comment RANGEMPTY, RANGESUB, and RANGESUM are non-local procedures;
begin
  real L1, R1, L2, R2, L3, R3, L4, R4;
  RANGEMPTY (aL, aR, cL, cR, L1, R1);
  RANGEMPTY (bL, bU, dL, dU, L2, R2);
  RANGESUB (L1, R1, L2, R2, eL, eR);
  RANGEMPTY (aL, aR, dL, dU, L3, R3);
  RANGEMPTY (bL, bU, cL, cR, L4, R4);
  RANGESUM (L3, R3, L4, R4, fL, fU);
end RNGMPYC;
procedure RNGDVDC (aL, aR, bL, bU, cL, cR, dL, dU, eL, eR, fL, fU);
  real aL, aR, bL, bU, cL, cR, dL, dU, eL, eR, fL, fU;
comment RNGMPYC, RANGESQR, RANGESUM, and RANGEDVD are non-local procedures;
begin
  real L1, R1, L2, R2, L3, R3, L4, R4, L5, R5;
  RNGMPYC (aL, aR, bL, bU, cL, cR, -dU, -dL, L1, R1, L2, R2);
  RANGESQR (cL, cR, L3, R3);
  RANGESQR (dL, dU, L4, R4);
  RANGESUM (L3, R3, L4, R4, L5, R5);
  RANGEDVD (L1, R1, L5, R5, eL, eR);
  RANGEDVD (L2, R2, L5, R5, fL, fU)
end RNGDVDC
end

```

EXAMPLE

```

real procedure CORRECTION (p); real p;
comment CORRECTION and the procedures below are intended for use with single-precision normalized floating-point arithmetic for machines in which the mantissa of a floating-point number is expressible to s significant figures, base b. Limitations of the machine or requirements of the user will limit the range of p to  $b^m \leq |p| < b^{n+1}$  for some integers m and n. Appropriate integers must replace s, b, m and n below. Signal is a non-local label. The procedures of the example would be included in the same block as the range procedures above;

```

```

begin
  integer w;
  for w := m step 1 until n do
1: begin
  if (b ↑ w ≤ abs (p))  $\wedge$  (abs (p) < b ↑ (w + 1)) then
2: begin
  CORRECTION := b ↑ (w+1-s); go to exit
  end 2
end 1;
go to signal;
exit: end CORRECTION;
real procedure ADJUSTSUM (w, v, i); integer i;
  real w, v;
comment ADJUSTSUM exemplifies a possible procedure for use with machines which, when operating in floating point addition, simply shift out any lower order digits that may not be used. No attempt is made here to examine the possibility that every digit that is dropped is zero. CORRECTION is a non-local real procedure which gives an upper bound to the magnitude of the error involved in the machine representation of a number;
begin
  real r, cw, cv, cr;
  r := w + v;
  if w = 0  $\vee$  v = 0 then go to 1;
  cw := CORRECTION (w);
  cv := CORRECTION (v);
  cr := CORRECTION (r);
  if cw = cv  $\wedge$  cr ≤ cw then go to 1;
  if sign (i × sign (w) × sign (v) × sign (r)) = -1 then go to 1;
  ADJUSTSUM := r + i × MAX (cw, cv, cr); go to exit;
1: ADJUSTSUM := r;
exit: end ADJUSTSUM;
real procedure ADJUSTPROD (p, i); real p; integer i;
comment ADJUSTPROD is for machines which truncate when lower order digits are dropped. CORRECTION is a non-local real procedure;
begin
  if p × i ≤ 0 then
1: begin
  ADJUSTPROD := p; go to out
  end 1;
  ADJUSTPROD := p + i × CORRECTION (p);
out: end ADJUSTPROD;
comment Although ordinarily rounded arithmetic is preferable to truncated (chopped) arithmetic, for these range procedures truncated arithmetic leads to closer bounds than rounding does.

```

* These procedures were written and tested in the Burroughs 220 version of the ALGOL language in the summer of 1960 at Stanford University. The typing and editorial work were done under Office of Naval Research Contract Nonr-225(37). The author wishes to thank Professor George E. Forsythe for encouraging this work and for assistance with the syntax of ALGOL 60.

ALGORITHM 62 A SET OF ASSOCIATE LEGENDRE POLYNOMIALS OF THE SECOND KIND*

JOHN R. HERNDON
Stanford Research Institute, Menlo Park, California

comment This procedure places a set of values of $Q_{n^m}(x)$ in the array Q[] for values of n from 0 to nmax for a particular value of m and a value of x which is real if ri is 0 and is purely imaginary, ix, otherwise. R[] will contain the set of ratios of successive values of Q. These ratios may be especially valuable when the $Q_{n^m}(x)$ of the smallest size is so small as to underflow the machine representation (e.g. 10^{-60} if 10^{-51} were the smallest representable

number). 9.9×10^{15} is used to represent infinity. Imaginary values of x may not be negative and real values of x may not be smaller than 1.

Values of $Q_{n,m}(x)$ may be calculated easily by hypergeometric series if x is not too small nor $(n - m)$ too large. $Q_{n,m}(x)$ can be computed from an appropriate set of values of $P_{n,m}(x)$ if x is near 1.0 or x is near 0. Loss of significant digits occurs for x as small as 1.1 if n is larger than 10. Loss of significant digits is a major difficulty in using finite polynomial representations also if n is larger than m . However, QLEG has been tested in regions of x and n both large and small;

```

procedure QLEG(m, nmax, x, ri, R, Q); value m, nmax, x, ri;
real m, nmax, x, ri; real array R, Q;
begin real t, i, n, q0, s;
n := 20;
if nmax > 13 then
  n := nmax + 7;
if ri = 0 then
  begin if m = 0 then
    Q[0] := 0.5 × log((x + 1)/(x - 1))
    else
    begin t := -1.0/sqrt(x × x - 1);
    q0 := 0;
    Q[0] := t;
    for i := 1 step 1 until m do
      begin s := (x+x)×(i-1)×t
      ×Q[0]+(3i-i×i-2)×q0;
      q0 := Q[0];
      Q[0] := s end end;
    if x = 1 then
      Q[0] := 9.9 ↑ 45;
    R[n + 1] := x - sqrt(x × x - 1);
    for i := n step -1 until 1 do
      R[i] := (i + m)/((i + i + 1) × x
      +(m - i - 1) × R[i + 1]);
    go to the end;
  if m = 0 then
    begin if x < 0.5 then
      Q[0] := arctan(x) - 1.5707963 else
      Q[0] := - arctan(1/x) end else
    begin t := 1/sqrt(x × x + 1);
    q0 := 0;
    Q[0] := t;
    for i := 2 step 1 until m do
      begin s := (x + x) × (i - 1) × t × Q[0]
      +(3i + i × i - 2) × q0;
      q0 := Q[0];
      Q[0] := s end end;
    R[n + 1] := x - sqrt(x × x + 1);
    for i := n step -1 until 1 do
      R[i] := (i + m)/((i - m + 1) × R[i + 1]
      -(i + i + 1) × x);
    for i := 1 step 2 until nmax do
      R[i] := - R[i];
  the: for i := 1 step 1 until nmax do
    Q[i] := Q[i - 1] × R[i];
  end QLEG;

```

* This procedure was developed in part under the sponsorship of the Air Force Cambridge Research Center.

ALGORITHM 63

PARTITION

C. A. R. HOARE

Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

```

procedure partition (A,M,N,I,J); value M,N;
array A; integer M,N,I,J;

```

comment I and J are output variables, and A is the array (with subscript bounds M:N) which is operated upon by this procedure. Partition takes the value X of a random element of the array A, and rearranges the values of the elements of the array in such a way that there exist integers I and J with the following properties:

```

M ≤ J < I ≤ N provided M < N
A[R] ≤ X for M ≤ R ≤ J
A[R] = X for J < R < I
A[R] ≥ X for I ≤ R ≤ N

```

The procedure uses an integer procedure random (M,N) which chooses equiprobably a random integer F between M and N, and also a procedure exchange, which exchanges the values of its two parameters;

```

begin real X; integer F;
F := random (M,N); X := A[F];
I := M; J := N;
up: for I := I step 1 until N do
  if X < A[I] then go to down;
  I := N;
down: for J := J step -1 until M do
  if A[J] < X then go to change;
  J := M;
change: if I < J then begin exchange (A[I], A[J]);
  I := I + 1; J := J - 1;
  go to up
  end
else if I < F then begin exchange (A[I], A[F]);
  I := I + 1
  end
else if F < J then begin exchange (A[F], A[J]);
  J := J - 1
  end;
end partition

```

ALGORITHM 64

QUICKSORT

C. A. R. HOARE

Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

```

procedure quicksort (A,M,N); value M,N;
array A; integer M,N;

```

comment Quicksort is a very fast and convenient method of sorting an array in the random-access store of a computer. The entire contents of the store may be sorted, since no extra space is required. The average number of comparisons made is $2(M-N) \ln (N-M)$, and the average number of exchanges is one sixth this amount. Suitable refinements of this method will be desirable for its implementation on any actual computer;

```

begin integer I,J;
if M < N then begin partition (A,M,N,I,J);
  quicksort (A,M,J);
  quicksort (A, I, N)
end
end quicksort

```

ALGORITHM 65

FIND

C. A. R. HOARE

Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

```

procedure find (A,M,N,K); value M,N,K;
array A; integer M,N,K;

```

comment Find will assign to A [K] the value which it would have if the array A [M:N] had been sorted. The array A will be partly sorted, and subsequent entries will be faster than the first;

```

begin      integer I,J;
  if M < N then begin partition (A, M, N, I, J);
    if K≤I then find (A,M,I,K)
    else if J≤K then find (A,J,N,K)
  end
end      find

```

ALGORITHM 66

INVRS

JOHN CAFFREY

Director of Research, Palo Alto Unified School District,
Palo Alto, California

```

procedure Invrs (t) size : (n);  value n;  real array t;  integer;
comment Inverts a positive definite symmetric matrix t, of
order n, by a simplified variant of the square root method. Re-
places the  $n(n+1)/2$  diagonal and superdiagonal elements of t
with elements of  $t^{-1}$ , leaving subdiagonal elements unchanged.
Advantages: only n temporary storage registers are required, no
identity matrix is used, no square roots are computed, only n
divisions are performed, and, as n becomes large, the number of
multiplications approaches  $n^3/2$ ;
begin integer i, j, s;  real array v[1:n-1];  real y, pivot;
for s := 0 step 1 until n-1 do
begin pivot := 1.0/t[1,1];
begin pivot := 1.0/t[1,1];
  comment If t[1,1] ≤ 0, t is not positive defi-
  nite;
  for i := 2 step 1 until n do v[i-1] := t[1, i];
  for i := 1 step 1 until n-1 do
    begin t[i,n] := y := -v[i] × pivot;
    for j := i step 1 until n-1 do
      t[i, j] := t[i + 1, j + 1] + v[j] × y
    end;
  t[n,n] := -pivot
end;
  comment At this point, elements of  $t^{-1}$  occupy
  the original array space but with signs reversed,
  and the following statements effect a simple re-
  flection;
  for i := 1 step 1 until n do
  for j := i step 1 until n do t[i,j] := -t[i,j]
end Invrs

```

ALGORITHM 67

CRAM

JOHN CAFFREY

Director of Research, Palo Alto Unified School District,
Palo Alto, California

```

procedure CRAM (n, r, a) Result: (f);  value n, r;  integer
  n, r;  real array a, f;
comment CRAM stores, via an unspecified input procedure
READ, the diagonal and superdiagonal elements of a square sym-
metric matrix e, of order n, as a pseudo-array of dimension
1:n(n + 1)/2. READ (u) puts one number into u. Elements e[i, j]
are addressable as a[c + j], where c =  $(2n - i)(i - 1)/2$  and c[i + 1]
may be found as c[i] + n - i. Since c[1] = 0, it is simpler to develop
a table of the c[i] by recursion, as shown in the sequence labelled
"table". Further manipulation of the elements so stored is illus-
trated by premultiplying a rectangular matrix f, of order n, r, by
the matrix e, replacing the elements of f with the new values, re-
quiring a temporary storage array v of dimension 1:n;

```

```

begin integer i, j, k, m;  real array v[1:n];  real s;
  integer array c[1:n];
table: j := -n;  k := n + 1;  for i := 1 step 1 until n do
begin
  j := j + k - i;  c[i] := j end;
load: for i := 1 step 1 until n do
begin for j := i step 1 until n do READ (v[j]);  m :=
c[i];
  for k := i step 1 until n do a[m + k] := v[k] end;
premult: for j := 1 step 1 until r do
begin for i := 1 step 1 until n do
begin s := 0.0;
  for k := 1 step 1 until i do
begin m := c[k];  s := s + a[m + i]
  × f[k, j] end;
  for k := i + 1 step 1 until n do
    s := s + a[m + k] × f[k, j];  v[i] = s
end;
  for k := 1 step 1 until n do f[k, j] = v[k]
end;
end CRAM

```

REMARK ON ALGORITHM 53

Nth ROOTS OF A COMPLEX NUMBER (John R.
Herndon, *Comm. ACM* 4, Apr. 1961)

C. W. NESTOR, JR.

Oak Ridge National Laboratory, Oak Ridge, Tennessee

A considerable saving of machine time for $N \geq 3$ would result from the use of the recursion formulas for the sine and cosine in place of an entry into a sine-cosine subroutine in the do loop associated with the Nth roots of a complex number. That is, one could use

$$\begin{aligned} \sin(n+1)\theta &= \sin n\theta \cos\theta + \cos n\theta \sin\theta \\ \cos(n+1)\theta &= \cos n\theta \cos\theta - \sin n\theta \sin\theta, \end{aligned}$$

at the cost of some additional storage.

We have found this procedure to be very efficient in problems dealing with Fourier analysis, as suggested by G. Goerzel in chapter 24 of *Mathematical Methods for Digital Computers*.

Contributions to this department must be in the form stated in the Algorithms Department policy statement (*Communications*, February, 1960) except that ALGOL 60 notation should be used (see *Communications*, May, 1960). Contributions should be sent in duplicate to J. H. Wegstein, Computation Laboratory, National Bureau of Standards, Washington 25, D. C. Algorithms should be in the Publication form of ALGOL 60 and written in a style patterned after the most recent algorithms appearing in this department.

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.

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.