常微分方程式の初期値問題における多倍長計算の必要性についての一考察

幸谷智紀* tkouya@cs.sist.ac.jp

2000 年 1 月 17 日 (月)

1 初めに

浮動小数点数を用いる数値計算において，仮数部の桁数を自在に操作できる場合を多倍長計算，または任意精度計算と呼ぶ。ここでは前者の名称を使用する。

浮動小数点計算は速度が要求されるため，ハードウェア，即ち PC や WS であれば CPU が直接実行することが多い。現在広く使用されている浮動小数点演算の規格である IEEE754 では，32bit 長の単精度 (仮数部 24bit[1])，64bit 長の倍精度 (仮数部 53bit)，80bit 長の拡張倍精度 (仮数部 64bit) の 3 種類の浮動小数点数フォーマットが規定されているが，これら浮動小数点数の四則演算は，整数よりも複雑なアルゴリズムを使用しなければならないため，ソフトウェアだけでは高速性に限りがあるからである。ハードウェアの発展は特に PC において著しいものがあるが，現段階においても IEEE754 規格の桁数を越えた多倍長計算はコスト高になる。

それでもなお多倍長計算を行う理由としては,「より高精度な数値解を求めるため」というのが一番多い理由であろう。しかし，問題自体に取り立てて悪条件性がない問題の数値解の桁数を増やすだけの多倍長計算では少々寂しい上，コスト高になる計算を行う意味付けにおいても説得力に欠ける。初期誤差や丸め誤差のためにハードウェアでの浮動小数点計算では必要な桁数を得ることができない悪条件問題に適用してこそ多倍長計算の重要性を示すことができる。

しかし，それでも計算時間は短いに超したことはない。そのため，計算桁数はなるべく少なくしたい。その目安となるようなコスト (計算時間) と計算桁数の具体的なベンチマークテストは，多倍長計算においては必要不可欠である。

本稿では，常微分方程式の初期値問題における悪条件問題として，Rössler Model を取り上げる。同時に，今回使用した多倍長計算パッケージの性能評価も併せて行い，この問題に対する多倍長計算の必要性と計算コストについて論じる材料を提供したい。

*静岡理工科大学

[1] 単精度・倍精度はケチ表現の 1bit 分を含む。

2 四則演算のベンチマークテスト

以下，本稿で示される数値実験は全て，次の計算機環境下で行ったものである。

1. 多倍長計算パッケージ: gmp 2.0.2
2. 本体: (CPU)AMD K6-2 300MHz, (RAM)64MB, (OS)Laser5 Linux 6.0
3. コンパイラ: gcc egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)2.7

多倍長計算パッケージに gmp[1] を選んだ理由は，桁数を自在に伸縮できることと，MPPACK と比べてみて四則演算の性能が良かったことにある。以下にその結果を示す。MPPACK は，2000 年 1 月現在の PHASE サーバ (http://phase.etl.go.jp/) に置いてあるものを使用した。なお，gmp は 2 進の桁数 (bit 数)，MPPACK は 10 進で桁数を指定する。参考までに，IEEE754 浮動小数点数の結果も載せておく。表の数字は全て MFLOPS である。

ベンチマークテストには自作の BASE Bench プログラムを使用した。これは乱数を発生させて二つの配列に格納し四則演算を行い，もう一つの配列に演算結果を代入する。これを繰り返して四則演算の MFLOPS 値を計算するという単純なプログラムである。メモリキャッシュの効果などについては特段考慮していない。

	IEEE		**MPPACK**
	single	double	40(decimal digits)
ADD	5.12	5.46	0.07
SUB	4.56	4.82	0.03
MUL	5.12	5.46	0.004
DIV	2.28	2.28	0.001

gmp

bits	64	128	256	512	1024
ADD	1.60	1.44	1.30	1.00	0.69
SUB	1.82	1.64	1.32	0.93	0.59
MUL	0.72	0.46	0.21	0.08	0.02
DIV	0.36	0.24	0.12	0.05	0.02

以上のように，四則演算だけの結果を見る限り，gmpが優れていると言える。

しかし，gmpはCプログラムのパッケージであり，初等関数も殆ど用意されていない。対して，MPPACKはC++のクラスとして提供されており，通常の演算子はそのまま使用できる。様々な関数群もあらかじめ用意されているから，使い勝手の上では断然優れている。

3 Rössler Modelの計算

Rössler ModelはChaos現象が現れる簡単な3次元力学系の一つである [5]

$$\frac{d}{dx}\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} -(y_2+y_3) \\ y_1+\alpha y_2 \\ \beta + y_3(y_1-\mu) \end{bmatrix} \qquad (1)$$

ここでは$\alpha=\beta=1/5$と固定して考える。μを3, 4, 5, 5.7としていくにつれ，この常微分方程式の解の周期が増加していくことが知られている [5]。また，Rössler Modelは，解の挙動が複雑になるにつれて収束性が悪くなる。

常微分方程式の初期値問題では，真の解が不明であるとき，どこまでが丸め誤差でどこまでが打切り誤差であるかを調べるには次のようにして考える。

1. 積分区間を固定し，刻み幅を小さくしていき (例えば，$1/2, 1/2^2, 1/2^3 \ldots$とする)，それぞれ数値解を求める。
2. 刻み幅が小さくなるにつれて，数値解の上の桁から数値が一致していく。これを真の解として採用する。
3. 刻み幅に合わせて，一致する桁が増えてきているときには，不一致桁の上位が打切り誤差を現している。
4. ある程度を超えると，刻み幅を小さくしても一致する桁数が増えないか，一致する桁数が減少し出す。このとき，不一致桁の上位が丸め誤差を現している。

今回は全て，初期値を$y(0)=[1\ 0\ 0]^T$とし，積分区間を$[0, 500]$に設定した。

3.1 IEEE754浮動小数点計算

まず，7段6次の陽的Runge-Kutta法と3段6次陰的Runge-Kutta法を使用して (1) をIEEE754倍精度浮動小数点数を使用して計算した。その結果を以下に示す。

Explicit : 7 step (6th order), $\mu=5$

	$h=1/4096$	$h=1/8192$
RN	$-4.8295189\mathrm{e}+00$	$-4.6293495\mathrm{e}+00$
RZ	$-6.0714505\mathrm{e}+00$	$-5.2090559\mathrm{e}+00$
RP	$-4.2363258\mathrm{e}+00$	$-3.5081142\mathrm{e}+00$
RM	$-5.1634064\mathrm{e}+00$	$-3.8688926\mathrm{e}+00$

Implicit : 3 step (6th order), $\mu=5$

	$h=1/4096$	$h=1/8192$
RN	$-3.8470654\mathrm{e}+00$	$-3.5698751\mathrm{e}+00$
RZ	$-6.1574318\mathrm{e}+00$	$-5.2671008\mathrm{e}+00$
RP	$-3.4452750\mathrm{e}+00$	$-4.6235796\mathrm{e}+00$
RM	$-3.4044998\mathrm{e}+00$	$-3.5246481\mathrm{e}+00$

上の表は$\mu=5$のときのy_1の値を，RN, RM, RP, RZモードでそれぞれ計算したものある。浮動小数点演算過程での丸めのモードを変更することによって，丸め誤差の大きさを精度を増やすことなく示すことができる [2]。全ての桁が異なっており，明らかに丸め誤差が増大していることが分かる。

また，刻み幅を増やしても，丸め誤差のorderには殆ど変化が無いことも同時に分かる。実際，もっと小さい刻み幅で計算された数値解でも，orderには明らかな変化は見られなかった。従って，ここで現れた丸め誤差は，単なる計算量に比例する蓄積によるものでは無く，元の常微分方程式 (1) の性質に依存するものである，と言える。このように，Rössler Modelの数値計算は，IEEE754浮動小数点数の桁数では不足である。$\mu=5.7$の場合は，更に丸め誤差の影響が大きくなり，精度が必要であれば多倍長計算を行う必要がある。

3.2 多倍長計算

IEEE浮動小数点数を用いた計算では，丸め誤差が全ての桁を覆い尽くしてしまい，その刻み幅で打ち切り誤差が十分に小さくなっているかどうかは判然としない。よって，多倍長計算で丸め誤差を追いやった後は，打ち切り誤差について考慮しなくてはならない。

$\mu=5.7$とし，先程と同じ7段6次陽的Runge-Kutta法を使用して，$x=500$でのyの値を計算した結果を以下の表に示す。

Explicit Runge-Kutta(7 steps, 6th order), 1/4096

	bits	
y_1	64	0.4200216265525560936e1
	128	0.3779764086196105713852908351536162851 7e1
	256	0.3779764086196105713673207388594309445894436861302922830788916187009183132686 1e1
	512	0.3779764086196105713673207388594309445894436861302922830788375823363511520282013984473937946615373388171563011098099672045367750871589300081703851108862 09e1

	bits	
y_2	64	0.4206069565641866605e1
	128	0.3842817261146448174534417352280712689 4e1
	256	0.3842817261146448174397373459622778803305049010855746326823194673383241688270 9e1
	512	0.3842817261146448174397373459622778803305049010855746326822791547874863034813779978274762266601926616607477019185714391769274294999379636046225626945224133e1

	bits	
y_3	64	0.8881089410076764788e1
	128	0.1043908998702490491828525217961576275 5e2
	256	0.1043908998702490491881550371970416229503536865436137065994270224646036084238e2
	512	0.1043908998702490491881550371970416229503536865436137065994426202353313690460282662773677378369078188945384237276424166129113969613151133762761770592852607e2

Explicit Runge-Kutta(7 steps, 6th order), 1/8192

	bits	
y_1	64	0.897092088142246549 2e0
	128	0.3779764055200011269763298297883138840 9e1
	256	0.3779764055200011269404896495632578252244670045204063231143785806260 8729623888e1
	512	0.3779764055200011269404896495632578252244670045204063231142734552931089301283674221682661851212136246918694400351208276401155490746826699092645607936398473e1

	bits	
y_2	64	0.1916131702986955354e1
	128	0.3842817237508147273850245598081543424 3e1
	256	0.3842817237508147273576920529705388424564656645478124376478903817578559911427 2e1
	512	0.3842817237508147273576920529705388424564656645478124376478102108752210586541297215637430589581620554016940858661449329117817158444750620020692407415706 61e1

	bits	
y_3	64	0.1228232061156121429e2
	128	0.1043909007848644408530949624497423314 9e2
	256	0.1043909007848644408636704819630627772344221331066543708864269970711005129839 8e2
	512	0.1043909007848644408636704819630627772344221331066543708864580168651773236312522813427674896625746427001850834901652944089469767296123972527269722037232769e2

上の表から，次のことが分かる。

1. 丸め誤差は 10 進 18〜19 桁程度の大きさである。故に，64bit 計算の結果は全く信用できない。
2. 128bit 以上の計算結果のうち，同じ計算桁数で刻み幅を小さくしていった結果の不一致桁の上位は打ち切り誤差を示している。刻み幅を 1/2 にすると 10 進 2 桁ずつ減少していく。

従って，128bit 以上の計算結果のうち刻み幅が 1/8192 のものも，打ち切り誤差は更に縮小させることができると予想される。しかし，陽的 Runge-Kutta 法で更に次数を上げるためにはより多くの段数を必要とし，段数と次数の差は拡大していく。そのため，多倍長計算においてはあまり効率の良い方法ではない。アルゴリズムが単純で，可変次数の公式が望まれる。

そのために補外法を使用する [4]。以下の計算は 128bit の浮動小数点数を使用して行われたものである。補外表の大きさは 4 段とした。この補外には Romberg 数列を使用しているため，4 段では 8 次程度の公式に相当する [4]。

Extrapolation 4 Stages, 128bit

	Stepsize	
y_1	1/4096	0.3779764054707608944829206100026606988 2e1
	1/8192	0.3779764054707609054935470156108709125 9e1
y_2	1/4096	0.3842817237132630470402267325061171541 2e1
	1/8192	0.3842817237132630554371717147968642325 5e1
y_3	1/4096	0.1043909007993939726868504282125266 79e2
	1/8192	0.1043909007993939694378964988426831359 6e2

以上の結果より，一致している桁は上から 16〜17 桁である。全 38 桁のうち，下位桁 18〜19 桁は丸め誤差であることが示されており，桁の一致している上位桁は正確であると考えて良い。従って，ほぼ IEEE754 倍精度程度の精度を得ていると言える。

最後に計算時間を示す。同次数での比較のため，3 段の補外法の計算時間も載せておく。

Computation Time (sec)

Stepsize: 1/1024			
	Runge-Kutta	Extrapolation	
	7-6	3 stage	4 stage
gmp 64 bits	146.69	167.08	364.3
gmp 128 bits	194.53	204.82	444.4
gmp 256 bits	328.71	308.55	664.8
Stepsize: 1/2048			
gmp 64 bits	293.32	334.13	727.1
gmp 128 bits	388.35	409.62	888.8
gmp 256 bits	657.34	617.09	1330.0

4 段の補外法は 7 段 6 次の陽的 Runge-Kutta 法の倍近い時間が必要であるが，同次数の 3 段の場合はほぼ同時間で済むことが分かる。

一般に，補外法は固定次数の解法に比較して計算時間が多いと思われているが，こと多倍長計算においては，固定次数の解法以上の精度を得ることができるという利点があり，桁数の変化にも柔軟に対応できる。更に刻み幅制御と次数制御 (段数制御) を加えることで，変動の激しい問題に対しても固定次数の解法以上の精度を得ることが可能であることも強調しておきたい [4]。

4 今後の課題

今後の課題としては，Rössler Model に関するものと，多倍長計算一般に対するものがあげられる。

4.1 Rössler Model について

Rössler Model の数値計算において，丸め誤差の増加が著しい原因を，アルゴリズムに沿った形で初等的に追求する。Rössler Model と同様の性質を持つ Lorenz Model についても同様に調べてみたい。最終的には丸め誤差に関する悪条件性を定量的に示す指標を見つけることができれば，と考えている。

なお，Chaos ついての Survey が十分でないため，以上の事柄は既知である可能性がある。御指摘頂ければ幸いである。

4.2 多倍長計算について

多倍長計算にかかる時間は，計算機環境とソフトウェアとしての構造に大きく依存するため，一律に決まるものではない。しかし，今回のように一定のベンチマークテストを行うことで，実際の計算に費やされる時間をある程度は予測することが可能である。従って，問題の数値的性質によって必要な精度を得るための桁数を知ることができれば,「計算コストの最適化」が可能になる。多倍長計算が現実的であるかどうかは，計算コストの最適値によって判断できる。

今回使用した多倍長計算パッケージ gmp では，同じ桁数でも乗除算が CPU の 10 倍程度の時間コストが必要であることが示されている。しかし，多倍長計算をハードウェアでサポートするという話は，寡聞にして聞いたことがない。現状ではソフトウェアによる多倍長計算を行わねばならない。

では，ソフトウェアによる多倍長計算は，どの程度の速度を出すことができるのか。当然，計算機環境によって左右されるところが大きいのであろうが，もしさらなるスピードアップが可能であるというのであれば，計算コストの最適値も下がることになる。今回のような丸め誤差によって引き起こされる悪条件問題への対応策としても，多倍長計算を用いるというアプローチは単純であるが有力な解となり得る。

今後は他の悪条件問題に対しても多倍長計算を適用し，精度と計算コストとの関係を定量的に調査してみたいと考えている。

参考文献

[1] T.Grandlund, TMG Datakonsult, GNU MP, The GNU Multiple Precision Arithmetic Library, Manuscript, 1996.

[2] 幸谷智紀・永坂秀子, IEEE754 規格を用いた丸め誤差の測定法について, 日本応用数理学会論文誌, 1997.

[3] 幸谷智紀，常微分方程式および固有値問題の高精度計算法の研究，博士論文 (日本大学), 1997.

[4] 室伏誠, 有限桁計算における Richardson の補外法による丸め誤差評価の研究, 博士論文 (日本大学), 1998.

[5] 下條隆嗣, カオス力学入門, 近代科学社, 1992.