

ベクトル型インターフェースの対数ルーチン

寒川 光

日本アイ・ビー・エム株式会社 東京基礎研究所

RISC 型ワークステーションの環境では、初等数学関数はスカラー型インターフェースのものが使用されているが、多くの引数に対してまとめて関数値を求める場合は、ベクトル型インターフェースにすると、アンローリングなどの技法が利用できるので高速化できる。本稿ではベクトル型の対数関数について述べる。Fortran や C などの言語ライブラリが提供する関数の精度は非常に高いので、ベクトル版も同様の精度を保ったまま、約 2 倍の性能を達成した。ベクトル型から出発して、これとビット単位に同一の結果を与えるスカラー型の関数を作ることはむずかしくないで、コンパイラのループ変換と組み合わせることで、コンパイラが自動的にこの機能を利用することが可能なことも分かった。

Logarithmic Routine with Vector Interface

Hikaru Samukawa

IBM Research, Tokyo Research Laboratory

Though elementary mathematical functions with scalar type interface are used in RISC workstation environments, it is possible to tune these routines with unrolling technique if the interface is changed to vector type interface, in which many function values are computed for many arguments in a single call. In this paper, we describe a logarithmic function with vector type. Since a very high accuracy is provided by intrinsic functions of Fortran/C library, our vector type routine achieves doubled performance with keeping the same level accuracy. We found that it is possible for compiler to use vector library with loop transformation techniques because it is not difficult to provide a bitwise identical results between vector and scalar libraries if the development is started from vector type.

1 はじめに

数値解析プログラムの中には、1つのライブラリ関数で、計算時間全体の半分近くを占めるものがある。このようなプログラムのチューニングには、どうしてもその関数自身を書き改める必要がある。一般にこのようなケースでは、ループによって多くの関数値が計算される。

```
do i = 1, n
  x = ...
  y = sin x + ...
  z(i) = ... + y
enddo
```

このループを分割すると、次の例の *sinv* のように、多くの関数値をまとめて計算するベクトル型インターフェースの関数サブルーチンを切り出せる。

```
do i = 1, n
  x(i) = ...
enddo
call sinv(x, y, n)
do i = 1, n
  z(i) = ... + y(i)
enddo
```

sinv はアンローリングが適用できる。すなわちベクトル型の関数を自作すれば高速化できるが、精度と速度を両立させることが意外に難しい。

精度 言語ライブラリが提供する組込み関数の精度は非常に高く、これと同等の精度を実現することはあまり簡単ではない。

速度 高精度を実現したプログラムにアンローリングが適用できるかどうかは疑問である。

もし精度的にルーズなルーチンを自作して組込み関数を置き換えると、プログラムの他の部分で同じ引数に対して組込み関数によって求めた値と、自作ルーチンとが異なる結果を出し、 $\sin a \neq \sin a$ という奇妙な現象が現れる危険性がある。これを避けるためには、同じアルゴリズムによるビット単位に同一の結果を与えるスカラー型も自作し、その関数をすべて置き換える必要がある。しかし逆関数との対応や、関数間の相互の関係 ($\sin$, $\cos$ など) もあるので、1つの関数だけの精度を下げにくい。そこで“精度はライブラリと同程度としたい”という要請が

生まれる。

関数の精度に関しては、“数学的に正確な関数値を想定し、これをもっとも近くの浮動小数点数に正しく丸める (correctly round)” という基準で考えることにする。IBM の AIX 環境で言語ライブラリが提供している組込み関数の精度は非常に高く、倍精度の数学関数は、4 倍精度の関数値を倍精度に丸めた結果に対し、すべての区間（後述）で約 99.9% の一致をみる。これはこの確率で、倍精度の関数値が正しく丸められているとみなすことができる。これを関数のもつ相対誤差に換算すると 2^{-63} 以下と考えられる。これは倍精度の仮数部が 53 ビットあるもので、正しく丸められる確率は

$$P_r = 2^{-63-53} = 2^{-10} \approx 0.001$$

となって、99.9% の一致を実現できるからである¹。

この高精度は、Accurate Table Method と呼ばれるテーブル駆動型のアルゴリズムによって実現される。ここでは区間分割によって多くの縮小区間を得るが、全区間で同じ近似式を用いているとはかぎらない。したがってこれを単純にベクトル型インターフェースに変更しても、アンローリングを適用しにくい。

Accurate Table Method は、Fortran のライブラリが実行時ライブラリになって記憶容量に余裕がでてきた、1980 年代から使用されるようになった。これらの数学ライブラリでは、単精度用のルーチンの一部が倍精度で書かれるなど、精度に高い配慮がなされ、System/370 環境の VS Fortran の単精度の指数関数では 100% 正しく丸めることに成功した例もある [1]。

なお IBM の AIX 環境では、通常のスカラー型のインターフェースに比較して高速の MASS と呼ばれるベクトル型インターフェースの数学ライブラリが提供されている。

2 近似式の計算と精度の制御

本節では Accurate Table Method の概要を、“正確な丸め”を実現する方法に焦点を当てるために、平方根と対数関数を比較して述べる。

(1) 基本周期への還元

三角関数では $[-\pi/4, \pi/4]$ を基本周期にとれること

¹ 通常の倍精度の演算よりも、10 ビット程度の精度補強を備えたアルゴリズムが採用されている。

は直観的にも分かるが、平方根でも $[1, 4)$ とすれば、 $x = 4^n f$ なので、 $\sqrt{x} = 2^n \sqrt{f}$ となって擬似的な周期関数として扱える。対数関数も $[0.5, 1)$ あるいは $[1, 2)$ を基本周期とすれば、 $\log x = \log f + n \log 2$ の形にできる（以下、引数 x を基本周期に変換して得られる変数を f で表す。また $\log$ は自然対数とする）。しかし対数関数の場合は、 $x = 1$ の近傍で精度を制御しにくいので、両方の周期を半分ずつ用いて $[0.75, 1.5)$ が用いられる。

$$x = 2^n f \quad (0.75 \leq f < 1.5) \quad (1)$$

(2) 縮小区間

平方根の場合、 f を 2 のべき乗個の縮小区間に分割するには、仮数部の上位数ビットを用いる。そして各区間の中点を代表点 f_i とし、修正ニュートン法の初期値 $\sqrt{f_i}$ と f_i の逆数をテーブル化しておけば、4 回の（除算なしの）反復で IEEE 規格（倍精度）に合致した丸め方式の平方根 $\sqrt{f}$ を求められる [2]。

対数関数は $\log f = \log f_i + \log(1 + (f - f_i)/f_i)$ なので、

$$z = \frac{f - f_i}{f_i} \quad (2)$$

を縮小区間での変数として、 $\log(1 + z)$ を近似する。

$$\log f = \log f_i + \log(1 + z) \quad (3)$$

これに $n \log 2$ を加え $\log x$ が得られる。

(3) Accurate Table Method

平方根も対数も、縮小区間でテーブル参照する点は似ているが、平方根では参照値が高精度である必要はない。しかし対数では $\log f$ は参照値に補正項を加える形になっている。

$$\log(f) = \text{Tab}(f_i) + \text{Corr}(f, f_i) \quad (4)$$

これを正確に計算するためには、絶対値の大きいほうの $\log f_i$ が正確（倍精度プラス 10 数ビット）でなければならない。 $\text{Tab}(f_i)$ を 4 倍精度で用意しても、4 倍精度の演算によらないと利用できない。そこで代表点 f_i を区間の中点からわずかに振って、 $\log f_i$ が倍精度で正確に表現されるようにする。この関係を図 1 に示した。区間の中点から ϵ_i だけずれた点を f_i とし、 $\log f_i$ を 2 進数で表わしたとき、54 ビット目以降にゼロがいくつも並ぶようにするのである。

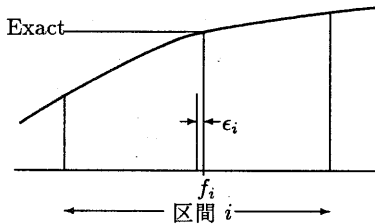


図 1: 縮小区間と代表点

このように Accurate Table Method では $\log f_i$ と, ϵ_i の近似値である ϵ_i の 2 つのテーブルが必要となる。なお ϵ_i は区間の中点 f_{mi} に比べて絶対値が十分小さな値で, f_i を 2 進数で表したとき, 54 ビット目以降に現れる。

以上, 平方根と比較して対数関数で“正確な丸め”を実現するポイントを述べた。Accurate Table Method を用いても, 依然として次の 2 点に困難は残る。

縮小区間の変数の計算 z を陽に求めなければ $\log f$ は計算できないが (式 (3)), 式 (2) には除算が含まれるため z を正確に求めることは容易ではない。

もとの周期での関数値の計算 平方根の場合の 2^n を掛ける操作は, 指数部に n をセットすることで正確に実現できるが, 対数の場合は無理数である $n \log 2$ を加える操作になるので誤差の混入が避けられない (式 (3))。

テーブルを用いずに, 基本周期全体で被近似関数を近似すると, 前者の問題は解消するが後者はより困難になる。特に基本周期の隣の周期での精度が下がる。これは 1.5 を少し越えた引数に対し, 基本周期での関数値と $\log 2$ が異符号で, 桁数が近い値になっているので, 両者を加えると桁落ちしやすいからである ($\log 1.6 = \log 0.8 + \log 2 = -0.2231 \dots + 0.6931 \dots$)。Accurate Table Method では $n \log 2$ を加える操作を分解できるのでこの問題に対処しやすい (後述)。

また, 広い区間を近似しようとする, 除算を用いた式が必要になる (例えば $\log x = \log(x-1)/(x+1)$) として, 奇関数に変換する) [3]。しかし RISC プロセッサでは, 除算は乗算に比べてかなり時間がかかるので速度の面でも不利になる²。このように Accurate

²RS/6000 の場合, 乗加算はレイテンシが 2 クロック, 依存性のない 2 つの乗加算もスルーブットが 2 クロックであるが, 除

Table Method は, 正しい丸めは実現できないものの, 精度, 速度の両面で, 基本周期全体をまとめて近似する方法よりも有利な点が多い。

3 実装例

本節でははじめに Accurate Table Method の対数関数への適用について述べ [4]³, 次にベクトル型の実装例を述べる。

3.1 スカラー型インターフェース

基本周期は, $[0.75, 1.5)$ を使い, 縮小区間の幅は $x = 1.0$ の近傍で細かく, 外側で粗くしている (区間幅として $1/256$ と $1/192$ を使い分ける)。ただし $x = 1.0$ の最近傍 $[127/128, 129/128)$ ではテーブル参照なしとしている。また区間代表点 f_i は, 式 (4) の $Tab(f_i)$ 項とはやや異なり, 次の 2 条件を満たす値としている。

- $\log f_i$ を 2 進数で表現したとき, 50 ビットめから連続する 14 ビットがゼロ
- f_i の逆数を 2 進数で表現したとき, 53 ビットめから連続する 10 ビットがゼロ

2 つめの条件は z をできるだけ正確に求めるために加えられた。

縮小区間の数 (テーブルのエントリ数) は 192 で, 各エントリに 3 つの値 ϵ_i , $\log f_i$, $1/f_i$ の近似値が格納される。近似式は, 縮小区間で 6 次多項式, 1.0 の最近傍で 7 次多項式を用いる。ただし定数項はゼロ, 1 次の項の係数は 1 である。

$$\log(1+z) = c_6 z^6 + \dots + c_2 z^2 + z \quad (5)$$

基本周期からの変換に用いる定数は $\log 2 = H + h$ と表したとき, 16 進数で $H = 3fe62e42feffa3800$, $h = 3d2ef35793c76730$ と分割し, 式 (5) の最後の $+z$ の前に h を計算する。すなわち

$$(\dots) \cdot z^2 + n \cdot h + z + \log f_i + n \cdot H \quad (6)$$

を左から右へ, 絶対値の小さいものから計算する。

算は 16 クロック程度かかる。多項式の次数に換算すると, 近似式に現れる 1 回の除算は, スカラー型では多項式の次数 7 に, ベクトル型なら次数 15 と時間的に同等になる。

³この方法は AIX 環境で実装されているものとは異なる。

以上をまとめると、式 (1), (2), (5) の z^2 まで、(6) の順に計算するが、引数がどの縮小区間に入るかで処理が異なる (if ブロックの問題)。

さらに、IEEE 浮動小数点形式では非数 (Not a Number: NaN), 無限大 (Inf), 不正規化数などの特殊な値が用意されている。これらを含め、特別な処理が要求される値に対する出力は、AIX 環境では、 $\log(\pm 0)$ は $-\text{Inf}$, $\log(\text{負数})$ は NaNQ (静止 NaN), $\log(+\text{Inf})$ は $+\text{Inf}$, $\log(\text{NaN})$ は NaNQ としている。また、最小の正の正規化数 ($2.23 \dots \times 10^{-308}$) に対する対数は $-708.396 \dots$ であるが、最小の正の不正規化数 ($4.94 \dots \times 10^{-324}$) まで考慮すると $-744.44 \dots$ に広がる。これらの処理にも if ブロックが必要になる⁴。

3.2 ベクトル型インターフェース

高速化はアンローリングによって行う。式 (5) の Horner 法による計算の最初の部分を例とすると、

$$\begin{aligned} p &= c6 * z + c5 \\ p &= p * z + c4 \end{aligned}$$

という依存性のある乗加算になるが、これに 4 ウェイのアンローリングを適用すると、連続する 4 つの引数に対して、次のように展開される [5]。

$$\begin{aligned} p0 &= c6 * z0 + c5 \\ p1 &= c6 * z1 + c5 \\ p2 &= c6 * z2 + c5 \\ p3 &= c6 * z3 + c5 \\ p0 &= p0 * z0 + c4 \\ p1 &= p1 * z1 + c4 \\ p2 &= p2 * z2 + c4 \\ p3 &= p3 * z3 + c4 \end{aligned}$$

RS/6000 のプロセッサは乗加算命令を 2 段の算術パイプライン演算器で実行し、また POWER2 機種のプロセッサは 2 つの算術パイプラインを備えるので、4 ウェイのアンローリングにより並列性を 4 にすれば、この部分に関しては最大 4 倍の速度が期待される。ところが、上述したスカラー型では、引数がどの縮小区間にあるかで処理が異なる。これを処理ごとに if ブロックでプログラミングするとアンローリングを適用できない。そこでベクトル型では全区間で同じ形の近似式を用いることが必要になる。

⁴IEEE 浮動小数点形式では 4 つの丸めモード (最近点, $+\text{Inf}$ 側, $-\text{Inf}$ 側, 切捨て) が選択でき、平方根は“数学的に正確な値”を指定されたモードで丸めた結果を返すように規定されているが、対数関数などに対しては (AIX 環境では) このようには作られていない。

なおベクトル型特有の考慮点として、次の 2 点を付記する。

テーブルサイズ テーブルのエントリ数は、多くの関数値を連続して計算するので、多くてもキャッシュミスのペナルティは相対的に小さい。

依存性 依存性があっても、アンローリングを適用することで解消されるものは避ける必要はない。

これらの点に注意して以下のアルゴリズムを用いた関数サブルーチン $dlogv$ と、あわせて同じテーブルを用いてビット単位に同一の結果を出力するスカラー型関数 $dlogs$ を作成する。

(1) 縮小区間

引数の仮数部からテーブルのインデックスを if ブロックなしで作るために、基本周期を $[0.75, 1.5)$ とした場合、縮小区間幅は、 f が 1 以下の場合は 1 以上の場合の半分になる (1 以上では区間幅が $1/512$, 1 以下では $1/1024$ の 512 エントリ)。また 1 の最近傍 $[255/256, 257/256]$ は 1 以上では 2 区間、1 以下では 4 区間が、 $f_i = 1.0$ を代表点とする。

(2) 近似式

区間 $[255/256, 257/256]$ で被近似関数を 8 次多項式によって最良近似した [6]。“8 次”は 1 の最近傍の区間での精度を、ライブラリの関数と同程度に維持するために必要になる。区間幅の異なる区間に対しては、区間幅に適した次数と係数を用いるのが最良近似の通常の使用法であるが、同一の処理とするために、すべての区間でこの 8 次多項式を用いた。

(3) テーブルの作成

縮小区間の中点 f_{mi} で 4 倍精度で求めた $\log f_{mi}$ を倍精度に丸めた値を a とすると、関数 $F(x) = \log x - a$ をゼロとする x をニュートン法で求めて f_i の近似値とする。 $a = \log f_i$, $\epsilon_i = f_{mi} - f_i$ を倍精度に丸めた ϵ_i , f_i の逆数を倍精度に丸めた値、および f_{mi} の 4 つをテーブル化する。 f_{mi} をテーブルに含めることで、1 の最近傍の区間のための if ブロックを省くことができる。すなわち 1 の近傍の 6 つの区間では、 $f_{mi} = 1$, $\epsilon_i = 0$, $\log f_i = 0$, $1/f_i = 1$ を入れておく。

(4) 加算の精度の改良

式 (6) などの加算で発生する誤差を次のように改善する。

$$\begin{aligned} c &= a + b \\ cor &= (c - a) - b \end{aligned}$$

...
 $c = c - cor$

この方法は式 (6) の場合、 a 、 b の絶対値の大小関係が既知なので適用しやすい。また依存性の強い計算であるが、アンローリングによって解消されるので、計算速度への影響は大きくない。これらの補正項は、最終結果を計算する段階で補正項同士を加えてから、最終結果に反映させる。

(5) 縮小区間の変数の算出

z を f_i の逆数の近似値から求めるために、上記の方法とニュートン法の反復を合わせて用いた。除算 $z = a/b$ を逆数 $p \approx 1/b$ を用いて $z \approx a * p$ と計算すると、正確な丸めの確率が約 20% 低下する。ニュートン法反復はこれに対し、補正項 $cor = -(b * z - a) * p$ を算出し、式 (5) の $+z$ の計算に反映させ、精度を改良する。

(6) 特殊な値の処理

アンローリングして扱われる、連続する 4 つの引数のうち 1 つでも、最大の正規化数 (7fefff...f) より大きい、正の最小の正規化数 (0010000...0) よりも小さいものがあるれば、これらはすべてスカラー型関数 $dlog$ s によって処理する。

4 精度の評価と計算速度

4.1 精度の評価

Accurate Table Method では、関数の定義域全体に平均的に変数を分散させて、平均的な精度を調べても、精度が高すぎて検定しにくい。これは基本周期の外側では関数値の絶対値が大きくなるので、基本周期よりも高い精度が得られるからである。また基本周期全体を乱数によって検定しても、99.99% 以上の点に対して正しく丸められた結果を得る。

そこで特定の比較的狭い区間を限定して調べる方法を使った。各区間で $dlogv$ による値を、Fortran ライブラリによる 4 倍精度 ($qlog$) の値と比較し、正しく丸められている点を数える。同時に Fortran ライブラリの倍精度関数 ($dlog$) も同様の方法で検定し、同程度の精度が達成されているかどうかを調べた。区間の選び方は次の 2 とおりである。

縮小区間ごとの検定 $dlogv$ の各縮小区間で一様乱数による 100000 点を計算する⁵。

⁵ 10 万点を 100 万点に増やしても検定結果にあまり変化がみ

縮小区間に依存しない検定 基本区間を 10 区間に等分割し、それぞれで一様乱数による 100000 点を計算し検定する。ここで最も成績の悪い区間をさらに 10 区間に等分割し、それぞれで検定する。この操作をもう一度行う。

はじめの検定法は、縮小区間の採り方の異なるアルゴリズムを比較するのには適当でない。2 番目の検定法は、基本区間から精度の悪そうな $(1.5 - 0.75) \times 10^{-3}$ 幅の区間を選び出し、そこで評価するものである。この区間幅は、512 区間の場合の負の縮小区間幅 $1/1028$ と同程度である。

はじめの検定法でもっとも成績の悪い区間は、1 の最近傍の外側で、99.894% であった (この区間で $dlog$ は 99.852%)。 $dlog$ のもっとも悪い成績は 99.747% だったが、 $dlog$ のほうが成績の良かった区間の数は 512 区間中 381 と半数を越えた。

2 番目の検定法で最も成績の悪い区間は (0.996124, 0.996125) で、99.851% だった ($dlog$ はこの区間で 99.850%)。これらの結果から、ほぼライブラリの組込み関数と同程度の精度が達成されたと判定した。

精度改良の効果については、例えばニュートン法反復による改良をなしにすると、最悪の区間は 3 番目の区間 (1 の最近傍の隣) に現れ、96.342% であった (改良付きではテーブルを参照する縮小区間で最悪のものは 99.988%)。縮小区間での変数 z を 4 倍精度 (除算も 4 倍精度) で求め、式 (6) の $+z + \log f_i$ を 4 倍精度で計算すると 99.998% になる。100% との差で見ると、“正しく丸められない確率は 4 倍精度計算で 0.002%、倍精度では 3.7% である。両者の間にあって、ニュートン法反復というわずかな計算時間で、この確率を 0.01% まで回復できる”のであるから、精度と性能の妥協点としては効果的といえることができる。

なお、縮小区間の数を 256 に減らした版でも、正しく丸められる点数の確率は同程度であったが、 $dlog$ のほうが成績の良かった区間の数は 256 区間中 223 とかなり多くなり、精度不足と判定される (実用上は差は微少と思われる)。

4.2 計算速度

計算速度は IBM の RISC System/6000 の 595 型 (POWER2, 135MHz) で測定した。コンパイラは XL Fortran の V5.1 を、オプション “-O3 -” されなかったもので、サンプリング数として十分と判断した。

qarch=pwr2-qstrict”で使用した。“-O3”は最適化のレベルが3を意味し、ビット単位の一致を無視した最適化を適用する。この場合“-qstrict”によって、ビット単位の一致の条件を付加しないと、精度改良のためのコードが移動され、精度が改良されない。

(1) ライブラリとの比較

表1に *dlog* を1とした計算時間の比を示した。なお計測に用いたデータには、引数に負の数や NaN などの特殊な値は含まれない。またデータ点数は 2000000 点である。

ベクトル型の *dlogv* は 1.8 倍の速度を達成している。なおこの速度は、1 関数値を約 33 クロック周期で求めている。これは除算 2 回程度の時間である。スカラー型の *dlogs* と比較すると 2.2 倍である。なお、ライブラリの 4 倍精度は、倍精度の 3.2 倍の時間で計算できている。

また、この速度は縮小区間の数を 256 に減らした版でもほぼ同様であった。すなわち、テーブルのサイズを、キャッシュミスを誘発するほど大きくしなければ、縮小区間の数は多いほうが精度的に有利で、これを節約して、式の次数を上げると、逆効果になる可能性が高い。本稿で述べた方法も、多項式の次数は 1 の最近傍で決定し、他の縮小区間はこれよりも高い精度が実現されるように選んでいる。

(2) 条件付きルーチン

変数の範囲に何らかの条件が付けられると性能が高められる。例えば、変数がすべて正の正規化された値である、ということが保証されていれば、特殊な値を考慮するための *if* ブロックを省くことができ、12% の速度向上が得られる。

もし引数の範囲が基本周期に入っていればさらに速くできるだろう（これは対数関数ではあまり考えられないが、三角関数ではありうる）。完全楕円積分を対数関数を介して近似する方法があるが [3]、この場合は引数は (0,1) 区間に限定でき、基本周期を (0.5,1] にとれる。

5 おわりに

ライブラリの組込み関数と同程度の精度をもつベクトル型インターフェースの初等数学関数を作成し、アンローリングによって高速化した。性能は多項式の計算で使用される Horner 法から直観的に期待さ

表 1: 計算時間比

関数名	時間比	クロック数	備考
<i>dlog</i>	1.00	59	倍精度 library
<i>qlog</i>	3.22	193	4 倍精度 library
<i>dlogv</i>	0.57	33	ベクトル型
<i>dlogs</i>	1.26	75	スカラー型
<i>dlogv</i>	0.50	29	特殊数処理なし
<i>dlogv</i>	0.46	27	(0,1) 区間

れるほどの差は得られなかった。これは関数のカーネルとみなせる Horner 法による計算が、関数全体の計算時間に占める割合が、テーブル駆動型では比較的少ない（テーブルを引くために引数の指数部や仮数部を取り出す処理などが重い）ことに最大の理由がある。したがって、一般のユーザがチューニングを目的に時間をかけてプログラム開発することは、よほど特殊な場合を除いては考えにくい。

“ベクトル型でライブラリの数学関数と同一の結果を保証することは難しい（性能の利得がなくなる）が、ベクトル型を前提に、スカラー型をこれに合わせることは簡単”なことも分かった。したがって、このアプローチで言語ライブラリが両者を用意し、ループ内部で繰返し呼ばれる部分を、コンパイラが自動的にベクトル型の関数サブルーチンを呼び出すようにするのが現実的な利用法と考える。

参考文献

- [1] Agarwal, R. C., Cooley, J. W., Gustavson, F. G., Shearer, J. B., Slishman, G., and Tuckerman, B.: New Scalar and Vector Elementary Functions for the IBM System/370, IBM J. Res. Develop. 30, pp. 126-144 (1986).
- [2] Markstein, P. W.: Computation of Elementary Functions on the IBM RISC System/6000 Processor, IBM J. Res. Develop. 34, pp. 111-119 (1990).
- [3] 大野 豊, 磯田和男監修: 新版 数値計算ハンドブック, オーム社, (1990).
- [4] Gal, S. and Bachelis, B.: An Accurate Elementary Mathematical Library for the IEEE Floating Point Standard, ACM Trans. Math. Softw. 17, pp. 26-45 (1991).
- [5] 寒川 光: RISC 超高速化プログラミング技法, 共立出版, (1995).
- [6] 浜田穂積: 近似式のプログラミング, 培風館, (1995).