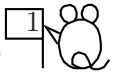
数値計算講義　第1２回
多倍長演算・精度保証計算

カーネンコ　アレクセイ
金子　晃

kanenko@mbk.nifty.com
alexei.kanenko@docomo.ne.jp
http://www.kanenko.com/

多倍長演算



通常のプログラミング言語でサポートされている変数の桁数以上の精度で計算すること.
無限精度演算とか任意精度演算ともいう.
原理: ソロバンを繋げれば (部屋に納まる限り) いくらでも大きな桁数で計算できる.

i.e. 変数を繋げれば (メモリーの許す限り) いくらでも大きな桁数で計算できる.

```
KETA(1)   KETA(2)   KETA(3)   KETA(4)   KETA(5)
```

任意精度浮動小数の最も簡単なフォーマット

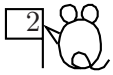
- 正規化された浮動小数の仮数部を十進4 桁ずつ整数配列に格納する.
- 小数点の位置を冪指数として記憶する.

この他に, 全体の符号と仮数部の桁数を記憶する必要があるので
結局, 多倍長浮動小数 A とは次のような内容の整数配列 (long) となる:

A(1) ··· 全体の符号 (±1)
A(2) ··· 指数部 (十進, 符号付き)
A(3) ··· 仮数部の現在の長さ (使用しているユニット数)
A(4) ··· 仮数部の最初の4 桁 (十進 0 〜 9999)
⋮

※ C 言語の場合は, 適当な構造体を定義すれば, もっとプログラムし易くなる.

多倍長浮動小数の演算



加法 小数点の位置を大きい方の数に合わせて加える

例: $0.1234 \times 10^2 + 0.4323 \times 10^1$ のとき,
$= 0.1234 \times 10^2 + 0.04323 \times 10^2$

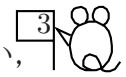
桁揃え 立場により二つに分かれる:

(1) 小数点以下4 桁に精度を固定しているとき,
このときは, 第2 の数を丸めて加える:
$= 0.1234 \times 10^2 + 0.0432 \times 10^2 = 0.1666 \times 10^2$

(2) 精度はもう少しとってあり, 二つの数は正確
(i.e. 上記の部分の後ろに 0 が精度の分だけ続いている)
このときは, 仮数部の長さを1 増やして続ける:
$= 0.1234 \times 10^2 + 0.04323 \times 10^2 = 0.16663000 \times 10^2$

`A(5)` には 3000 を入れるのが正しい. 間違えて 3 を入れると,
0.16660003×10^2 ができてしまう.

繰り上がり処理 加法の結果あるユニットで5 桁になったら必要となる.
繰り上がりは次々波及するかもしれないので, 汎用に書くのは結構大変.
cf. 全加算器の設計.
最上位のユニットで繰り上がりが起こると, 指数の調整が必要となる.
例: $0.1234 \times 10^4 + 0.9876 \times 10^4 = 1.1110 \times 10^4 = 0.1111 \times 10^5$
4 桁ずつの格納法だと, この変更は大変
(すべてのユニットで1 桁のずらしが必要となる). なので, 冪は 4 の倍数とし
$= 0.00011110 \times 10^8$ でアレンジされたものとしてしまうのが簡単.
(ただしこの種の処理が続くと有効桁数が損なわれるという欠点がある.)

減法 同様だが，引けない場合は上から借りて来る必要が有る．


便法として，負になってもよいから対応する部位から引いてしまい，
後で一括アレンジするという流儀も有る．
加減算だけが続くときは, この方が高速, かつ1 万回程度の演算数なら，
各ユニットにも途中結果を格納する余裕がある．

実際の計算では, データは符号付きなので,
加法· 減法のどちらを実行するかは, 第2 オペランドの符号と合わせて決まる．
答の符号は最上位ユニットが正規化後にも正にならないとき反転する．
この場合は正規化をやり直さねばならない．

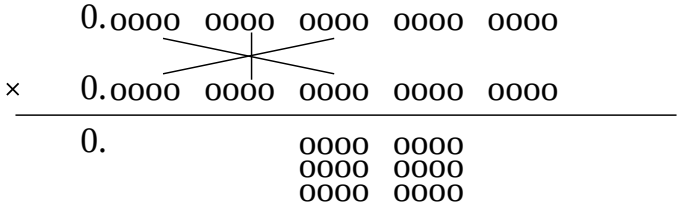
乗法 指数部は単なる和. 仮数部は基本的に整数の乗算と同様
例: $0.1234 \times 10^4 * 0.5678 \times 10^8 = 0.1234 * 0.5678 \times 10^{12}$

```
    1234
x   5678
-----------
    6170
     7404
      8638
       9872
-----------
    7006652
```

右の計算より, 仮数部は 0.07006652 となる．
簡易アレンジ法の下では,
(1) 精度が4 桁に指定されているときは, 答 0.0701×10^{12}
(2) 精度が8 桁以上のときは, 答 $0.07006652 \times 10^{12}$
となる．
一般に, $C = A \times B$ とし, A の仮数の第 k ユニットを A_k と記せば

$$C_k = \sum_{j=1}^{k} (A_j \times B_{k-j+1}\text{の上位4 桁} + A_j \times B_{k-j}\text{の下位4 桁})$$

となる (ただし, 右辺で添字が範囲外となった因子は 0 とみなす.)
特に，中央付近のユニットでは総ユニット数程度の加算が生ずる．
よって乗算結果は一回毎にアレンジするのがよい．



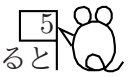
除法 指数部は引き算.
仮数部の除法は仮商を立てる小学生の筆算式にやればできるが, 低速.
反復法, 特に Newton 法の援用など, 種々の工夫が行われているが,
いずれにしても他の演算に比して非常に時間がかかる.

組み込み函数の実装 単精度や倍精度で `sqrt`, `exp`, `sin`, `cos`, `log` などの
組み込み函数がどのように実装されているかを調べ, 真似をする.
チューニングが進みすぎているものは真似ができないので, 基本に帰る.

例:

- `sqrt(x)` Newton 法を適用した通常のライブラリ函数の実装法が
  そのまま使える. しかも, 割り算を避けた工夫が多倍長では非常に有効.
- `exp(x)` 概算で `exp(y)`$\times 10^n$, $-3 < y < 0$ の形にする:
  $n = \lceil \dfrac{x}{\log 10} \rceil$, $y = x - n \log 10$,
  `exp(y)` は Taylor 展開で求める.
  通常の倍精度浮動小数用ライブラリでは, 同様に 2 冪を括り出した後,
  `exp(y)` の計算には, 倍精度が保証された複雑な係数の近似多項式などが
  使われるが, それは倍精度を越えたら使えない:
  数値計算の中には, 函数近似法というトピックがある.

C++ による多倍長演算の実装



多倍長浮動小数のクラス infprec を定義し, その上の演算を実装すると
- 演算子をオーバーロードできる.
- 多倍長数に対する組み込み函数も定義して追加できる.

等の C++ の特徴が使える.
この結果, 倍精度浮動小数に対するのとほとんど同じ表記で
多倍長数での計算が可能となる:

```
infprec A, B, C;
...
C=A*B-3*A+2*B/A;
....
```

高速に実装するには, 多倍長浮動小数型の引数のコピーが
頻発しないように, サブルーチン間のデータの受渡しの書き方を
工夫する必要がある.
ただの配列 A なら, 普通に引数に書けば, A のアドレスが渡る (参照渡し).
しかし, A が infprec 構造体宣言されていると, デフォールトでは A の内容が
コピーされてサブルーチンに渡される.
これを防ぐには,

```
void sub(infprec &A){
......
```

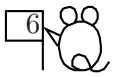
のような書き方で, 参照渡しを宣言するとよい.
(ポインターではないので, 指すアドレスは変化せず, 安全.)
更に, A の内容を sub の中で変更されたく無い場合は,

```
void sub(const infprec &A){
......
```

とすると, アドレス参照なのに sub の中で A の要素の値が修正不可能となる.

Q C 言語流に infprec 宣言をポインタで行うことも可能.
このときは, 生成・破壊演算子でメモリを確保, 開放するように書く.
また, 構造体の内部を private にして C++のセキュリティ機能を使うべきである.

既存の多倍長演算システム



UBASIC 老舗だが, X86 (特に昔の NEC PC) のアセンブラで書かれており , 著者に Windows への移植の意志が無かったので, 今は使われない.

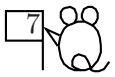
PARI ボルドー大で開発され C 言語で書かれた, 主に整数論の計算用ツールだが, 初等函数の多倍長浮動小数演算もサポート .
`libpari.a` の形でライブラリとしても提供されており , この中の函数を自分のプログラムにリンクして使える他, ソースが公開されているので, 必要な部分を取って来て自分のプログラムに取り込んだりもできる .
(Risa/Asir も多倍長演算にこれを利用している .)

GMP GNU MultiPrecision library. gcc 用のライブラリとして, 2000 年頃から出ている .
今後はフリーの多倍長演算ライブラリの主流になると思われる

その他個人の作品 小生を含めて多くの人が作って個人的に使っている .
最近のお勧め : **exflib**
`http://www-an.acs.i.kyoto-u.ac.jp/~fujiwara/exflib/`

数式処理ソフトに組み込まれた多倍長演算機能 主な数式処理ソフト Mathematica, Maple, Macsyma (Maxima), Matlab などは, いずれもその中で精度を任意に指定した多倍長浮動小数の演算が可能.
しかし , これらを , 自分で書いた C のプログラムとリンクするのは困難.

GMP の使用例 – C 言語編 (gmptest.c)



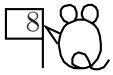

```
#include<stdio.h>
#include<stdlib.h>
#include<gmp.h>                /* ヘッダファイルの挿入 */
int main(void)
{
  int i;
  mpf_t  s,t;                  /* 型宣言 */
  mpf_set_default_prec(1000);
  mpf_init(s);                 /* 変数を 0 で初期化 */
  mpf_init_set_str(t,"1e0",10); /* 変数を 1 で初期化 (10 は radix)*/
  for (i=1;i<=10000;i++){
     mpf_add(s,s,t);           /* s <- s+t */
     mpf_div_ui(t,t,i);        /* t <- t*i */
  }
  mpf_out_str(stdout,10,1000,s); /* 結果の出力 (10 は radix) */
  mpf_clear(t);                 /* 必須 */
  mpf_clear(s);
  return 0;
}
```

コンパイルは5 階に GMP が標準でインストールされたので次のようにする :

```
gcc gmptest.c -lgmp -o gmptest
```

ライブラリ libgmp.a は C 言語用で, C++ 用は libgmpxx.a
このライブラリでは, 他に, 整数型 mpz_t, 有理数型 mpq_t をサポート .
指令の詳細はディレクトリ内のマニュアル gmp-man-4.3.0.pdf を参照せよ .

π の計算



古代エジプト　$3,\ \left(\frac{4}{3}\right)^4 \fallingdotseq 3.16$

Archimedes $\frac{223}{71} < \pi < \frac{22}{7}$ (内接正 96 角形の辺長)

劉徽 (3 世紀)　$\frac{22}{7} = 3.1428571\cdots,\ \frac{157}{50} = 3.14,\ \frac{3927}{1250} = 3.1416$

祖沖之(5 世紀)　約率 $\frac{22}{7}$, 密率 $\frac{355}{113} = 3.14159292\cdots,$
$3.1415926 < \pi < 3.1415927$

Āryabhaṭa (インド，5 世紀) 3.1416

Adriaan Anthoniszoon (1527–1607)　1585 年に $\frac{355}{113}$

François Viète (Vieta) (フランス, 1540–1603) 代数学の父.

公式　$\frac{2}{\pi} = \prod_{n=2}^{\infty} \cos\frac{\pi}{2^n} = \sqrt{\frac{1}{2}}\sqrt{\frac{1}{2}+\frac{1}{2}\sqrt{\frac{1}{2}}}\sqrt{\frac{1}{2}+\frac{1}{2}\sqrt{\frac{1}{2}+\frac{1}{2}\sqrt{\frac{1}{2}}}}\cdots$

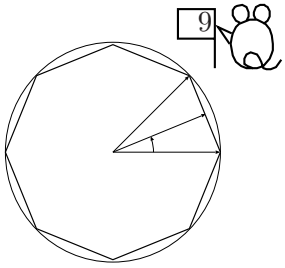
を用いて 1593 年に π を小数点以下 9 桁計算.

Ludolph van Ceulen (オランダ，1540–1610)

正 2^{62} 角形の周長を用いて小数点以下 35 桁計算.

関孝和 小数点以下 14 桁

正 2^{15}, 2^{16}, 2^{17} 角形の周長を 15 桁ほど計算し，
補外により，14 桁を正しく求めた．



単位円に内接する正 2^n 角形の半周長は

$$L_n = 2^n \sin\frac{\pi}{2^n}, \quad L_{2^2} = 2\sqrt{2}.$$

半角公式より $\sin\dfrac{\pi}{2^{n-1}} = 2\sin\dfrac{\pi}{2^n}\cos\dfrac{\pi}{2^n}$

$$\therefore\ L_n = \frac{L_{n-1}}{\cos\dfrac{\pi}{2^n}} = \cdots = \frac{L_{2^2}}{\displaystyle\prod_{k=3}^{n}\cos\frac{\pi}{2^k}} = \frac{2\sqrt{2}}{\displaystyle\prod_{k=2}^{n}\cos\frac{\pi}{2^k}}\frac{1}{\sqrt{2}}$$

$$= \frac{2}{\displaystyle\prod_{k=2}^{n}\cos\frac{\pi}{2^k}} \overset{n\to\infty}{\longrightarrow} \pi.$$

$\cos\dfrac{\pi}{2^n}$ は半角公式で漸化式が作れる：

$$\cos\frac{\pi}{2^n} = \sqrt{\frac{1}{2} + \frac{1}{2}\cos\frac{\pi}{2^{n-1}}}, \qquad \cos\frac{\pi}{2^2} = \frac{1}{\sqrt{2}}$$

正 2^{15} 角形の半辺長: $3.14159264877698566948\cdots$
正 2^{16} 角形の半辺長: $3.14159265238659134580\cdots$
正 2^{17} 角形の半辺長: $3.14159265328899276527\cdots$
今, 正 N 角形の半辺長が $c_0+\dfrac{c_1}{N}+\dfrac{c_2}{N^2}+\cdots$ という
漸近展開を 持つと 仮定し ,
$c_0=\pi$ が半円周の長さと すれば,

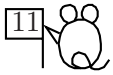
$$3.14159264877698566948\cdots=c_0+\frac{c_1}{2^{15}}+\frac{c_2}{2^{30}}+\cdots$$
$$3.14159265238659134580\cdots=c_0+\frac{c_1}{2^{16}}+\frac{c_2}{2^{32}}+\cdots$$
$$3.14159265328899276527\cdots=c_0+\frac{c_1}{2^{17}}+\frac{c_2}{2^{34}}+\cdots$$

この三式から c_1, c_2 を 消去すると ,

$$\begin{aligned}6c_0+\frac{c_1}{2^{15}}+\cdots&=8\times 3.14159265328899276527\cdots\\&\quad-2\times 3.14159265238659134580\cdots\ ,\\3c_0+\frac{c_1}{2^{15}}+\cdots&=4\times 3.14159265238659134580\cdots\\&\quad-3.14159264877698566948\cdots,\\\therefore\quad c_0+\cdots&=\frac{8}{3}\times 3.14159265328899276527\cdots\\&\quad-2\times 3.14159265238659134580\cdots\\&\quad+\tfrac{1}{3}\times 3.14159264877698566948\cdots\\&=3.14159265358979323894\cdots\end{aligned}$$

左辺の $\cdots$ は $\dfrac{1}{2^{45}}$ のオーダー.

cf: 正 2^{31} 角形の半辺長: $3.14159265358979323734\cdots$
正 2^{35} 角形の半辺長: $3.14159265358979323845\cdots$



A. Sharp (1653-1742) 1699 下記公式により小数点以下 71 桁:

$$\frac{\pi}{6} = \mathrm{Arctan}\,\frac{1}{\sqrt{3}} = \frac{1}{\sqrt{3}}\left(1 - \frac{1}{3\cdot 3} + \frac{1}{5\cdot 3^2} - + \cdots\right)$$

J. Machin (1680–1751) 1706 下記公式を導き 100 桁計算:

$$\begin{aligned}\frac{\pi}{4} &= 4\mathrm{Arctan}\,\frac{1}{5} - \mathrm{Arctan}\,\frac{1}{239}\\ &= 4\left(\frac{1}{5} - \frac{1}{3\cdot 5^3} + \frac{1}{5\cdot 5^5} - + \cdots\right)\\ &\quad - \left(\frac{1}{239} - \frac{1}{3\cdot 239^3} + \frac{1}{5\cdot 239^5} - + \cdots\right)\end{aligned}$$

※ $\mathrm{Arctan}\,x = x - \frac{x^3}{3} + \frac{x^5}{5} - + \cdots \quad (-1 < x \leq 1)$

松永良弼 1739 小数点以下 49 桁. 下記公式を導き利用:

$$\pi = 3\left(1 + \frac{1^2}{4\cdot 6} + \frac{1^2\cdot 3^2}{4\cdot 6\cdot 8\cdot 10} + \frac{1^2\cdot 3^2\cdot 5^2}{4\cdot 6\cdot 8\cdot 10\cdot 12\cdot 14} + \cdots\right)$$

W.Shanks 1873 Machin の公式により小数点以下 707 桁計算
この計算があまりに圧巻だったので, 以後電子計算機の発明まで誰も挑戦しようとする者は居なかった.

Ludolph の計算について, 正 2^{62} 角形の半辺長:
3.14159265358979323846264338327950288395418471219027
cf. π の真値:
3.14159265358979323846264338327950288419716939937510
ちょうど 35 桁まで真値なので, どのように答えたか興味深い!

松永の計算について, 松永の級数の値:
70 項までの和:
3.14159265358979323846264338327950288419716939801448866932 2
78 項までの和:
3.14159265358979323846264338327950288419716939937508814202 3
79 項までの和:
3.14159265358979323846264338327950288419716939937510148411 9
80 項までの和:
3.14159265358979323846264338327950288419716939937510475684 2
松永は“ $\cdots$3751 微強 ”と記しているので,
実際には 50 桁計算したとみてよいであろう.
本当に 80 項計算したのか, それとも
関孝和の補外法のような加速法を援用したのか?

コンピュータによる π の計算

1949 世界最初 (?) の電子計算機 ENIAC (1946) で 2037 桁 (70 時間).
Shanks の値の 528 桁以降が誤っているのを発見
1961 D.Shanks-J.W.wrench IBM 7090 により 10 万桁 (約9 時間)
1967 Guilloud-Dichampt CDC 6600 により 50 万桁 (約 28 時間)
1973 Guilloud-Bouyer CDC 7600 により 100 万桁 (約 23 時間)
1981 三好-金田 FACOM M-200 により 200 万桁 (約 137 時間)
1983 後-金田 HITAC S-810/20 で 1000 万桁(約 24 時間)
1985 Gosper Symbolics 3670 により 1752 万桁 (約 28 時間?)
1986 Bailey CRAY-2 により 2936 万桁 (約 28 時間)
1988 金田-田村 HITAC S-820/80 で2 億桁(約 35 時間)
1989 G.V.Chudnovsky IBM-3090 で 10 億桁(2 ヶ月?)
1991 G.V.Chudnovsky 自家製計算機で 22 億桁(?)
1995 高橋-金田 HITAC S-3800/480 で 64 億桁(約 117 時間)
1996 G.V.Chudnovsky 自家製計算機で 80 億桁(一週間?)
1997 高橋-金田 HITAC SR2201 で 515 億桁(約 29 時間)
1999 高橋-金田 HITAC SR8000 で 2000 億桁(約 37 時間)
2002.12.6 金田 & 日立チーム SR8000/MPP で 1 兆 2411 億桁 (約 600 時間)

ENIAC の計算は Machin の項式を用いていたが,
現在では, 算術幾何平均の改良型を用いる :
2) $a_0 = a = 1, b_0 = b = 1/\sqrt{2}$ を初項とする二つの数列を

$$a_n = \frac{a_{n-1} + b_{n-1}}{2}, \quad b_n = \sqrt{a_{n-1} b_{n-1}}$$

で定めるとき, $\pi = \lim_{n\to\infty} \dfrac{2a_n b_n}{1 - \sum_{j=0}^{n} 2^j (a_j^2 - b_j^2)}$

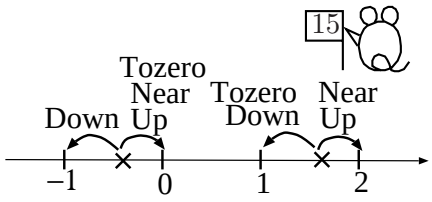
精度保証計算

精度保証計算とは, 実数 x を有限浮動小数で近似するとき,
$x_0 < x < x_1$ と上下から挟むことにより, 最終的な答の誤差の大きさを
保証付きで与えるもの.
昔は区間演算と呼んだ. 実現するには, 非常に計算量がかかり,
非実用的だった.
粗い計算だと, 演算の度に区間がどんどん大きくなり,
すぐに挟んでも意味の無いような大きさに区間が広がってしまう.

最近の CPU は切り捨て, 切り上げのモード変更ができるので,
それぞれのモードで1 回ずつ同じ計算をするだけで, i.e.
普通の計算の2 倍の計算量で精度保証が可能となる.
うまく使うと, 数学の定理の実用的証明の道具となる

CPU の丸めモード：
通常, CPU は正しい計算値を維持するため, 少し多めの桁で計算し,
結果を返すときに対応する変数のフォーマットに合わせて丸める.
デフォールトは四捨五入 Near, i.e.
指定のフォーマットで表現可能な最も近い値に丸める.
この他に, 切り上げ Up, 切り捨て Down, 0 に最も近い値 Tozero, の
計4 種が選択可能.
これらは実数の集合 $\boldsymbol{R}$ から表現可能な浮動小数の集合 $\boldsymbol{F}$ への写像として,
IEEE754 できちんと規定されている.

- `Near(-x)=-Near(x)`,
  `Up(-x)=-Down(x)`, `Down(-x)=-Up(x)`,
- `x` が表現可能浮動小数なら
  任意の丸め方 Marume について
  `Marume(x)=x`,
- 演算 `?` $\in$ `{+, -, *, /}` の全てについて
  `Marume(x?y)=Marume(x)?Marume(y)`



- `sqrt` までは IEEE754 に規定があるが，`exp`，`sin` など，
  他の組み込み函数については規定が無いので，
  CPU の丸めモードを UP や DOWN に変えただけでは，
  上下から挟めるかどうかは現状では期待できない．
  $\Longrightarrow$ 自分で書くか, Matlab などのソフトに組み込まれたものを
  利用するしかない．

例: 計算機では 0.1 を 10 回足しても 1 にならない (cf. 藤代先生の基礎講義),
というのを種々のモードで実験してみる．
最近接丸め (四捨五入) 方式 (標準) のとき
$s =$ `1.000000000000000e-01 = 9A 99 99 99 99 99 B9 3F`
$\sum_{i=1}^{10} s =$ `9.999999999999999e-01 = FF FF FF FF FF FF EF 3F`
上への丸め (切り上げ) 方式を指定したとき
$s =$ `1.000000000000000e-01 = 9A 99 99 99 99 99 B9 3F`
$\sum_{i=1}^{10} s =$ `1.000000000000001e+00 = 03 00 00 00 00 00 F0 3F`
下への丸め (切り捨て) 方式を指定したとき
$s =$ `9.999999999999999e-02 = 99 99 99 99 99 99 B9 3F`
$\sum_{i=1}^{10} s =$ `9.999999999999998e-01 = FE FF FF FF FF FF EF 3F`

球を空間にできるだけ密に詰める方法は,
果物屋さんがオレンジを積んでいるやりかただろう. (Kepler 1611)

不思議なことに, 最下層の一段の並べ方が図 a でも図 b でも,
その後に無駄無く積み上げてできた結果は3 次元的には同じになる!
図 c は William Barlow (1845–1934) によるその説明図.

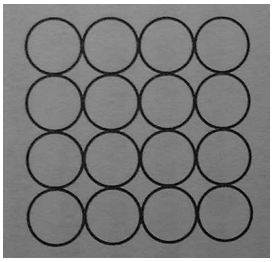


図 a

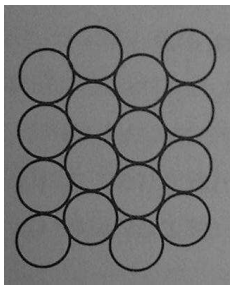


図 b

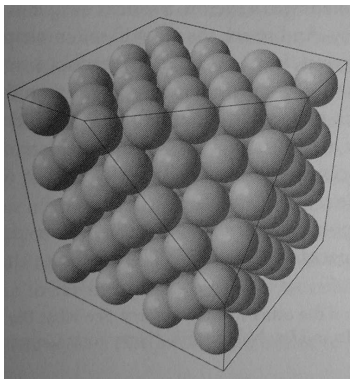


図 c

これより詰められないことを, Kepler が予想して以来約 400 年振りに
Thomas C. Hales (1998) が精度保証付き数値計算により証明した.
(F. Tóth が有限個のパラメータの最小問題に変換していた.
その最小解は, Kepler 問題の解の空間充填率を主要項とする漸近式を持つ.
Hales はその誤差項を改良し, それを厳密に評価することにより,
理論的最小解と Kepler 予想の解が一致することを示した.)

現実の CPU での精度保証計算

INTEL x86 の場合: (プログラム見本 `roundx86.c`)
CPU のモードを保持するレジスター (CW で表される) の第 10,11 ビットが丸めのモードを制御している. (第 8,9 ビットは精度を制御.)
下記の函数 (インラインアセンブラ) は `__fldcw`, `__fnstcw` の名前で `fenv.h` に定義されているので, これをインクルードするだけでもよい.

```
void setround(int mode){
    __asm __volatile ("fldcw %0" : : "m"(mode))
}
int getround(int *mode)
    __asm("fnstcw %0" : "=m" (*(mode)))
    return *mode;
}
#define FE_TONEAREST    0x0000
#define FE_DOWNWARD     0x0400
#define FE_UPWARD       0x0800
#define FE_TOWARDZERO   0x0c00
```

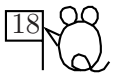
- AMD64 互換の 64 ビットマシンでは, 上のデータは共通だが, 10 バイト長の long double で実験しないと差が見えない.

PowerPC の場合: (プログラム見本 `roundppc.c`)
C 言語用のライブラリ函数が用意されている.

```
typedef unsigned long fenv_t;
void  fesetenv(const fenv_t *);  /* 丸めモード設定 */
void  fegetenv(fenv_t *);        /* 丸めモード問い合わせ */
#define  FE_TONEAREST     0x00000000
#define  FE_TOWARDZERO    0x00000001
#define  FE_UPWARD        0x00000002
#define  FE_DOWNWARD      0x00000003
```

- `roundppc.c` は現在のインテルマックでは, コンパイルしても正常に動かない. 昔の Power Mac でコンパイルされたバイナリ roundppc が, エミュレーションで正しく動くので, これで PowerPC を味わってください.

本日の講義内容の自習課題



1 Kerosoft 社の多倍長演算ライブラリを使ってみる.
`inflib.f`, `infdec.f` がライブラリで, `exp.f`, `log.f`, `tri.f` などがそれを利用したプログラムである. コンパイル例:

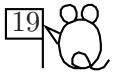
```
g77 -c infdec.f inflib.f
g77 log.f inflib.o infdec.o -o log
./log
```

2 GMP を使った C プログラム例を実行してみる.
コンパイルの仕方は

```
gcc gmptest.c -lgmp -o gmptest
./gmptest
```

3 CPU の丸めモードを変更する実験をしてみる.
`roundx64.c` は, gcc で普通にコンパイルできる.
`roundppc.c` の方はコンパイルできないので，roundppc を実行するだけにせよ.
PowerMac を持っている人は `roundppc.c` を持ち帰って
コンパイル・実行してみよ.
同様に，32 ビットの Intel CPU 用の roundx86.c は5 階ではコンパイルはできるが
実行しても何も変わらない．これは，CPU が丸めを 80 ビットでやっているため,
64 ビットに落としたときに，みな同じ値になってしまうためのようである.
32 ビット Windows 機を持っている人は `roundx86.c` を持ち帰って
コンパイル・実行してみよ.

本日の範囲のプログラム練習問題



問題 12.1 Kerosoft 社の多倍長演算ライブラリを使って
$x - \cos x = 0$ の解を 100 桁近似計算せよ.

問題 12.2 C 言語で次の Machin の級数を実装し, π を 1000 桁計算せよ:

$$\pi = 16\Big(\frac{1}{5} - \frac{1}{3\cdot 5^3} + \cdots + \frac{(-1)^n}{(2n+1)\cdot 5^{2n+1}} + \cdots\Big)$$
$$- 4\Big(\frac{1}{239} - \frac{1}{3}\frac{1}{239^3} + \cdots + \frac{(-1)^n}{2n+1}\frac{1}{239^{2n+1}} + \cdots\Big).$$

[これは次問の多倍長演算ライブラリを構築しなくても単独で解ける.]

問題 12.3 GMP を利用して何か興味を持った数を 100 桁計算してみよ.
(例えば, 第5 回の課題に有った $\sin x = x^2$ の正の解など.)

問題 12.4 多倍長数の加減乗算を実行できるライブラリを C 言語で作れ.
それを用いて $\sqrt{2}$ を 1000 桁計算せよ.
[第4 回に紹介した Newton 法と割り算を避ける工夫を組み合わせた
方法によれば, 多倍長数同士の除算を使わずに計算できる.]