A MACHINE-INDEPENDENT ALGOL PROCEDURE FOR ACCURATE FLOATING-POINT SUMMATION

Michael A. Malcolm
Stanford University

Prepared for:
Office of Naval Research
National Science Foundation

June 1973
A MACHINE-INDEPENDENT ALGOL PROCEDURE
FOR ACCURATE FLOATING-POINT SUMMATION

by

Michael A. Malcolm

STAN-CS-73-374
June 1973

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

DISTRIBUTION STATEMENT A
Approved for public release; Distribution Unlimited
A MACHINE-INDEPENDENT ALGOL PROCEDURE
FOR ACCURATE FLOATING-POINT SUMMATION

Michael A. Malcolm

June 1973

This work was supported by the Office of Naval Research, Contract N00014-67-A-0112-0029, and NSF Contract GJ29988X.
procedure sum (x, n, m, result, fail);
value n, m; integer n, m; real result;
array x; label fail;
begin comment This Algol 60 procedure is an implementation of the floating-point summation technique described in Malcolm (1971). This implementation is machine-independent in the sense that it will work on any computer having a floating-point number system $F$ characterized as follows: Each number $x \in F$ has a radix-$\beta$ $t$-digit fraction where $t \geq 1$. The radix $\beta$ can be any positive integer greater than 1. The exponent $e$ is assumed to lie in the range $$b \leq e \leq B,$$
where $b < 0$ and $B > t$. Each nonzero $x \in F$ has the representation
$$x = \pm d_1 d_2 \ldots d_t \beta^e,$$
where $d_1, \ldots, d_t$ are integers satisfying
$$0 \leq d_i \leq \beta - 1, \ (i=1, \ldots, t).$$
The number 0 is contained in $F$, but no assumption is made about its representation. All floating-point operations (e.g., addition and multiplication) are assumed to result in either 0 or a normalized floating-point number contained in $F$. The machine may do either proper rounding or chopping (truncation). (Note that this definition of $F$ excludes machines using extra-length accumulators for intermediate arithmetic. However, this algorithm is seldom needed on such machines.)

The parameters $p$ and $t$ of $F$ are automatically computed at execution time by a technique described in Malcolm (1972). Since the range of the floating-point exponent cannot be determined automatically,
the input parameter $m$ is used for allocating the set of accumulators used by the algorithm.

Provided no overflow or underflow occurs, and none of the $x[i]$ are larger than $10^m$, or smaller than $10^{-m}$, in magnitude, and $n \leq \beta^{t+1}/16$, where $t = \lfloor t/2 \rfloor$, then

$$n \approx \sum_{i=1}^{\infty} x[i]$$

is returned with nearly full-precision accuracy. The bound on the relative error is given by Theorem 2 in Malcolm (1971) as

$$\frac{f(t+1)}{\log_\beta 16} \beta^{1-t}.$$ 

If any of the $x[i]$ are larger than $10^m$ or smaller than $10^{-m}$, then the error exit fail is taken.;

**Boolean** rnd; **integer** beta, t, t2, mu, L, U;

**procedure** ENVEON (beta, t, rnd);

**Boolean** rnd; **integer** beta, t;

begin comment This procedure is an Algol 60 translation of the (first) Fortran subroutine ENVRON given in Malcolm (1972);

**real** a, b, e;

for $e := 2, 2 \times e$ while $(a+1)-a=1$ do $a := e$;

for $e := 2, 2 \times e$ while $a+b=a$ do $b := e$;

beta := $(a+b)-a$; rnd := $a+(beta-1) > a$; $t := 0$;

for $a := 1$, beta $x a$ while $(a+1)-a=1$ do $t := t+1$

end ENVEON;

ENVEON (beta, t, rnd); $t2 := t+2$; $mu := ln(16)/ln(beta)$;

$U := \text{entier}(m \times ln(10)/(ln(beta) \times mu)) + 1$;

$L := \text{entier}((-m \times ln(10)/ln(beta) - t2)/mu)$;
In the notation of Malcolm (1971), \( l = t_2 \) is the padding that each of the numbers added to the accumulators will have. Each of the \( x[i] \) will be split into two halves (i.e. \( q=2 \)) having the last \( t_2 \) digits equal zero. The variable \( nu \) above is used for \( v \) defined in Equation (2) of Malcolm (1971). The value for \( nu \) computed above is rather arbitrary and was chosen to make \( nu \) sufficiently smaller than \( t_2 \). The variables \( U \) and \( L \) are the upper and lower bounds on the indices of the accumulators which are declared in the following block. They are chosen to allow the \( x[i] \) to range from \( 10^{-m} \) to \( 10^m \) in magnitude. In slightly different notation, they are

\[
U = \lfloor m/(v \log_{10} g) \rfloor , \\
L = \lfloor (-m/\log_{10} g - \lfloor t/2 \rfloor)/v \rfloor ;
\]

```plaintext
begin array accumulators[L:U]; integer ex;
    real xL, xH;

integer procedure e(x);
value x; real x;
begin comment This procedure computes the exponent \( e \) of the floating-point number \( x \) .
    real y, q; integer ex;
    x := abs(x); ex := 0; for y := 1,q
    while \( x > y \) do begin ex := ex+1; q := beta*y; end;
    for y := q, y/beta while \( x < y \) do ex := ex-1;
    e := ex
end e;
```
comment initialize the array of accumulators;
for i:=L step 1 until U do accumulators[i] := 0;
comment accumulate the nonzero x[i]s;
for i:=1 step 1 until n do if x[i]#0 then
begin ex := e(x[i]);
   if entier(ex/mu)>U V ex-t2<x then go to fail;
   comment Now the x[i] is split into a high- and low-order part, xH and xL. The method used here is to add the proper power of β to x[i] to force it to preshift t2 digits to the right and then either truncate or round the last t2 significant digits. Then the same power of β is subtracted to cause a post normalization which brings in t2 trailing zero digits. The resulting high-order part of x[i] is then subtracted from x[i] to produce the low-order part such that the sum of the high- and low-order parts is exactly equal to x[i]. This method of splitting a floating-point number into two halves is similar to that given by Dekker (1971). ;
   xH := beta(ex-l-t2); xH := (xH+x[i]) - xH;
   xL := x[i] - xH;
   comment xH and xL can now be added to the appropriate accumulators. ;
   accumulators[entier(ex/mu)] := xH;
   accumulators[entier((ex-t2)/mu)] := xL
end; comment Now sum the accumulators in decreasing order. ;
result := 0;
for i:=L step -1 until U do
result := result + accumulators[i]

end

end  sum

5
References

