

MalcolmのCascading Accumulatorを改良した 高速無誤差加算を含む高精度計算用Javaツールパッケージの開発

八 尋 秀 一

九州女子大学人間科学部人間発達学科、北九州市八幡西区自由ヶ丘1-1 (〒807-8586)

(2014年11月13日受付、2014年12月18日受理)

要 旨

通常の数値計算において、計算誤差の問題は大きな問題である。計算のアルゴリズムによっては誤差が拡大し、誤った計算結果を出す場合があり、結果の検証に多大な時間を要することもある。信頼できる計算結果を出すためには、高精度で素早く検証できる計算システムが要求されている。一般の様々な数値計算を含む計算システム、たとえば、分子軌道計算システム、流体計算システム、天体の運動シミュレーションなどにおいては、ますます大規模計算化しつつあり、果たして結果がどこまで信頼できるのか疑問視する声もある。マルコムのCascading Accumulator (カスケーディングアキュムレータ) から発展させた高速無誤差加算を含む高精度固定小数点方式の計算システムを紹介する。計算の基本は倍精度浮動小数点計算 (double) であり、計算にも適していると評価されているJava 言語上で構築した。

1 序論

現代の数値計算の世界において、計算誤差の問題は重要な問題として認識されている。特に、加算や減算を繰り返して総和を求めるときに発生する情報落ちや桁落ちの問題は、時に深刻な問題を引き起こすことで知られている。現代の数値計算の主流は倍精度浮動小数点計算であるが、単純化のため7桁の有効精度を持つ10進数での四捨五入計算で説明することにする。これはほとんど単精度浮動小数点計算 (有効精度約7桁) と同じと考えてよい。大きな数値同士の差を求める計算、たとえば、 $900000.8 - 900000.7 = 0.1$ の計算の場合、有効桁7桁の数値から1桁の有効精度しかない計算結果が得られる。これを桁落ちという。大きな数値と小さな数値の和を計算した場合、たとえば、 $900000.8 + 0.04440022 = 900000.84440022$ となるはずであるが、計算結果は900000.8であり、0.04440022の小さな数値情報が抜け落ちてしまう。これを情報落ちという。これらの計算の組み合わせとして、 $900000.8 + 0.04440022 + 0.04440022 + 0.04440022 - 900000.7 = 0.23320066$ となるはずであるが、計算結果は0.1となり、計算誤差は深刻な状況になってしまう。この計算は、倍精度で計算すれば何の問題もなく正しく計算できるが、約倍の桁数 (15から16桁) 以上の数値の場合に同様なことが起こる。 $10^{20} + 0.5 - 10^{20}$ の計算は単精度でも倍精度でも計算結果は0になり、正しく計算できない。

過去の数値計算の歴史において、このような問題を解決するための様々な工夫が行なわれてきた[1, 2, 3]。近年、distillation 法と呼ばれる計算法[4, 5]が登場し、最も高速と言われている。この方法は、総和 $S = \sum_{i=1}^N x_i$ の計算を行なうにあたり、 $a > b > 0$ として、

$$\begin{aligned} x &= fl(a + b) \\ y &= fl((a - x) + b) \end{aligned}$$

を求めると、

$$x + y = a + b$$

が厳密に成立することを利用する。ここで fl は、浮動小数点演算により計算することを意味し、桁落ちや情報落ちさらに桁数繰上げ時に起こる丸め誤差などを含む計算である。この無誤差変換の原理に基づいて、

$$\sum_{i=1}^N x_i = \sum_{i=1}^N x'_i = \sum_{i=1}^N x''_i = \dots$$

のように変換を繰り返すことで、 x_i の配列データのどこかに正確な加算結果が蓄積されるようになる。それゆえ、蒸留 (distillation) によって有用なデータが凝縮されるような意味合いでdistillation法と呼ばれる[3, 4, 5]。この方法の原理を理解すればすぐにわかることであるが、総和を求めるための数値データは配列で保持していなければならない。行列計算などのような配列を基本とする計算には最適の計算法と言える。しかし、配列を使用しないが、膨大なデータの加算をするような計算、たとえば、ある区間の数値積分を台形公式などのような方法で求める場合、加算すべき中間データを加算直前にその都度計算しながら加算を繰り返して総和を求めるのが普通である。distillation法を使うことは可能であるが、分割数個の配列を用意しなければならず、メモリ使用量の問題から膨大な分割数を指定できない。また、総和を求める個数があらかじめ決まっていない場合もあり、配列を利用することのメリットがあまりないこともある。このような場合、Malcolmのカスケーディングアキュムレータ法が有用である。加算データの配列を用意する必要がないので、膨大な量の加算データを処理することが可能である。近年のdistillation法の方がMalcolmの方法より高速であると報告されている[5]、この方法でも十分高速に総和を求めることができる[7]。

本研究において、Javaプログラム言語上でMalcolmのカスケーディングアキュムレータ法による加算プログラムを製作した。この方法は機種依存性が高いということから一時期敬遠されていたが、近年、McNameeがC言語上で構築した[6]。IEEE754 標準規格が制定された経緯もあり、機種依存性の問題はなくなりかけているため、その有用性が再認識されつつあると思われる。

近年のコンピュータは倍精度浮動小数点数の高速処理を目的とした専用プロセッサを内蔵するようになった。それゆえ、整数化して論理演算を繰り返すよりも、倍精度浮動小数点数のまま計算したほうが速い場合があると言われている。そこで、本研究ではさらに、カス

ケーディングアキュムレータそのものを活用して高精度計算に応用できないか検討した。そして、倍精度浮動小数点計算を基本ロジックとして動作する応用プログラムをJava言語上に構築したので報告する。

2 概念上の基本原理と実質原理

Cascading Accumulator (カスケディングアキュムレータ CA) は、図1に示すように、固定小数点2進データを適切なサイズの区切りビット数 n_d で分割し、N個のレジスター（倍精度浮動小数点数）に分配されることを概念上の基本原理とする。実際の計算は浮動小数点演算プロセッサを使用して高速処理し、最終計算結果が正しければ、計算途中の各レジスターの中身については情報の欠落が起きない範囲で概念上の原理の枠からはみ出して高速化される。それゆえ、概念上の基本原理はアキュムレータの理解の一助となるが、実際とは大きく異なる。

あるアキュムレータ型データをAで表し、それを構成するレジスターを a_i で表すと、

$$A = \sum_{i=1}^{N_{acc}} a_i \quad (1)$$

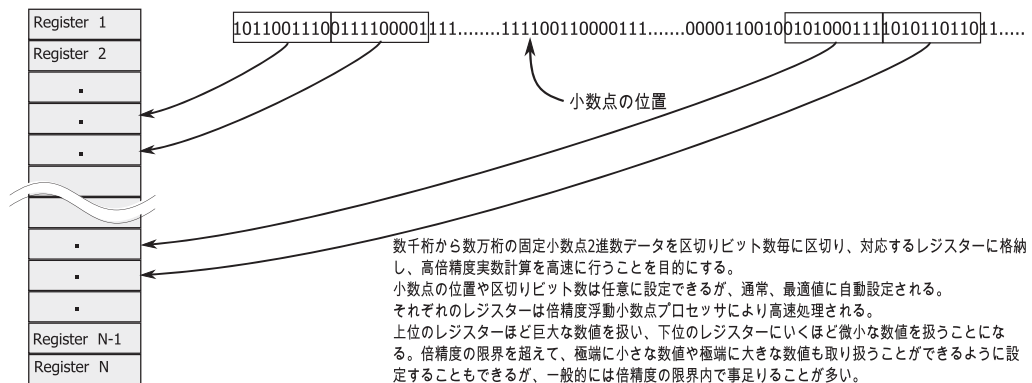
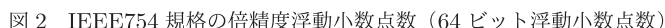


図1 カスケディングアキュムレータの概念上の基本原理

と記述できる。概念上の基本原理は常に成立しなくてもよいが、この式は常に成立するようにしているので実質原理と言える。各レジスターは符号が一致していなくてもよく、格納されているデータの大小関係も順番どおりでなくてもよい。0のビットデータが並んだ部分に対応するレジスターもあるので、数値的にzero となっているレジスターも存在している。通常、プログラム言語においてdoubleで記述される倍精度浮動小数点数は、図2のような形式で表現されている。Java言語はこのIEEE754規格に従っている。（以降は、倍精度浮動小数点数のことをdouble で表現する。）



正規化数 1.110111×2^e の場合、小数部 110111 を仮数部に、e を指数部に表すが、e + 1023 のパイアスされた値が実際に格納される。

[illegible]

が取りうる値の範囲である。指数部がすべて1（2047）の場合、無限大もしくはNaNを意味し、もはや数値を意味しないことになっている。正規化数に限定すれば、

このdouble の数値範囲を超えてアキュムレータを使用する場合、スケーリングファクター Sを設定している。

しかし、各レジスタ毎にスケーリングファクターを設定するのはメモリ上の無駄が大きいので、1023 の指数単位でスケーリングファクターをレジスタブロックごとに増減することで対応している。具体的には、

のような対応となる。現在のところ、スケーリングファクターを使用しないバージョンと使用するバージョンが存在しているが、double の数値範囲で十分であるプロジェクトが多いので、スケーリングファクターを必要とする計算問題は少ないであろう。以下の議論はスケーリングファクターを使用しない場合を想定している。

3 高速無誤差加算

3.1 高速無誤差加算とは

倍精度浮動小数点数(double)の加算を繰り返し行なうと誤差が発生するが、加算回数が少ない場合はさほど問題になることはない。doubleは10進表現で15 から16 桁の有効精度があり、加算回数が数千程度なら十分な精度の計算結果が得られるのが普通である。しかし、膨大な個数の加算を繰り返すと、しだいに誤差が堆積し、無視できなくなる場合がある。た、性質の悪い数列の計算においては、たとえ加算回数が少なくても、計算結果がまったく信じられない値を示すようになることもしばしば発生する。そのような場合、通常、多倍精度計算に切り替えて計算を行なうようにすると事態が改善されるが、計算時間はどんどん膨らみ、計算結果を得るのに時間がかかりすぎることになる。ここで紹介するのは、double型データの加算を繰り返して総和を求める場合に高速処理する無誤差加算プログラムである。Cascading Accumulator (CA) の機能の一つであり、double のデータを無誤差加算方式で加算を繰り返すため、一切の加算による誤差は発生しない。

$$S = \sum_{i=1}^N x_i \quad (6)$$

のどんな膨大なNにも対応し、無誤差加算による計算結果を得る。(ただし、計算途中の各レジスタの値が $2^{1024} \approx 10^{308}$ を超えた場合の結果は保証されないので、このような膨大な数値が発生することが想定される場合は、数値全体をスケーリングするなどの回避策が必要になる。) 、もともとのxiはdoubleなので合計値はdoubleの有効精度以上に精度が上がることはないと思われるが、 unnecessary 誤差の導入は避けられる。特に、積分計算において絶大な威力を発揮する。表1は次の積分を台形公式により計算したものである。この積分は π を与えるので、計算結果の正しさが確認できる。

$$S = \int_0^1 \frac{4}{1+x^2} dx \quad (7)$$

表1 式(7)の倍精度計算の結果 $\pi = 3.1415926535897932 \dots$

n	double	Cascading Accumulator(CA)	BigDecimal(128bits)
262144	3.1415926535873995	3.1415926535873680	3.1415926535873680
1048576	3.1415926535896660	3.1415926535896417	3.1415926535896417
4194304	3.1415926535897930	3.1415926535897840	3.1415926535897840
16777216	3.1415926535903624	3.1415926535897927	3.1415926535897927
67108864	3.1415926535890550	3.1415926535897930	3.1415926535897930
268435456	3.1415926535898740	3.1415926535897930	3.1415926535897930
1073741824	3.1415926535901244	3.1415926535897930	3.1415926535897930
4294967296	3.1415926535897670	3.1415926535897930	-
17179869184	3.1415926535895213	3.1415926535897930	-

分割数 n を増やすと計算精度は上がるはずであるが、doubleの計算では誤差が堆積し最後の数桁が一致しない。不思議なことだが、 $n=4194304$ のとき、たまたま正しい値と一致しているが、分割数をさらに増やすと最後の5桁が微妙に振動して収束しない。しかし、CAを使用すると、きれいに正しい値に収束していることがわかる。分割数を増やしすぎても計算時間が膨大になるだけであまり意味があるとも思えないが、収束状況を見て正しい結果が得られているかどうか判断できることは、大変重要な意味を持つ。最後の桁が0となって真の値と微妙に異なっているのは、64ビット浮動小数点数の有効桁を超えているためであり、この値がdoubleでの最適値となっている。ついでに、Java言語に用意されているBigDecimal(128bits)を使用したときの結果も載せている。ぴったりとCAと一致していることがわかり、CAの計算処理が正しく行なわれていることが確認できる。BigDecimalの最後の2つが欠けているのは、計算時間があまりにも膨大になり途中で計算を打ち切ったためである。

次の式の計算は、大きな値が混在し、かつ±の値が入り乱れ、さらに計算結果が小さな値になる、「性質の悪い計算式」である。

$$S = a + \sum_{i=0}^n \{f(n) - g(n, i) + g(n, n - i + 1)\} - a \quad (8)$$

ここで、 $a = 10^{20}$ 、 $f(n) = 0.12345678901234567/n$ 、 $g(n, i) = (4.5678901234567890/n) \times i$ である。計算が正しければ、 n の値に関係なく計算結果は常に0.12345678901234567となるようになっている。表2はこの式を順序どおりに計算した場合の計算結果を示している。

表2 式(8)の倍精度計算の結果

n	double	CA	BigDecimal(128bits)
1	0.0	0.12345678901234566	0.12345678901230000
4	0.0	0.12345678901234566	0.12345678901240000
16	0.0	0.12345678901234566	0.12345678901280000
64	0.0	0.12345678901234566	0.12345678901120000
256	0.0	0.12345678901234566	0.12345678901770000
1024	0.0	0.12345678901234566	0.1234567899190000
4096	0.0	0.12345678901234566	0.1234567889010000
4294967296	-	0.12345678901234566	-

doubleは、最初の段階から計算結果は0で、このような計算にはまったく対応できていない。BigDecimal(128bits) (4倍精度に相当)はある程度対応できているが、誤差が非常に大きい。それに関わらず、CAはこのような性質の悪い計算式でも正確に計算できていることがわかる。CAの計算時間はたかだかdoubleの7.8倍であり、BigDecimalの約100倍に比べるとたいへん実用的である。

CAを使った無誤差加算のJavaプログラムは、以下のsample 1、

```
program sample 1.
class sample1 {

    static double func(long i,double x, ....){ // user defined function
    .....
    return ..;
    }

    public static void main(String args[]){
        long N=100000000;
        double x;
        long i;
        Accumulator a = new Accumulator(); // generates an Accumulator obj.
        for(i=0;i<N;i++) {
            x=func(i,...);
            a.add(x);
        }
        double S = a.getSum(); // extracts a sum_value from Accumulator a.
        System.out.println(" Sum = "+S+" Num = "+N);
    }
}
```

のようになる。このプログラムはユーザ関数が未定義なのでこのままでは実行できないが、intやdoubleの型宣言と同じようにAccumulatorクラスオブジェクトを生成し、用意されているメソッドを呼び出すだけで容易にCAが利用できるようになっている。CAには他にも様々なメソッドが用意されている。

3.2 無誤差加算の原理

Malcolmのカスケディングアキュムレータ法[2]は有用な計算法としてあったが、データの指数部を直接調べることが必要なためコンピュータの機種依存性が高いことからしばらく敬遠されていた。IEEE754国際標準規格が制定され、ほとんどのコンピュータがこの規格に準拠するようになってから、その有用性を認めようとする動きがあり、McNameelはMalcolmの方法をC言語で作成し、他の方法より最も高速であることを示した[5]。しかし、公表されたC プログラムは完全ではなかったため、yahiroはその改良版をJava言語で作成し、無誤差加算を実現した[6]。今回作成したものは、このMalcolmの流れを汲む計算法を改

良し、誤差が全く発生しない無誤差加算をJava 言語で実現したものである。そして、さらにこのCascadingAccumulatorそのものを高倍精度固定小数点数として他の計算に活用できるように拡張を行っている。

さて、無誤差加算の原理を簡単に説明する。 $S = \sum_{i=1}^N x_i$ における右辺の x_i が加算されるデータであり、double型の浮動小数点で与えられる。 $\mathbf{A} = \sum_{i=1}^{N_{acc}} a_i$ は加算に使用するアキュムレータを表し、 N_{acc} 個のレジスタ a_i で構成されるアキュムレータAに x_i が次々と加算され、加算結果がアキュムレータAに蓄積される。各レジスタ a_i はdouble型の浮動小数点であるが、加算時には x_i は2個のデータに分割され、それぞれの指数部の値から対応するレジスタを決定し、加算処理される。このとき、無誤差加算が保障された形で加算が行われる。加算したレジスタの結果がある一定値以上になると上位レジスタにデータを移動し、無誤差加算が常年实现するよう工夫がしてある。まず、double型のデータ x の仮数部52ビットを上26ビット下26ビットに分割する関数Splitを以下のように定義する。(実質的には53ビット情報の分割なので27ビットと26ビット情報に分割され、それぞれ先頭1ビットが省略されて仮数部がセットされる。)

関数 Split: $(x_{upper}, x_{lower}) = Split(x)$:

$x_{upper} = x \text{ AND "FFFFFFFFFC0000000";}$

$x_{lower} = x - x_{upper};$

プログラムコードは単にビット操作のANDを用いて2個のデータに分離する。しかし、実際にはJavaは直接のビット操作を許していないので、他の代用方法で行っている。

無誤差加算のメソッドaddは、以下のようなロジックで処理される。

メソッド add: $\mathbf{A} = \mathbf{A} + x \quad \equiv \mathbf{A}.add(x)$

$(x_{upper}, x_{lower}) = Split(x);$

$i = Exponent_of_x_{upper}/n_d; j = Exponent_of_x_{lower}/n_d;$

$a_i = a_i + x_{upper}; a_j = a_j + x_{lower};$

$k = Exponent_of_a_i/n_d; l = Exponent_of_a_j/n_d;$

repeat while ($k > i + \text{gap}$)

$(p, q) = Split(a_i);$

$a_k = a_k + p; a_i = q;$

$i = k; k = Exponent_of_a_k/n_d;$

repeat while ($l > j + \text{gap}$)

$(p, q) = Split(a_j);$

$a_l = a_l + p; a_j = q;$

$j = l; l = Exponent_of_a_l/n_d;$

Java言語はC言語と異なり、浮動小数点計算の計算ロジックは丸め処理時に最近接値にビット値を割り当てる手法のみがサポートされているので、計算結果の仮数部最後のビットがC言語と異なる場合があります、従来のdistillation法で採用されている計算ロジックが使えない。それゆえ、本研究で採用したロジックはdistillation法と一部異なり、直接ビット情報を操作するロジックを含んでいる。しかし、それでも十分高速性を保った計算が可能となっている。本研究で開発したJava高精度計算ツールパッケージの詳しい解説は、別の論文で言及する予定である。また、 $\sum_{i=1}^N x_i y_i$ のベクトルの内積 (dot product) を求める無誤差計算も行えるようにしている。

4 CAを使った高精度計算の例

CAには様々な演算機能があり、それらを使った計算例のいくつかを紹介する。次のように指数関数は級数展開で、

$$\exp(x) = \sum_{k=0}^{\infty} x^k / k! \quad (9)$$

となることは、数学では常識であるが、実際に計算できるのかとなると別問題である。 x が小さい場合は容易に収束するが、 x が大きくなると急激に膨大な数値が計算途中に発生するようになり、すぐに計算不能になる。

$$f_n(x) = \sum_{k=0}^n x^k / k! \quad (10)$$

として、 n を大きくすれば $\exp(x)$ に近づくはずであるが、実際にどうなるのかdoubleとCAを使った計算で比較すると、表3のようになる。 n を増やすと途中までは同じ計算結果を示しているが、 $n=1024$ のあたりから計算結果が大きく異なるようになり、doubleの計算結果は極端に大きなとんでもない値のまま変化しなくなったが、CAは $n=4096$ 付近で正しい解に収束していることがわかる。これは、極端に大きな数値と小さな数値の加算時と極端に大きな数値同士の差をとる時の情報落ちや桁落ちの問題が大きな要因となり、doubleでは正しい計算値に収束しない。CAにも限界があり、 x の絶対値が800を超えたあたりから正しい答えを出さなくなる。一般的には、 $x = n_x + \delta x$ と置き、 $\exp(x) = e^{n_x} \times e^{\delta x}$ として、 $e^{\delta x}$ を7から8項の級数展開で計算するのが普通であり、上述のような計算は行わない。しかし、物理や数学の世界では級数展開は大変重要な数学的アイテムであり、様々な数理論において活用されているので、CAは様々な数理論の検証において強力な力を発揮すると思われる。もちろん、他の様々な高精度計算プログラムが世の中存在しているので、CAが特にと言うわけではないが、Java言語を活用している多くの利用者にとっては、容易に使えるという点で有用であろう。sinやcosなどの様々な関数の級数展開も同様に計算できるが、計算限界も同様である。

表3 $f_n(x) = \sum_{k=0}^n x^k/k!$ の計算 : $x = -600$, $\exp(-600) = 2.6503965530043108 \times 10^{-261}$

n	double	CA
4	-3.5820599000000000e+07	-3.5820599000000000e+07
8	-5.4901283501063130e+15	-5.4901283501063130e+15
16	-3.5077448931788916e+29	-3.5077448931788920e+29
32	-1.5337513922487072e+52	-1.5337513922487070e+52
64	-4.8180040714491170e+87	-4.8180040714491180e+87
128	-1.8313482293286640e+139	-1.8313482293286635e+139
256	-5.6162598336649170e+203	-5.6162598336649100e+203
512	-3.4322212850586140e+255	-3.4322212850586055e+255
1024	-9.2042661989186100e+242	-7.8128904660818660e+204
2048	-9.2042661989186100e+242	-2.0835021409819763e-205
4096	-9.2042661989186100e+242	2.6503965530043108e-261
8192	-9.2042661989186100e+242	2.6503965530043108e-261
16384	-9.2042661989186100e+242	2.6503965530043108e-261

$$\sin x = \sum_{k=0}^{\infty} (-1)^k x^{2k+1}/(2k+1)!$$

$$\cos x = \sum_{k=0}^{\infty} (-1)^k x^{2k}/(2k)!$$

であり、

$$f_n(x) = \sum_{k=0}^n (-1)^k x^{2k+1}/(2k+1)!$$

$$g_n(x) = \sum_{k=0}^n (-1)^k x^{2k}/(2k)!$$

とすると、

$$F_n(x) = 1.0 - \frac{x^2}{2n(2n+1)}$$

$$F_j(x) = 1.0 - \frac{x^2 F_{j+1}(x)}{2j(2j+1)}$$

$$f_n(x) = x F_1(x)$$

の後退漸化式を使うと便利である。これは、級数展開の数値計算において一般的に使われているホーナー法を漸化式にしたものである。同様に、

$$G_n(x) = 1.0 - \frac{x^2}{2n(2n-1)}$$

$$G_j(x) = 1.0 - \frac{x^2 G_{j+1}(x)}{2j(2j-1)}$$

$$g_n(x) = G_1(x)$$

である。これらの漸化式を使うと、 x の絶対値が1400ぐらいまでCAによる計算が可能となる。 x の値によって最適の n 値が変わり、おおよそ、 $n = 1.4|x| + 30$ とするとよいようである。 \sin, \cos は周期関数なのでこのような計算は実際的ではないが、級数展開をともに計算することがかなり可能であることを示した。

5 各種組み込みメソッド（組み込み関数）

各種組み込みメソッドは生成したアキュムレータオブジェクトと密接に関係している。アキュムレータ同士の和や積が登場すると混乱をきたすため、それらを明確に区別するためにthis表現を使用しているが、Aをこの生成したインスタンスthisに対応したアキュムレータオブジェクトとしている。

$$\mathbf{A} = \mathbf{A} + x$$

は、add(double x) メソッドの働きを示している。また、

$$\mathbf{A} = \mathbf{A} + \mathbf{X} \times \mathbf{Y}$$

は、muladd(Accumulator X, Accumulator Y) メソッドの働きを表している。

5.1 add(double x)

double型の実数値 x を無誤差加算方式でアキュムレータthisに加算する。 $\sum_{i=1}^N x_i$ の無誤差計算をサポートする。加算する個数に限界は定めていないが、総和が限界値(2^{1024}) を超えた場合は結果は保証されない。

使い方例：

```
Accumulator a=new Accumulator();
a.init();
a.add(10.25678901112);
.....
double sum=a.getSum();
```

5.2 muladd(double x, double y)

double 型の実数値 x, y の積をとってアキュムレータthis に無誤差加算する。 $\sum_{i=1}^N x_i y_i$ のベクトルの内積を求める無誤差計算をサポートする。積をとる場合、64 ビットの限界を超

えて情報の欠落が起きないようにしてあるので、無誤差積和計算を保障する。総和の限界はaddと同じである。

使い方例：

```
Accumulator a=new Accumulator();
a.init();
a.muladd(10.25678901112,0.0012305903812);
.....
double sum=a.getSum();
```

5.3 mul(double x)

double型の実数xとアキュムレータthisの積をとり、アキュムレータthisに保存する。

$$\mathbf{A} = x \times \mathbf{A}$$

5.4 div(int m)

int型の整数mでアキュムレータthisを割り、アキュムレータthisに保存する。

$$\mathbf{A} = \mathbf{A}/m$$

5.5 init()

アキュムレータthis のすべてのレジスタを0に初期化する。

5.6 double getSum()

アキュムレータthis のすべてのレジスタの合計値を求め、double 型の実数値を返す。

5.7 add(Accumulator X)

アキュムレータX をアキュムレータthis に加算する。 $\mathbf{A} = \mathbf{A} + \mathbf{X}$

5.8 muladd(Accumulator X, Accumulator Y)

アキュムレータXとアキュムレータYの積をとり、アキュムレータthisに保存する。

$$\mathbf{A} = \mathbf{A} + \mathbf{X} \times \mathbf{Y}$$

5.9 その他のメソッド

現在、整数による除算はあるが、 \mathbf{X}/\mathbf{Y} のようなアキュムレータ同士の除算が未完成となっている。これが完成すれば、加減乗除の基本演算はすべて揃うことになる。現在、計算ロジックの設計は完了し、あとはプログラム製作のみであるが、完成は数ヵ月後の見込みである。また、他の有用なメソッドも順次揃えていく予定にしている。

参考文献

- [1] D. R. Ross, Communications of the ACM, 8 (1965), pp.32-33
- [2] M.A. Malcolm , Comm. Ass. Comp. Math., 14(1971), pp. 731-736.
- [3] N. J. Higham, Accuracy and Stability of Numerical Algorithms 2nd ed. , SIAM, 2002, 4 章でsum-mation の歴史について言及している。
- [4] J. Demmel and Y. Hida,, SIAM J. Sci. Comput., 25 (2003), pp. 1214-1248.
- [5] S. M. Rump, T. Ogita, and S. Oishi, SIAM J. Sci. Comput., 31(2008), pp.189-224.
- [6] J. M. McNamee, ACM SIGSAM Bulletin, 38(2004), pp. 1-7.
- [7] 八尋秀一, 情報処理学会創立50周年記念(第72回)全国大会講演論文集 1 巻(2010), pp.37-38.

Construction of a Java tool package for high precision calculations including error-free summation devised by improving Malcolm's Cascading Accumulator

Shuichi YAHIRO

Department of Human Development, Faculty of Humanities,
Kyushu Women's University.
1-1 Jiyugaoka, Yahatanishi-ku, Kitakyushu-shi, 807-8586, Japan

Received November 13, 2014 ; Accepted December 18, 2014

Abstract

It is a big problem that arithmetic errors must occur in scientific calculations for various fields. Sometimes these errors enlarge themselves at some calculation algorithms, and cause important problems, and may result in enormous time consuming for verification. It is required for many scientists, the verification system which calculate the arithmetic equation of concern exactly and rapidly. For example, molecular orbital calculation system, fluid calculation system and the simulation of astrological phenomenon are increasing their sizes of calculation and the trustability of them may not be guaranteed. I introduce the high precision calculation system which is devised by improving Malcolm's cascading accumulator and produce very high precision computations of fixed point arithmetic numbers. This system is constructed by the basic computations of double precision floating points and developed on Java language regarded generally to be appropriate for scientific calculations.