ALGORITHMS TO REVEAL PROPERTIES OF FLOATING-POINT ARITHMETIC

BY

MICHAEL A. MALCOLM

STAN-CS-71-211

MARCH, 1971

COMPUTER SCIENCE DEPARTMENT

School of Humanities and Sciences

STANFORD UNIVERSITY

ALGORITHMS TO REVEAL PROPERTIES OF FLOATING-POINT ARITHMETIC

by

Michael A. Malcolm

Abstract

Two algorithms are presented in the form of Fortran subroutines. Each subroutine computes the radix and number of digits of the floating-point numbers and whether rounding or chopping is done by the machine on which it is run. The methods are shown to work on any "reasonable" floating-point computer.

Keywords: Floating-Point Arithmetic

High-Level Languages

Philosophy of Language Design

This research was sponsored by the Office of Naval Research under grant number N00014-67-A-0112-0029, The National Science Foundation under grant number NSF GJ 408 and the Atomic Energy Commission under grant number AT (04-3) 326, PA 30.

1. Introduction

A large percentage of the practical numerical algorithms in use today require some information about the actual floating-point number system on which they are implemented. For example, a zero finder must use some sort of "machine epsilon" to determine when it has found a "zero". An iterative improvement subroutine in a linear system solver must stop iterating when the corrections no longer affect the answer. Some other algorithms which require this type of information are: eigenvalue-eigenvector routines, ordinary differential equation solvers, function minimizers, etc.

Usually this information is supplied to the algorithm in one of two ways : It is either passed as a parameter or it is imbedded in one or more constants. In the first case, each user is faced with the problem of understanding another confusing parameter in the calling sequence and he is likely to not know what to use for a "machine epsilon". In the second case, the "magic" numbers in a program are often not understood by people reading or translating the program. When the subroutine is moved from one machine to another, these numbers are seldom changed to reflect the properties of the new machine -- even when the author of the original program provides explicit comments in the program telling what the constants mean and how to change them.

Since one of the original motivations for designing and implementing high-level languages was to allow a program written for one machine to run on other machines, I think that this problem reflects a serious shortcoming of languages like Fortran and Algol. Such languages should provide standard functions which return information pertinent to the machine.

However, given these shortcomings, it is reasonable to ask: How can information about the number system of a computer be determined automatically? That is, can a subroutine written in Fortran compute this information?

The Fortran subroutine given in the next section partially solves the problem for a large class of floating-point systems. Another Fortran subroutine, presented in Section 3, solves the same problem for a more restricted set of floating-point systems.

2. The Fortran Subroutine ENVRON

For the remainder of this paper, a floating-point number system F will be characterized as follows: Each number will have a radix β and a t-digit mantissa where $t \geq 1$. Usually β is 2 ; 8 , 10 or 16 , but β will only be restricted to be a positive integer greater than 1 . The exponent e is assumed to lie in the range

$$m \leq e \leq M ,$$

where $m \leq 0$ and $M > t$. Each nonzero $x \epsilon F$ has the representation

$$x = \pm .d_1 d_2 \ldots d_t \cdot \beta^e ,$$

where $d_1, \ldots, d_t$ are integers satisfying

$$0 \leq d_i \leq \beta - 1 , \quad (i = 1, \ldots, t) .$$

The number 0 belongs to F . No assumption is made about the representation of 0 ; however it is usually represented by

$$0 = + .00 \ldots 0 \cdot \beta^m .$$

If $x \neq 0$ and $d_1 \neq 0$, then x is said to be normalized. All floating-point operations (e.g., addition and multiplication) are assumed to result in either 0 or a normalized floating-point number.

The machine will do either proper rounding or chopping (truncation).

The <u>machine epsilon</u> mentioned in the previous section is the smallest positive floating-point number ϵ such that $\epsilon \oplus 1 > 1$, where $\oplus$ denotes floating-point addition. Thus, one could compute ϵ from β and t.

The Fortran subroutine shown in Figure 1 can be called with the Fortran statement

```
CALL ENVRON (IB, IT, IR)
```

If the Fortran program is running on a machine with a floating-point number system of the type just described, then the actual parameters will be returned with the values

$$IB = \beta ,$$

$$IT = t ,$$

$$IR = \begin{cases} 0 , & \text{if the machine does chopping,} \\ 1 , & \text{if the machine does proper rounding.} \end{cases}$$

```
      SUBROUTINE ENVRON(BETA,T,RND)
      INTEGER BETA, T, RND
      RND = 1
      A = 2.
      B = 2.
  100 IF ((A+1.)-A.NE.1.) GO TO 200
      A = 2.*A
      GO TO 100
  200 IF (A+B.NE.A) GO TO 300
      B = 2.*B
      GO TO 200
  300 BETA = (A+B) - A
      IF (A+(BETA-1).EQ.A) RND = 0
      T = 0
      A = 1
  400 T = T+1
      A = A*BETA
      IF ((A+1)-A.EQ.1) GO TO 400
      RETURN
      END
```

Figure 1

Suppose the machine on which ENVRON is executing has the floating-point system F . Then the consecutive integers

$$0,1,2,\ldots,\beta^t$$

can be represented exactly in F . Integers larger than β^t which can be represented exactly are

$$\beta^t+\beta,\ \beta^t+2\beta,\ \beta^t+3\beta,\ \ldots\ \beta^{t+1},\ \beta^{t+1}+\beta^2,\ \ldots$$

Thus, the difference between neighboring floating-point numbers in the interval $[\beta^t,\beta^{t+1}]$ is β . The first part of ENVRON (lines 4 through 8) tests successive powers of 2 until a floating-point number (A) in this interval is found. Lines 9 through 12 add successive powers of 2 to A until the next floating-point number $(A+\beta)$ is found and then β is computed by subtracting these two numbers. To determine whether rounding or chopping is being done (line 13), $\beta-1$ is added to A . Now, since A is in the interval

$$\beta^t \leq A < \beta^{t+1} \quad ,$$

the number t can be computed by

$$t = \lfloor \log_\beta A \rfloor \ .$$

However, possible inaccuracies in computing the logarithm are avoided by determining the power of β required to shift the least significant digit of an integer out of the mantissa. The smallest such exponent is equal to t .

The time required for ENVRON to execute is roughly proportional to $\log_2 \beta$. For any practical application, the execution time is negligible.

It is important to note that the algorithm used in ENVRON does not rely upon the use of guard digits in the floating-point additions.

The author believes this algorithm to be a very efficient way of computing β , t and whether the floating-point system rounds or chops.

3. A Special Algorithm for the Cases β = 2, 4, 8, 10 and 16

After using the technique described in the previous section to determine the number $A \in [\beta^t, \beta^{t+1})$, the following trick can be used . to determine both β and whether rounding or chopping is done:

1. Set B:=A+15 (B is another floating-point representation).
2. If B=A , then β=16 and chopping is done.
3. If B=A+8 , then β=8 and chopping is done.
4. If B=A+10 , then β=10 and chopping is done.
5. If B=A+12 , then β=4 and chopping is done.
6. If B=A+14 , then β=2 and chopping is done.
7. If B=A+16 , then rounding is done and β is either 2, 4, 8 or 16.
8. If B=A+20 , then β=10 and rounding is done.

The case B=A+16 can be resolved by

1. Set B:=A+5.
2. If B=A , then β=16 .
3. If B=A+4 , then β=4 .
4. If B=A+6 , then β=2 .
5. If B=A+8 , then β=8 .

A For-bran subroutine incorporating this idea and using the same name and calling sequence as the subroutine given in Section 2 is shown in Figure 2. Although the code is longer than the version in Figure 1, the execution time for this version is slightly smaller.

```
      SUBROUTINE ENVRON(BETA,T,RND)
      INTEGER BETA, T, RND
C
C  THIS VERSION WORKS FOR MACHINES WITH BASE 2, 4,8, 10 OR 16
C
      RND = 0
      A = 2.
 10   IF ((A+1.)-A.NE.1.) GO TO 20
      A = 2.*A
      GO TO 10
 20   I = IFIX((A+15.)-A) + 1
      GO TO (30,1,1,1,1,1,1,40,1,50,1,60,1,70,1,80,1,1,1,90),I
 1    STOP
 30   BETA = 16
      GO TO 100
 40   BETA = 8
      GO TO 100
 50   BETA = 10
      GO TO 100
 60   BETA=4
      GO TO 100
 70   BETA=2
      GO TO 100
 80   RND = 1
      I = IFIX((A+5.)-A) + 1
      GO TO (82,1,1,1,84,1,86,1,88),I
 82   bETA = 16
      GO TO 100
 84   BETA=4
      GO TO 100
 86   BETA=2
      GO TO 100
 88   BETA=8
      GO TO 100
 go   RND = 1
      BETA = 10
 100  T = 0
      A = 1
 110  T = T+1
      A = A*BETA
      IF ((A+1)-A.EQ.1) GO TO 110
      RETURN
      END
```

Figure2

4. Conclusions

The algorithms given in Figures 1 and 2 will determine certain characteristics of the floating-point number system of any machine currently in use (at least those floating-point machines of which the author is aware). Specifically, the number base, number of digits and whether rounding or chopping is done, can be computed automatically.

Programs and subroutines in general use, such as library routines, should avoid additional parameters in the calling sequence and magic constants in the code by using one of these subroutines for computing the floating-point environment of the current machine. This not only makes the code more readable but the portability of the program is greatly increased. The additional execution time required to call such a routine is insignificant compared to these advantages.

Unfortunately, it is not possible to write a general subroutine to compute upper and lower bounds for the floating-point exponent (m and M). If underflow and overflow conditions were handled in some uniform manner, it would be possible to do so. Some programs make use of the values of m and M. Thus it would be worthwhile for software manufacturers to consider ways of providing such information automatically. Automatic determination of properties of the integer arithmetic system would also be useful. A good universal random number generator could be written if it were possible to automatically determine the magnitude of the largest representable integer.

Other desirable environmental parameters are listed in a paper by Redish and Ward.

5. Acknowledgment

The author would like to thank Professor Cleve Moler for arousing his interest in this problem. Mr. Richard Sites contributed some of the ideas which led to an earlier version of ENVRON. The author would like to thank Professor Robert Floyd for questioning the "optimality" of this earlier version and for a discussion which led to improved versions.

6. Bibliography

Redish, K. A. and Ward, W., "Environment Enquiries for Numerical Analysis", SIGNUM Newsletter **6** (1), January 1971, 10-15.