数値解析 2
第1回

静岡理工科大学
情報学部　コンピュータシステム学科
幸谷智紀
https://na-inet.jp/mpna/

概要

- 目的：「多倍長精度数値計算」をネタに「高性能計算(HPC)手法」を学ぶ。
- 教科書：なし。PowerPoint資料に基づいて講義＋演習
- 参考書：幸谷「多倍長精度数値計算」（森北出版）
- 内容：C/C++プログラミングを行いながら，HPCテクニックを学ぶ。
- 出欠：カードリーダー＋毎回の小テスト（本日の課題）
- 中間レポートあり（１～2回）
- 最終試験：15回目の講義日に実施。内容は事前に周知。

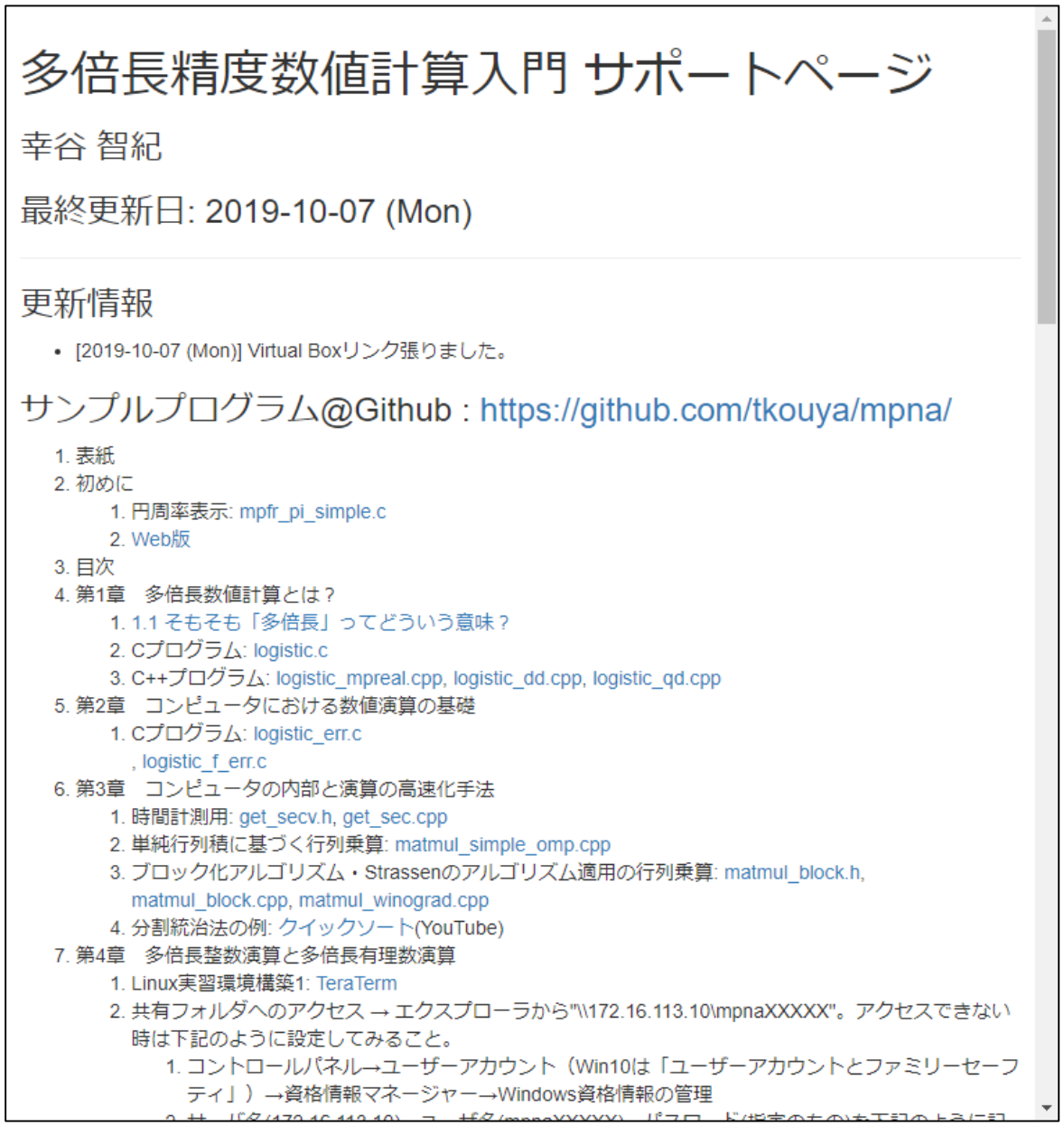
シラバス・・・あくまで予定

1. 多倍長数値計算とは？
2. コンピュータの内部と演算の高速化手法
3. 多倍長自然数演算と多倍長整数演算
4. 多倍長整数演算とその応用
5. 多倍長有理数演算とその応用
6. GMPを用いたRSA暗号の実装

→中間レポート

7. GMPのMPF型とMPFR
8. マルチコンポーネント型多倍長演算の実装
9. QDライブラリの利用法
10. 基本線型計算
11. 連立一次方程式の求解(1/2)
12. 連立一次方程式の求解(2/2)
13. べき乗法と逆べき乗法の実装(1/2)
14. べき乗法と逆べき乗法の実装(2/2)
15. 講義のまとめと最終試験

サポートページ https://na-inet.jp/mpna/



多倍長精度数値計算入門 サポートページ

幸谷 智紀

最終更新日: 2019-10-07 (Mon)

更新情報

- [2019-10-07 (Mon)] Virtual Boxリンク張りました。

サンプルプログラム@Github : https://github.com/tkouya/mpna/

1. 表紙
2. 初めに
    1. 円周率表示: mpfr_pi_simple.c
    2. Web版
3. 目次
4. 第1章　多倍長数値計算とは？
    1. 1.1 そもそも「多倍長」ってどういう意味？
    2. Cプログラム: logistic.c
    3. C++プログラム: logistic_mpreal.cpp, logistic_dd.cpp, logistic_qd.cpp
5. 第2章　コンピュータにおける数値演算の基礎
    1. Cプログラム: logistic_err.c
       , logistic_f_err.c
6. 第3章　コンピュータの内部と演算の高速化手法
    1. 時間計測用: get_secv.h, get_sec.cpp
    2. 単純行列積に基づく行列乗算: matmul_simple_omp.cpp
    3. ブロック化アルゴリズム・Strassenのアルゴリズム適用の行列乗算: matmul_block.h, matmul_block.cpp, matmul_winograd.cpp
    4. 分割統治法の例: クイックソート(YouTube)
7. 第4章　多倍長整数演算と多倍長有理数演算
    1. Linux実習環境構築1: TeraTerm
    2. 共有フォルダへのアクセス → エクスプローラから"\\172.16.113.10\mpnaXXXXX"。アクセスできない時は下記のように設定してみること。
        1. コントロールパネル→ユーザーアカウント（Win10は「ユーザーアカウントとファミリーセーフティ」）→資格情報マネージャー→Windows資格情報の管理

- 講義資料
- サンプルプログラム
- 参考サイト

等々・・・

基本，「多倍長精度数値計算」の内容に沿って進める。

実習環境

- 基本，Linux上でコマンドラインベースのC, C++プログラミングを行う。
- 実習環境としては次の2方式を取る。
  - 学内のLinuxマシンを提供・・・アカウントも発行する
  - 自分のLinux環境を用意する。Windows10上にWSLでUbuntuをインストールする等。
- 自分が作成したプログラムやデータのバックアップは適宜行うこと。

本日のお題: 多倍長数値計算とは？

- 標準的なデータ型・・・一つの機械語命令で演算処理が可能
  - 整数型 (integer)・・・8〜128 ビット長の 2 進整数
    - 符号付き整数型 (signed integer)
      - *int* 型
      - *long int* 型 (*long* 型)
    - 符号なし整数型 (unsigned integer)
      - *unsigned int* 型
      - *unsigned long int* 型 (*unsigned long* 型)
  - 実数型（浮動小数点型, floating-point number)・・・32 ビット
    - *float* 型 ・・・IEEE754 単精度 (binary32, 仮数部長 2 進 24 ビット，10 進 7 桁の有限桁小数)
    - *double* 型・・・IEEE754 倍精度 (binary64, 仮数部長 2 進 53 ビット，10 進 15 桁の有限桁小数)

→これを繋ぎ合わせて標準より長い桁数をサポートするのが「多倍長」整数・浮動小数点数演算＝（広義の）多倍長計算

多倍長数値計算の目的

- 多倍長（精度）計算は計算処理に時間がかかる
- 多倍長整数演算
  - 整数論における理論の検証・・・ディオファントス問題
  - 暗号処理の実装・・・RSA暗号→「計算処理に時間がかかる」＝セキュリティの確保＝解読に要する時間を延ばす→デメリットを生かす

- 多倍長（精度）浮動小数点演算
  - アルゴリズムの正しさの検証・・・精度保証など
  - 悪条件問題の計算精度向上→ロジスティック写像に基づく数列計算

多倍長精度浮動小数点演算の活用例

以下，桁落ちする計算の例として，ロジスティック写像

$$f(x) = 4x(1-x) \tag{1.1}$$

を使った，漸化式

$$x_{i+1} := f(x_i) = 4x_i(1-x_i) \tag{1.2}$$

によって導出される実数列 $\{x_i\}_{i=0}^{100}$ を，IEEE754 単精度 (以下，単精度。2 進 24 ビット, 10 進 7 桁) と倍精度 (2 進 53 ビット，10 進 15 桁) 精度の浮動小数点演算を使用して求めた結果を示す。プログラムは例えば C 言語で記述すると図 1.1 のようになる。初期値は $x_0 = 0.7501$ と指定し，$x_0, x_{10}, x_{20}, ..., x_{100}$ まで出力するようにしている。

Cプログラム例 logistic.c

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int i;
6
7    double x[102]; // 倍精度
8    x[0] = 0.7501;
9
10   for(i = 0; i <= 100; i++)
11   {
12     x[i+1] = 4 * x[i] * (1 - x[i]);
13     if(i%10 == 0)
14       printf("%5d␣%25.17e\n", i, x[i]);
15   }
16
17   return 0;
18 }
```

質問　単精度計算の例はどこまで正しい？

0	7.50100016593933105e-01	7.50099999999999989e-01
10	8.44516813755035400e-01	8.44495953602201199e-01
20	1.22903920710086823e-01	1.42939724528399537e-01
30	4.97778177261352539e-01	8.54296020314155413e-01
40	5.81643998622894287e-01	7.74995884542777125e-01
50	5.58012068271636963e-01	7.95132827423636751e-02
60	2.28748228400945663e-02	2.73872762849587226e-01
70	9.98548150062561035e-01	9.97021556611396687e-01
80	9.42180097103118896e-01	3.52785425069070790e-01
90	2.12938979268074036e-01	6.35558111134618908e-01
100	7.56864011287689209e-01	5.15390006286616020e-01

図 1.1　ロジスティック写像を計算する C プログラム: logistic.c

解答例）断続的に桁落ちが発生する

単精度計算と倍精度計算の結果を見比べた時の違いを観察してみると

- 初期値 x_0 の有効桁数が 7 桁から 15 桁に増えている。
- x_{20} ではもう最初の 1 桁程度しか精度がない
- x_{30} 以降の両者の値は全く異なっている

ということが分かる。つまり，単精度計算した場合の x_{30} 以降の計算はどうも怪しい，と勘ぐらねばならない。

この数列の計算は断続的な桁落ちが無限に発生する悪条件な計算の一例である。

理由は，$x_i \in (0,1)$ であれば，$1-x_i$ のところで，少しずつではあるが有効桁数が減り，しかも i が大きくなっても常に $x_i \in (0,1)$ が保たれているため，この有効桁数の減少が発生し続けて正しい値とのズレが大きくなっていくからである。

多倍長精度浮動小数点演算で解決

- C++のプログラムとして実装したものを使用
  - 演算子(＋，－，＊，／）が使用可能（オーバーロード機能）
  - 変数の初期化，消去，入出力が楽

①MPFR++ & MPFR/GMPを使用した場合・・・細かく計算精度桁数を設定できる

②QDを使用した場合・・・DD(double-double)とQD(quad-double)の2種類で比較

①MPFR++ & MPFR/GMP 128 bits vs. 192 bits

```
$ ./logistic_mpreal 128
num_bits = 128, num_decimal = 39
    0, 7.5009999999999999999999999999999999996e-01
   10, 8.4449595360221744753714870256154137269870e-01
   20, 1.4293972451230765528428175723130626119290e-01
   30, 8.5429600370442189166613118425886484760370e-01
   40, 7.7497575311820124128022346123682417226480e-01
   50, 9.3375332197703029055189687555128488838990e-02
   60, 4.0822016829087813187106146216161441532350e-01
   70, 7.1511999705058574878953745604654169664480e-02
   80, 4.6325330290077567460258670989545944685970e-01
   90, 1.3344050120841884955920667169676700883140e-03
  100, 7.8817989391884034027630801238868292349520e-02
```

```
$ ./logistic_mpreal 192
num_bits = 192, num_decimal = 58
    0, 7.500999999999999999999999999999999999999999999999999999995e-01
   10, 8.444959536022174475371487025615413726740195229122912799424 1e-01
   20, 1.429397245123076552842817572313062830737696920021411860913 8e-01
   30, 8.542960037044218916661311842588874437479758900823439332189 9e-01
   40, 7.749757531182012412802234612642116848151131368064124449676 5e-01
   50, 9.337533219770302905518966801516823476282297739532594560921 2e-02
   60, 4.082201682908781318710276618195268673035299106652459595690 1e-01
   70, 7.151199970505857489709934641759321838380427001443199468594 1e-02
   80, 4.632533029007757105599346745832075369053407137864153672343 8e-01
   90, 1.334405012086884046469201215007086192015724905345023559454 8e-03
  100, 7.881798937150990680670477040248033397691309758757782392702 4e-02
```

128 ビットの結果と，192 ビットの結果を比較し，一致する桁は有効桁と考えてよい。これによって，$x_{100} = 0.0788179893\cdots$ であろうという見当がついてくる。

logistic_mpreal.cpp

```cpp
#include <iostream>
#include <iomanip>

// MPFR/GMP + mpreal.h
#include "mpreal.h"

using namespace std;
using namespace mpfr;

int main(int argc, char *argv[])
{
  int i;
  int num_bits, num_decimal;

  // 引数チェック
  if(argc <= 1)
  {
    cerr << "Usage:␣" << argv[0] << "␣[num_bits]" << endl;
    return 0;
  }

  // 計算桁数設定
  num_bits = atoi(argv[1]);
  if(num_bits <= 24)
    num_bits = 24;

  num_decimal = (int)ceil(log10(2.0) * (double)num_bits);
  mpreal::set_default_prec(num_bits);

  cout << "num_bits␣=␣" << num_bits << ",␣num_decimal␣=␣" << num_decimal << endl;

  mpreal x[102];

  // 初期値
  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) << scientific << i << setprecision(num_decimal) << ",␣" << x[i] << endl
          ;
  }

  return 0;
}
```

②QD

```
1  #include <iostream>
2  #include <iomanip>
3  #include "qd_real.h" // dd_realクラスとqd_realクラス
4
5  using namespace std;
6
7  int main()
8  {
9    int i;
10   qd_real x[102]; // DD精度
11
12   // 初期値
13   x[0] = "0.7501";
14
15   fpu_fix_start(NULL);
16
17   for(i = 0; i <= 100; i++)
18   {
19     x[i + 1] = 4 * x[i] * (1 - x[i]);
20     if((i % 10) == 0)
21       cout << setw(5) << i << setprecision(64) << ",␣"
22   }
23
24   return 0;
25 }
```

```
$ ./logistic_dd
    0, 7.5009999999999999999999999999990e-01
   10, 8.4449595360221744753714870256178 2e-01
   20, 1.4293972451230765528428175700522 4e-01
   30, 8.5429600370442189166613095091084 7e-01
   40, 7.7497575311820124127994063273762 7e-01
   50, 9.3375332197703029256977185520600 4e-02
   60, 4.0822016829087848092430269383726 2e-01
   70, 7.1511999704871186811654212952273 7e-02
   80, 4.6325330252944721821252995636892 7e-01
   90, 1.3343771755058556524889959962941 2e-03
  100, 7.9028519640321121569181021148982 7e-02
```

```
$ ./logistic_qd
    0, 7.5010000000000000000000000000000000000000000000000000000000000000e-01
   10, 8.4449595360221744753714870256154137267401952291229128000784836889e-01
   20, 1.4293972451230765528428175723130628307376969200214112138765683432e-01
   30, 8.5429600370442189166613118425888744374797589008227714686828830936e-01
   40, 7.7497575311820124128022346126421168481511313672546437683759904859e-01
   50, 9.3375332197703029055189668015168234762823035148728377335712513474e-02
   60, 4.0822016829087813187102766181952686730362981285202267088068037662e-01
   70, 7.1511999705058574897099346417593218330172114550014128305492366822e-02
   80, 4.6325330290077571055993467458320743062778639025185340006762860292e-01
   90, 1.3344050120868840464692012149991190706624483976174704690161639369e-03
  100, 7.8817989371509906806704770462699247076819063446303925907861408889e-02
```

図 1.3　QD を用いたロジスティック写像計算プログラム: logistic_dd.cpp

GNU MP(GMP), MPFR, QDの歴史

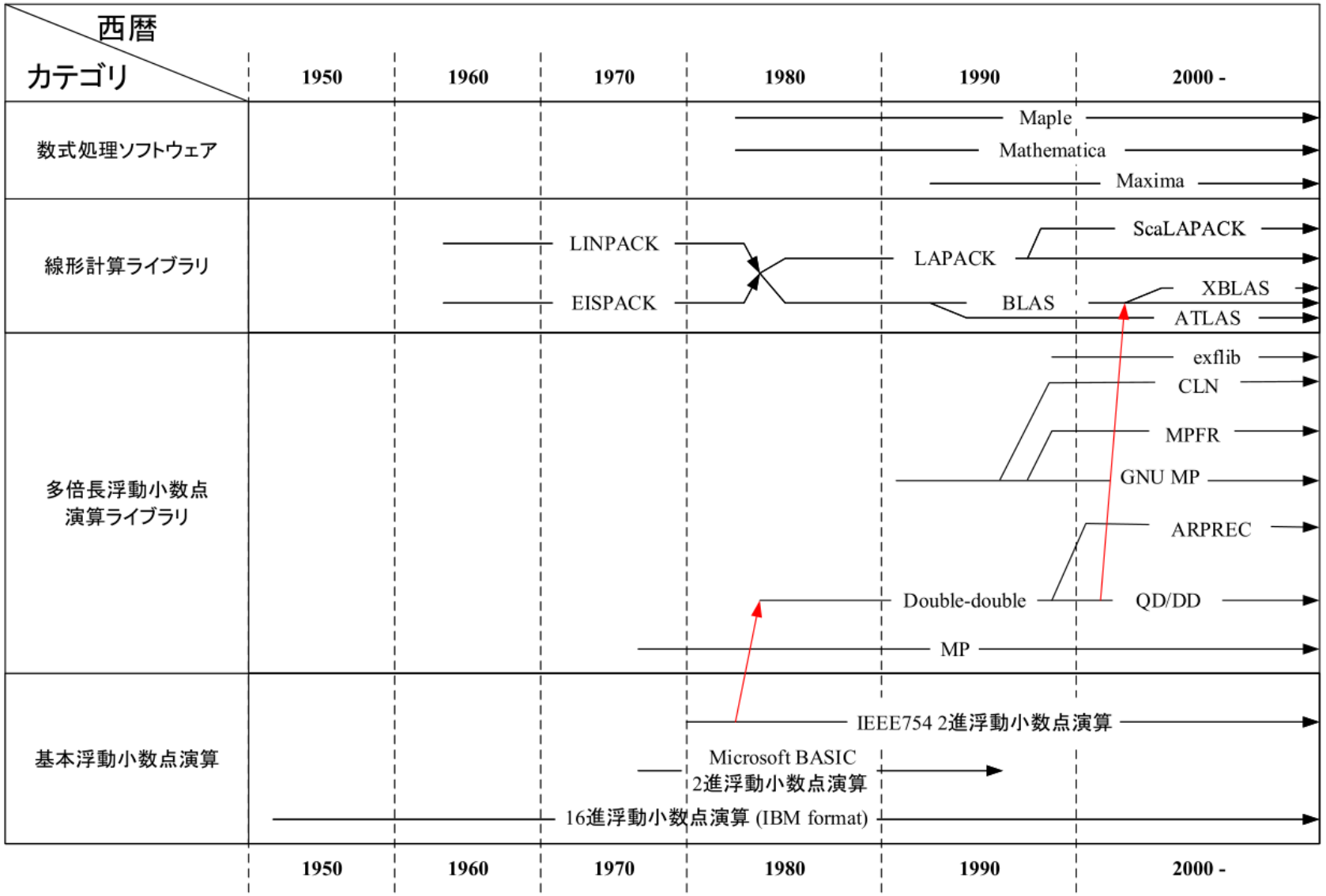


図 1.9　多倍長浮動小数点演算ライブラリの歴史

- それぞれ20年以上の歴史を持つソフトウェア
- 全てオープンソース
- GMPは多倍長整数，有理数，浮動小数点数演算を高速な自然数演算で支える
- MPFRはGMPの自然数演算ベースの多倍長浮動小数点演算ライブラリ
- QDはマルチコンポーネントタイプの固定精度浮動小数点演算ライブラリ

本日の課題

- どんなソフトウェアを用いてもいいので，100! と 1000! の値を求めよ。

- 解答は，大きい方から10桁以上を記述し，位取りも正確に行うこと。
- 必ずしもプログラムを組まなくてもよく，Webサイトを使用しても良い。
- 使用したソフトウェア名，Webサイト名は明記すること。