Java言語による誤差伝搬の起きない加算処理アルゴリズム

八尋　秀一
九州女子大学

1．　序論

n個の浮動小数点数の和の計算 $\sum_{i=1}^{n} x_i$ において丸め誤差が伝播し、時として悲惨な計算結果になることは良く知られ、計算精度を上げるための様々な方法が考案されている。Higham は、それまでの方法論を詳細に調査し、Malcolmによる cascading accumulator 法と Kahan の加算公式を利用した distillation 法を除き、その他の方法は過酷な幾つかのテスト計算に耐えられないことを示した。また、Malcolm の方法はマシン依存性が高いという欠点があることを示唆した。その後、distillation 法の数多くの研究が発表され、完全に加算時の誤差が伝播しない方法が確立したかのように思える。この方法は、$\sum_{i=1}^{n} x_i y_i$ 型の内積に対しても適用され、パラレルコンピュータとの相性も良いとされている。しかしながら、この方法の問題点は加算されるデータを配列で保持しておく必要があることである。nを大きくすればするほどコンピュータのメモリ資源を浪費し、プログラミングも容易ではない。

近年、IEEE 754 規格に対応する CPU や言語が普及してきた。それゆえ、浮動小数点数の内部規格が統一され、マシン依存の浮動小数点計算はなくなりつつあると思われる。特に Java プログラムは Java 仮想コンピュータ（JVM）で動作するので、プログラムのマシン依存はほとんどなく、さらに IEEE 754 規格を遵守している。そのようなことから、過去の Fortran の世界で一時期はマシン依存性が高すぎると敬遠されていた Malcolm の方法が見直される時期であるように思われる。McNaee は、このことを察知してか、Malcolm の方法が最も高速、高精度であると主張し、この方法の C 言語コードを近年公表している。

Accurate Floating-Point Summation Easily Available to Java Programming
Shuichi Yahiro , Kyushu Women's University

今回の研究のきっかけは、「簡単な方法で和の計算誤差を少なくできないの？」という単純な自分自身への問いかけであった。誤差発生の主要原因は、大きな数に小さな数を加算するときに起きる。小さな数の浮動小数点仮数部（Mantissa）の下位ビットの情報が消えてしまうことが原因である。ならば、同じ大きさのデータ同士を加算することで誤差は少なくできるはずである。小さいものは小さいもの同士、大きなものは大きなもの同士の加算を繰り返せばよい。配列で加算データを保持する必要もなく、簡単にそれを実現する方法はと考えたときに思いついたのが、Malcolm の cascading accumulator 法に大変よく似た方法であった。

Malcolm の方法は、単精度の加算データを倍精度に型変換し、用意された 60 個ほどの倍精度 accumulator のどれかに加算することにより、1ビットの情報も紛失せずに加算を繰り返すことができる。なぜなら、単精度から型変換された倍精度データの仮数部下位ビットに 30 個の 0 が並び、よほどのことがないかぎり加算による丸め誤差は発生しない。accumulator のどれに加算するかを決めるのは加算データの浮動小数点指数部（Exponent）の値であり、加算の繰返し数が相当大きなある一定値を超えると上位 accumulator へのデータ移動が行なわれ、加算による情報損失が起きないよう配慮している。しかし、我々が思いついたのは、複数の accumulator を使うのは同じであるが、加算データと同じ精度の accumulator を用意する方法である。それぞれの accumulator には値域が厳密に設定され、加算後値域を超えると一つ上の accumulator へ加算によりデータ移動を行ない、さらにそれも値域を超えるとまた一つ上の accumulator へと加算を繰り返していくものである。そうすることにより、ほぼ同じ大きさの値同士の加算を実現する。仮数部の下位2，3ビットの誤差は無視するという大雑把なものであるが、Java 言語での実装は簡単である。この方

法は誤差がいくらか発生するという点ではMalcolmの方法より劣るが、加算データの倍の精度の計算を必要としないことは大きな利点である。特にJava言語では4倍精度はサポートされておらず、非常に遅く、使いづらい任意精度のBigDecimalが用意されているのみである。

いくつかのテスト計算を行なった結果、全ての加算データが正値の場合はたいへん満足のいく結果であった。しかし、正負両方のデータが混在するデータの和を計算すると、誤差は飛躍的に増大した。しかしながら、この方法はある種の定積分で被積分関数が常に正または負とわかっている場合の数値積分などの利用に関しては十分活用できる。そこで、このプログラムを簡易版ExactSumとして発表することにした。ただし、使用法は正値のみまたはその反対に限定される。

誤差伝播がどのようにして起こるのか、幾つかの計算結果を丹念に調査した結果から次のように言える。「誤差が増大する原因は加算時の丸めによる仮数部下位ビット情報の消失と減算により発生する情報落ちが相互に影響し合い拡大していく。」とすると、例え仮数部末端の1ビットでも無視できないことになる。そこで、上記の方法に改良を行ない、確実に加算を行なう方法はないであろうかと考えてみることにした。そして、浮動小数点仮数部を上位ビット、下位ビットに分割することで問題解決を図ることを考案した。最初の方法をExactSum、改良版をSExactSumとして、プログラムを作成した。

2. 倍精度計算による検証

初級数値積分において、台形法やシンプソン法は一般常識であるが、分割数を増やしすぎると通常あまり良くない結果となる。表1は台形法により以下の定積分を倍精度で行なった場合を示したものである。

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

表1．式（1）の台形法による倍精度計算の結果

π =3.1415926535897932

n	double	ExactSum	SExactSum
1048576	3.1415926535896660	3.1415926535896417	3.1415926535896417
4194304	3.1415926535897930	3.1415926535897840	3.1415926535897840
16777216	3.1415926535903624	3.1415926535897922	3.1415926535897927
67108864	3.1415926535890550	3.1415926535897936	3.1415926535897930
268435456	3.1415926535898740	3.1415926535897930	3.1415926535897930
1073741824	3.1415926535901244	3.1415926535897930	3.1415926535897930
4294967296	3.1415926535897670	3.1415926535897936	3.1415926535897930
17179869184	3.1415926535895213	3.1415926535897930	3.1415926535897930

積分値の正しい値はπである。

表中のdoubleは通常の倍精度計算の結果を示したものである。nが4194304のとき、倍精度限界の正解値と偶然の一致を見せているが、以降は変動しながら誤差が拡大している。ExactSum、SExactSum共に、nの増大と共に正解値に近づき、以降はほぼ倍精度限界の正解値で一定となった。

表1の結果は、全ての要素が正であるため加算のみであり、一般的な$\sum_{i=1}^{n} x_i$の計算に対応していない。そこで、加算、減算を両方含んだ系の計算を行なうことにした。次式は、誤差が簡単に拡大しそうな「性質の悪い計算式」である。

$$S=a+\sum_{i=1}^{n}\{f(i)-g(i)+g(n-i+1)\}-a \quad (2)$$

ここで、

a =1e20 , f(i) = 0.12345678901234567/ n,

g(i) = (4.5678901234567890/ n) × i

である。nの値をどのように変えてもSの理論値は0.12345678901234567となるようになっている。計算は式（2）の示す順序のとおりに行なった。計算結果を表2に示す。

表2．式（2）の倍精度での計算

n	double	ExactSum	SExactSum
1048576	0.0	0.12345678913388270	0.12345678901234566
4194304	0.0	0.12345679011195898	0.12345678901234566
16777216	0.0	0.12345678978314195	0.12345678901234566
67108864	0.0	0.12345678717839523	0.12345678901234566
268435456	0.0	0.12345677864702509	0.12345678901234566
1073741824	0.0	0.12345685933418807	0.12345678901234566

通常の計算doubleでは全て0となってしまったが、SExactSumは常に正しい結果を出している。ExactSumはnが大きくなるにつれ誤差が拡大している。