多倍長計算ライブラリGMPでの べき乗関数のプログラム開発

田中　　寛※

1 はじめに

コンピュテーショナル・エコノミクスにおいて重要な概念である効用関数や生産関数として用いられるものにコブダグラス型がある。例えば、Kを資本量Lを労働量としたときの生産量は

$K^{\alpha}L^{\beta}$

と表される。この式で、αは資本分配率、βは労働分配率といわれ、ともに０と１の間の数である。ここで確認すべきことは、べき乗の表現が現れることである。コンピュテーショナルである以上、実証研究であっても、理論モデル計算であっても、それが何らかの形でべき乗の計算がコンピュータ上でなされることである。

多倍長計算は、もともとは円周率πを求める興味から関心がもたれたものである。しかし、近年はもっと実用的な大規模計算にも利用できるPCのハードウェアとソフトウェアの環境が整ってきている。その中で、多倍長計算ライブラリGMPが、フリーソフトファウンデーション(FSF)のGNUプロジェクトの一つの成果として公開されている。筆者も、Markowitz投資理論を日本の株式市場に適応する際に約3000×3000の連立一次方程式を解くのに利用[1)]した。GMPライブラリは、四則演算を含めてC言語の関数の集まりである。その関数のインターフェース仕様は、GMPプロジェクトのWebページ[2)]にある。多倍長計算の精度は、PCのコンピュータ資源の限りでプログラムから関数を使って任意に指定することが出来る。

コンピュテーショナル・エコノミクスで必須であるべき乗計算が、現在のGMPライブラリには含まれていない。関数mpf_pow_uiという名の関数はあるが、この関数は指定する精度での指数が整数乗の計算の機能しかない。いま必要とする指数が０以上１以下の小数であるようなべき乗の計算には対応していない。また、１以上で小数点のある数が指数であるべき乗の計算にも使えない。もっとも、C言語自体には標準関数として関数powがあり、その精度は10進数で15桁である。本論文の目的は、指数が小数点のある数である数についてのべき乗計算に対応する多倍長計算をするべき乗関数のプログラムを開発することである。

まず、２においてべき乗計算のアルゴリズムを明らかにする。このアルゴリズムは、アルゴリズムの教科書でも、インタネット上のWebページの検索によってでも、さらに、オープンソースであるはずのC言語のソースプログラム中でも見つけることが出来なかった。３においては、２で明らかにするアルゴリズムに従って、三つのべき乗計算のプログラムを開発する。最初に、２で開発するアルゴリズムを忠実に実現するプログラムを通常のC言語で記述する。このプログラムにはアルゴリズムの正しさを確かめることと多倍長計算のプログラム作成のひな型としての役割がある。次に、このひな型に従って関数ライブラリGMPを用いて書き換える。最後に、GMPで書き換えられたプログラムを同時並行処理のためにスレッド技術[3)]を用いてさらに書き換える。

2 べき乗計算のアルゴリズム

べき乗計算の基本方針は、指数である数を小

※ 青森公立大学教授

数点以上の数と以下の数の和とし、両者の２進数表現をともに一般的に得て、仮数の指数乗の積としてそれぞれのべき乗計算を必要な精度で計算し、最後に両者の計算結果を掛け合わせる。

仮数xを０以上の小数点数、指数yを０を含む任意の小数点数とするときに、

⑴　x^y

の値を数値計算することを考える。

２.１．べき乗を計算するアルゴリズム

指数yを２進数で表現すれば、nを０から無限大の正の整数、mを１から正の無限大の自然数として、

⑵　$y=\sum_{n=0}^{\infty} b_n \cdot 2^n + \sum_{m=1}^{\infty} c_m \cdot 2^{-m}$

と一般的に必ず表すことができる。係数b_nとc_mはともに０か１である。０以上の整数nについての和は、yが有限である限り上限があり無限大までの和は必要がない。このnの最大値をn_maxとする。一方、自然数mについての和は、yが特別の値でない限り和を無限大まで行わなければならない。

このように表現されるyを(1)式に挿入すると、指数の和は二つのΣの積であらわされる。すなわち、⑴式は

$$x^{(\sum_{n=0}^{\infty} b_n \cdot 2^n)} \cdot x^{(\sum_{m=1}^{\infty} c_m \cdot 2^{-m})}$$

となる。それぞれの仮数xのべき乗の指数としてある肩の部分の記号Σはともに和を意味するので、べき乗の性質より積の記号Πを用いて、

⑶　$\prod_{n=0}^{\infty} (x^{(b_n \cdot 2^n)}) \cdot \prod_{m=1}^{\infty} (x^{(c_m \cdot 2^{-m})})$

と書きかえることが出来る。nとmでの個々の項の値を計算し最後にそれらすべての積をとれば、べき乗が計算できることを⑶式は示している。

⑶式において、個々の項の係数b_nおよびc_mは０か１であり、０の場合には指数が全体として０となり項の値は１となり、最後の積の計算には影響を与えない。したがって、ここまでで残されている問題はb_nおよびc_mが１である項の計算である。

⑶式のnについての項は、b_nが１の場合、

$$x^{(2^n)}$$

である。ここで、

$$n=(n-1)+1$$

とすると、

$$2^n=2^{((n-1)+1)}$$

となる。べき乗の性質を利用すると、

$$2^n=2^{(n-1)} \cdot 2^1=2 \cdot 2^{(n-1)}=2^{(n-1)}+2^{(n-1)}$$

である。この結果、今計算したい⑶式のnの項は

$$x^{(2^n)}=x^{(2^{(n-1)}+2^{(n-1)})}$$

となり、もう一度べき乗の指数の加算が二つの項の乗算となることを利用すると、

⑷　$x^{(2^n)}=x^{(2^{(n-1)})} \cdot x^{(2^{(n-1)})}=(x^{(2^{(n-1)})})^2$

が得られる。この式の意味するところは、nの項$x^{(2^n)}$を計算するのには、2^nを求めさらに$x^{(2^n)}$を計算する必要はなく、(n-1)の項$x^{(2^{(n-1)})}$を単に２乗すればよいということを示している。nが０の場合は、文字通り2^0は１であり、$x^{(2^0)}$はxである。

⑶式のmについての項は、c_mが１の場合、

$$x^{(2^{(-m)})}$$

である。⑷式の導出に当たって、nは０以上の整数と一応考えたが、導出過程の途中でその条件は用いてはいない。そこで、nが負の数についても⑷式が成立するとして、mが１から無限大の整数に対して、

$$n=-m+1$$

という変数変換を⑷式に行うと、

$$x^{(2^{(-m+1)})}=(x^{(2^{(-m+1-1)})})^2$$

が得られる。この式を整理すると、

$$x^{(2^{(-(m-1))})}=(x^{(2^{(-m)})})^2$$

となる。この式の右辺は、今計算したい項の２乗である。そこで、左右の両辺を入れ替え、正の２乗根をとると、

$$x^{(2^{(-m)})}=(x^{(2^{(-(m-1))})})^{(1/2)}$$

となる。⑶式のmの項$x^{(2^{(-m)})}$を計算には、$2^{(-m)}$を求めさらに$x^{(2^{(-m)})}$を計算する必要はなく、(m-1)の項$x^{(2^{(-(m-1))})}$の２乗根をとればよいことを示している。

以上の結果、⑶式自体の各項は自由な順番で計算でき最後に積をとればよいように見えるが、各項の値の計算方法の手法によって、nの各項はn=0のxから始めてn=1でx^2などとnをn_maxまで順に計算しなければならなく、また、mの各項はm=1のxの２乗根から始めてmの無限大までの次々の２乗根の計算となる。ただし、b_nないしc_mが０の場合は、項の値としては１となることを忘れてはならない。

２.２．２乗根を求めるアルゴリズム

べき乗(1)式を計算するには、(3)式のmの項からxの２乗根を順次計算する必要のあることが分かった。

今、ある定数ａの２乗根 $a^{(1/2)}$ を求める手順ととして、まず $a^{(-1/2)}$ を関数の解を求めるニュートン法によって計算し、最後にaを掛ける[4]ことにする。

関数

(5)　$f(x)=1/x^2-a$

によって

$f(x)=0$

を解くことによって

$x=a^{(-1/2)}$

が得られる。しかし、これだけで具体的数値が得られるわけではない。そこでニュートン法の手順に従うことが必要となる。

解であるかどうか分からない適当な x_0 を取る。(5)式によって対応する $f(x_0)$ を計算する。また、(5)式のxについての微分をすると、

(6)　$f'(x)=-2/x^3$

がえられる。xy平面で点 $(x_0,f(x_0))$ を通り傾きが $f'(x_0)$ の直線の式は、

(7)　$y-f(x_0)=f'(x_0)(x-x_0)$

で表される。この直線がx軸と交わる点を x_1 とする。すなわち、(7)式において $y=0$ となる x_1 を求める。

$0-f(x_0)=f'(x_0)(x_1-x_0)$

から

$x_1=x_0-f(x_0)/f'(x_0)$

となる。この式において右辺は x_0 だけで表されているので数値的に計算できる。その結果、x_1 も数値として表すことが出来る。(5)式と(6)式から x_0 の具体的表現を入れて整理すると、

$x_1=x_0(3-ax_0^2)/2$

が得られる。同様の手順を繰り返して、$x_2,x_3,\ldots$ と具体的な数値を求めていけば、$a^{(-1/2)}$ に限りなく近い数値が得られることになる。最後に、得られた $a^{(-1/2)}$ の近似値にaを掛けて $a^{(1/2)}$ を求める。以上がニュートン法による２乗根の求め方である。

３　多倍長ライブラリGMPを使ったプログラム開発

任意の精度でべき乗計算が可能な関数の開発を目指して、次の順序でプログラム開発を行なった。まず、２で述べたアルゴリズムで標準的なC言語の範囲で動作する関数を開発する。ついで、開発したそのプログラムの演算を多倍長計算のライブラリGMPの対応する関数に置き換える。最後に、２のアルゴリズムは並行処理が可能であるので、スレッド技術を用いる多倍長計算のプログラムを作成する。

プログラミングの成果物は、C言語で記述されたソースプログラムである。正当に動作するプログラムは、必要なすべての内容表現がそのソースプログラム中に含まれている。したがって、ソースプログラムだけを見れば動作のすべてを理解できるようになっていることになる。しかし、ソースプログラムはコンピュータとして動作させる言語で記述されるので、人がにわかに理解するのは容易ではない。そこで、ここではソースプログラムを示すとともに、その解説を行う。ソースプログラムを書いた人にとっても、メモを残しておくという意味もある。書いた人にとってもプログラムのあらゆることを記憶し続けることは、多くの場合可能でないのが現実である。

３.１．C言語標準関数powを再現するpow_double

プログラミングの最初の作業として、２で述べたアルゴリズムに従う普通のC言語の範囲内で記述するプログラムを作成する。この作業は学問的に価値のあるものではない。しかし、２で述べたアルゴリズムの妥当性を、C言語標準関数としてあるべき乗関数powの計算結果と比較することによって、確認できるという意味がひとつある。また、以後に述べる多倍長計算関数ライブラリを用いる開発の際に、このプログラムは有用なひな型として利用することもできる。したがって、この関数pow_doubleの開発作業は重要であり疎かにはできない。

関数の仮引数は、べき乗の基数が最初の、また、その指数が次のものである。この順序は、powと同じにしてある。基数xは、ゼロ以上でなけれ

ば一般的な指数yに対する計算は数学的に不可能である。特定のyに対しては数学的には計算可能かもしれないが、xが負の値の場合計算できないこととしてエラー表示をする。xがゼロの場合は、yが何であってもべき乗計算結果はゼロとする。この結果、xが正の場合のみ次のステップに進むことになる。指数yの値は正、ゼロ、負が考えうる。yがゼロの場合は、べき乗の計算結果を１とする。yが負の値の場合、yの絶対値をとり、その値についてのべき乗計算を行い、その計算結果で１を割る。この計算の途中にあるべき乗計算は、通常の場合あまりなされない再帰的関数呼び出しとなる。C言語にはこの機能がある。yが正の値の場合が実際のべき乗計算が必要となる。この段階で、指数を整数部分と小数部分の和として分離し、それぞれに対応する関数を呼び出し、それぞれの関数処理の終了後に、計算結果を掛け合わせる。この処理方法は、指数は整数部分と小数部分との和で表現され、指数での和は基数の整数乗と基数の小数乗の積となることから、正しいことになっていることが分かる。

べき乗計算の指数の整数部分について計算する関数は、sei_doubleである。引数は、計算したいべき乗計算の基数のｘと指数のｙである。指数についての引数は指数の整数部分ではないことに注意が必要である。関数 pow_doubleを記述するに当たっては、10進数での精度の桁数の定数 PREC_DOUBLEと相対精度の定数EPS_DOUBLEが、プリプロセッサで定義されているものとする。yの整数部分が０の場合は、単に１を関数の戻り値とする。またyの整数部分が１の場合は、xを関数の戻り値とする。配列kには、１から始めて順に２倍した値を記憶する。同時にyの２進数表記でのn_maxを求める。因みにn_maxは、yの２進数表記の最大桁である。次に、その最大桁から始めて、yの２進数表記を配列bに順に求める。そして、小さい桁数から始めてn_maxまで、配列bの値によってべき乗計算を行う。配列bの値が０の場合は、対応するべき乗はxの０乗なので１をかける。配列bの値が１の場合は、基数xの対応する配列kの要素すなわち２進数の基数の数乗をして、べき乗計算の数zに掛けることをする。最後に、得られたべき乗計算の結果zを関数の戻り値とする。

べき乗計算の指数の小数部分について計算する関数は、shou_doubleである。この関数の引数も、基数xと指数yである。まず最初にやるべきことは、yの小数部分だけを得ることである。yを整数型の変数mにコピーし、y自身から今得たmを引くことによって行う。指数が少数部分の場合のべき乗計算は、指数が１以上の場合に比べて複雑ではない。まず、n_maxに相当するものを計算する必要がない。少数部分の最上位桁の２進数は0.5であると決まっている。したがって、yを２進数で表現するための処理は、0.5から始めて順に２で割って行って、その数とyの残りの数と比較すればよい。大きいか等しければその桁の2進数cは１であり、小さければcは０である。そして、cが１であれば１に初期化されているzに前の桁の計算までxの２乗根を次々とかける。cが０であればzに１をかけるが、このことは何もしないことと同じである。最後に、xについての２乗根計算の相対誤差の絶対値がEPS_DOUBLEよりも小さくなったら繰り返し処理を停止する。そして、得られたべき乗計算の結果zを関数の戻り値とする。

２乗根の計算には、C言語標準関数として関数sqrtがあるが、多倍長計算のプログラム開発を視野に入れて、２で述べたアルゴリズムを実現する関数sqrt_doubleを用いる。この関数の記述における仮引数は、いまその２乗根を求めたい数aとする。ニュートン法の繰り返しを始める初期値としてsqrtで計算される値を用いる。このようにとれば、次に行う繰り返しは意味がないと考えられるかもしれないが、プログラミングの次のステップとして多倍長計算を行って精度の高い２乗根の値を求める際には、このように初期値をとることが生きてくることになる。ニュートン法により、x_{n-1}から次の近似値 x_n を求める。そして、相対誤差の絶対値がEPS_DOUBLE以下になるまでニュートン法の繰り返しを続ける。繰り返しが終了したら、得られた結果にａを掛けて関数の戻り値とする。

以上に述べてきたプログラムを図１に示す。

３.２. 多倍長計算ライブラリGMPを利用するpow_gmp

多倍長計算用のライブラリGMPを利用して、指定する任意の精度のべき乗計算を行う。関数

名をpow_gmpとする。この関数を呼び出すに当たっては、精度の10進数での桁数PREC_GMPと精度の数の大きさEPS_GMPをプリプロセッサ命令#defineで指定されていなければならない。また、GMP関数mpf_set_default_precで２進数での桁数を設定すると便利である。２進数での桁数は、PREC_GMP*log(10)/log(2)で計算できる。設定される桁数での計算が、GMP関数の利用において最低保証される。

小数点のある数の定義は、GMPでは型定義mpf_tで行う。その内部構造はプログラムでGMPを利用する側では知る必要がない。GMPには整数の多倍長計算機能もあるが、ほとんどあらゆる計算では必要としない。このべき乗計算でも用いないことにする。

関数pow_gmpの引数は、pow_doubleの場合のように基数のxと指数のyとする。また、pow_doubleでは計算結果を戻り値としたが、ここでは計算結果zも引数として、関数の戻り値はvoidとする。また、引数の順番は、GMPの多くの関数に合わせて、計算結果のzを一番目の引数、基数xを二番目の、指数yを三番目の引数とする。通常の関数記述においては、仮引数の定義がなされると、実行時に記憶領域にその定義に従う名前の領域がとられて実引数の値がその領域にコピーされる。しかし、仮引数の定義がmpf_tの場合、当然実引数もmpf_tで定義されているのであるが、仮引数の領域が新たに実引数とは別にとられるのではなく、仮引数の名前で実引数の記憶領域が直接利用されるようである。このことは、型mpf_tの内部構造に由来していると考えられる。このようなことは、仮引数がポインタ型の場合に起きることはよく知られている。

関数pow_doubleでの変数と配列のdoubleの定義を、関数pow_gmpではmpf_tに置き換える。ただし、戻り値の型がdoubleの関数定義があれば、先に述べた理由によりその型の定義をvoidとする。また、当然のことながら、その関数の記述での型定義もvoidとする。そして、その戻り値に相当するmpf_t型の変数を実引数の最初のものとすることは先に述べた。

仮引数を除くすべてのmpf_t型の変数と配列は初期化しないと、それらはプログラム中で利用可能とはならない。GMP関数mpf_initで初期化すると、値０に設定される。値０以外に設定したい場合は、関数mpf_set系の関数を用いる。配列のすべての要素も関数mpf_initによって個々に初期化する。

四則演算の計算については、最初の実引数を演算結果を入れる変数を指定し、次の実引数を演算の対象となる変数を指定し、最後の実引数として演算する変数を指定することは、GMP関数においてすべて共通である。そして、その関数の戻り値の型は、当然のことながらvoidである。加算はmpf_add系、減算はmpf_sub系、乗算はmpf_mul系、除算はmpf_div系のそれぞれの関数を必要に応じて用いる。pow_doubleにおいて一つの式で表現できたものでも、演算結果を別々の変数に記憶して、後で掛け合わせるようにするような複数の行で表現しなければならないものもある。

関数mpf_sgnを含むmpf_tの型の変数を比較する系のGMP関数の戻り値は、通常のC言語の型であるint型である。このようになっている理由は、これらの関数をC言語体系のif文の中で従来のアルゴリズムの流れで直接に利用できるようにするためである。そして、処理結果は、負の値、ゼロ、正の値に分かれるのは、通常の場合と同じである。

以上のことを、関数pow_doubleに対して変換して得られた関数pow_gmpを、図２に示す。GMP関数のプログラムとして得られたものをまず示し、pow_double.cのプログラムを//でコメントアウトしたものを次に示している。

３.３．スレッドを利用するpow_thread

べき乗計算の２で述べたアルゴリズムにおいて、指数の整数部分の計算と小数点以下の部分の計算は全く並行して独立に行うことが出来る。そこで、CPU上で作動しているOSの機能を用いて、並行処理をべき乗計算の多倍長プログラムで実行させることを考える。並行処理には、プロセスを複数起動する方法と、一つのプロセス中でスレッドを複数同時に起動させる二つの方法がある。ここでは、簡易に並行処理のできるスレッドを選択することにする。

スレッドを動作するようにするには<pthread.h>をプリプロセッサでインクルードする。スレッド識別子thread_seiとthread_shouを定義する。また、並行処理するvoid* 型の関数sei_threadと関数shou_threadの定義を行なう。そして、関数thread_createを呼び出して二つの関数を二つ同時並行で実行する。最後に、関数pthread_joinを呼び出して、二つの関数の終了の同期をとる。同期をとるということは、二つのそれぞれのスレッドの終了を待って次の処理に進むという意味である。

多倍長ライブラリGMPで記述したプログラムpow_gmp.cに、スレッド処理を加えるプログラミングを行う際に、問題が発生する。それは、関数pthread_createを呼び出す際に、スレッドとして動かす関数の引数の数が一つでしかもvoid* 型でなければならないことである。プログラムpow_gmp.cでは三つのmpf_t型の引数z,x,yであったものを、三つの要素をもつ構造体xyzのxyz_seiとxyz_shouを記憶領域に取るという形で、一つの名前で扱うようにできるようにしてこの問題を解決する。当然のことながら、構造体のそれぞれの要素のGMPライブラリでの初期化と基数xと指数yの値の転送も必要となる。また、関数処理の最後に構造体の要素zに計算結果を入れる。

プログラムpow_gmp.cをスレッドで扱えるようにしたプログラムpow_thread.cのソースプログラムを図３に示す。ただし、pow_gmpとは違ってひな型となった元のpow_doubleのソースは、読みづらくなるのでコメントアウトした形で残してはいない。

４　計算例と考察

４.１．計算例

完成したソースプログラムをコンパイルし動作させる実行環境は、表１のとおりである。CPUはインテル製のcore i7 3770K BOXである。このCPUには４個のコアが実在するが、スレッディング技術により８個のコアとして利用できる。このCPUをオーバークロックして定格3.5GHzのところを4.6GHzにしている。マザーボードは、ASUS製のMAXIMUS IV EXTREMEである。このマザーボードはCPUをオーバークロックで使うのに特化しているものである。メモリは32GiB(16BiB×２)をマザーボードのメモリソケットに挿入している。しかし、インストールしてあるOSが32ビットであるために15.8GiBまでしか認識されない。OSはUbuntu13.10の32ビット版である。32ビット版を使う理由は、コンパイラの64ビット版がまだ完成されていないので、コンパイラの32ビット版を32ビットのOS上で動作させることによってコンパイル作業の安定を担保するためである。コンパイラは、gccのバージョン4.8.1である。OSがインストールされる際に同時に導入される仕様で用いている。多倍長計算ライブラリGMPは、OSをPCに導入するだけではシステムに組み込まれないので、バージョン5.1.0のソースプログラムをGMPのWebページからダウンロードし別途インストールしている。なお、Windowsではスレッドを動作させることが出来ないので、CygwinなどのC言語の利用環境を用いることはできない。

適当な例題とは、べき乗の基数は正の数ならば何でもよい。たぶん１以上であまり大きすぎない数が良いであろう。また、べき乗の指数は、整数だけではそもそもGMPにも関数があるのだから、小数部分がある必要がある。そこで、基数として5.83、指数として8.01をとることにする。

関数のパフォーマンスを測るために二つの方法を用いる。一つは、UNIXのtimeコマンドであり、もう一つはC言語の標準関数である関数clockによって、同時に同じ処理に要する時間を測る。timeコマンドは、処理に要した時間を得ることが出来る。関数clockは処理のために使われる時間の総和が得られる。二つの方法によって得られる時間の違いは、スレッドによって並行処理が行われることによって生じる。関数clockは、並行処理の時間の和が得られるのに対して、timeコマンドは一番長いスレッドの処理時間である。したがって、timeコマンドで得られる時間のほうが短いことになる。

パフォーマンスを比較するべき乗の関数は、C言語の標準関数pow、で作成した三つの関数pow_double、関数pow_gmpおよび関数pow_threadである。一回だけの実行では、処理時間の測定限度以下の時間にしかならないので、以下に述べる

特別の場合を除いて、各関数での同じ処理を10万回繰り返させることにする。

関数pow_doubleにおいて２乗根の計算に自作の関数sqrt_doubleを用いているが、この関数をC言語標準関数sqrtに変更すると、パフォーマンスにどのような影響があるかは調べてみる価値はある。また、関数pow_gmpにおいて2乗根の計算にこれも自作の関数sqrt_gmpを用いているが、この関数をGMPライブラリのmpf_sqrtに変更して、パフォーマンスに対する影響も興味がある。

以上のことを行なうソースプログラムtest_powを図４に示す。四つのそれぞれのプログラムを順に10万回だけ呼び出す間に、関数clockを呼び出して時間を測定している。このプログラムをコンパイルし実行すると、その測定結果が画面に表示される。その実行の際に、timeコマンドを実行プログラム名の指定する前につけると、画面の先の測定結果の後に、全体とユザーとシステムの経過時間が三行でそれぞれ表示される。

表１．本研究でプログラム開発に利用したPCの主要な仕様。

CPU	intel core i7 3770K BOX
マザーボード	ASUS MAXIMUS IV EXTREME
メモリ	DDR3 PC3-10600 32GiB
OS	Ubuntu 13.10
コンパイラ	gcc4.8.1
多倍長計算ライブラリ	GMP5.1.0

４.２．考察

実行結果の画面を図５に示す。べき乗の計算結果は、関数 pow と関数pow_doubleでは15桁まで一致している。また、関数pow_gmpと関数pow_threadでは表示された全97桁が完全に一致している。この計算は、90桁の精度のはずであるが、二つの関数の違いはスレッドプログラミングを用いるかどうかの違いであり、処理内容は全く同じことを行なっているので、画面で同じ表示になっていることは当然の結果といえる。また、90桁の計算結果は、15桁までが関数powおよび関数pow_doubleの結果と同じである。このことは、90桁の多倍長計算が正しいであろうことが推測できる有力な根拠である。

関数powはC言語の標準関数である。その関数を10万回繰り返しても関数clockで処理時間の測定が可能ではない。非常にパフォーマンスに優れている。関数pow_doubleを10万回繰り返すと、関数clockで約0.30秒の処理時間になる。そこで、両方の関数を1000万回繰り返すと、関数powでは0.74秒、関数pow_doubleでは30.98秒の処理時間になる。約40倍以上のパフォーマンスの差がある。関数pow_doubleを作成した理由は、関数powの代替のためではなく、多倍長関数を作成するひな型であることを忘れてはならない。２乗根の計算も自作のものを関数pow_doubleで用いているが、これをC言語標準関数sqrtで変えると、1000万回繰り返す関数pow_doubleの処理時間は30.60秒となった。この差は僅かである。なお、PCでの処理時間測定は、PCの情報資源などの状況いわゆる実行環境から影響を受ける。そのため、処理時間の大きさにある程度の幅があるものと考えるのが妥当である。

関数pow_doubleをひな型として、多倍長ライブラリを用いて書き換えたのが、関数pow_gmpである。例題として、doubleの精度の約6倍の90桁を指定してべき乗計算を行った。関数pow_doubleを10万回繰り返す処理時間が0.31秒であるのに対して、関数pow_gmpを10万回繰り返す時間が28.55秒である。約６倍の精度の計算結果を得るために、約90倍以上の処理時間を要する。多倍長ライブラリGMPには、mpf_t型のデータを整数乗する関数mpf_pow_uiがある。この関数で関数sei_gmpを置き換えて実行すると、10万回実行する時間は28.01秒である。実行時間の測定のばらつきを考えると、この結果は公開されているGMPライブラリと同程度である関数sei_gmpの高い性能を示しているのではないだろうか。

関数 pow_gmp を構成する関数 sei_gmp と関数shou_gmpとの部分を、スッレドを用いて同時並行で処理できるようにしたのが、関数pow_threadである。純粋に同時並行処理だけならば、関数clockで測る処理時間が変わらないはずであるが、関数pow_gmpは28.77秒に対して、関数pow_threadは35.04秒である。このことは、スレッドをマルチコアのCPUで実行するために別途必要となる動作がかなりあることを示している。このソフトウェアでのオーバーヘッドが避けられないこと

が、ハードウェアとしてのCPUの進歩がマルチコアでしか実現していないことの矛盾である。マルチコアの多重化が進めば進むほどオーバーヘッドが増えることになる。ただし、マルチコアで処理しているので、処理時間がこのオーバヘッド分が単純に増えるわけではない。関数clockで得られる処理時間の総和は64.04秒であるが、経過時間を得るtimeコマンドでは59.05秒である。すなわち、スレッドで約５秒だけ処理時間が短くなったが、オーバーヘッドで６秒以上長くなっているのである。この結果、スレッドの効果がオーバーヘッドによって完全に無効となっていることが分かる。現状でのスレッド技術は、実用性の面で疑問符が付くのは避けられない。

パフォーマンスの詳細な検討の結果、多倍長ライブラリGMPを用いる関数pow_gmpが実用的であることが分かった。

（2013年12月2日受付、2014年1月14日受理）

謝　辞

本研究は、青森公立大学河野秀孝教授の多倍長精度でべき乗計算をする関数のプログラムを使いたいという要求から始まった。ここに感謝申し上げる。

参考文献

1)田中寛，「Markowitz型投資理論の数値解法」，青森公立大学経営経済学研究２no2，2-8(2002).
2)http://www.gmplib.org/
3)例えばhttp://www.tsoftware.jp/nptl/
4)例えばhttp://www.finetune.co.jp/~lyuka/technote/fract/sqrt.html

図１．関数pow_doubleのソースプログラム。C言語標準関数powの実行結果との比較のためと、多倍長計算用ライブラリGMPに書き換えるためのひな型である。

```
#include <math.h>
#define PREC_DOUBLE 16
#define EPS_DOUBLE 1.0e-16

double pow_double(x,y)
double x,y;
{
   double sei_double(),shou_double();

   if(x<0)
      printf("x=%f must be greater than 0.¥n",x);
   else if(x<=0)
      return 0;
   else if(y<0)
      return 1/pow_double(x,-y);
   else if(y>0)
      return sei_double(x,y)*shou_double(x,y);
   else if(y>=0)
      return 1;
}

double sei_double(x,y)
double x;
double y;
{
   int n_max,n,b[PREC_DOUBLE];
   double k[PREC_DOUBLE],y_p,x2,z;

   if(y>=0 && y<1)return 1;
   if(y>=1 && y<2)return x;
   y_p=y; k[0]=1;
   for(n=1;n<PREC_DOUBLE;n++)if((y_p-=(k[n]=2*k[n-1]))<0) break;
   n_max=n;

   for(n=0;n<PREC_DOUBLE;n++) b[n]=0;
   y_p=y;
   for(n=n_max;n>=0;n--) if(y_p>=k[n]) {y_p-=k[n]; b[n]=1;} else {b[n]=0;}

   x2=x; z=1;
   for(n=0;n<=n_max;n++){
      z*=b[n]?x2:1;
      x2*=x2;
   }

   return z;
}

double shou_double(x,y)
double x;
double y;
{
   int c,m;
   double k,x2,x_p,y_p,z;
   double sqrt_double();

   m=y; y-=m;
//   x2=x; z=1; x_p=sqrt(x2); y_p=y; k=1; m=0;
   x2=x; z=1; x_p=sqrt_double(x2); y_p=y; k=1; m=0;
   while(1){
      k/=2;
      if(y_p>=k){y_p-=k; c=1;} else c=0;
      z*=c?x_p:1;
      x2=x_p; x_p=sqrt_double(x_p);
      m++;
      if(fabs(x2-x_p)/x<EPS_DOUBLE)break;
   }

   return z;
}

double sqrt_double(a)
double a;
{
   double xn1,xn;

   xn=1/sqrt(a);
   xn1=1;

   while(fabs((xn1-xn)/xn)>EPS_DOUBLE){xn1=xn; xn*=(3-a*xn*xn)/2;}
   return (xn*a);
}
```

図２．関数 pow_gmp のソースプログラム。Pow_doubleのソースプログラムをGMPライブラリの記述の後に注記記号//で示している。

```
#include <math.h>

void pow_gmp(z,x,y)
mpf_t x,y,z;
{
   void sei_gmp(),shou_gmp();
   mpf_t y_p,z1,z2;

   mpf_init(y_p); mpf_init(z1); mpf_init(z2);

   if(mpf_sgn(x)<0)   //if(x<0)
      gmp_printf("x=%Ff must be greater than 0.¥n",x);
   else if(mpf_sgn(x)<=0)   //if(x<=0)
      mpf_set_ui(z,0);   //return 0;
   else if(mpf_sgn(y)<0){   //if(y<0)
      mpf_neg(y_p,y);
      pow_gmp(z,x,y_p);
      mpf_ui_div(z,1,z);
      //return 1/pow_double(x,-y);
   } else if(mpf_sgn(y)>0){   //if(y>0)
//       sei_gmp(z1,x,y); shou_gmp(z2,x,y); mpf_mul(z,z1,z2);
      {int m; m=mpf_get_d(y); mpf_pow_ui(z1,x,m); shou_gmp(z2,x,y); mpf_mul(z,z1,z2);}
      //return sei_double(x,y)*shou_double(x,y);
   } else if(mpf_sgn(y)>=0)   //if(y>=0)
      mpf_set_ui(z,1);   //return 1;
}

void sei_gmp(z,x,y)
mpf_t x,y,z;
{
   int n_max,n,b[PREC_GMP];
   mpf_t k[PREC_GMP],y_p,x2;

   mpf_init(y_p); mpf_init(x2);
   for(n=0;n<PREC_GMP;n++)mpf_init(k[n]);

   if(mpf_cmp_ui(y,0)>=0) if(mpf_cmp_ui(y,1)<0) {mpf_set_ui(z,1); return;}
   //if(y>=0 && y<1)return 1;
   if(mpf_cmp_ui(y,1)>=0) if(mpf_cmp_ui(y,2)<0) {mpf_set(z,x); return;}
   //if(y>=1 && y<2)return x;
   mpf_set(y_p,y); mpf_set_ui(k[0],1);   //y_p=y; k[0]=1;
   for(n=1;n<PREC_GMP;n++){
      mpf_mul_ui(k[n],k[n-1],2);
      mpf_sub(y_p,y_p,k[n]);
      if(mpf_sgn(y_p)<0) break;   //if((y_p-=(k[n]=2*k[n-1]))<0) break;
   }
   n_max=n;

   for(n=0;n<PREC_GMP;n++) b[n]=0;
   mpf_set(y_p,y);   //y_p=y;
   for(n=n_max;n>=0;n--){
      if(mpf_cmp(y_p,k[n])>=0){
         mpf_sub(y_p,y_p,k[n]); b[n]=1;
      } else
         b[n]=0;   //if(y_p>=k[n]) {y_p-=k[n]; b[n]=1;} else {b[n]=0;}
   }

   mpf_set(x2,x); mpf_set_ui(z,1);   //x2=x; z=1;
   for(n=0;n<=n_max;n++){
      if(b[n]) mpf_mul(z,z,x2);   //z*=b[n]?x2:1;
      mpf_mul(x2,x2,x2);   //x2*=x2;
   }

   //return z;
}

void shou_gmp(z,x,y)
mpf_t x,y,z;
{
   int c,m;
   mpf_t k,x2,x_p,y_p,z_p;
   void sqrt_gmp();

   mpf_init(k); mpf_init(x2); mpf_init(x_p); mpf_init(y_p); mpf_init(z_p);

   m=mpf_get_d(y); mpf_sub_ui(y_p,y,m);   //m=y; y-=m;
   mpf_set(x2,x); mpf_set_ui(z,1); sqrt_gmp(x_p,x2); mpf_set_ui(k,1); m=0;
   //x2=x; z=1; x_p=sqrt_double(x2); y_p=y; k=1; m=0;
   while(1){
      mpf_div_ui(k,k,2);   //k/=2;
```

```
    if(mpf_cmp(y_p,k)>=0){mpf_sub(y_p,y_p,k); c=1;} else c=0;
    //if(y_p>=k){y_p-=k; c=1;} else c=0;
    if(c) mpf_mul(z,z,x_p);   //z*=j?x_p:1;
    mpf_set(x2,x_p); sqrt_gmp(x_p,x_p);   //x2=x_p; x_p=sqrt_double(x_p);
//    mpf_set(x2,x_p); mpf_sqrt(x_p,x_p);   //x2=x_p; x_p=sqrt_double(x_p);
    m++;
    mpf_sub(z_p,x2,x_p); mpf_abs(z_p,z_p); mpf_div(z_p,z_p,x);
    if(mpf_cmp_d(z_p,EPS_GMP)<0)break;
    //if(fabs(x2-x_p)/x<EPS_DOUBLE)break;
  }

  //return z;
}

void sqrt_gmp(z,a)
mpf_t a,z;
{
  mpf_t xn,xn1,z_p;   //double xn,xn1;

  mpf_init(xn); mpf_init(xn1); mpf_init(z_p);

  mpf_set_d(xn,1/sqrt(mpf_get_d(a)));   //xn=1/sqrt(a);
  mpf_set_ui(xn1,1);   //xn1=1;

  while(1){
    mpf_sub(z_p,xn1,xn); mpf_abs(z_p,z_p); mpf_div(z_p,z_p,xn);
    if(mpf_cmp_d(z_p,EPS_GMP)<0)break;
    mpf_set(xn1,xn); mpf_mul(z_p,xn,xn); mpf_mul(z_p,z_p,a);
    mpf_ui_sub(z_p,3,z_p); mpf_div_ui(z_p,z_p,2); mpf_mul(xn,xn,z_p);
  }
  //while(fabs((xn1-xn)/xn)>EPS_DOUBLE){xn1=xn; xn*=(3-a*xn*xn)/2;}
  mpf_mul(z,xn,a);   //return (xn*a);
}
```

図３．関数pow_threadのソースプログラム。図２のソースプログラムをスレッド技術を用いて書き換えた。

```
#include <math.h>
#include <gmp.h>
#include <pthread.h>

typedef struct
{
   mpf_t x;
   mpf_t y;
   mpf_t z;
} xyz;

void pow_thread(z,x,y)
mpf_t x,y,z;
{
   pthread_t thread_sei,thread_shou;

   void *sei_thread(),*shou_thread();
   mpf_t y_p,z1,z2;
   xyz xyz_sei,xyz_shou;

   mpf_init(y_p); mpf_init(z1); mpf_init(z2);

   if(mpf_sgn(x)<0)    //if(x<0)
      gmp_printf("x=%Ff must be greater than 0.¥n",x);
   else if(mpf_sgn(x)<=0)    //if(x<=0)
      mpf_set_ui(z,0);    //return 0;
   else if(mpf_sgn(y)<0){    //if(y<0)
      mpf_neg(y_p,y);
      pow_thread(z,x,y_p);
      mpf_ui_div(z,1,z);
      //return 1/pow_double(x,-y);
   } else if(mpf_sgn(y)>0){    //if(y>0)
      mpf_init(xyz_sei.x); mpf_init(xyz_sei.y); mpf_init(xyz_sei.z);
      mpf_set(xyz_sei.x,x); mpf_set(xyz_sei.y,y);
      pthread_create(&thread_sei,NULL,sei_thread,&xyz_sei);
      mpf_init(xyz_shou.x); mpf_init(xyz_shou.y); mpf_init(xyz_shou.z);
      mpf_set(xyz_shou.x,x); mpf_set(xyz_shou.y,y);
      pthread_create(&thread_shou,NULL,shou_thread,&xyz_shou);
      pthread_join(thread_sei,NULL);
      pthread_join(thread_shou,NULL);
      mpf_mul(z,xyz_sei.z,xyz_shou.z);
   } else if(mpf_sgn(y)>=0)    //if(y>=0)
      mpf_set_ui(z,1);    //return 1;
}

void *sei_thread(xyz_sei)
void *xyz_sei;
{
   int n_max,n,b[PREC_GMP];
   mpf_t k[PREC_GMP],y_p,x2;
   mpf_t x,y,z;

   mpf_init(x); mpf_init(y); mpf_init(z);
   mpf_set(x,((xyz *)xyz_sei)->x); mpf_set(y,((xyz *)xyz_sei)->y);

   mpf_init(y_p); mpf_init(x2);
   for(n=0;n<PREC_GMP;n++)mpf_init(k[n]);

   if(mpf_cmp_ui(y,0)>=0) if(mpf_cmp_ui(y,1)<0) {mpf_set_ui(z,1); return;}
   if(mpf_cmp_ui(y,1)>=0) if(mpf_cmp_ui(y,2)<0) {mpf_set(z,x); return;}
   mpf_set(y_p,y); mpf_set_ui(k[0],1);
   for(n=1;n<PREC_GMP;n++){
      mpf_mul_ui(k[n],k[n-1],2);
      mpf_sub(y_p,y_p,k[n]);
      if(mpf_sgn(y_p)<0) break;
   }
   n_max=n;

   for(n=0;n<PREC_GMP;n++) b[n]=0;
   mpf_set(y_p,y);
   for(n=n_max;n>=0;n--){
      if(mpf_cmp(y_p,k[n])>=0){
         mpf_sub(y_p,y_p,k[n]); b[n]=1;
      } else
         b[n]=0;
   }

   mpf_set(x2,x); mpf_set_ui(z,1);
   for(n=0;n<=n_max;n++){
      if(b[n]) mpf_mul(z,z,x2);
```

```
    mpf_mul(x2,x2,x2);
  }
  mpf_set(((xyz *)xyz_sei)->z,z);
}

void *shou_thread(xyz_shou)
void *xyz_shou;
{
  int c,m;
  mpf_t k,x2,x_p,y_p,z_p;
  void sqrt_thread();
  mpf_t x,y,z;

  mpf_init(x); mpf_init(y); mpf_init(z);
  mpf_set(x,((xyz *)xyz_shou)->x); mpf_set(y,((xyz *)xyz_shou)->y);

  mpf_init(k); mpf_init(x2); mpf_init(x_p); mpf_init(y_p); mpf_init(z_p);

  m=mpf_get_d(y); mpf_sub_ui(y_p,y,m);
  mpf_set(x2,x); mpf_set_ui(z,1); sqrt_thread(x_p,x2); mpf_set_ui(k,1); m=0;
  while(1){
    mpf_div_ui(k,k,2);
    if(mpf_cmp(y_p,k)>=0){mpf_sub(y_p,y_p,k); c=1;} else c=0;
    if(c) mpf_mul(z,z,x_p);
    mpf_set(x2,x_p); sqrt_thread(x_p,x_p);
    m++;
    mpf_sub(z_p,x2,x_p); mpf_abs(z_p,z_p); mpf_div(z_p,z_p,x);
    if(mpf_cmp_d(z_p,EPS_GMP)<0)break;
  }

  mpf_set(((xyz *)xyz_shou)->z,z);
}

void sqrt_thread(z,a)
mpf_t a,z;
{
  mpf_t xn,xn1,z_p;

  mpf_init(xn); mpf_init(xn1); mpf_init(z_p);

  mpf_set_d(xn,1/sqrt(mpf_get_d(a)));
  mpf_set_ui(xn1,1);

  while(1){
    mpf_sub(z_p,xn1,xn); mpf_abs(z_p,z_p); mpf_div(z_p,z_p,xn);
    if(mpf_cmp_d(z_p,EPS_GMP)<0)break;
    mpf_set(xn1,xn); mpf_mul(z_p,xn,xn); mpf_mul(z_p,z_p,a);
    mpf_ui_sub(z_p,3,z_p); mpf_div_ui(z_p,z_p,2); mpf_mul(xn,xn,z_p);
  }
  mpf_mul(z,xn,a);
}
```

図４．四つの関数すなわちpow、pow_double、pow_gmp、 pow_threadを、一連として実行させるためのソースプログラム。

```
//*  gcc test_pow.c -lgmp -lm -lpthread

#define PREC_GMP 90
#define EPS_GMP 1e-90
#define NUM 100000

#include <stdio.h>
#include <math.h>
#include <gmp.h>
#include "pow_double.c"
#include "pow_gmp.c"
#include "pow_thread.c"

#include <sys/times.h>
#include <time.h>

main()
{
   int i;
   double x,y;
   mpf_t xp,yp,z;

   clock_t t;

   t=clock(); printf("t=%ld   %f secs¥n",(long) t,(double) t/CLOCKS_PER_SEC);

   x=5.38;
   y=8.01;

   t=clock(); printf("t=%ld   %f secs¥n",(long) t,(double) t/CLOCKS_PER_SEC);
   for(i=0;i<NUM;i++) pow(x,y);
   printf("pow          x=%f y=%f z=%25.20f¥n",x,y,pow(x,y));
   t=clock(); printf("t=%ld   %f secs¥n",(long) t,(double) t/CLOCKS_PER_SEC);
   for(i=0;i<NUM;i++) pow_double(x,y);
   printf("power_double x=%f y=%f z=%25.20f¥n",x,y,pow_double(x,y));
   t=clock(); printf("t=%ld   %f secs¥n",(long) t,(double) t/CLOCKS_PER_SEC);
   mpf_set_default_prec(PREC_GMP*log(10)/log(2));
   mpf_init_set_d(xp,x); mpf_init_set_d(yp,y); mpf_init(z);
   for(i=0;i<NUM;i++) pow_gmp(z,xp,yp);
   pow_gmp(z,xp,yp);
   gmp_printf("power_gmp    x=%Ff y=%Ff z=%97.90Ff¥n",xp,yp,z);
   t=clock(); printf("t=%ld   %f secs¥n",(long) t,(double) t/CLOCKS_PER_SEC);
   for(i=0;i<NUM;i++) pow_thread(z,xp,yp);
   pow_thread(z,xp,yp);
   gmp_printf("power_thread x=%Ff y=%Ff z=%97.90Ff¥n",xp,yp,z);
   t=clock(); printf("t=%ld   %f secs¥n",(long) t,(double) t/CLOCKS_PER_SEC);
}
```

図５．図４で示すソースプログラムをコンパイルし実行したときの画面。それぞれの関数での計算結果の比較が出来る。また、関数**clock**による処理時間の測定と**time**コマンドによる経過時間の観測結果を見ることが出来る。

```
t=0    0.000000 secs
t=0    0.000000 secs
pow          x=5.380000 y=8.010000 z=713782.71703627938404679298
t=0    0.000000 secs
power_double x=5.380000 y=8.010000 z=713782.71703627915121614933
t=310000    0.310000 secs
power_gmp    x=5.380000 y=8.010000
z=713782.71703627943494662082437321733914796430896094281354376749364708125116548714719083632649406 2
t=29080000    29.080000 secs
power_thread x=5.380000 y=8.010000
z=713782.71703627943494662082437321733914796430896094281354376749364708125116548714719083632649406 2
t=64040000    64.040000 secs

real    1m1.106s
user    0m59.052s
sys     0m5.132s
```

Program Development of The Exponentiation Function Using The Multi-precision Calculation Library GMP

Hiroshi TANAKA

Abstract

Indispensable exponentiation calculation of computational economics is not included in the present GMP library. Although there is a function named function mpf_pow_ui in GMP, there is only a function of calculation of integer power. It does not correspond to calculation of the exponentiation of number whose index is 0 or more and 1 or less decimal. Moreover, it cannot be used for calculation of the exponentiation whose index is the number of a decimal point or more by one. But the C language itself has the function pow as a standard function, and the accuracy is 15 figures in a decimal number. The purpose of this paper is to develop the program of a exponentiation function which procedures multi-precision exponentiation calculation of a number whose index is a number with a general decimal point.