多倍長計算ノート

桂田 祐史

2010 年 9 月 9 日, 2012 年 10 月 28 日, 2016 年 11 月 13 日, 2017 年 9 月 13 日

http://nalab.mind.meiji.ac.jp/~mk/labo/text/on-multiprecision/

1 はじめに

1.1 多倍長計算とは何か

1.2 多倍長計算を何のために使うか

円周率の計算をして、競争に参加したり神様のメッセージを読み解くんだというような人は別にして。

計算の規模が大きくなって来て、**悪条件**の問題を解く必要が生じた、というケースもあるが、それ以外に**逆問題**など、もともと**非適切** (**illposed**) な問題に挑戦する人が出て来たこと、また**精度保証付き数値計算**をする人達からの要請もある。

今井仁司の**無限精度数値計算** (スペクトル法のように望む次数の精度を実現出来る離散化手法と多倍長計算を併用した数値計算技法のこと。非適切問題を解くために用いる。今井 [1])

1.3 アルゴリズム

小学校で学ぶ筆算と同じアルゴリズムを用いると、桁数を n として、計算量は $O(n^2)$ になり、これで計算をするのは実際的ではない。FFT を用いると $O(n \log n)$ の計算量で済むので、n が大きい場合は FFT を用いるのが普通である。しかし n があまり大きくないときは、FFT の計算量が最小とはならない。

後述する GMP では、n によってアルゴリズムを切り替えている (付属するマニュアル中の “Algorithms” を一読することを強く推奨する)。

1.3.1 Karatsuba 法

Karatsuba (ソ連, 1962 年)

$$X = x_1 b + x_0, \quad Y = y_1 b + y_0$$

とするとき

$$\begin{aligned} XY &= (x_1 b + x_0)(y_1 b + y_0) \\ &= x_1 y_1 b^2 + (x_1 y_0 + x_0 y_1) b + x_0 y_0 \\ &= x_1 y_1 b^2 + [x_1 y_1 + x_0 y_0 - (x_1 - x_0)(y_1 - y_0)]\, b + x_0 y_0 \end{aligned}$$

であるから、

$$
\begin{aligned}
s_0 &:= x_0 y_0, \\
s_2 &:= x_1 y_1, \\
XY &:= s_2 b^2 + [s_2 + s_0 - (x_1 - x_0)(y_1 - y_0)]\, b + s_0
\end{aligned}
$$

と計算すると、乗算 3 回で XY が計算できる (b の冪をかけるのは桁ずらしで済むことに注意)。

4 回の積と 2 回の和が、3 回の積と 3 回の和と 3 回の差になった。再帰的に繰り返すと、極限では、乗算の計算量は $O(n^{\log_2 3})$ $(\log_2 3 = 1.58496\cdots)$ となる。

資料によっては

$$
XY = x_1 y_1 b^2 + [(x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0]\, b + x_0 y_0
$$

に基いた説明をしている。あれれ。

1.3.2 FFT による乗算

(準備中)

1.4 利用可能なパッケージ

現在利用可能な多倍長計算パッケージとして

GMP (the GNU Multiple Precision arithmetic library) GMP[1] というサイトがある。

exflib

FMLIB

Mathematica も整数演算は GMP を使っているそうである。

2 exflib を使ってみる

2.1 入手＆インストール

exflib[2] という WWW サイトから、必要なものを入手する。

2.1.1 Mac

`exflib-intelmac-20091230.pkg.tar.bz2` を使ってすいすい (`tar xjf exflib-intelmac-20091230.p` で `exflib-intelmac-20091230.pkg` が現れるので、それをダブルクリックする)。

試してみる

```
cp -pr /usr/local/share/exflib/ ~
cd ~/exflib/sample-cxx
make
```

[1] `http://gmplib.org/`
[2] `http://www-an.acs.i.kyoto-u.ac.jp/~fujiwara/exflib/`

napier.cpp

```
// -*- C++ -*-
// $Id: napier.cpp,v 1.1.1.1 2002/04/07 07:18:10 fujiwara Exp $

#define PRECISION 1000
#include <iostream>
#include "exfloat.h"
using namespace std;

int main()
{
  exfloat term, e, old_e;
  long i;

  old_e = 1;
  e = 2;
  term = 1;

  for(i = 2; e != old_e; i++){
    term /= i;
    old_e = e;
    e += term;
  }

  cout << setiosflags(ios::scientific) << e << '\n';
  cout << setiosflags(ios::fixed) << setprecision(100) << e << '\n';

  e = "#NAPIER";
  cout << e << endl;

  return 0;
}

// Local Variables:
// compile-command: "g++ -Wall -O2 -L.. -o napier napier.cpp -lexf"
// End:
```

2.1.2 Cygwin

exflib-win32-dll-20091230.zip を入手して、展開し、install.bat を実行すれば良い。

もっとも単に .dll ファイルを C:¥WINDOWS¥system32 にコピーしているだけなので、手動でやってもよいかも知れない。Cygwin のシェルでは、

例えば

```
unzip exflib-win32-dll-20091230.zip
cd exflib-win32-dll-20091230
cp exfloat.dll /cygdrive/c/WINDOWS/system32
```

という調子。

g++ で使うのならば、次のようにするのが良い？

例えば

```
cp exfloat.h /usr/local/include
cp exfloat.dll /usr/local/lib/libexfloat.dll
```

これで

```
g++ なんとか.cpp -lexfloat
```

という感じでコンパイル出来るようになる。

2.2 使い方

コンパイルは -lexfloat というオプションをつけるくらい (簡単)。
以下プログラムの書き方についてのメモ。

```
#define PRECISION 1000
#include <exfloat.h>
```

これで 10 進 1000 桁の精度の exfloat というクラスが使えるようになる。
x が exfloat 型の変数であるとき、整数式ならば `x=1;` のように代入が出来るが、double の式を代入したりは出来ない (`x=1.23;` や、y が double 型の変数の場合の `x=y;` は駄目)。

- 定数ならば `x = "1.23";` のように文字列で代入出来る。
- 一般には明示的にキャストするのが良い: `x = static_cast<exfloat>(y);`
- exfloat 型のデータを double に丸めるにもキャストが使える。

```
double x;
exfloat y;
....
x = static_cast<double>(y)
```

- 表示する桁数や、固定小数形式・指数形式のどちらで表示するかを指定するには、double 型のデータと同様のやり方が使える。

表示桁数

```
std::cout << std::setprecision(1000);
```

固定小数形式 (fixed-point format)

```
std::cout << std::setiosflags(std::ios::fixed);
```

指数形式 (scientific notation, exponential notation)

```
std::cout << std::setiosflags(std::ios::scientific);
```

test-exflib-1.cpp

```
// test-exflib-1.cpp

#define PRECISION 50
#include <iostream>
#include <exfloat.h>
using namespace std;

int main(void)
{
  exfloat x, y;

  x = "1/3";
  cout << setprecision(PRECISION) << x << endl;

  y = 1;
  y /= 3;
  cout << y << endl;

  x = "#PI";
  cout << x << endl;

  return 0;
}
```

実行

```
[katsurada:~] mk% g++ test-exflib-1.cpp -lexfloat
[katsurada:~] mk% ./a.out
0.33333333333333333333333333333333333333333333333333
0.33333333333333333333333333333333333333333333333333
3.14159265358979323846264338327950288419716939937510
[katsurada:~] mk%
```

2.3 複素数の場合

exflib のサイトでは、C++ のテンプレート・ライブラリィと exflib は協調して動くように書いてある。

test-cpp-complex-2.cpp

```
// test-cpp-complex-2.cpp

#include <iostream>
#include <iomanip> // setprecision()
#include <complex>
#include <math.h>

#define PRECISION 50
#include <exfloat.h>

using namespace std;

int main(void)
{
  complex<exfloat> I(0,1), z, one;
  one = 1;
  cout << "I=" << I << endl;
  cout << "I^2=" << I * I << endl;
  cout << setprecision(PRECISION+10) << "sqrt(I)=" << sqrt(I) << endl;
  cout << "sqrt(I) の自乗=" << sqrt(I)*sqrt(I) << endl;
  z=one+I;
  cout << "z=" << z << ", z^2=" << z*z << endl;
  return 0;
}
```

```
[katsurada:~/study-c-c++-2010] mk% g++ test-cpp-complex-2.cpp -lexfloat
[katsurada:~/study-c-c++-2010] mk% ./a.out
I=(0,1)
I^2=(-1,0)
sqrt(I)=(0.70710678118654752440084436210484903928483593768847,0.70710678118654752440084436210484903928
sqrt(I)=(0,1)
z=(1,1), z^2=(0,2)
[katsurada:~/study-c-c++-2010] mk%
```

sqrt() のような関数もちゃんと動くのは大したものだ (少し驚いた)。

3 GMP

GMP[3] (**GNU Multiple Precision arithmetic library**) は、名前が示しているように GNU Project でサポートされている多倍長計算パッケージで、C 言語とアセンブリ言語で記述されている。

付属するマニュアル (WWW サイトでも入手可能) が十分詳しい。アルゴリズムについても 1 章を使って説明されている。

普段使っている Mac には、気がついたときには既に GMP が入っていた (いつインストールしたのか覚えていない、何かの依存関係？)。何でも GCC のインストールには GMP が必要になっているそうで、UNIX 風の環境ではあるのが当たり前に近いライブラリィなのかもしれない。

mpz_t	整数型
mpq_t	有理数型
mpf_t	浮動小数点数型 (Version 2 から)
mpfr_t	浮動小数点数型 (Version 4 から, IEEE 754 互換)

[3] http://gmplib.org/

```
typedef  struct  {
   mpfr_prec_t    _mpfr_prec;
   mpfr_sign_t    _mpfr_sign;
   mp_exp_t          _mpfr_exp;
   mp_limb_t      *_mpfr_d;
}  __mpfr_struct;
```

Verion 4 から C++ インターフェイスが導入された。

- 初期化 mpz_init(変数名) または mpz_init2(変数名, ビット数)

```
mpz_t i,j;

mpz_init(i);
mpz_init2(j, 1024);
```

- 定数の代入 mpz_set_str(変数名, 文字列, 基数)

```
mpz_t i;

mpz_init(i);
mpz_set_str(i, "1234567890123456", 10);
```

- 演算 mpz_add(), mpz_sub(), mpz_mul()
- 出力 mpz_out_str(stdout, 基数, mpz 型の式)

pi.c — 100 万桁

```
/*
The Gauss AGM algorithm using Schonhage variation.
[1] Jorg Arndt, Christoph Haenel, "Pi Unleashed", pp. 93, Springer, 2001.
(C) 2003 Hironobu SUZUKI, licensed by GPL2
$Id: pi.c,v 1.1 2003/09/16 01:20:41 hironobu Exp hironobu $
*/
#include <stdio.h>
#include <gmp.h>
#include <sys/timeb.h>
#include <stdlib.h>

#define LOG_TEN_TWO  3.32192809488736234789
#define bprec(n) (int)(((n+10)*LOG_TEN_TWO)+2)

int
main (int ac, char *av[])
{
  long int k, loopmax;
  mpf_t A, B, a, b, s, s_1, t, t1, t2, t3, c2;
  long int prec, dprec;
  struct timeb start_t, end_t;

  dprec = 1000000L; /* decimal precision */
  prec = bprec(dprec); /* binary precision (plus alpha) */
  mpf_set_default_prec (prec);
  loopmax = 21;

  mpf_init (A); /* big A */
  mpf_init (B); /* big B */
  mpf_init (a); /* a */
  mpf_init (b); /* b */
  mpf_init (s); /* s(n) */
  mpf_init (s_1); /* s(n-1) */
  mpf_init (t); /* temporary */
  mpf_init (t1); /* temporary */
  mpf_init (t2); /* temporary */
  mpf_init (t3); /* temporary */
  mpf_init (c2); /* 2 constant */
  mpf_set_ui (A, 1);
  mpf_set_ui (a, 1);
  mpf_set_ui (t1, 1);
  mpf_div_ui (B, t1, 2);
  mpf_div_ui (s, t1, 2);
  mpf_set_ui (t1, 10);
  mpf_set_ui (c2, 2);
  for (k = 1; k <= loopmax; k++)
    {
      mpf_add (t1, A, B); /* (A+B) */
      mpf_div_ui (t, t1, 4); /*  t = (A+B)/4 */
      mpf_sqrt (b, B); /* b = sqrt(B) */
      mpf_add (t1, a, b); /* (a+b) */
      mpf_div_ui (a, t1, 2); /* a = (a+b)/2 */
      mpf_mul (A, a, a); /* A = a * a */
      mpf_sub (t1, A, t); /*  (A-t) */
      mpf_mul_ui (B, t1, 2); /* B = (A - t) * 2 */
      mpf_sub (t1, B, A); /* (B-A) */
      mpf_pow_ui (t2, c2, k); /* 2^k */
      mpf_mul (t3, t1, t2); /* (B-A) * 2^k  */
      mpf_set (s_1, s);
      mpf_add (s, s_1, t3); /* s = s + (B-A) * 2^k */
    }
  mpf_add (t1, a, b);
  mpf_pow_ui (t2, t1, 2);
  mpf_mul_ui (t1, s, 2);
  mpf_div (A, t2, t1);
  mpf_out_str (stdout, 10, dprec + 10, A);

  exit(0);
}
```

MacBook Air でのコンパイルと実行

```
[katsurada:~/study-c-c++-2010] mk% gcc pi-gmp.c -lgmp
[katsurada:~/study-c-c++-2010] mk% time ./a.out>pi.out
5.446u 0.093s 0:05.55 99.6%     0+0k 0+3io 0pf+0w
[katsurada:~/study-c-c++-2010] mk%
```

5 秒強で 100 万桁の数値を求めることが出来ている。

4 MPFR

MPFR については、MPFR[4] というサイトがある。

Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, “MPFR: A Multiple-Precision Binary Floating-Point Library With Correct-Rounding” という INRIA のレポートがある。その abstract によると、MPFR は、C 言語で記述され、GMP に基いた、多倍長浮動小数点演算ライブラリィである。正しい丸めと例外を用意することで、IEEE 754 標準の考え方を多倍長演算に拡張した点に特徴があるとしている。

また C99 に含まれるすべての数学関数を多倍長でサポートしている (底が e, 2, 10 の対数関数と指数関数と `log1p()` $(= \log(1+x))$ と `expm1()` $(= \exp(x) - 1)$, 三角関数、逆三角函数、双曲線函数、逆双曲線関数、ガンマ関数、ゼータ関数、誤差関数 (?`erfc()`)、算術幾何平均、冪乗 `pow()` $(= x^y)$。すべて「正しく丸められる」そうで、精度保証数値計算家にとってはありがたいだろう…と思ったら既に MPFR に基づく区間演算ライブラリィがいくつかあるそうだ。

使用しているアルゴリズムについては、THE MPFR TEAM によって “THE MPFR LIBRARY: ALGORITHMS AND PROOFS” という文書が用意されている。“proofs” が入っているところが流石。

普段使っている MacBook Air には、気がついたときには既に MPFR が入っていた。おいおい、どういうことだ (調査中)。GMP に MPFR が取り込まれたという説もある。どこかまっさらなマシンに自分でインストールしてみないと分からない。

[4] `http://www.mpfr.org/`

napier-MPFR.c

```
/*
 * napier-MPFR.c --- MPFR ライブラリィを使って自然対数の底を求める。
 *  http://www.mpfr.org/sample.html
 */

#include <stdio.h>

#include <gmp.h>
#include <mpfr.h>

int main (void)
{
  unsigned int i;
  mpfr_t s, t, u;

  mpfr_init2 (t, 200);
  mpfr_set_d (t, 1.0, GMP_RNDD);
  mpfr_init2 (s, 200);
  mpfr_set_d (s, 1.0, GMP_RNDD);
  mpfr_init2 (u, 200);
  for (i = 1; i <= 100; i++)
    {
      mpfr_mul_ui (t, t, i, GMP_RNDU);
      mpfr_set_d (u, 1.0, GMP_RNDD);
      mpfr_div (u, u, t, GMP_RNDD);
      mpfr_add (s, s, u, GMP_RNDD);
    }
  printf ("Sum is ");
  mpfr_out_str (stdout, 10, 0, s, GMP_RNDD);
  putchar ('\n');
  mpfr_clear (s);
  mpfr_clear (t);
  mpfr_clear (u);
  return 0;
}
```

MacBook Air でのコンパイルと実行

```
katsurada:~/study-c-c++-2010 mk$ gcc napier-MPFR.c -lmpfr -lgmp
katsurada:~/study-c-c++-2010 mk$ ./a.out
Sum is 2.71828182845904523536028747135266249775724709369995957496691 31
katsurada:~/study-c-c++-2010 mk$
```

- mpfr_t 変数名リスト; で変数を定義出来る。
- mpfr_init2(変数名, ビット数) で変数の精度を指定できる。
- mpfr_set_d(変数名, double 型の数値, GMP_RNDD) で、double 型の数値を変数に代入する。_d は double を意味する。
- GMP_RNDN, GMP_RNDZ, GMP_RNDU, GMP_RNDD, GMP_RND_MAX という丸めモードがある。
- mpfr_mul(z,x,y,GMP_RNDD), mpfr_mul_d(z,x,y,GMP_RNDD), mpfr_mul_ui(z,x,y,GMP_RNDD) はいずれも、z=x*y という意味だが y の型が違う。
- mpfr_out_str(stdout, 基数, 0, 数値, GMP_RNDN)

(2016/5/10 記) kv[5] の MPFR ラッパーというのを試してみた (http://nalab.mind.meiji.ac.jp/~mk/labo/text/studying-kv/node15.html)。

5 CLN

(準備中)

[5] http://verifiedby.me/kv

6 FMLIB

6.1 FMLIB とは

David M. Smith が作成した Fortran 90 用の多倍長計算パッケージである。採用しているアルゴリズムは一連の論文に示されている ([4], [5], [6], [7], [8], [9])。

Fortran 90 では演算子も多重定義されている。

IM	整数型
FM	浮動小数点型
ZM	複素数型

```
use FMZM
```

FMLIB の多倍長型変数の宣言

```
type(IM) :: i,j
type(IM) :: x,y
type(ZM) :: z
```

```
call FMSET(100)
```

表示

```
call IMPRNT(i)
call FMPRNT(x)
call ZMPRNT(z)
```

書式付きの表示

```
character(LEN=32) :: ST
type(FM) :: x

ST=FM_FORMAT('F32.24',x)
write(*,*) ST
```

型変換

IM_IM	多倍長整数型
IM_FM	多倍長浮動小数点型
IM_ZM	多倍長複素数型
IM_INT	多倍長整数型
IM_SP	単精度浮動小数点型
IM_DP	倍精度複素数型
IM_SPZ	単精度複素数型
IM_DPZ	倍精度複素数型

高次代数方程式の数値解法ライブラリの研究開発[6] では FMLIB を採用している。

一度自分でプログラムを書いてみるのだな。

[6] http://www.ss.isc.tohoku.ac.jp/refer/pdf_data/v39-3p51-67.pdf

6.2 FMLIB のインストール

Multiple Precision Computation (by Dr. David M. Smith)[7] が FMLIB の WWW サイトである。

FM.f90, FMZM.f90, FMSAVE.f90, TestFM.f90, SampleFM.f90, SAMPLE.CHK, ReadMe をダウンロードする。いずれも .TXT という拡張子がついているが、それは手動で削除する (ファイル名を変更する) ことが求められている。

次は MacBook Air で作成したときの自作 Makefile である (見れば分かるが、FM.f90, FMZM.f90, FMSAVE.f90 という 3 つのソースプログラムがライブラリィの実体である)。

Makefile

```
OBJS1   =       FMSAVE.o FMZM90.o FM.o
OBJS2   =       TestFM.o SampleFM.o
OBJS    =       $(OBJS1) $(OBJS2)
SRCS    =       FMSAVE.f90 FMZM90.f90 FM.f90 TestFM.f90 SampleFM.f90
FMLIB   =       libfmlib.a
PROGS   =       TestFM SampleFM
FC      =       gfortran
FFLAGS  =       -O

.SUFFIXES: .f90 .o

.f90.o:
        $(FC) $(FFLAGS) -c $<

all: $(PROGS)

fmlib.a: $(OBJS1)
        ar cr $@ $(OBJS1)
        ranlib $@

TestFM: TestFM.o fmlib.a
        $(FC) $(FFLAGS) -o $@ TestFM.o fmlib.a

SampleFM: SampleFM.o fmlib.a
        $(FC) $(FFLAGS) -o $@ SampleFM.o fmlib.a

test: TestFM
        ./TestFM
```

7 Multiple Precision Toolbox for MATLAB

どの程度使えるのか未知数であるが、まあ試してみよう。

(数日経過)

確かに mp という多倍長実数クラスを定義出来たが、それを用いた行列演算が激遅 (何か間違えているだろうか？…)。現時点では、連立 1 次方程式を多倍長で解こう、という目的には全然向かないようである。自分で必要なものを書き足せばいいのか？

7.1 インストール

2012 年 10 月、Mac Book Air にインストールしたときの記録。

[7] http://myweb.lmu.edu/dmsmith/fmlib.html

1. 前準備として GMP, MPFR をインストールする。MacPorts に任せれば良いので簡単である（はず）。

```
sudo port install gmp
sudo port install mpfr
```

最初、付属文書の指示に従い、ヘッダーファイルやライブラリィファイルを /usr/local/include に入れようと思ったが、古い gmp.h や mpfr.h があった。言うことに従わないことにする。

2. Multiple Precision Toolbox for MATLAB[8] から mptoolbox_1.1.zip を入手し、中身をコピーする.

ファイルを解いて、コピー

```
mkdir mptoolbox_1.1
cd mptoolbox_1.1
unzip ../mptoolbox_1.1.zip
sudo tar cf - . | (cd /Applications/MATLAB_R2010b.app/; tar xpf -)
```

(丸ごと /Applications/MATLAB_R2010b.app/ にコピーするというのは、付属文書の指示なのだけど、かなり乱暴である。既存のファイルを上書きしないことを確認してからコピーしたが、後から考えてみると、動作に必要なのは mp@ 以下のファイルだけみたいなので、mp@ だけをコピーするものかもしれない。)

3. Mex のセットアップをする。まずは

[8] http://www.mathworks.com/matlabcentral/fileexchange/6446

mex のセットアップを始める

```
>> mex -setup

    Options files control which compiler to use, the compiler and link command
    options, and the runtime libraries to link against.

    Using the 'mex -setup' command selects an options file that is
    placed in ~/.matlab/R2010b and used by default for 'mex'. An options
    file in the current working directory or specified on the command line
    overrides the default options file in ~/.matlab/R2010b.

    To override the default options file, use the 'mex -f' command
    (see 'mex -help' for more information).

The options files available for mex are:

  1: /Applications/MATLAB_R2010b.app/bin/gccopts.sh :
      Template Options file for building gcc MEX-files

  2: /Applications/MATLAB_R2010b.app/bin/mexopts.sh :
      Template Options file for building MEX-files via the system ANSI
compiler

  0: Exit with no changes

Enter the number of the compiler (0-2):
1

/Applications/MATLAB_R2010b.app/bin/gccopts.sh is being copied to
/Users/mk/.matlab/R2010b/mexopts.sh

********************************************************************************
  Warning: The MATLAB C and Fortran API has changed to support MATLAB
           variables with more than 2^32-1 elements.  In the near future
           you will be required to update your code to utilize the new
           API. You can find more information about this at:
           http://www.mathworks.com/support/solutions/en/data/1-5C27B9/?solution=1-5C27B9
           Building with the -largeArrayDims option enables the new API.
********************************************************************************
>>
```

メッセージに書いてあるように、~/.matlab/R2010b/mexopts.sh というファイルが用意される。これを修正する。Arch が maci64 のところをチェックする。以下、C の場合のみ考える (C++, Fortran については後日)。

C コンパイラーを指定している部分

```
CC='gcc-4.0'
SDKROOT='/Developer/SDKs/MacOSX10.5.sdk'
MACOSX_DEPLOYMENT_TARGET='10.5'
```

ところで使っているシステムにはどんなのが入っているのか？

```
gcc --version
which gcc
ls -l /usr/bin/gcc*
```

これで 4.2.1 というバージョンで、コマンド名が gcc-4.2 であることを知る。

```
ls /Developer/SDKs/
```

これで /Developer/SDKs/MacOSX10.6.sdk というのがあることを知る.

こんなふうに修正

```
            CC='gcc-4.2'
            SDKROOT='/Developer/SDKs/MacOSX10.6.sdk'
            MACOSX_DEPLOYMENT_TARGET='10.6'
```

GMP, MPFR を見つけ出せるように、CFLAGS に -I/opt/local/include を加えておく。具体的には次のような行を追加する。

```
            CFLAGS="$CFLAGS  -I/opt/local/include"
```

ライブラリィもやるべきなのかな？

```
CLIBS="-L/opt/local/lib $MLIBS"
```

とか。

こんなふうにしたはず。

```
[chronos:~/.matlab/R2010b] mk% diff mexopts.sh.org mexopts.sh
239,241c239,241
<             CC='gcc-4.0'
<             SDKROOT='/Developer/SDKs/MacOSX10.5.sdk'
<             MACOSX_DEPLOYMENT_TARGET='10.5'
---
>             CC='gcc-4.2'
>             SDKROOT='/Developer/SDKs/MacOSX10.6.sdk'
>             MACOSX_DEPLOYMENT_TARGET='10.6'
244a245,246
> # The next line was added by mk.
>             CFLAGS="$CFLAGS  -I/opt/local/include"
[chronos:~/.matlab/R2010b] mk%
```

4. コンパイルする。

```
>> cd /Applications/MATLAB_R2010b.app/@mp/private/
>> mp_compile_all
```

コンパイルすると山のように警告が出るが、全部同じ内容である。int であるべきとこ

ろに mp_exp_t がある、後者は実は前者を typedef したものなので何の問題も生じない。ポリシーからすると黙らせたいのだけど (警告を無視していると、大事なことを見落とす可能性があるので、警告には出来る限りていねいに対応することにしている)、どうやれば良いか分からない。

5. 後はチェック。

```
>> cd ../..
>> mp_TESTING
>> mp_TESTING2
```

Hilbert 行列のところの精度が precision をあげても良くならない。なぜだろう。

6. 日常的に利用するには、MATLAB に/Applications/MATLAB_R2010b.app/@mp というディレクトリィを登録しておくのだろう。File → Set Path かな？

8 gcc の __float128

(2016/2/25)
「__float128 の利用法」[9] がわかりやすい。

gcc の version 4.6 以降では__float128, __complex128 というのが使えるとか。IEEE 754 2008 で導入された binary128 だとか。

[9] http://maikaze.cafe.coocan.jp/float128.html

```c
/*
 * testfloat128.c -- __float128 のテスト
 *   gcc testfloat128.c -lquadmath
 */

#include <stdio.h>
#include <quadmath.h>
#define MAXN (100)

int main(void)
{
  int n;
  char message[128];
  __float128 e, newe, x;
  x = 1.0Q; e = x;
  for (n = 1; n <= MAXN; n++) {
    x /= n;
    newe = e + x;
    quadmath_snprintf(message, sizeof(message), "%.40Qf", e);
    puts(message);
    if (e == newe)
      break;
    else
      e = newe;
  }
  printf("2.7182818284590452353602874713526624977572470937000 --- N[E,50]\n");
  printf("10 進法で 34 桁程度の精度があるというけれど、本当？\n");
  return 0;
}
```

```
[chronos:~/work] mk% gcc testfloat128.c -lquadmath
[chronos:~/work] mk% ./a.out
1.0000000000000000000000000000000000000000
2.0000000000000000000000000000000000000000
2.5000000000000000000000000000000000000000
2.6666666666666666666666666666666665382713
2.7083333333333333333333333333333330765427
2.7166666666666666666666666666666663841969
2.7180555555555555555555555555555554613990
(途中略)
2.7182818284590452353602874713492665146805
2.7182818284590452353602874713525463733758
2.7182818284590452353602874713526596180565
2.7182818284590452353602874713526634699164
2.7182818284590452353602874713526624977572470937000 --- N[E,50]
10 進法で 34 桁程度の精度があるというけれど、本当？
[chronos:~/work] mk%
```

なるほど、なるほど。

(2016/12/1)

__float128 と double で、1000 次正方行列同士の乗算にどれくらい時間がかかるか。

```
/*
 * gcc -O3 -funroll-loops -o testfloat128 testfloat128.c -I/usr/local/include -L/usr/l
 */

#include <stdio.h>
#include <math.h>
#include <matrix.h>
#include <quadmath.h>

int main(void)
{
  int i, j, k, n;
  char message[128];
  __float128 norm;
  matrix128 a,b,c;
  n = 1000;
  a = new_matrix128(n,n);
  b = new_matrix128(n,n);
  c = new_matrix128(n,n);
  if (a == NULL || b == NULL || c == NULL) {
    fprintf(stderr, "cannot allocate\n");
    exit(1);
  }
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
      a[i][j] = i + j;
      b[i][j] = i - j;
    }
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) {
      c[i][j] = 0;
      for (k = 0; k < n; k++)
c[i][j] += a[i][k] * b[k][j];
    }
  norm = 0;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      norm += fabsq(c[i][j]);
  quadmath_snprintf(message, sizeof(message), "%.7Qf", norm);
  printf("%s\n", message);
  return 0;
}
```

```
[chronos:~/work] mk% ccmg testfloat128.c -lquadmath ; time ./testfloat128
257388424419000.0000000
122.528u 0.179s 2:02.73 99.9% 0+0k 0+0io 2pf+0w
```

double バージョンを作ってみたが、あまりスピード出ない (5〜10 秒)。MATLAB だと、掛け算 1 秒もかからない。

9 その他

そのうちに…

- **mpmath**[10]
  Python 用の多倍長ライブラリィ

- MPACK[11] by 中田真秀
  「MPACK (MBLAS/MLAPACK) 高精度 BLAS/LAPACK ライブラリの作成」(2010)[12] — 要精読
  「高精度線形代数演算ライブラリ MPACK 0.8.0 の紹介」(2013)[13] — スライド 43 ページ目の「多倍長精度演算ライブラリのまとめ」は、2010 のと少し違う？いずれにせよ、色々な多倍長演算ライブラリィの速さ比較をしているのは興味深い。

- QD[14]
  解説のありがたい日本語訳「Double-Double(DD) および Quad-Double(QD) 演算ライブラリ」 by 幸谷智紀[15] がある。

10 2016 年秋

最近、久しぶりに色々試した。

- 精度保証付数値計算ライブラリ kv[16] の MPFR ラッパーを試してみた (2016 年春)。MPFR ラッパー[17]

- MPFR への C++ インターフェイスである MPFR C++[18] を試してみた。

- MPFR への C++ インターフェイスである Programming mpfr::real[19] を試してみた。

exflib の MATLAB 拡張も “Comming soon!” だと言うし、色々楽しみだ。

(2017/9/12 記: exflib のサイトに “2017/05/07 : MATLAB version released” と書いてあった。今度やってみよう。)

[10] http://mpmath.org/
[11] http://mplapack.sourceforge.net/
[12] http://nakatamaho.riken.jp/slides/mpack-0.6.4.pdf
[13] http://www.slideshare.net/NakataMaho/mpack-080/
[14] http://crd-legacy.lbl.gov/~dhbailey/mpdist/
[15] http://na-inet.jp/na/qd_ja.pdf
[16] http://verifiedby.me/kv/
[17] http://nalab.mind.meiji.ac.jp/~mk/labo/text/studying-kv/node15.html
[18] http://www.holoborodko.com/pavel/mpfr/
[19] http://chschneider.eu/programming/mpfr_real/

10.1 MPFR C++ by Pavel Holoborodko

GNU MPFR を使ってみよう、と考えた。C++ のインターフェイスはどうなっているんだっけ？(本当は、MPFR 自体が C++ インターフェイス持っていれば良いんだよな…) The GNU MPFR Library[20] を見たら、百花繚乱になっているらしいことが分かった。うーん。まずは “MPFR C++” の検索でトップに出て来た MPFR C++ - Pavel Holoborodko[21] をやってみる。

ライセンスは GPL である。と書いてあるけれど、

> MPFR C++ is free for usage in free projects. If you intend to use it in commercial application please contact author for permission.

とも書いてある。はてな。

boost[22] で使われていると書いてある (本当？)。

Eigen[23] で使うには、

```
#include <unsupported/Eigen/MPRealSupport>
```

だそうな。

とにかくやってみよう。`mpfrc++-3.6.2.zip` (2015 年 7 月に出来た？) を入手して

```
mkdir mpfrc++-3.6.2
cd mpfrc++-3.6.2
unzip ../mpfrc++-3.6.2.zip
```

実にあっさりしている。実体は、`mpreal.h` というインクルード・ファイル一つ (3100 行くらい)。

MPFR 自体は、MacPorts でインストールされているので、`/opt/local/include`, `/opt/local/lib` に置かれている。

`mpreal.h` はどこに置くのが良いか。自分で導入したものなので `/usr/local/include` が自然か。ここに置いておくと、LLVM の g++ は無指定で見てくれる。一方で、MacPorts で入れた GCC の g++ は、`/opt/local/include` を無指定で見てくれる。そこで

```
sudo cp -p mpreal.h /usr/local/include
sudo ln -s /usr/local/include/mpreal.h /opt/local/include
```

とした。これで `mpreal.h` を見るために `-I` は必要ない。

サンプル・プログラムは唯一 `example.cpp` がついている (107 行)。それはあっさり動いた。

$a_1 = 1$, $a_{n+1} = 3a_n + 2$ ($n \in \mathbf{N}$) で定義される数列の第 100 項を計算してみる (すごい勢いで大きくなるので、double では正確に計算できない)。

[20] `http://www.mpfr.org/`
[21] `http://www.holoborodko.com/pavel/mpfr/`
[22] `http://www.boost.org/`
[23] `http://eigen.tuxfamily.org/`

```
/*
 * test-mpfrcc++.cpp
 * 準備
 *   (1) http://www.holoborodko.com/pavel/mpfr/ から mpfrc++-3.6.2.zip 入手
 *       unzip mpfrc++-3.6.2.zip として出て来る mpreal.h を適当な場所におく。
 *   (2) MPFR を用意する。MacPorts を使っていると、/opt/local にあるだろう。
 * コンパイル
 *   g++ -I/opt/local/include test-mpfrc++.cpp -L/opt/local/lib -lmpfr
 */

#include <iostream>
#include "mpreal.h"

using namespace std;

int main(void)
{
  using mpfr::mpreal;
  const int digits = 50;
  mpreal::set_default_prec(mpfr::digits2bits(digits));
  mpreal a;
  int i;
  cout.precision(digits);
  a = 1;
  for (i = 2; i <= 100; i++)
    a = 3 * a + 2;
  cout << a << endl;

  return 0;
}
```

```
g++-mp-6 test-mpfrc++.cpp -L/opt/local/lib -lmpfr

あるいは

g++-mp-5 -std=c+11 test-mpfrc++.cpp -L/opt/local/lib -lmpfr

あるいは

g++ -I/opt/local/include test-mpfrc++.cpp -L/opt/local/lib -lmpfr
```

でコンパイルする。これであっさり動いた (343585013821340887357640753177080848468071681333 という結果が表示される)。

次は、自然対数の底 e を 1000 桁計算してみよう。画面に表示する前に文字列にして、50 桁ずつわけて表示する。

e1000.cpp

```
/*
 *   g++ -I/opt/local/include e1000.cpp -L/opt/local/lib -lmpfr
 */

#include <iostream>
#include <string>
#include "mpreal.h"

using namespace std;

int main(void)
{
  int n, maxn = 20;
  using mpfr::mpreal;
  const int digits = 1000 + 10; // 10 は余裕
  mpreal::set_default_prec(mpfr::digits2bits(digits));
  mpreal t, s, news;
  string str;
  t = 1; s = t;
  for (n = 1; n < 1000; n++) {
    t /= n;
    news = t + s;
    if (news == s) {
       break;
    }
    s = news;
  }
  cout.precision(digits);
  // cout << "e=" << s << endl;
  str = s.toString();
  cout << "e=" << str.substr(0, 2) << endl;
  for (n = 0; n < digits; n += 50) {
    cout << "    " << str.substr(n + 2, 50) << endl;
  }
  return 0;
}
```

e1000 の実行結果

```
[chronos:math/new-sotsuken-text/on-multiprecision] mk% ./a.out
e=2.
    71828182845904523536028747135266249775724709369995
    95749669676277240766303535475945713821785251664274
    27466391932003059921817413596629043572900334295260
    59563073813232862794349076323382988075319525101901
    15738341879307021540891499348841675092447614606680
    82264800168477411853742345442437107539077744992069
    55170276183860626133138458300075204493382656029760
    67371132007093287091274437470472306969772093101416
    92836819025515108657463772111252389784425056953696
    77078544996996794686445490598793163688923009879312
    77361782154249992295763514822082698951936680331825
    28869398496465105820939239829488793320362509443117
    30123819706841614039701983767932068328237646480429
    53118023287825098194558153017567173613320698112509
    96181881593041690351598888519345807273866738589422
    87922849989208680582574927961048419844436346324496
    84875602336248270419786232090021609902353043699418
    49146314093431738143640546253152096183690888707016
    76839642437814059271456354906130310720851038375051
    01157477041718986106873969655212671546889570350354
    0212340769
[chronos:math/new-sotsuken-text/on-multiprecision] mk%
```

Mathematica で `N[E,1001]` として表示したものと比較して、一致することを確認出来た。

10.2 mpfr::real by Christian Schneider

The GNU MPFR Library[24] のリストにあった Programming mpfr::real[25] を試す。そのサイトから `mpfr_real_v0.0.9-alpha.tar.gz` を持って来た。2012 年版だ。

```
tar xzf mpfr_real_v0.0.9-alpha.tar.gz
cd mpfr_real_v0.0.9-alpha
```

`real.hpp` というのが、唯一コンパイルに必要なファイルであるとか (4700 行くらい)。

```
sudo cp -p readl.hpp mpfr_header_wrapper.hpp /usr/local/include
sudo ln -s /usr/local/include/real.hpp /opt/local/include
sudo ln -s /usr/local/include/mpfr_header_wapper.hpp /opt/local/include
```

“Comparison with Other MPFR C++ Wrappers” という項があり、MPFR C++ - Pavel Holoborodko[26] との比較が書いてある。こちらは cmath にある数学関数は完備している、とのこと (MPFR C++ の方はどうなんだろう？)。

テストしてみる。比較用の出力結果を添えたテスト・プログラムが用意されている (こういうのは他の人も見習ってほしいものだ)。

`real_operator_test.cpp` から。

[24] http://www.mpfr.org/
[25] http://chschneider.eu/programming/mpfr_real/
[26] www.holoborodko.com/pavel/mpfr/

```
g++ -I/opt/local/include real_operator_test.cpp -L/opt/local/lib -lmpfr -lgmp

g++-mp-6 real_operator_test.cpp -L/opt/local/lib -lmpfr -lgmp

g++-mp-5 real_operator_test.cpp -L/opt/local/lib -lmpfr -lgmp
```

でコンパイルする。いずれの場合も

```
./a.out > real_operator_test.txt
diff real_operator_test.txt test_output/real_operator_test.txt
```

で実行して、動作確認した。問題なし。

次は real_function_test.cpp を試す。

```
g++ -I/opt/local/include real_function_test.cpp -L/opt/local/lib -lmpfr -lgmp

g++-mp-6 real_function_test.cpp -L/opt/local/lib -lmpfr -lgmp

g++-mp-5 real_function_test.cpp -L/opt/local/lib -lmpfr -lgmp
```

でコンパイルする。

実はオリジナルの real_function_test.cpp のままでは、LLVM の g++ ではコンパイル出来たが、g++-mp-5 や g++-mp-6 ではコンパイル出来なかった。次のように real_function_test.cpp を書き換えた。

```
$ diff  real_function_test.cpp.org real_function_test.cpp
457c457
<               << std::setw(17) << isinf(a1);
---
>               << std::setw(17) << std::isinf(a1);
459c459
<     std::cout << std::setw(20) << type2char(isinf(a1));
---
>     std::cout << std::setw(20) << type2char(std::isinf(a1));
495c495
<               << std::setw(17) << isnan(a1);
---
>               << std::setw(17) << std::isnan(a1);
497c497
<     std::cout << std::setw(20) << type2char(isnan(a1));
---
>     std::cout << std::setw(20) << type2char(std::isnan(a1));
```

これで g++-mp-5, g++-mp-6 でもコンパイル出来るようになった。

比較結果。

1 g++-mp-5 の場合、

```
diff real_function_test.txt-5 test_output/real_function_test.txt
32c32
< acosh(a1)           double     0.70711              nan               double
---
> acosh(a1)           double     0.70711             -nan               double
```

-nan て何でしょうね。

2 g++-mp-6 の場合、たくさん相違点がある。int が bool になっている。

3 LLVM g++ の場合、たくさん相違点がある。g++-mp-6 と同じように int が bool になっている他に、frexp() 絡み。

何か良く分からない。

11 2017 年晩夏

11.1 exflib for MATLAB

(準備中)

11.2 Arb

(準備中)

参考文献

[1] 今井 仁司, 無限精度計算が切り開く応用数値解析の未来, 数理解析研究所講究録, 1566 巻, 2007 年, pp. 96–118.

[2] 今井 仁司, 応用解析における多倍長計算, 数学, Vol. 55, No. 3, pp. 316–325 (2003).

[3] 幸谷 智紀, PC Cluster 上における多倍長数値計算ライブラリ BNCpack の並列分散化— MPIBNCpack について —, lc.linux.or.jp/paper/lc2003/CP-14.pdf

[4] David M. Smith, Efficient Multiple-Precision Evaluation of Elementary Functions, Mathematics of Computation, 52 (1989), pp. 131–134.

[5] David M. Smith, A Fortran Package For Floating-Point Multiple-Precision Arithmetic Transactions on Mathematical Software, 17 (1991), pp. 273–283.

[6] David M. Smith, A Multiple-Precision Division Algorithm Mathematics of Computation, 66 (1996), pp. 157–163.

[7] David M. Smith, Multiple Precision Complex Arithmetic and Functions Transactions on Mathematical Software, 24 (1998), pp. 359–367.

[8] David M. Smith, Multiple-Precision Gamma Function and Related Functions Transactions on Mathematical Software, 27 (2001), pp. 377–387.

[9] David M. Smith, Using Multiple-Precision Arithmetic Computing in Science and Engineering, 5 (July, 2003), pp. 88–93.

[10] the GMP developers, GNU MP (the manual of GMP), http://gmplib.org/ (1991–2010).