LISP-based "Big-float" system is not slow

Yasumasa Kanada* and Tateaki Sasaki**

*The Institute of Plasma Physics, University of Nagoya
Chikusaku, Nagoya 464, Japan

**The Institute of Physical and Chemical Research
Wako-shi, Saitama 351, Japan

Abstract

The efficiency of evaluation is investigated on two "big-float" systems, a LISP-interpreter-based system and a FORTRAN-compiler-based system. The results show that the LISP-interpreter-based system is quite efficient. Timing data on basic functions sqrt(x) and exp(x) and basic constants e and pi are also reported.

Key words and phrases: Multiple Precision Arithmetic, LISP, Extended Precision, Floating-Point Arithmetic
CR categories: 5.12

1. Introduction

Multiple-precision floating-point arithmetic (or big-float arithmetic) is one of the most powerful and basic facilities for numeric-algebraic computations, and it should be equipped in the general systems for symbolic and algebraic manipulation[1] (or SAM). Big-float arithmetic was already implemented on some systems as Fateman's system on MACSYMA[2], Pinkert's system on SAC-1[3] and Sasaki's system on REDUCE-2[4,12]. In his paper in 1976, Fateman points out the several obvious uses of floa3ting-point arithmetic system within a SAM system;

1) When exact rational answers are not necessary, the floating-point approximation is typically easier to read, write, compute with, and comprehend. For example, 3900231685776981/2251799813685248 is approximately 1.7320508.
2) Systems allowing non-rational functions such as sine, logarithm, etc., are unable to evaluate arbitrary constant expressions to an exact numeric (real) value anyway, and so floating-point approximations are frequently as appropriate or more appropriate than rational approximations.
3) Typical scientific calculations may begin with floating-point approximations as data; conversion to exact rational arithmetic may be an unnecessary and counterproductive transformation.

The same reasons can be applied to the usages of big-float systems in SAM systems when highly precise floating-point numbers are concerned. For, fixed preci-

記号処理 11-4 (1980. 3. 21)

sion real arithmetic, as in FORTRAN, is not sufficient to evaluate long or non-regular expression for which algebraic systems are best suited to manipulate[5]. Moreover, most of the algebraic system-oriented numerical algorithms require big-float arithmetic[10-11].

Most of the big-float systems on SAM systems are written in LISP (SAC-1 in the above examples is written in FORTRAN). On the other hand, many portable multiple-precision arithmetic packages[6-7] are written in FORTRAN. Among others, Brent's package[7] is elaborative as well as famous and widely distributed. Then, the following questions will naturally arise. Which system is faster, LISP-based or FORTRAN-based? Which is more convenient to use? Which is more useful in actual applications? As for the last question, it is too early to give a definite answer. Our answer to the second question is that LISP-based systems are more convenient to use because of the LISP facility of automatic memory management and of easeness of writing programs. As for the first question, a very common answer will be that the FORTRAN-based systems are faster than the LISP-based systems. Then, how faster the FORTRAN-based systems are? Nobody answered to this question yet.

Thus, we compared the speeds of three big-float systems, one is Brent's package and the others are Sasaki's system (machine-independent version) and its modified version (machine-dependent version) which we implemented on the HLISP system[8].

Brent's package was implemented on a FACOM 230-75 computer at the Institute of Physical and Chemical

Research, and Sasaki's system was implemented on a FACOM M-200 computer at the Institute of Plasma Physics, University of Nagoya. The later version of HLISP system, which is the fastest among the series of HLISP system and which we used in our experiments, is written in assembly language of FACOM M-200, and the Institute of Physical and Chemical Research has the license of Brent's system. Therefore, we cannot obtain timing data for the both big-float systems on the same machine. However, we can perform a relative comparison of both systems by referring to Table 1, which indicates the approximate CPU times of executing basic machine operations such as integer addition, subtraction, multiplication, etc. Since operations in both systems are dominated by integer arithmetic, floating-point operations are out of interests in our case.

Table 1

2. Brief survey of the systems concerned

In the Brent's system, T-digit big-float normalized numbers are stored in integer array of dimension T+2, with the following conventions:

word 1 = sign (0, -1, or +1)
word 2 = exponent (to base B)
words 3 to T+2 = normalized fraction.

Where T>1, B>1, and $8B^2-1$ must be representable as a single-precision integer. The base B and number of digits T may be varied dynamically. If sign = 0, the words 2 to T+2 are undefined. The exponent lies in (-M,

M), where M is set by the user, with the restriction that 4M is representable as a single-precision integer. In our experiments, base B is set to 10000 according to Brent's recommendation, because the word length of 230-75 is 36 bits. Selection of 65536 for B is also possible in our experiments, and this value is recommended for efficient memory usage and execution speed.

In the HLISP system, 10^8-base arithmetic is adopted. Namely, the big-integer of

$$\pm(b_1 \times 10^{8*(n-1)} + b_2 \times 10^{8*(n-2)} + \ldots + b_n \times 10^{8*0})$$

is represented by the S-expression,

$$(\pm n \;\; b_n \;\; b_{n-1} \;\; \ldots \;\; b_2 \;.\; b_1),$$

through a virtual address which points to the S-expression. Here $b_1 > 0$, $n > 1$, and b_2, .., b_n are non-negative integers such that $0 \leq b_i < 10^8$, for $i = 2 \ldots n$. Basic integers have the magnitude less than 10^8 and are represented by using virtual addresses which do not point to any S-expressions.

In the same way, in our machine-dependent version of the big-float system, a big-float number being assumed to be of the form of $m \times 10^e$ is represented by using a virtual address which points to an S-expression,

(e . m).

Here the mantissa m and the exponent e are signed big-integers. On the other hand, in the machine-independent version, every big-float number of $m \times 10^e$ is represented as a LISP S-expression

(!:BF!: . (m . e)).

Here, !:BF!: is an indicator distinguishing the big-float data type from others.

Any big-float number in our systems (machine-independent and -dependent versions) is represented in as short a form as possible, unless it has zeros accidentally at its rightmost places by the result of arithmetic operations. So, the decimal 1.234, for example, is represented as

(!:BF!: . (1234 . -3)) or --> (1234 . -3)

and not as

(!:BF!: . (12340000000000000000 . -19)), or
--> (12340000000000000000 . -19)

even if we are calculating a number up to twenty digits. Here '--> (L)' denotes the representation of the S-expression (L) by virtual addressing method mentioned above. For the reasons for taking the concise representation of big-float number as above, see reference 4.

Corresponding to this base-10 representation, base-10 arithmetic is adopted in our systems. The base-10 is chosen not only since base-10 arithmetic is the most familiar to scientists and engineers, who we assume will be the main users of the system, but also since base-10 representation allows us to represent any decimal accurately. For example, 0.1 or 0.3 can not be represented in base-2^k representation. In more detail, see references 4 and 12 for our big-float system and reference 8 for the HLISP system.

3. Timings of LISP-based system

Timing data by machine-independent version and machine-dependent version of the LISP-based system are given in Tables 2 and 3, respectively. We can see that the machine-dependent version is about 2 times faster than the machine-independent one.

Important and effective modifications made on the machine-independent version giving to the machine-dependent version are the followings:

1. N-arg EXPRs of big-integer arguments to 2-arg SUBRs.

   Examples. MAX, MIN, PLUS, TIMES, DIFFERENCE, QUOTIENT ---> MAX2, MIN2, P+, P-, P*, P/.

2. Basic EXPRs (constructor, deconstructor and predicate for big-float numbers) to SUBRs.

   Examples. MAKE!:BF, MT!:, EP!:, BFP!: ---> ENFLOAT(arg), CDR(DEFLOAT(arg)), CAR(DEFLOAT(arg)), FLOATP(arg).
   (MT!: and EP!: are functions which return exponent and mantissa part of a given big-float number, respectively. Being given a dotted pair of two integer as its argument, ENFLOAT changes the argument into a datum of big-float data type and returns the big-float number. Conversely, DEFLOAT changes its argument which is a big-float number into the dotted pair of the exponent and the mantissa.)

3. Precision counting routine of type EXPR to SUBR.

   Machine-independent version uses EXPLODE in precision counting routine and the routine is used in all precision handling routines. The EXPLODE is a time and space consuming function. Machine-dependent precision counting routine (HOWLONG) does not use EXPLODE but counts the length of the (big-)integer representing the mantissa.

4. Precision changing routines of type EXPRs to SUBRs.

   In the machine-independent version, the precision is changed by dividing or multiplying the mantissa by a power of 10. The machine-dependent version performes this by using FPSHIFT which shifts the digits of a (big-)integer.

Note that, all SUBRs needed for these modifications are not newly programmed. They are standard SUBRs in HLISP! Most effective modifications are on precision handling routines. These modifications achieved the speed-up ratio of 1.3. Timing data of logarithmic function calculations are given in reference 13, so we omit the data in the Tables.

Table 2

Table 3

4. Timings of FORTRAN-based system

Timing data for the Brent's FORTRAN system are given in Table 4. We did not change the system in anyway except for the replacement of clock routine. Then, the data in Table 4 are not biased in favor of the FORTRAN-based system.

Table 4

5. Comparison of FORTRAN-based and LISP-based systems

A glance at Tables 3 and 4 suggests superiority of the LISP-based system over the FORTRAN-based system, even if we consider the normalization factor 2 due to the differences of machine operation speeds. However, both systems are not compared on the same conditions. There are many differences between the systems. First, timing data in Table 3 are for machine-dependent version while those in Table 4 are for machine-independent system. Second, our systems were operated by a LISP-interpreter while Brent's system was operated by a FORTRAN-compiler. Third, the big-float numbers are represented with base-10^8 in our system while base-10^4 is used in the FORTRAN-based system. Fourth, different algorithms are used in both systems. And fifth, different precision handling methods are adopted. It is necessary to consider these differences for the fair comparisons of the systems.

In order to facilitate a fair comparison, we measured the execution time of basic big-float operations such as big-float addition, subtraction, multiplication and division on both system. The results are given in Table 5. Except for the data for big-float division, the results in Table 5 are quite reasonable. Both systems use the same algorithms for addition, subtraction, and multiplication. That is, the classical O(P) and $O(P^2)$ algorithms are used for addition-subtraction and multiplication, respectively, where P is the precision of the operands. Since the big-integer routines in the HLISP system are written in the ASSEMBLY language of M-200, the handicap of our system of being operated on a LISP-interpreter will be negligible for large P. Since, the addition, subtraction and multiplication times for both systems will correctly reflect the ratio 2 of basic machine operation speeds. We can see from Table 5 that this is in fact true. Table 5 shows, however, that the handicap between an interpreter and a compiler becomes apparent for small P.

The division algorithm used in the HLISP system is the algorithm D in the book of Knuth[9]. We see the algorithm is quite efficient. On the other hand, in Brent's system, the quotient of X/Y is calculated by first computing 1/Y by Newton's method (the result is approximate) and then multiply it by X. The algorithm for reciprocation has the asymptotic time complexity of O(P). However, as we can see from Table 5, this O(P) algorithm is quite slow in practical cases.

It is interesting to observe that the ratio of execution speeds of basic arithmetic operations is nearly equal to the ratio of machine speeds in spite of the difference in big-integer representations in both systems.

Reminding of section 2, we can see that Brent's system requires 2 times more basic machine arithmetic operation than our system. While, extra machine operation for handling lists are necessary in our system. Table 5 indicates that the total number of machine operations are almost the same in both systems.

Note that, nearly the same memory is used for representing a big-float number in our system and Brent's system: our system and Brent's system use 2+(P+3)/4 and 2+((P+7)/8)*2 words, respectively, to represent a big-float number of precision P.

With the data in Table 5 in mind, we may regard the data in Table 3 as showing a high efficiency of our LISP-based system. A detailed comparison of the data in Tables 3 and 4 shows that the ratio of the computation times of a given constants or functions on our and Brent's systems is almost constant for $P \geq 400$. However, the ratio depends strongly on the problems we tested. This means that the efficiency of big-float evaluation depends essentially on programming and not on the language on which the system is based. In fact, we know that our routine for exp(x) and Brent's routines for e and pi can be speeded-up considerably by improving programs. We must comment, however, that a fancy precision handling method as in reference 4 is necessary for writing efficient programs.

Table 5

6. Conclusion

A LISP-interpreter-based big-float system is quite efficient. It's speed can certainly be reduced within 2 times of the speed of a FORTRAN-compiler-based systems. Compared with the FORTRAN-based systems, the LISP-based systems have many merits in that they are basically interpretive systems and garbage collectors manage free storage automatically. These merits will push away the small demerit of the LISP-based system in the execution speed.

7. Acknowledgement

The authors would like to thank Mr. N. Inada for obtaining timing data on a FACOM 230-75.

	add/sub	mult	div	store	load
FACOM 230-75	108	450	2,340	270	108
FACOM M-200	54	270	920	108	54

Table 1. Comparisons of machine operation times (in nano-seconds) for integers on two FACOM machines which we used for our experiments. "Operands on the cache registers" is assumed. M-200 is nearly the same machine with Amdahl 470 V/8.

P	e	pi	sqrt(2)	exp(0.5)
20	2	1	42	92
40	5	5	50	132
50	5	5	51	144
60	5	5	51	176
80	5	5	61	213
100	5	5	62	266
150	5	5	74	391
200	5	5	79	525
400	135	276	107	1,305
600	215	489	152	2,669
800	312	768	174	4,688
1000	414	1,087	198	7,461

Table 2. Timing data for the machine-independent version of the LISP-based system on a FACOM M-200. The numbers listed are computation times (in milli-seconds) of evaluating e, pi, sqrt(2), exp(0.5) to the P-th decimal place. Garbage collection times are not included. Constants e and pi are pre-computed to the 210-th decimal place and stored. Making computation times of e and pi very small for P less than 201.

P	e	pi	sqrt(2)	exp(0.5)
20	1	1	9	24
40	1	1	11	35
50	1	1	11	37
60	1	1	11	43
80	1	1	14	54
100	1	1	15	68
150	1	1	19	102
200	1	1	20	145
400	60	151	36	437
600	111	303	62	980
800	176	510	81	1,853
1000	256	775	98	3,096

Table 3. Timing data for the machine-dependent version of the LISP-based system on a FACOM M-200. The numbers listed are computation times (in milli-seconds) of evaluating e, pi, sqrt(2), exp(0.5) to the P-th decimal place. Garbage collection times are not included. Constants e and pi are pre-computed to the 210-th decimal place and stored. Making computation times of e and pi very small for P less than 201.

P	e	pi	sqrt(2)	exp(0.5)
20	13	14	6	9
40	19	25	9	16
50	22	30	11	19
60	25	37	12	23
80	31	52	13	32
100	39	69	16	42
150	62	125	20	80
200	87	186	25	123
400	230	591	49	445
600	428	1,179	75	990
800	680	1,982	111	1,851
1000	982	2,975	151	3,086

Table 4. Timing data for the FORTRAN-based system on a FACOM 230-75. The numbers listed are computation times (in milli-seconds) of evaluating e, pi, sqrt(2), exp(0.5) to the P-th decimal place. Note that the FACOM 230-75 is about 2 times slower than the FACOM M-200.

P	Add	Sub	Mult	Div
20	0.153 / 0.12	0.155 / 0.15	0.155 / 0.37	0.799 / 4.3
40	0.158 / 0.14	0.160 / 0.17	0.214 / 0.67	0.934 / 6.3
50	0.165 / 0.15	0.167 / 0.18	0.290 / 0.79	1.029 / 7.8
60	0.166 / 0.16	0.170 / 0.20	0.330 / 0.93	1.126 / 8.4
80	0.172 / 0.18	0.177 / 0.22	0.431 / 1.35	1.345 / 9.8
100	0.180 / 0.20	0.183 / 0.25	0.620 / 1.71	1.565 / 12.6
150	0.193 / 0.26	0.201 / 0.32	1.106 / 3.29	2.257 / 17.9
200	0.212 / 0.31	0.217 / 0.39	1.755 / 4.85	3.234 / 24.1
400	0.278 / 0.53	0.295 / 0.66	6.252 / 18.3	8.640 / 57.9
600	0.337 / 0.74	0.362 / 0.93	13.52 / 35.2	16.96 / 104
800	0.402 / 0.96	0.439 / 1.21	23.54 / 53.1	28.08 / 168
1000	0.461 / 1.18	0.505 / 1.49	36.44 / 81.6	42.00 / 248

Table 5. Timing data for basic big-float operations (left columns for the LISP-based system on a FACOM M-200 and right columns for the FORTRAN-based system on a FACOM 230-75). The constants e and pi of precision P(i.e., big-float numbers having P figures) are added, subtracted, multiplied and divided to give the result of precision P. The numbers listed are the execution times (in milliseconds) of these calculations. Garbage collection times are not included.

References

1) W. S. Brown and A. C. Hearn, "Applications of Symbolic Algebraic Computation", Computer Phys. Comm. 17, pp. 207-215 (1979)

2) R. J. Fateman, "The MACSYMA "Big-Floating-Point" Arithmetic System", Proc. ACM SYMSAC '76, pp. 209-213 (1976)

3) J. R. Pinkert, "SAC-1 Variable Precision Floating Point Arithmetic", Proc. ACM 75, pp. 274-276 (1975)

4) T. Sasaki, "An Arbitrary Precision Real Arithmetic Package in REDUCE", Lecture notes in Computer Science 72 (Proc. ACM EUROSAM '79) Springer-Verlag, pp. 358-368 (1979)

5) E. W. Ng, "Symbolic-Numeric Interface: A Review", ibid, pp. 330-345 (1976)

6) W. T. Wyatt Jr., D. W. Lozier and D. J. Orsen, "A Portable Extended Precision Arithmetic Package and Library with FORTRAN Precompiler", Mathematical Software II, Purde University, May 1974

7) R. P. Brent, "A FORTRAN Multiple-Precision Arithmetic Package", ACM Trans. Math. Software 4, pp. 57-70 (1978)

8) Y. Kanada, "HLISP and Supplementary HLISP-REDUCE manual", Nagoya University, Feb 1979

9) D. E. Knuth, "The art of Computer Programming, vol 2: Seminumerical algorithms", Addison Wesley, Reading Mass., 1969.

10) P. L. Richman, "Automatic Error Analysis for Determining Precision", CACM 15, pp. 813-817 (1972)

11) J. R. Pinkert, "Interval Arithmetic Applied to Polynomial Remainder Sequences", Proc. ACM SYMSAC '76, pp. 214-218 (1976)

12) T. Sasaki, "Manual for Arbitrary Precision Real Arithmetic System in REDUCE", Operating Report of Symbolic Computation Group, University of Utah, May 1979.

13) T. Sasaki and Y. Kanada, "Practically Fast Multiple-Precision Evaluation of Log(x)", preprint.