24. LISP上の任意精度浮動小数演算システム

佐々木建昭(理研), 金田康正(名大プラズマ研究所),
Tateaki Sasaki Yasumasa Kanada
稲田信幸(理研)
Nobuyuki Inada

1. はじめに

任意精度浮動小数演算システム(以下、big-floatシステムと呼ぶことにする)が必要とされるのはどういう場合であろうか? 数値計算のユーザーが、「桁落ちのため答の正確さが著しく悪くなった。もっと高精度に計算してくれるシステムが欲しい」などというのをよく耳にする。しかし、彼らの多く(おそらく99%以上)は答の精度を10桁かせいぜい20桁程度必要としているだけであろう。それが、桁落ちのため望ましい精度が得られないと嘆いているのである。このような場合には、単精度から倍精度、あるいは倍精度から4倍精度にすることにより解決できるし、しかも計算手順を工夫するだけで桁落ちを防げることも多い。しかしながら, 非常に精度のよい数表を作るとか, 関数値の非常に小さい差異を調べるなどという場合には、どうしても高精度の演算システムが必要になる。すなわち、big-floatシステムが必要とされるのは、元来高精度の答が必要なときである。そのような必要性はそう度々あることではないが、なすべき計算の性質上、その必要性が高い分野がある。それは代数的計算の分野である。

代数的計算システム(数式処理システム)では、正確な答を得る計算、正確でなくても解析的な答を出したり、数値計算ではうまく成し得ない計算が実行できる。そこでは、非常に桁数の多い整数、あるいは有理数を扱うことが多い。そのため、数式処理システムには任意精度整数演算システム(以下, big-numberシステムと呼ぶ)を装備することが常識になっている。しかしながら, big-numberシステムだけではプログラミング上不便なことが多い。それは小数を扱うときである。big-numberシステムがあれば任意の有理数は扱えるが, 数値が有理数としては扱えない場合、あるいは有理数として扱う必要性がない場合が、代数的計算においてはよく現れる。たとえば、非常に近接した2根を代数的方法で分離する場合とか、特殊関数の関数値を計算する場合などである。このような場合には小数を有理数で表現したり近似したりするよりも、初めから浮動小数で表現し演算した方が見易いし、プログラミングも容易で、しかもメモリー使用効率, 計算効率がはるかに高くなる。この点についてより詳しく知りたい方は、文献1の解説を見られたい。

本稿では、数式処理システムへの組込を目的にして筆者らが作成した2つのbig-floatシステムを紹介する。これらは多くの数式処理システムのホスト言語であるLISPで書かれているが、精度の扱いなど重要な点でいくつか異なっている。システムを作成する上で底の選択、精度の扱い、基本組込関数のアルゴリズムの違いがシステムの実行効率、使い勝手に大きく影響する。これらの違いによる速度のちがいを数値的に比較しながら、我々のシステムを説明する。また、FORTRAN-basedの代表的システムとの比較も試みる。我々のシステムの使い勝手は諸兄に御判断頂く以外にないが、その速度は現存のLISP-basedのシステムの中では、世界最高であると自負している。

2. Big-float システムの現状

Big-floatシステムに明るくない読者のために簡単にその歴史を解説しておく。この解説は完全ではなく、著者らの知識不足あるいは紙面の都合上、記述しきれぬ部分もあることをあらかじめお断りしておく。

1960年代の中頃からbig-floatシステムに関する文献が数多くみられるようになった。1966年のTiemari と Suokonautio の ALGOL 60 における big-floatの話、1967年の Ehrman (SLAC) による IBM/360 上での big-float システムの話、同年の Lawson (Jet Propulsion Lab.) の一連の論文などがそうである。より詳しい文献リストについては文献3の末尾を参照されたい。これらのシステムの多くはFORTRANで書かれているが、big-float数の基本演算の各々がサブルーチンとして書かれているため、演算毎にサブルーチン・コールが必要で、プログラミングが繁雑になる。プレコンパイル方式により、プログラミングの繁雑さを取除き、ポータビリティを重視して設計したシステムにWyattら (National Bureau of Standards) のもの[2]とBrent (オーストラリア大学) のもの[3]がある。特にBrentのシステムには、πやγなどの定数や$\exp$, $\sin$などの初等関数、特殊関数の高速アルゴリズムに関する最新の成果[4]が最も多く取入れられている。

一方、数式処理システム指向のbig-floatシステムとしては、1974年MACSYMAに組込んだFatemanのシステム[5]、1975年のPinkertのSAC-1上のシステム[6]、1978年の金田のHLISP上のシステム[7]、1979年の佐々木のREDUCE上のシステム[8]がある。このうち、PinkertのシステムはSAC-1語で書かれているが、SAC-1自身はFORTRANで書かれた数式処理システムである。Fatemanと金田のシステムはLISP(詳しく言えばMACLISP及びHLISP)で書かれている。佐々木のシステムはREDUCEの記号演算用言語(RLISP)で書かれているが、この言語はLISPと一対一に対応し、REDUCEを通すことによりStandard LISPに変換されるので、LISPで書かれていると言ってさしつかえない。これらのシステムの他に、1979年に東大の小野がFORTRANの拡張としてのbig-floatシステムを作成している[9]。彼のシステムはFORTRAN→LISPへのトランスレータで、全てのシステムがLISPで書かれている。実際のbig-float演算関数はHLISPで書かれているので、本質的にLISP上のシステムと考えてよい。また、後藤らがbig-float数を基本的なデータ型の一つにもつ数式処理専用機を製作中である[10]。

3. システムの基本的構造

我々のシステムにおいては、big-float数はLISPのS式として表現される。このことはLISP上のbig-floatシステムすべてに共通している。任意の小数n(詳しく言えば10進表示での小数)は適当な整数mとeとにより

$$(1) \qquad n = m \cdot 10^{e}$$

と表現できる。ここで、用語上疑義があるかも知れぬが、mを仮数、eを指数と呼ぶことにする。mとeの選び方には無限の自由度があることに注意されたい。たとえば1.234は1234×10^{-3}, 12340×10^{-4}, 等々と表現できる。(1)式に従い、

big-float数を2整数のペアで表現する。金田のシステムでは

(2)　　　　$(e \cdot m)$

なるリストでこれを表示する。このままではbig-float数と通常のリストとの区別がつかないので、実在しないアドレスを指す仮想ポインタ(視点を変えれば、タグ付リストとも見なせる)で上記リストをポイントし、big-float数とみなす。又big-numberも同じような方法で実現されている。佐々木のシステムでは、

(3)　　　　(!:BF!: . (m . e))

なる、表示子!:BF!:をもつリストでbig-float数を定義する。

さて、我々は任意精度の浮動小数を扱っているのであるから、仮数mのみならず指数eもbig-numberとなり得る。したがって、big-float数の演算にはbig-numberの演算機構が必要であるが、幸い大半のLISPシステムにはこの機能が備わっているのでそれを利用する。

我々のシステムは基本的演算ルーチン(四則演算など)，数の型(整数型，実数型など)と精度の変換ルーチンなどに加え、π，e，$\sqrt{2}$などの定数ルーチン，$\sqrt{x}$, $\exp(x)$, $\sin(x)$などの基本関数ルーチンを含んでいる。

4. 底について

big-float数の外部表現において(1)式のように底10を採用し，内部表現として(2)あるいは(3)式を採用したことは、我々は全面的に底として10を採用したことを意味する。理工学においては底10が一般的に用いられている現状からして、外部表現では(1)式に異論はなかろうが、内部表現ではいろいろと異論があろう。その最たるものは、「何故2進表示を用いないのか？　2進表示では桁ずらしがビット・シフト演算で高速に実行できるのに」あるいは「2進表示の方がスペースの節約になるのに」であろう。実際，Fatemanは、底として2を採用している。この論点に関する著者らの見解は完全には一致していないが、最大公約論は「ホスト言語を想定しているHLISPがbig-number表現において10^8を底としているから」である。もしもbig-numberが

(4)　　　　$n_0 + n_1 \cdot 10^k + n_2 \cdot 10^{2k} + \cdots, \quad 0 \leqq n_i < 10^k$

と表現されていれば、この数の桁数を勘定したり桁ずらししたりするには10進表示の方がはるかに便利である。Big-floatシステムでは桁数の勘定と桁ずらしが演算速度に大きく影響するので、我々の場合、底として10を選択する方が有利である。さらに、底を10とすることは、入出力時に数の表現型をいちいち変換しなくてもよいというメリットがあることを付け加えておく。(特に出力の場合、2進表示であれば、桁数が長いと、変換時間が無視できなくなる。)底10と2の違いによる計算速度の違いの総合的比較はしていないが、我々のシステムにおいて、桁数勘定ルーチンをシステム・デペンデントに書き直した時、システムは約1.3倍強高速化したことを報告しておく。

底として10を選択することは佐々木のシステムにおいてはより積極的意味をもつ。このシステムでは高速化のためbig-float数をできるだけ短く表現する。底

が10ならば、たとえば0.1や0.03は(!:BF!:.(1.-1))及び(!:BF!:.(3.-2))と短く表現できるが、底2ではこれらの数は、(3)の内部表現を用いて正確には表現できないからである(次節参照)。

5. 精度の扱い

(1)式において、仮数部mの(10進での)桁数を精度(precision)と呼ぶ。Big-floatシステムでは精度は当然任意で可変であるが、その扱い方には種々の方法があり、システムの実行効率と使い勝手に大きく影響する。

Big-floatシステムにおける精度の最もポピュラーな扱い方は全体精度(global precision)を設けることである。全体精度とは、その名の通り、全ての big-float 数の精度を規定する。この方法は従来の固定精度システムの最も自然な拡張であり、Brent, Fateman, 金田など多くの人々のシステムで採用されている。これらのシステムでは、ユーザーは最初に全体精度を彼の欲する値にセットしておけば、あとは精度のことを余り気にしなくてよいから、使い勝手の点で優れている。しかし、効率の良いプログラムを書くという点では不利である。

一方、全体精度を設ける方法と正反対なのが、個々の基本演算毎にその答の精度を指定する方法である。この方法だと、精度を常に気にしなければならぬためユーザーの負担は増すが、次節に述べるごとく、効率のよいプログラムを書くという観点からは有利である。この方法を全面的に採用したのは Pinkert であるが、彼の論文からもシステム(定数及び基本関数ルーチンを含まず)からも、彼が精度の扱いをどの程度真剣に考察したかは読みとれない。

佐々木は上記2方法の折中案を採用した。彼はシステムの高速化を最大の設計方針とし、次の2点を重視した: ① Big-float 数の表現をできるだけ短くし、個々の基本演算に要する時間を短縮する。②精度制御を可能な限り自由にし、実行効率の高いプログラム作成を可能にする。まず前者について説明する。たとえば big-float 数としての0.2と0.03の積を考えよう。全体精度を設けるシステムでは、全体精度がたとえば20のとき、これらは(2)式流表現では(-20. 200000000000000000000)と(-21. 300000000000000000000)で表現される。従って、積の計算には20桁の整数の積とその結果の丸めが必要になる。一方、仮数をできるだけ短かく表現する方式では、前記2数は(3)式流表現では(!:BF!:.(2.-1))と、(!:BF!:.(3.-2))と表現される。従って、積の計算には単精度整数の演算で事足りる。どちらが高速かは明白であろう。しかしながら、後者の方式では、精度はもはやその数の"正確さの度合"を意味しないから、基本演算においては、被演算数は一般に正確な数として扱う必要がある。そのため、丸めはユーザーの責任で実行するのが望ましい。こうすればプログラマの負担は増すであろうが、可能な限り正確に計算を進めることができ、しかも不必要な丸めのステップが省略できる。佐々木はこれらの長所短所を考慮して、二種類の基本演算ルーチン群(+,-,*,/,**)を用意した。第一種のルーチン群では、除算以外の関数は最後の桁まで正確に計算した答を返す。除算では割り切れない場合もあるので、答の桁数はユーザーが指定する必要がある。第二種のルーチン群では、演算後の答の精度は全体精度がセットされているときにはその値に、そうでないときには被演算数の精度の大きい方にセットされる。上述したごとく、第一種のルーチン

群を使えば効率的なプログラムが書けるが、ユーザーが精度を細かい点まで制御しなければならぬ。それが嫌な人は第二種のルーチン群をどうぞ、という訳である。次に、設計方針の第二の点、すなわち「精度制御の自由化」は効率的なプログラムを書く上で非常に重要であるが、この点は次節で説明しよう。

6. 効率的プログラム

まず、数 n のオーダーを次のように定義する：

$$10^k \leqq |n| < 10^{k+1} \text{ のとき } k, \quad n = 0 \text{ のとき } 0.$$

数の構成要素が仮数と指数の二つあることに対応して，big-float数の数としての性質は大きさを表わすオーダーと精度の二つであると言えよう。Big-float数を最大限制御できるためには、少くとも精度とオーダーの二つの観点から数を制御できねばならず、またそれで十分であると考えられる。

さて、精度を制御すると効率的なプログラムが書ける例として、Newtonの反復公式による平方根の計算を考えよう：

$$(4) \qquad y_{n+1} = \frac{1}{2}(y_n + x/y_n).$$

この反復公式では $n \to \infty$ とともに y_n は $\sqrt{x}$ に収束するが，一回反復する毎に y_n の精度は約2倍づつ増加する：なぜなら $y_n = \sqrt{x}\ (1+\varepsilon)$，$|\varepsilon| \ll 1$ とすると、$y_{n+1} \approx \sqrt{x}\,(1+\varepsilon^2/2)$ となるので。一方，反復の初期には答の精度は低いのが普通であるから、反復の最初から y_n の精度を最終の答の精度に等しくして計算するのは無駄な計算をすることになる。初期値 y_0 の精度は1か2程度にしておき，y_n の精度が最終の答の精度に達するまで、各反復後に y_n の精度を2倍づつに増やして計算させれば，無駄な計算が省けて効率が上がる。

次に、オーダーに着目して big-float数の精度を制御すべき例として、Taylor級数を利用した対数関数の計算を考えよう：

$$(5) \qquad \log(1+x) = \sum_{n=1}^{\infty} \frac{x^n}{n}, \quad |x| \ll 1.$$

この級数の第 n 項のオーダーは n の増加とともに減小するから，全ての項を同じ精度で計算するのは無駄な計算をすることになる。今の場合、最終の答のオーダー O の近似値は容易に知れる。従って、最終の答の精度を P まで欲しいときには第 n 項をオーダー O-P-G，ただし G は保護桁数，の桁まで（すなわち、一番右端の桁の数字のオーダーが O-P-G となるように）計算させれば、無駄な計算が省けて効率が上がる。

精度を実行時に変えることはたいていの big-float システムで可能である。もちろん我々のシステムでもそうである。しかし、数のオーダーに着目して精度を制御するルーチンは、現在のところ佐々木のシステムだけが備えている。精度を制御しながら計算をすれば数倍の高速化が達成できることを8節に示す。

桁落ちによる精度の損失がある場合には、その損失をカバーするため、保護桁数（guard digit）をより多くとらねばならない。もし，実際に桁落ちが起きそうな箇所でのみ高精度に計算させることができれば、計算の効率が上がる。その

ためには、佐々木の第一種の基本演算ルーチンのように、正確な結果を丸めなしに返すルーチンと丸めのルーチンがあればよい。たとえば、$|x|\sim 1$のとき$1-x^2$を計算するのに、まずx^2を正確に計算し1から引いたのち、結果を丸めるならば桁落ちは生じない。

7. 定数および基本関数ルーチン

我々のシステムの定数ルーチンとしては、π, eに加えて2, 3, 5, 10の平方根、立方根、及び対数などがある。これらの定数は一度計算されると自動的にその値が記憶され、次回からは、記憶された数より高精度の定数が要求されない限り、再計算を行わない。πとeは整数の四則演算で計算し、最後にbig-float数に変換する。πはMachinの公式[11]を利用するものとBrentの公式[4]を用いるものと二種類ある。eは階乗の逆数和で計算する。他の定数は$\sqrt{x}$や$\log x$などの基本関数ルーチンを利用して計算する。

基本関数ルーチンとしては、平方根および立方根関数、指数及び対数関数、三角関数及び逆三角関数がある。平方根および立方根関数はNewtonの反復公式で、exp, log, sin, cos, 及びatanは引数を適当な値域内に変換したあと、それらのTaylor級数を勘定することにより計算する。$\tan(x)$は、$|\sin(x)| \leqq 1/\sqrt{2}$のとき$\tan(x) = \pm\sin(x)/\sqrt{1-\sin^2(x)}$で、そうでないときは$\tan(x) = \pm\sqrt{1-\cos^2(x)}/\cos(x)$で計算する。$a\sin(x)$, $x \neq \pm 1$, 及び$a\cos(x)$, $x \neq 0$, の計算には$a\tan(x/\sqrt{1-x^2})$および$a\tan(\sqrt{1-x^2}/x)$がそれぞれ用いられる。

さて、$M(k)$をk桁の2数の積に要する単精度演算の個数とするとき、上記の計算法で$\exp(x)$, $\sin(x)$, $\cos(x)$をk桁まで勘定する際の計算の複雑さは、$O(M(k)k/\log(k))$であるが、$a\tan(x)$と$\log(x)$については$O(M(k)k)$である。すなわち、前の3つの関数については、Taylor級数に$1/n!$の係数が含まれるから収束は速いが、後の2つの関数のTaylor級数の係数は$1/n$なので収束が遅いのである。そのため、$a\sin(x)$及び$a\cos(x)$の計算の複雑さも$O(M(k)k)$となる。しかしながら、$\log(x)$及び$a\sin(x)$を計算するのに、$x=\exp(y)$及び$x=\sin(y)$をNewton法でyについて解くならば、これらの関数の計算の複雑さを$\exp(x)$及び$\sin(x)$の計算の複雑さと同じにまで、すなわち$O(M(k)k/\log(k))$にまで改善することが可能である。ただし、Newton法で解く際には、平方根関数などの計算と同様、精度を最適に制御する必要がある(前章の(4)式以後を参照のこと)。そのため、佐々木のシステムでは$\log(x)$と逆三角関数に対しては、Newtonの反復法を利用するルーチンも備えている。Newton法を利用するルーチンを、Taylor級数を利用するルーチンと比較すると、精度50桁では約3倍遅い。しかし計算の複雑さがよいから、精度が大きくなるにしたがい、Taylor級数を利用するルーチンより速くなるであろう。

定数と初等関数の計算については、Brentらが計算の複雑さ$O(M(k)\log(k))$の「高速」アルゴリズムを提案している[4]。そのうち、計算ステップ数からみて実際上最も高速なπの計算アルゴリズムをテストしたところ、精度200では、Machinの公式に基づくルーチンより10倍程遅いことが判明した。従って、Brentらの高速アルゴリズムは、πの計算でさえ精度が数千以上でないと実用的でない。(その後の実験により、1万桁の計算においてもMachinの公式に基づくルーチ

この方が速い事が判明した。ただし garbage collection時間を除いて。)

以上のように、我々の組み込んだルーチンの計算の複雑さは良くはないが、精度が1000程度以下では最も高速であろうと考えられる。次に次節に実際のシステムの計算速度を、いくつかの定数と基本関数のルーチンの速度で例示しよう。

8. システムの性能

我々のシステムが実際にどの程度の(速度)性能をもっているかをテストした。最後の表①～④は定数 e と π, 基本関数 $\sqrt{x}$, $\exp(x)$, $\log(x)$ をいくつかの桁数で計算したものである。又いくつかの結果について、桁数と処理時間の関係を図示しておいた。

表①は金田のシステムをFACOM M-200上でHLISPにより稼動させたもの。
このシステムについては、以下の点に注意されたい。……◎guard digit は2。
◎$\log(x)$はxを$[1, 1+1/e]$におとすため $\log(3)$は $\log(2)$より 3倍弱遅い。
◎$\log(x)$の計算には、eの計算は含まない。 ◎garbage collectionの時間($\log$, πでのみ重要)は含まず。 ◎$\exp(x)$の計算には、eの計算時間も含む。
2回目以降は、$\exp(x)$とeの時間差の時間でよい。(π, eの値を記憶するため)

表②は佐々木のシステム(Machine-independent版)を同様に稼動させたもの。

表③は佐々木のシステム(HLISP版)を同様に稼動させたもの。佐々木のシステムについては、以下の点に注意されたい。……◎guard digit は2。
◎$\log(x)$はxを$[1, e^{1/10}(\approx 1.105)]$の間におとす。 ◎$\log(x)$の計算には、$e$, $e^{\frac{1}{10}}$の計算を含まない。 ◎garbage collectionの時間は含まない。 ◎e, πは、前もって210桁まで計算し記憶してある。(ただし20桁までの値は、答が早く出る。) ◎$\exp(x)$の400桁以上の計算においては、eの計算時間も含む。2回目以降の計算時間は、$\exp(x)$とeの時間差だけになる。

表③は、稲田により理研にインプリメントされたBrentのシステム(FORTRANで書かれている)[3]を、FACOM 230-75上で稼動させたもの。このシステムについては以下の点に注意されたい。……◎guard digit は4以上。
◎$\log(2)$は、整数引数用の高速ルーチンを用いた。

なお、big-floatシステムで最も多用する整数の加算及び乗除算等の演算に要する時間は、FACOM 230-75の場合、マニュアルによると、加減算で108nsec, 乗算で450nsec、除算で2.34μsec, Storeが270nsec, Loadが108nsecである。同じくM-200の場合、ソフトウェアプログラムによる時間測定によると、加減算が54nsec, 乗算が270nsec, 除算が920nsec, Storeが108nsec, Loadが54nsecであった。

9. おわりに

FORTRANなどのように固定精度の浮動小数をハードウェアで演算するのに比べれば、現在ソフトウェアで実行するしか手のないbig-float数の演算は非常に高価なものにつく。そのためにも、少しでも高速なシステムを作成することは有意義であると信ずる。我々のシステムの処理速度は、LISPのインタプリタで動かしたにもかかわらず、FORTRANで直接書いたBrentのシステムと比較してもそう

見劣りしないことが判明した。従って、我々のシステムは十分高性能であると自負している。

システムの高速化にあたっては、精度の扱いを可能な限り自由にし、精度を自在に制御したことが大きく効いていることを強調しておきたい。(図を参照されたい。) この精度を制御し効率よいプログラムを書くということは、少くとも組込関数などでは、絶対に必要なことであろう。精度制御の自由をユーザーにまで認めるかどうかは議論の余地があろうが、big-float数の演算が高価な現在では、それも必要なことであろうと考える。ここで一言注意しておきたいのは、精度を制御しながらプログラムを書くことは、慣れてしまえば、FORTRANのプログラムを書くのと同様に、さして困難でも苦でもないことである。

我々は最近、本稿に述べた佐々木のシステムをもとにして、それをHLISP専用に修正し、考え得る限り高速化したシステムを名大プラズマ研と理研のHLISPに組込んだ。使い勝手はいざ知らず、「速いことは良いことだ」と信ずる故である。我々の現在の関心事は、「底として2を採用し、桁ずらしにビット・シフト演算を利用したらどの程度の高速化が達成できるか」である。この点を除けば、我々のシステムはソフトウェア的に達成可能な極点に近くまで高速化されていると考えている。従って、今後のテーマは、big-numberシステムのファームウェア化，ハードウェア化による高速化であろう。

なお、我々のシステムに興味ある方には、ドキュメント(ALGOL-likeの言語で書いたシステムルーチン一覧及びマニュアル[12]) とシステムを無料で開放するので、著者のいずれかに御一報頂きたい。

表①

	e	π	SQRT(2)	EXP(2) (EXP(2.1))	LOG(2) (LOG(3))
k=20	5	12	19	11 (34)	59 (28)
k=40	11	26	25	15 (64)	140 (52)
k=50	12	33	26	17 (79)	165 (68)
k=60	14	41	28	20 (95)	225 (85)
k=80	22	56	34	27 (140)	353 (132)
k=100	29	73	39	36 (194)	508 (184)
k=150	56	117	52	66 (373)	1,080 (387)
k=200	98	168	68	106 (644)	2,049 (721)
k=400	411	426	152	424 (2,969)	10,932 (3,882)
k=600	1,015	774	285	1,038 (8,021)	30,684 (11,254)
k=800	2,029	1,215	450	2,063 (17,022)	68,123 (24,909)
k=1000	3,558	1,749	648	3,596 (30,549)	127,797 (46,855)

kは精度（答の10進での桁数），数の単位はmsec.

表②

	e	π (Brent法)	SQRT(2)	EXP(2) (EXP(2.1))	LOG(2) (Newton法)
k=20	2	1	42	18 (97)	157 (650)
k=40	5	5	50	19 (153)	289 (1,076)
k=50	5	5	51	21 (180)	365 (1,196)
k=60	5	5	51	21 (208)	447 (1,301)
k=80	5	5	61	22 (270)	624 (1,939)
k=100	5	5	62	22 (316)	797 (2,182)
k=150	5	5	74	26 (470)	1,406 (3,795)
k=200	5	5	79	30 (629)	2,100 (4,757)
k=400	135	276 (1,784)	107	181 (1,559)	6,813 (13,343)
k=600	215	489 (2,788)	152	279 (2,704)	15,867 (34,170)
k=800	312	768 (4,030)	174	396 (4,220)	30,716 (53,212)
k=1000	414	1,087 (5,243)	198	522 (6,121)	53,386 (80,569)

kは精度（答の10進での桁数），数の単位はmsec.

表③

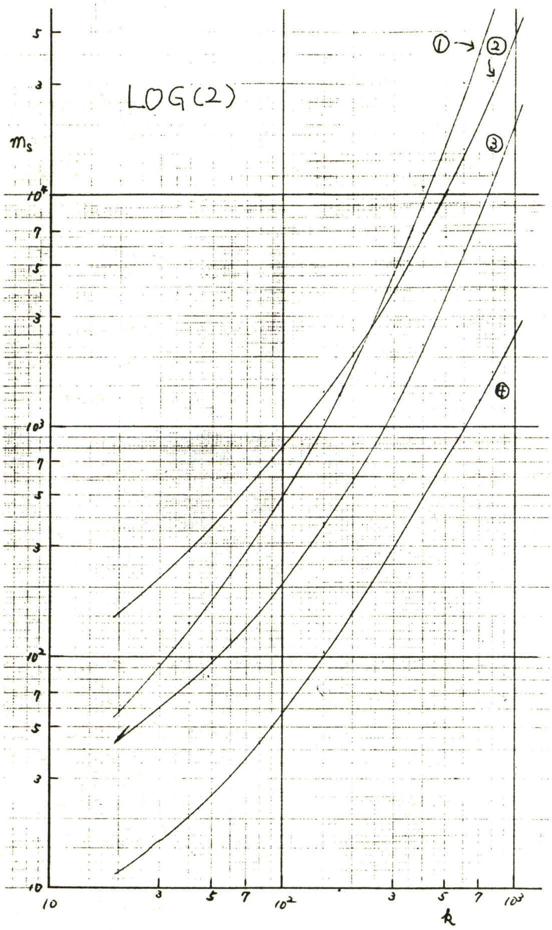
	e	π (Brent法)	SQRT(2)	EXP(2) (EXP(2.1))	LOG(2) (Newton法)
k=20	1	1	11	3 (24)	45 (166)
k=40	1	1	13	3 (38)	77 (280)
k=50	1	1	13	4 (45)	98 (310)
k=60	1	1	13	4 (52)	119 (336)
k=80	1	1	16	4 (67)	164 (509)
k=100	1	1	16	4 (80)	214 (583)
k=150	1	1	21	5 (124)	377 (1,051)
k=200	1	1	22	5 (171)	589 (1,376)
k=400	60	151 (526)	36	71 (504)	2,232 (4,676)
k=600	111	303 (943)	63	126 (993)	5,757 (13,519)
k=800	176	510 (1,438)	81	204 (1,738)	12,020 (22,521)
k=1000	256	775 (1,929)	100	300 (2,757)	21,846 (35,879)

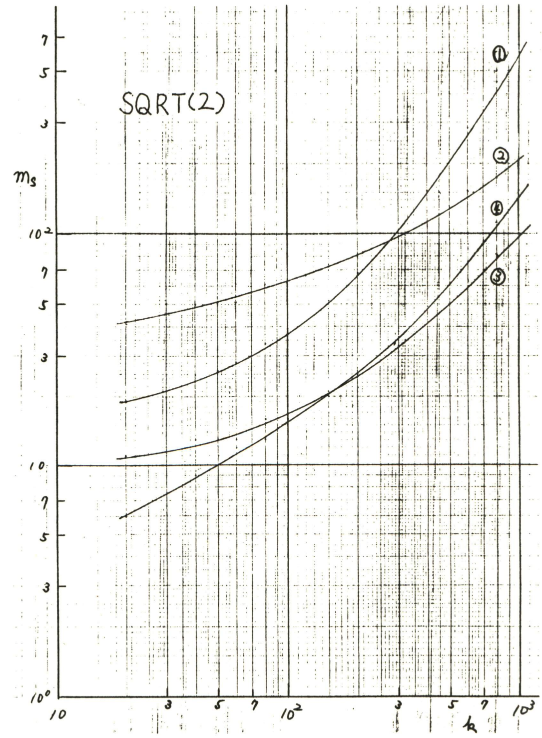
kは精度（答の10進での桁数），数の単位はmsec.

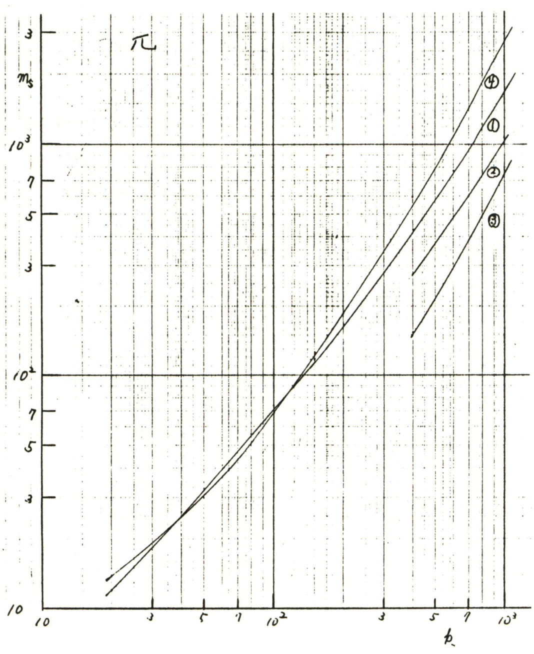
表④

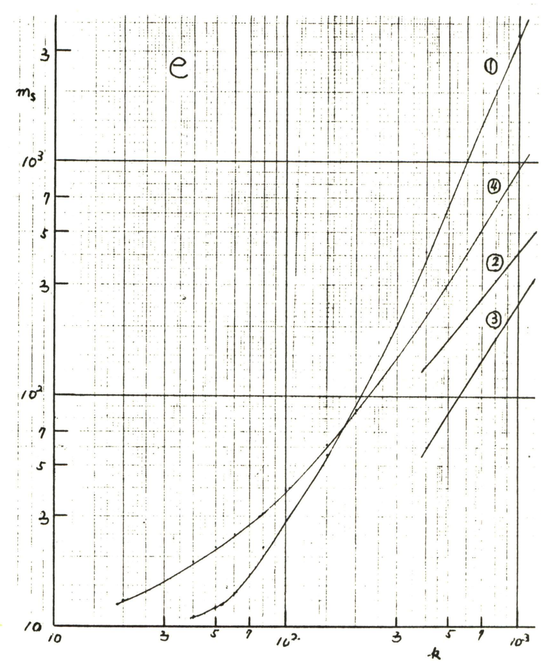
	e	π	SQRT(2)	EXP(2)	LOG(2)
k=20	13	14	6	14	12
k=40	19	25	9	20	21
k=50	22	30	11	22	26
k=60	25	37	12	26	31
k=80	31	52	13	33	43
k=100	39	69	16	41	58
k=150	62	125	20	67	104
k=200	87	186	25	93	157
k=400	230	591	49	250	504
k=600	428	1,179	75	467	1,014
k=800	680	1,982	111	733	1,726
k=1000	982	2,975	151	1,062	2,541

kは精度（答の10進での桁数），数の単位はmsec.









精度と計算時間
との関係図

LOG(2)	SQRT(2)
π	e

参考文献

1: W. S. BROWN and A. C. HEARN, "Applications of Symbolic Algebraic Computation", Computer Phys. Commun. 17, pp. 207 ~ 215 (1979).

2: 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, Purdue University, May 1974.

3: R. P. BRENT, "A FORTRAN multiple-Precision Arithmetic Package", ACM Trans. Math. Software, 4, pp. 57 ~ 70 (1978).

4: R. P. BRENT, "Fast multiple-precision Evaluation of Elementary Functions", JACM 23, pp. 242 - 251 (1976).

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

6: J. R. PINKERT, "SAC-1 Variable Precision Floating Point Arithmetic", Proc. ACM 75, pp. 274 ~ 276 (1975).

7: Y. KANADA, "HLISP and Supplementary HLISP-REDUCE manual", Nagoya University, Feb. 1979.

8: 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).

9: K. ONO, "BFORT -- A FORTRAN System with Arbitrary Precision Integer and Real Arithmetic", Master Thesis, University of Tokyo, Jan. 1979.

10: E. GOTO, T. IDA, K. HIRAKI, M. SUZUKI, N. INADA, "FLATS, a machine for numerical, symbolic and associative computing", Proc. 6th Symp. on Computer Architecture", pp. 102 - 110 (1979).

11: 森口繁一, 宇田川銈久, 一松 信, "数学公式" 第Ⅱ巻, 岩波全書 (1968).

12: T. SASAKI, "Manual for Arbitrary Precision Real Arithmetic System in REDUCE", Univ. of Utah, May 1979.

本 PDF ファイルは 1980 年発行の「第 21 回プログラミング・シンポジウム報告集」をスキャンし、項目ごとに整理して、情報処理学会電子図書館「情報学広場」に掲載するものです。

この出版物は情報処理学会への著作権譲渡がなされていませんが、情報処理学会公式 Web サイトの https://www.ipsj.or.jp/topics/Past_reports.html に下記「過去のプログラミング・シンポジウム報告集の利用許諾について」を掲載して、権利者の捜索をおこないました。そのうえで同意をいただいたもの、お申し出のなかったものを掲載しています。

過去のプログラミング・シンポジウム報告集の利用許諾について

情報処理学会発行の出版物著作権は平成 12 年から情報処理学会著作権規程に従い、学会に帰属することになっています。

プログラミング・シンポジウムの報告集は、情報処理学会と設立の事情が異なるため、この改訂がシンポジウム内部で徹底しておらず、情報処理学会の他の出版物が情報学広場 (=情報処理学会電子図書館) で公開されているにも拘らず、古い報告集には公開されていないものが少からずありました。

プログラミング・シンポジウムは昭和 59 年に情報処理学会の一部門になりましたが、それ以前の報告集も含め、この度学会の他の出版物と同様の扱いにしたいと考えます。過去のすべての報告集の論文について、著作権者（論文を執筆された故人の相続人）を探し出して利用許諾に関する同意を頂くことは困難ですので、一定期間の権利者捜索の努力をしたうえで、著作権者が見つからない場合も論文を情報学広場に掲載させていただきたいと思います。その後、著作権者が発見され、情報学広場への掲載の継続に同意が得られなかった場合には、当該論文については、掲載を停止致します。

この措置にご意見のある方は、プログラミング・シンポジウムの辻尚史運営委員長 (tsuji@math.s.chiba-u.ac.jp) までお申し出ください。

加えて、著作権者について情報をお持ちの方は事務局まで情報をお寄せくださいますようお願い申し上げます。

期間：2020 年 12 月 18 日 ～ 2021 年 3 月 19 日
掲載日：2020 年 12 月 18 日

プログラミング・シンポジウム委員会

情報処理学会著作権規程
https://www.ipsj.or.jp/copyright/ronbun/copyright.html