# 第4章　多倍長計算

**永坂**[a]「でしょうね。その後，計算機によって新しい代数方程式の問題なども出てきた。計算機の回路が全部10進で，データ作成もチェックも10進でやる場合は解がびっちり出るんです。それが2進回路になってからは，10進のデータを2進に変換したりする過程で誤差が出るんです。こうした誤差に対する認識は，外国ではしっかりしているんですが，日本人は今でもその辺の認識が甘くて，高精度計算すればよいと考える傾向が強いようですね。」

遠藤諭「計算機屋かく戦えり」（アスキー）

[a]元・日本大学教授

多倍長浮動小数点数 (multiple precision floating-point number) とは，仮数部の桁数をより多く指定できる浮動小数点数のことを指す。任意精度 (arbitrary precision) とも呼ぶ。このような浮動小数点数を扱う計算を一般に多倍長計算 (multiple precision arithmetic) と総称することにする。

前の章でも触れたように，現在の PC や WS の環境では IEEE754 standard に基づいた，仮数部の桁数が固定された浮動小数点数を扱うのが普通である。これらはハードウェア (CPU) で直接処理できるものであり，高速な演算が期待できる。反面，多倍長計算を行うには，現在の所，ソフトウェアで多くの命令を組み合わせて実現するほかなく，前者に比べると計算時間はずっと大きくなる。

しかし，数値計算の途中で発生する丸め誤差を減らす根本的な解決策は，前章でも示した通り

- アルゴリズムを改良（改変）する
- マシンイプシロン (丸め誤差の最小単位) を小さくする

他なく，特に後者は前者に比べて簡単な（安易な）ものである。現在の数式処理ソフトウェアで多倍長計算が利用できないものは皆無であり，プログラミング言語から簡単に利用できるライブラリとして提供されているものも少なくない。多倍長計算が利用できる環境はかなり整ってきており，数値計算にこれを利用しない手はない。

本章ではごく大ざっぱに，多倍長計算の現状とその利用方法について考察を行う。

## 4.1　多倍長計算の適用例

前章の例題 3.4.2 を，数式処理ソフトウェアの一つである MuPAD[24] を使って，再度計算を行ってみる。

```
x[0] := 0.7501:
for i from 0 to 100 step 1 do
    x[i + 1] := 4 * x[i] * (1 - x[i]):
end_for:
for i from 0 to 100 step 10 do
    print(i, x[i]);
end_for;
```

図 4.1: ロジスティック写像を計算する MuPAD スクリプト

計算には図 4.1 のスクリプトを用いた。MuPAD で計算桁数を指定するときには DIGITS というグローバル変数に 10 進数での桁数を指定する。このスクリプトの最初の行にこれを付加することで，それ以下の計算が全てその桁で実行される。

以下は DIGITS := 20(10 進 20 桁計算) としたときの結果である。

```
  0, 0.7501
 10, 0.84449595360221744753
 20, 0.14293972451230765528
 30, 0.85429600370442189166
 40, 0.77497575311820123972
 50, 0.093375332197704141841
 60, 0.40822016829280303573
 70, 0.071511998671680367325
 80, 0.46325125515737645908
 90, 0.0011853049045088519022
100, 0.41894573012901575815
```

これと前章の IEEE754 倍精度の計算結果と見比べると，どうも $x_{40}$ の頭 4 桁ほどは一致しているが，それ以下の桁は丸め誤差が桁落ちにより上がってしまっている，ということが分かる。$x_{50}$ より以降は全く信頼できない計算結果である。

以下は DIGITS := 50(10 進 50 桁計算) としたときの結果である。

```
  0, 0.7501
 10, 0.84449595360221744753714870256154137267401952291229
 20, 0.14293972451230765528428175723130628307376969200214
 30, 0.85429600370442189166613118425888744374797589008823
 40, 0.77497575311820124128022346126421168481511313675586
 50, 0.09337533219770302905518966801516823476282301346205 4
 60, 0.40822016829087813187102766181952686730359229911213
 70, 0.07151199970505857489709934641759321835031124057788 7
 80, 0.46325330290077571055993467458320747053550671068192
 90, 0.00133440501208688404646920121500211076211366084892 91
100, 0.07881798937150990680670477044008676265065876728396 6
```

ここにきて，ようやく $x_{90}$ の一桁目ぐらいは信頼しても良さそうだ，ぐらいのところまできた。しかし $x_{100}$ の数値はまだ信用できない。

以下は DIGITS := 100(10 進 100 桁計算) としたときの結果である。紙面の都合上，末尾の 10 桁は切り捨てている。

```
  0, 0.7501
 10, 0.844495953602217447537148702561541372674019522912291280007848388118986290823419270975121157
 20, 0.142939724512307655284281757231306283073769692002141121387637872878254442652405447154653323
 30, 0.854296003704421891666131184258887443747975890082277146848716525830873157517999293072280730
 40, 0.774975753118201241280223461264211684815113136725464353115716739919423777262939033931884043
 50, 0.093375332197703029055189668015168234762823035148745302006641168172968513199758645546034917 8
 60, 0.408220168290878131871027661819526867303629812881299069587191914174238088850476295365965331
 70, 0.071511999705058574897099346417593218330172098833077217051255851990381440740010828690073796 2
 80, 0.463253302900775710559934674583207430627755245547675072149154489741355095166291517037937750
 90, 0.001334405012086884046469201214999119068327678471079100334790947016197798656540387577848753 52
100, 0.078817989371509906806704770462699264724009445034603844115443436664272838000468909027496070 9
```

ここで，ようやく $x_{100} = 0.078817989371509906806704770 4...$ であろうと言うことが出来る[1]。

より長い桁で計算した結果と比較することで，丸め誤差の大きさを把握する方法が古くから使用されてきた。固定された桁数の計算だけで精度を一定範囲で保証する区間解析法や，打ち切り誤差も含めた精度保証の開発も盛んであるが，誤差を過大評価しがちになったり，適用できるアルゴリズムや問題が限定されるという難点もある。上記の手法も，後述するように計算時間が過大になるという難点はあるが，一定の見積もりを取る，という点ではどのような問題にも適用可能である。丸め誤差の影響で必要な精度を得ることが困難なケースでは，多倍長計算の利用も魅力的な選択肢の一つと言える。

## 4.2 多倍長計算環境を提供するソフトウェア

現在，コンピュータ上で多倍長計算を行うには，前節のように数式処理ソフトウェアを利用するか，プログラミング言語から呼び出し可能な多倍長計算ライブラリを利用することになる。

数式処理ソフトウェアの例として，前節では MuPAD を取り上げたが，これは後発のソフトウェアである。著名なものとしては前述したように Mathematica がある。ロジスティック写像を計算するスクリプトを Mathematica で記述すると図 4.2 のようになる。`N[`評価式，桁数`]` とすることで桁数の指定が可能になっている。

プログラム言語から，関数もしくはクラスライブラリとして利用可能な多倍長計算ライブラリも様々なものが提供されている。ここでは GNU MP(以下 GMP と略記する) の例を取り上げる。2006 年 8 月現在，Version 4.2.1 が最新版となっている。土台は C ネイティブの関数群からできており，テスト版ではあるが C++のクラスインターフェースも提供されている。図 4.3 は C 言語の関数を使った例と C++のクラスを使った例である。演算子の overload 機能を利用できるため，左では 3 行費やしていた漸化式の計算部分が，右では 1 行の式として表現でき，結果としては同じ処理を行っているのだが，右の方が短く記述できている。

多倍長計算を利用する立場としては，MuPAD や Mathematica のような数式処理ソフトウェアを使うか，GMP のようなライブラリを使用すると一見楽が出来るように思える。しかし，それは複雑な部分を表に出さないよう，black box 化されているだけのことであり，「見栄え」の恩恵はあるかも知れないが，計算時間や記憶容量の減少といった実質的な恩恵は殆ど期待できない。もし実を

[1] 解析的に真の値を求めることの出来ないものを数値計算で求める際に留意すべき事は，伊理の本 [13] の「第 4 章 たった 1 回だけの計算なんて … 」に詳しく例題付きで述べられている。

```
prec = 100;
x = Table[0, {i, 0, 101}];
x[[0]] = N[7501/10000, prec];
For[i = 0, i <= 100,
    x[[i + 1]] = 4*x[[i]]*(1 - x[[i]]);
    i++;
]
For[i = 0, i <= 100,
    Print[i, ", ", x[[i]]];
    i += 10;
]
```

図 4.2: ロジスティック写像を計算する Mathematica スクリプト

取ろうとするのであれば，様々なソフトウェアを比較したり，自身でプログラムを組み立てたりするなど，それなりに試行錯誤を要求されるのである。

## 4.3　現行の多倍長計算ソフトウェアの比較

以上見てきたように，多倍長計算を行う環境は整っている。しかし，多倍長計算は多くの機械語命令を多数組み合わせたソフトウェアで実現されているため，CPU で直接処理可能な浮動小数点演算に比べて処理に時間がかかる。ではどのぐらい速度が落ちるのか，またソフトウェア間で速度の相違はどの程度存在するだろうか。

以下の環境下でベンチマークテストを行ってみる。

**CPU** Intel Pentium III 750MHz

**RAM** 384MB

**OS** Windows XP Professional

**C compiler** GCC 2.95

**GMP** Ver.4.0.1 (`./configure; make; make install` でインストール)

**Software** MuPAD Pro 2.0, Mathematica 4.1

ベンチマークテストに使用したプログラムは，MPFR プロジェクト (`http://www.mpfr.org/`) で提供されているものを使用した。各演算に要する時間は，同じ演算を多数繰り返すことで得るようになっている。その結果を表 4.1 と表 4.2 に示す。時間は全てミリ秒である。

多倍長計算については，CPU のアーキテクチャに依存して様々なチューン方法が存在しうるため，上記の結果が他のコンピュータ環境で通用するかは不明である。しかしながら，このベンチマークテストを行った環境下では，ほぼ GMP の計算速度が最良のものであるといえる。他にも多

表 4.1: IEEE754 浮動小数点数の演算速度

| | IEEE 倍精度 |
|---|---|
| $x+y$ | 0.000010 |
| $x-y$ | 0.000011 |
| $x\times y$ | 0.000009 |
| $x/y$ | 0.000053 |
| $\sqrt{x}$ | 0.00157 |

表 4.2: 多倍長計算ソフトウェアの比較

| | MuPAD Pro | Mathematica | GMP |
|---|---|---|---|
| 10 進 100 桁 | | | |
| $x+y$ | 0.0060 | 0.0054 | 0.00040 |
| $x-y$ | 0.0070 | 0.0092 | 0.00050 |
| $x\times y$ | 0.011 | 0.011 | 0.0022 |
| $x/y$ | 0.012 | 0.073 | 0.0041 |
| $\sqrt{x}$ | 0.036 | 0.097 | 0.0076 |
| 10 進 1000 桁 | | | |
| $x+y$ | 0.010 | 0.0090 | 0.002 |
| $x-y$ | 0.020 | 0.0161 | 0.001 |
| $x\times y$ | 0.27 | 0.21 | 0.079 |
| $x/y$ | 0.32 | 0.63 | 0.13 |
| $\sqrt{x}$ | 0.29 | 0.90 | 0.085 |
| 10 進 10000 桁 | | | |
| $x+y$ | 0.10 | 0.040 | 0.01 |
| $x-y$ | 0.10 | 0.07 | 0.01 |
| $x\times y$ | 25 | 5.2 | 2.62 |
| $x/y$ | 26 | 25 | 5.53 |
| $\sqrt{x}$ | 93 | 29 | 3.52 |

```c
#include <stdio.h>
#include "gmp.h"

main()
{
    int i;
    mpf_t x[102];

    mpf_set_default_prec(2048);

    mpf_init_set_str(x[0], "0.7501", 10);
    for(i = 1; i <= 101; i++)
            mpf_init(x[i]);

    for(i = 0; i <= 100; i++)
    {
      mpf_ui_sub(x[i+1], 1UL, x[i]);
      mpf_mul(x[i+1], x[i], x[i+1]);
      mpf_mul_ui(x[i+1], x[i+1], 4UL);
      if(i%10 == 0)
       gmp_printf("%5d %58.50Fe\n", i,
x[i]);
    }
}
```

```c
#include <iostream>
#include <iomanip>
#include "gmpxx.h"

using namespace std;

main()
{
    mpf_set_default_prec(2048);

    int i;
    mpf_class x[102];

    x[0] = "0.7501";

    for(i = 0; i <= 100; i++)
    {
      x[i+1] = 4 * x[i] * (1 - x[i]);
      if(i%10 == 0)
        cout << setw(5) << i << ' ' \
<< setw(58) << scientific << \
setprecision(50) << x[i] << '\n';
    }
}
```

図 4.3: ロジスティック写像を計算する GMP を使った C(左)/C++(右) プログラム

倍長計算が可能なライブラリは幾つか存在するが，速度の order は GMP のそれと似たり寄ったりであり，四則演算レベルで見る限り，これ以上の劇的な速度向上が見込める状況にはない。GMP が安定したパフォーマンスを誇っているのは，桁数に応じて使用するアルゴリズムを変えているためである。詳細については GMP のマニュアル [1] を参照されたい。また，そこで使用されている各種の四則演算アルゴリズムについては Knuth[3] が詳しい。

以上の結果より，上記の環境下では CPU で直接処理できる演算に比べ，ソフトウェアによる多倍長計算は非常に速度が遅くなることが分かる。これはコンピュータのアーキテクチャに依らず，一般的な傾向である。

## 4.4　多倍長計算の存在意義

一般に数値計算は，ハードウェアで直接処理出来る，有効桁数が固定された浮動小数点数を使用して行われる。それ以上の精度が要求される場合のみ，ソフトウェアで実現される任意精度の浮動小数点演算（多倍長計算）が実行される。その理由としては

1. 計算速度が低下する
2. 現行の精度で間に合っている
3. 必要となる記憶領域が増える

ということが挙げられる。従って，多倍長計算は数値計算における一種のオプションとして考えられている。

歴史を降り返ってみると，ハードウェアの高機能化に合わせて，サポートされる浮動小数点数の有効桁数は増えてきている。ユーザにとっては，計算速度があまり変化しないのであれば，精度は高いに越したことはない。そこで，多倍長計算が利用できるようになってくると，次のような疑問が浮かんでくる。

1. 浮動小数点数の有効桁数に鋭敏なアルゴリズムは，多倍長計算を利用することで，現在最良とされている頑健なアルゴリズムよりも速度面で凌駕する可能性があるのではないか？
2. 悪条件問題において，浮動小数点数の精度が数値解の精度に与える影響はどの程度のものか？特に最良の精度との関係を図示するとどのようなものになるのか？

これらを理論的に追求するのは困難であると思われるが，数値実験を網羅的に行うことである程度の見積もりを得ることは可能であろう。但し前者については，前節のベンチマークテストの結果から，現在の所あまり期待できないと言ってよい。大雑把に図示すれば図 4.4 のようになろうか。

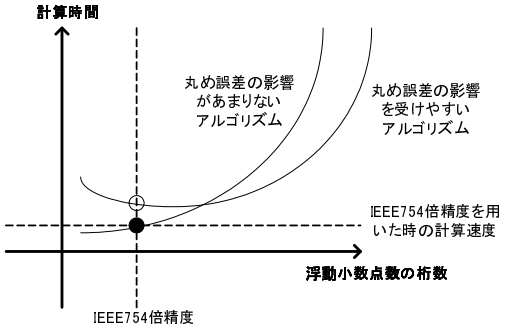


図 4.4: 浮動小数点数の桁数と計算時間との関係

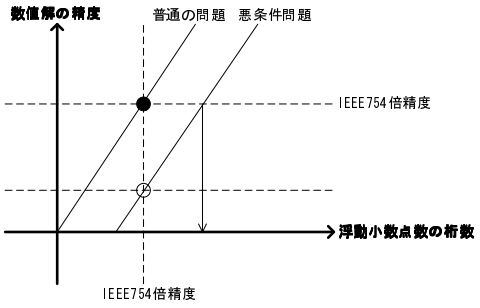


図 4.5: 浮動小数点数の桁数と数値解の精度との関係

従って現状では，悪条件問題において必要な精度を得るために多倍長計算を使用する，ということが，多倍長計算の一番の存在意義となろう (図 4.5)。それも，問題自体の初期誤差が任意に調整できる，限定された状況においてのみである。多倍長計算を適用するにあたっては，図 3.2 において，入力数値の変換誤差，計算途中に発生する丸め誤差，数値を出力する際に発生する出力（変換）誤差にのみ，効果があることをしっかり認識しておく必要がある。さもなければ，多大な誤差を含む数値を使って，無意味にコンピュータ資源を費やすことになるだろう。

## 演習問題

1. 例題 3.4.1 の解の公式に係数 $a, b, c$ をそのまま適用した場合，$x_2$ が 10 進 15 桁程度の精度を持つためには，10 進でどの程度の桁数での計算が必要か？また，多倍長計算が実行可能なソフトウェアを用いると，どの程度計算時間がかかるかを調べよ。

2. 本章で示した logistic 写像の多倍長計算にどの程度の計算時間を要するのかを調べよ。例えば Mathematica で実行した時はどうなるか？

## 参考図書

多倍長計算，特に四則演算レベルのアルゴリズムをきちんと議論した書籍としては

**準数値算法/算術演算**
D.E.Knuth/中川圭介　訳
サイエンス社
1986 年

が一番である。多倍長計算の実装は様々なものがあるが，ここで取り上げた GMP のマニュアルは参考になる。Version 4 以降はアルゴリズムや内部構造の解説も追加されている。多倍長計算のプログラムを組む際には一度目を通しておいて損はない。

**GNU MP**
`http://swox.com/gmp/`