

浮動小数点演算における精度見積りアルゴリズムとその評価

鈴木 弘

都立航空高専 電子工学科

大岩 元

慶應大学 環境情報学部

浮動小数点演算には必ず誤差が含まれるが、一般には、計算結果にどれだけ誤差が含まれているかをユーザーは知ることができない。そこで、計算過程で桁落ちを追跡し、最終的に桁落ちの影響を受けていない桁数（有効桁数）をユーザーに知らせるシステムを作成した。そのための桁落ち追跡アルゴリズムについて述べ、その有効性を評価する。

具体的には、C++でクラスとオペレータオーバーロードの機能を使い、実数（浮動小数点）変数から指数部を取り出し、演算の前後での指数部の変化を見ることにより桁落ちを判断し、さらに演算前の有効桁数を考慮し演算後の有効桁数を導く。これを最終的な計算結果を得るまで常に監視し続ける。利用者はプログラムの先頭で変数宣言するだけで、この精度見積りを使用できる。

A Program for Estimating Catastrophic Cancellation of Floating-Point Arithmetics

Hiroshi Suzuki

Department of Electronics Engineering

Tokyo Metropolitan College of Aeronautical Engineering

Hajime Ohiwa

Department of Environmental Information

Keio University

In the floating-point arithmetics, errors such as round-off, cancellation are inevitably included but user cannot know how much error is included in his computation. We have developed a system that monitors every cancellation and keep track of significant digits of every floating-point value.

Concretely, this system takes out the characteristic in a floating-point representation from a real number (floating-point) variable, judge the cancellation by change of characteristic, and introduce a significant number after an arithmetic operation. This number is used for the subsequent operations for obtaining a significant digits. We implement this system using a function of the class of C++ and the operator overload. A user can use this precision estimate system by declaring variables as such at the front of a program.

1. まえがき

計算機の処理速度の発展により、大型計算機やスーパーコンピュータを使った数値計算が盛んに行われている。

しかし、浮動小数点演算には、丸め、情報落ち、桁落ちなどの誤差を含み、従って、数値計算の結果にも誤差を含むことになる。場合によっては、演算を繰り返すうちに誤差が蓄積し、実際とは全く違った結果が出ていることもあり、利用者はそれを知らずに使っている可能性もある。また、計算結果の精度を上げるためには、倍精度や四倍精度など仮数部の大きい変数を使ったり、数式処理ソフトを使うことが考えられる。しかし、数式処理では表現できない問題もあり、プログラムによる方法では、どれくらいの精度で計算すれば良いかはわからず、さらに一般的には倍精度までしかサポートしていないシステムが多い。

このように浮動小数点演算には必ず誤差が含まれるが、一般には、どれだけ誤差が含まれているかをユーザーは知ることはできない。そこで誤差そのものではないが、最も大きな誤差が生じる可能性のある桁落ちに着目し、計算過程で桁落ちを追跡し、最終的に桁落ちの影響を受けていない桁数をユーザーに知らせるシステムを作成した。

初めに浮動小数点演算をソフト的にシミュレートし桁落ちを数えるという方法でシステムを作成した。これは仮数部長を自由に変更できるようにしてあり、精度を上げて計算できる。しかしソフトウェアシミュレーションのため処理速度が非常に遅いので実用的ではなかった。次に処理速度を上げ実用に耐えるシステムを目指し、メモリ内部の実数(浮動小数点)変数から指数部を取り出し、演算の前後での指数部の変化を見ることにより桁落ちを判断する方法でシステムを作成した。これは仮数部長は変更できないが、前者に比べて処理速度は速くなり、実用に耐えうるものとなった。

本論文は主に後者のメモリ読み取り方式について今回作成した計算精度見積りシステムとそのための桁落ち追跡アルゴリズムについて述べる。

2. 信用できる値を示すために

浮動小数点演算の3つの誤差(桁落ち、情報落ち、丸め)を演算が行われるたびに正確に把握し、計算結果が得られるまで累積していけば計算結果における誤差がわかるはずであるが、これは不可能に近い。

そこで桁落ちのみを考慮し、情報落ち・丸めについては考慮しないことにした。これは、桁落ちが仮数部の上位の桁に起こりうる誤差であるのに対し、情報落ち・丸めは仮数部の下位の桁で起こる誤差であり、誤差率は小さいと考えたからである。

桁落ちの誤差の累積の考え方であるが、桁落ちの影響を受けていない「有効桁」という考え方を導入した。まず、一回毎の各演算について桁落ちした桁数を数える。さらに演算の対象となった数が演算前に受けている桁落ちの影響を考慮し、演算後の数の桁落ちの影響を受けていない桁を求める。この操作を演算が行われる毎に繰り返し、最終的な計算結果の桁落ちの影響を受けていない桁数(これを「有効桁数」と定義する。)を求める。(3. 3節に詳しく述べる)この最終的な有効桁数が、計算結果がどれだけ誤差を含むかの指針になる。

情報落ちと丸めの誤差の影響については、誤差の追跡に計算時間を費やす割には、含まれる誤差が小さく、誤差の追跡にそれほど効果が無かった。

情報落ちと丸めの誤差の影響については、誤差の追跡に計算時間を費やす割には、含まれる誤差が小さく、誤差の追跡にそれほど効果が無かった。

3. 桁落ちの判断方法及び有効桁追跡の実現

3-1. 桁落ちの判断方法

演算の結果桁落ちが生じた場合は、仮数部は正規化され、結果的には指数部が減少する。このこ

とから、演算の前後での指数部の変化を見れば、何桁桁落ちしたかがわかる。

例えば、 $1.5625 - 1.5 = 0.0625$ の場合を考えてみると、1.5625、1.5 は2進数でそれぞれ以下のような仮数部と指数部からなる。(仮数部の(1)は隠れビット)

	仮数部	指数部
1.5625	(1) 10010000...	0111111111
1.5	(1) 10000000...	0111111111

減算した結果の 0.0625 は以下のような仮数部、指数部となる。

0.0625	(1) 00000000...	01111111011
--------	-----------------	-------------

従って指数部を比べることによって以下のように何桁桁落ちしたかが判断できる。

演算前の指数部	0111111111
- 演算後の指数部	01111111011
= 桁落ち	100 (4桁)

3-2. 指数部の読みとり

例えば以下のような変数宣言のプログラムを実行すると計算機のメモリ上に変数a, b, cのための領域が倍精度浮動小数点数なので、図のように8バイトずつ作られ、そのメモリ領域を使って計算される。

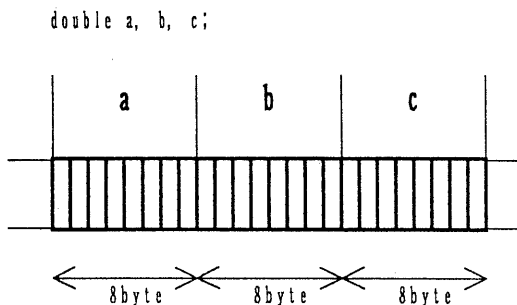


図-1 計算機のメモリ上の変数領域の一部

さらに変数1つの8バイトは、図のように符号1bit, 指数部11bit, 仮数部52bitからなる。

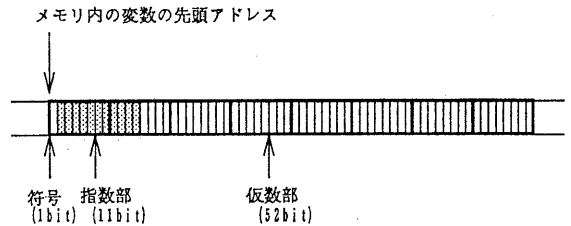


図-2 浮動小数点表現

プログラミング言語C++では、以下のようなプログラムで、変数の格納アドレスを求めることができ、さらに変数の値を普通の実数としての値ではなく、8バイトのビットイメージとしてそのまま取り出すことができる。またさらにそのビットイメージの中から指数部の部分を取り出すことももちろん可能である。

例:

```
double x;           // 実数変数 x
int xpart;          // 変数 x の指数部
unsigned long int *xp; // 変数のアドレスへのポインタ

xp = (unsigned long int *) &x;
// ポインタへの代入
xpart = (*xp >> 20) &x7fff;
// 指数部を取り出す
```

このようにして、浮動小数点変数の指数部のみを取り出すことができる。従って演算前に取り出した指数部と演算後に取り出した指数部を比較することによって桁落ちを判断できる。

3-3. 桁落ちの影響を受けていない桁数(有効桁数)の計算法

仮数部を表すビット列において、演算による桁落ちの影響を受けていない部分のビット長を「有効桁数」と定義する。

各変数の有効桁数は演算が行われる毎に桁落ちの影響を受けて変化する。これを毎回の演算毎に有効桁を計算・追跡すれば、その変数の最終的な

有効桁数がわかる。この有効桁数が計算精度に近い値になると考えられる。

浮動小数点演算の誤差には、丸め、情報落ち、桁落ちがあるが、今回は桁落ちのみを考慮した。すなわち、有効桁数が、そのまま計算精度を正確に示すわけではない。

さて、有効桁数の計算方法であるが、加減算の場合は、 $z = x - y$ として、 x, y の大きさによって2つのパターンに分かれる。

1つめは、指数部の大きさが同じ場合であり、この場合は x と y の有効桁数の小さいほうから桁落ちの桁数を引くことによって演算後の有効桁数を得る。

2つめは、指数部の大きさが異なる場合であり、この場合は、桁合わせが行われるので、指数部の小さいほうの有効桁数に桁合わせが行われる桁数を加えた上で、有効桁数の小さいほうから桁落ちの桁数(ただし、指数部の大きいほうと比較する)を引く。

乗除算の場合は、桁落ちは起こらないので、2数の有効桁数の少ないほうを新しい有効桁とする。

図3、図4に2数の演算における有効桁数の計算方法を図示する。

プログラムの実行経過で有効桁数がどのように変化するかを具体例で示す。以下のようなプログラムの場合、

```
a = 1.5625;
b = 1.5;
c = a - b;    // 0.0625
d = a - c;    // 1.5
e = a - d;    // 0.0625
```

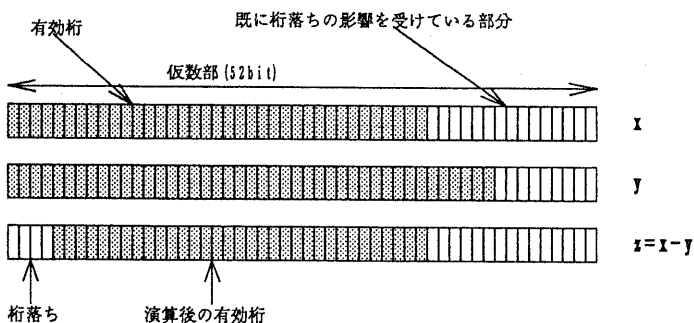


図3 指数部が同じ場合

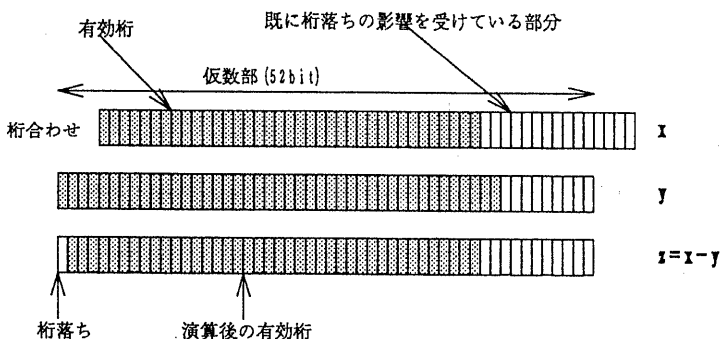


図4 指数部が異なる場合

$e = 16.0 + e \quad // \quad 16.0625$

a, b は定数の代入なので、仮数部の53桁有効桁数とする。減算された c は、4桁桁落ちするので有効桁数は49、 d は桁落ちは起こらないが c が49桁なので、49桁となる。次の e は d が49桁であり、さらに4桁桁落ちするので、有効桁数は45となる。さらにその次の e は仮数部の先頭に4桁追加され右に4桁シフトされるかたちになるので、有効桁数は4桁増え49となる。

このように演算前の有効桁数及び演算毎の桁落ちによって有効桁数が変化していく。上記の例の場合には、仮数部の下位の桁は0なのですべて正しい結果となっているが、実際には正しくない場合もあり、それが誤差となる。

3-4. 実現方法

今回の浮動小数点演算精度「お知らせ」システムはプログラミング言語C++を使って実現した。

C++のオペレータオーバーロードを使えば、例えば加算の演算子 + に対して、加算以外のことも定義できる。すなわち、加算と同時に演算前後の指数部を比較する機能を持たせれば、桁落ちを常に監視できるわけである。さらに、上記の指数部の変化から有効桁数を記録する処理を加え、そのクラスを **Double** とした。クラス **Double** は実数型の数値を表すメンバと、有効桁数を表すメンバ、及び四則演算のための関数や有効桁数を計算するためのメンバ関数からなる。

クラス **Double** を使ったプログラム例を以下に示す。ヘッダファイルのインクルードと変数の宣言部が違っているだけである。従って普通の C, C++ のプログラムがほとんど変更なしに、桁落ちの状況を見ることができる。

Double を使ったプログラム例

普通の C	Double の例
<code>#include <iostream.h></code>	<code>#include <Double.h></code>
<code>main ()</code>	<code>main ()</code>
<code>{</code>	<code>{</code>
<code>double a, b, c;</code>	<code>Double a, b, c;</code>
<code>a = 0.625;</code>	<code>a = 0.625;</code>
<code>b = 0.5;</code>	<code>b = 0.5;</code>
<code>c = a - b;</code>	<code>c = a - b;</code>
<code>cout << c;</code>	<code>cout << c;</code>
<code>}</code>	<code>}</code>

上記の例では **Double** の場合、変数 **c** は計算結果の 0.125 と有効桁数 51 (53 桁-桁落ち 2 桁) を知ることができる。

4. 性能評価

4-1. 実行速度比較

(1) 10,000,000 回の演算

普通の倍精度実数を使った場合とソフトウェアシミュレーションによる有効桁追跡及びメモリ読み取り方式による有効桁追跡の実効速度比較を表

1 に示す。一千万個のランダムな実数を SPARCstation2 で加減乗除したものである。比率は表中の加減乗除をもとに計算した。加減算がより多くの時間を要する理由は桁落ちの判断によるものである。

表 1 実行速度比較

	普通の double	ソフトウェアシミュレーション	メモリ読み取り方式
加算	3	36,224	124
減算	3	44,612	123
乗算	6	19,730	57
除算	14	64,710	65
加減乗除	22	123,770	350
比率	1	5,626	16

(10,000,000 回のランダムな数の演算、SPARCstation2 による。[単位: 秒])

(2) whetstone

ベンチマークテストとして有名な whetstone を使って実行速度比較を行った。なお、超越関数等は普通の double の演算ルーチンを使用し、有効桁数は演算前のままとした。理由としては、個々の関数ルーチンは比較的正確に計算すると判断し、実行速度を上げるためである。

	普通の double	メモリ読み取り方式
whetstone	7,692	1,250
比率	1	6

上記 2 つの結果から実行時間は演算の内容にもよるが、加減算が多ければ数十倍の時間を要するが、乗除算が比較的多く混在すれば、数倍の時間で済む。普通に行って 10 分の計算はこのシステムでは約 1 時間かかることになる。時間は余計に必要となるが、計算精度を知ることができる。

4-2. 有効桁数と実際の計算精度との比較

1) ヒルベルト行列を係数とする8元連立方程式の解

桁落ちすること有名なヒルベルト行列を解いてみる。

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{8} \\ \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{7} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{8} & \frac{1}{7} & \cdots & \frac{1}{15} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_8 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

(a) ガウスの消去法で解いた場合

結果を表2に示す。ここで精度とは、計算結果のうち実際に正しかった桁数である。

結果から、前進消去後まではほぼ有効桁数と精度とが一致するが、最終的な結果では、実際の計算精度よりも小さく見積もってしまった。これは、桁落ちが起こり、シフトして0が入った時にその0が偶然正しい場合もあるが、桁落ちしたので、有効ではないと判断してしまうからである。

表2 ヒルベルト行列での比較(ガウスの消去法)

	計算結果		前進消去後	
	有効桁	精度	有効桁	精度
x1	0	27	53	53
x2	3	26	53	53
x3	6	26	49	52
x4	9	26	44	42
x5	11	26	39	38
x6	13	26	33	39
x7	14	26	26	29
x8	14	26	19	29

(b) ガウス-ジョルダンの消去法で解いた場合

結果を表3に示す。やはり、実際の精度よりも小さく見積もってしまう。

表3 ヒルベルト行列での比較(ガウス-ジョルダン)

	有効桁	精度
x1	17	27
x2	15	26
x3	15	26
x4	15	26
x5	14	25
x6	15	26
x7	14	25
x8	14	25

2) 指数関数を次式で求めた場合

$$e^x = 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + \cdots$$

xの値として負の値で右辺を計算すると桁落ちする例である。結果を表4に示す。この例では有効桁数はほぼ精度に一致した。

表4 指数関数での比較

x	有効桁	精度
-4	44	42
-7	35	30
-9	30	30
-12	21	11
-17	7	3
-19	1	2
-21	0	0

3) 偏微分方程式

Laplace方程式 $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$ において

差分近似で解く。

正方形領域 ($0 \leq x \leq 1, 0 \leq y \leq 1$) の上で、境界条件として、

x軸上とy軸上で $u(x,y)=0$

直線 $x=1$ の上で $u(x,y)=y$

直線 $y=1$ の上で $u(x,y)=x$

を与え、分割数4（格子点25、未知の格子点9）で単純に連立方程式で解いた場合、およびSOR法で解いた場合の結果を表5に示す。どちらもそれほど桁落ちが無く、また、有効桁は精度にはあっている。ただし、SOR法の場合、解には桁落ちはあまり見られないが、計算途中の

$$\tilde{u}_{ij} = (u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}) / 4$$

$$u_{ij} = u_{ij} + \alpha(\tilde{u}_{ij} - u_{ij})$$

において、 $\tilde{u}_{ij} - u_{ij}$ の部分で桁落ちが起こる。これについて、計算途中で上式の有効桁を見て桁落ちの大きさを知ることは可能である。

表5 偏微分方程式での比較

	連立方程式		SOR法	
	有効桁	精度	有効桁	精度
u11	42	46	48	49
u12	45	49	47	50
u13	45	50	48	52
u21	45	48	47	51
u22	47	50	49	51
u23	48	53	48	53
u31	47	53	48	52
u32	49	53	48	53
u33	51	53	49	53

5. まとめ

桁落ちだけを計算過程のすべてについて追跡し、桁落ちの影響を受けていない部分を「有効桁」と定義し、有効桁と計算結果との関係を調べた。有効桁が、計算結果の誤差を含んだ無効な部分と誤差を含んでいない有効な部分を示すものと考えた。

そのための有効桁数追跡システムとしてソフトウェアシミュレーション方式とメモリ読み取り方式でシステムを作成した。メモリ読み取り方式はソフトウェアシミュレーションに比べ、実行速度

が速くなり、まだ数倍の時間を必要とはするが、実用に差し支えない速度となった。これらはC++が動作すれば、ほとんど機種で動作する。

有効桁追跡は、桁落ちした全ての部分を無効と考えたので、シフトして0が入った場合に偶然あっている場合もあり、誤差を大きめに評価してしまうことがある。また、丸め、情報落ちを考慮していないために、実際の精度よりもきびしく見積ってしまう場合もあり、実際の計算精度とは多少違う。しかし、桁落ちがあることを利用者に知らせ、注意を促すことは出来る。実行速度を多少犠牲にしても計算結果に信頼性を持たせる価値はあるのではないだろうか。

また、応用としてソフトウェアシミュレーション方式では、時間はかかるが仮数部を自由に大きくできるので、計算後の有効桁数を元にして仮数部を大きくし、再計算することによってより正確な値を求めるためのシステムをも考案した。[2]

将来的には、見積もり精度の向上、実行速度の高速化をはかり、ハードウェア化したい。

[参考文献]

- [1] 鈴木、大岩：桁落ちを考慮した浮動小数点演算の有効桁、情報処理学会 第19回数値解析シンポジウム 98-101, 1990
- [2] 鈴木、大岩：桁落ちを追跡する浮動小数点演算、情報処理学会第42回全国大会講演論文集1-57, 1991
- [3] 鈴木、大岩：浮動小数点演算における桁落ち追跡のアルゴリズム、情報処理学会 第48回(平成6年前期)全国大会講演論文集7L-1, 1994