なんとか C++ を使う

桂田 祐史

2018 年 7 月 1 日, 2022 年 10 月 17 日

http://nalab.mind.meiji.ac.jp/~mk/labo/text/nantoka-c++/

1 はじめに

C++は便利というか、C では済まなくて、C++を使わざるを得ないようなことがしばしばある。適当に相手をして来たのだけれど、学生も使うようになって来たので、昔書き散らした資料をまとめて見た。

整理不十分なことはなはだしいけれど…

C++ は演算子オーバーローディングが可能なので、それを用いたクラス・ライブラリィが使えそうな場合は、C よりも便利である。

新しい情報を入手しないと

- cpprefjp - C++日本語リファレンス[1]
- 「江添亮の詳説 C++17」[2]

2 基本的なプログラム

CUI ですむ簡単なプログラムを示す。

2.1 何もしないプログラム

donothing.cpp

```
/*
 * donothing.cpp
 */

int main()
{
  return 0;
}
```

- ファイル名末尾が .c でなく .C や .cpp である以外は C のプログラムと同じである。—

[1] https://cpprefjp.github.io/
[2] https://ezoeryou.github.io/cpp17book/

– C++ は基本的にはスーパー C なので、ほとんどのまともな C プログラムはそのまま C++ プログラムになる。

– C++ のソース・プログラムのファイル名末尾は .C (大文字！), .cpp, .cxx などが採用されている。GCC (g++) では .C である。

コンパイルと実行の結果

```
oyabun% g++ -o donothing donothing.cpp
oyabun% ./donothing
oyabun%
```

2.2 Hello world

C++ はスーパー C であるから、printf() を使うことも出来なくはないが、そうしないで、次のようにストリームを使うべきだという意見が強い。

hello.C

```
// hello.c

#include <iostream>
using namespace std;

int main(void)
{
  cout << "Hello, world." << endl;
  return 0;
}
```

- C++ で // を書くと、そこから行末まで注釈になる。
- 標準出力に文字列を書くには、cout << 文字列 とすればよい。
- 改行を表わすのに endl が使える。細かい話になるが、これを出力すると、出力バッファーをフラッシュすることが保証されている。
- cout の宣言をするには、#include <iostream> とすれば良い。(単にインクルードしただけでは、std::cout としないと使えない。using namespace std; とすることで、cout という名前が使えるようになる。)

細かい話 (ある文句)

ところで、C++ では、言語の規格がかなり流動してきた[3]。

```
#include <iostream>
using namespace std;
```

は以前は

[3]プログラミング言語は誕生してから、色々変化するのは普通だけれど、以前書いていたプログラムの書き換えを強制されるのは、C++ のやり方のまずいところだと個人的に考えている。ちょっと古い本に書いてあるプログラムは、今の規格では間違っているなんてことはザラである。C 言語ではそういうことは滅多になかった。

```
#include <iostream.h>
```

としていた。

例えば

hello2.cpp

```
// hello2.cpp

#include <iostream.h>

int main()
{
  cout << "Hello, world." << endl;
  return 0;
}
```

もっと古い本を見ると #include <stream.h> なんて書いているのもある。

2.3 整数の簡単な入出力と計算 — 五則演算

sisoku.cpp

```
//
// sisoku.cpp
//

#include <iostream>
using namespace std;

int main(void)
{
  int a, b, wa, sa, seki, syou, amari;
  cout << "二つの整数を入力してください: ";
  cin >> a >> b;
  wa = a + b;
  sa = a - b;
  seki = a * b;
  syou = a / b;
  amari = a % b;
  cout << "和=" << wa << ", 差=" << sa << ", 積=" << seki << ", 商=" << syou
       << ", 余り=" << amari << endl;
  return 0;
}
```

コンパイルと実行結果

```
oyabun% g++ -o sisoku sisoku.cpp
oyabun% ./sisoku
二つの整数を入力してください: 123 456
和=579, 差=-333, 積=56088, 商=0, 余り=123
oyabun%
```

- (もちろん) 整数型の変数の宣言、演算などは C と同じである。
- 入力は cin >> 変数の並び ; とする。

2.4 実数の簡単な入出力と計算 — 2 次方程式を解く

今回も (C 流の scanf() & printf() ではなくて) ストリームを使って入出力をする。ここで一つの困難が出現する。数値計算プログラミングでは、結果を数値で出力する際に、書式の指定をすることが必須であるが、C++ で書式の指定をするのは思いの外面倒である。C では %20.15f だけで済むことを一体どうやるのか、以下のサンプル・プログラムを見てもらいたい。世の中には、この理由から、C++ でプログラムを書く場合にも、ストリームではなくて printf() を使って結果を出力する、という人が結構いたようである。

(なお、「§2.10 入出力の書式 (とりあえずの例)」 も見よ。そちらの方が書いたのが新しい。)

(C の場合については、「C 言語これくらい覚えよう　§3.4 浮動小数点数の入出力と四則演算」[4] を見よ。)

quadratic_eq.cpp

```
/*
 * quadratic_eq.cpp
 */

#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main(void)
{
  double a,b,c,D;
  cout << setprecision(15);
  cout.setf(ios::fixed, ios::floatfield);
  cout << "a,b,c: ";
  cin >> a >> b >> c;
  D = b * b - 4 * a * c;
  if (D >= 0)
    cout << setw(20) << (-b+sqrt(D))/(2*a) << ", "
         << setw(20) << (-b-sqrt(D))/(2*a) << endl;
  else
    cout << setw(20) << -b/(2*a) << "±"
         << setw(20) << sqrt(-D)/(2*a) << " i" << endl;
  return 0;
}
```

コンパイルと実行結果

```
oyabun% g++ -o quadratic_eq quadratic_eq.cpp
oyabun% ./quadratic_eq
a,b,c: 1 2 1
  -1.000000000000000,   -1.000000000000000
oyabun% ./quadratic_eq
a,b,c: 1 1 1
  -0.500000000000000 ±    0.866025403784439 i
oyabun%
```

- setw(整数) で表示桁数の指定。
- setprecision(整数) で小数点以下の桁数の指定。
- 小数形式、指数形式の選択は

[4] http://nalab.mind.meiji.ac.jp/~mk/labo/text/cminimum/node10.html

```
cout.setf(ios::fixed, ios::floatfield);        小数形式
cout.setf(ios::scientific, ios::floatfield);   指数形式
cout << resetiosflags(ios::floatfield);        元に戻す
```

率直に言って、C++ のこのあたりの機能を設計した人間は馬鹿だったのだと思う。

(…乱暴なことを言うようだけれど、このせいで多くの人達に無用の混乱を与えて、長い時間を浪費させていると思っている。C 言語のやり方は、Fortran のやり方を踏襲したもので、うまいやり方ではないけれど、ほどほどの使い勝手だと思われる。)

参考: https://www.horstmann.com/ にある “The March of Progress”

1980: C

```
printf("%10.2f", x);
```

1988: C++

```
cout << setw(10) << setprecision(2) << fixed << x;
```

1996: Java

```
java.text.NumberFormat formatter = java.text.NumberFormat.getNumberInstance();
formatter.setMinimumFractionDigits(2);
formatter.setMaximumFractionDigits(2);
String s = formatter.format(x);
for (int i = s.length(); i < 10; i++) System.out.print(' ');
System.out.print(s);
```

2004: Java

```
System.out.printf("%10.2f", x);
```

(皮肉の説明をするのは無粋かもしれないけれど) Horstmann 先生の言いたいことは、古い C で簡潔に出来ていたことが、それから後に出た C++ や Java でどんどん複雑になって行った。大した進歩だこと。結局 2004 年の Java では、C 風のやり方が使えるようになった。それ見たことか (最初からそうすれば良いのに)、ということだろう。

2.5 制御 (1) for 文 — 数列, 級数

ここは C とあまり変わりないかも。

2.6 制御 (2) while 文, if 文 — 2 分法による方程式の解法

bisection.cpp

```
/*
 * bisection.cpp -- 二分法 (bisection method) で方程式 f(x)=0 を解く
 *   コンパイル: g++ -o bisection bisection.cpp -lm
 *   実行: ./bisection
 */

#include <iostream>
#include <cstdlib> // exit()
#include <iomanip>
#include <cmath>

using namespace std;

int main(void)
{
    int i, maxitr = 100;
    double alpha, beta, a, b, c, eps;
    double fa, fb, fc;
    double f(double);

    cout.setf(ios::scientific, ios::floatfield);

    cout << " 探す区間の左端α, 右端β, 許容精度ε=";
    cin >> alpha >> beta >> eps;

    a = alpha; b = beta;
    fa = f(a); fb = f(b);

    if (fa * fb > 0.0) {
        cout << " f(α) f(β) > 0 なのであきらめます。" << endl;
        exit(0);
    }
    else {
        for (i = 0; i < maxitr; i++) {
            c = (a + b) / 2; fc = f(c);
            if (fc == 0.0)
                break;
            else if (fa * fc <= 0.0) {
                /* 左側 [a,c] に根がある */
                b = c; fb = fc;
            } else {
                /* 左側 [a,c] には根がないかもしれない。[c,b] にあるはず */
                a = c; fa = fc;
            }
            cout << "f(" << setw(24) << setprecision(15) << a << ")="
                        << setw(9) << setprecision(2) << fa
                 << ", f(" << setw(24) << setprecision(15) << b << ")="
                           << setw(9) << setprecision(2) << fb << endl;
            if ((b - a) <= eps)
                break;
        }
        cout << "f(" << setw(24) << setprecision(15) << c << ")="
             << setw(9) << setprecision(2) << fc << endl;
    }
    return 0;
}

double f(double x)
{
    return cos(x) - x;
}
```

このプログラムの実行結果は以下のようになる。

コンパイルと実行結果

```
oyabun% g++ -i bisection bisection.cpp
oyabun% ./bisection
 探す区間の左端α, 右端β, 許容精度ε=0 1 1e-14
f(   5.000000000000000e-01)= 3.78e-01, f(   1.000000000000000e+00)=-4.60e-01
f(   5.000000000000000e-01)= 3.78e-01, f(   7.500000000000000e-01)=-1.83e-02
f(   6.250000000000000e-01)= 1.86e-01, f(   7.500000000000000e-01)=-1.83e-02
f(   6.875000000000000e-01)= 8.53e-02, f(   7.500000000000000e-01)=-1.83e-02
f(   7.187500000000000e-01)= 3.39e-02, f(   7.500000000000000e-01)=-1.83e-02
f(   7.343750000000000e-01)= 7.87e-03, f(   7.500000000000000e-01)=-1.83e-02
f(   7.343750000000000e-01)= 7.87e-03, f(   7.421875000000000e-01)=-5.20e-03
f(   7.382812500000000e-01)= 1.35e-03, f(   7.421875000000000e-01)=-5.20e-03
f(   7.382812500000000e-01)= 1.35e-03, f(   7.402343750000000e-01)=-1.92e-03
f(   7.382812500000000e-01)= 1.35e-03, f(   7.392578125000000e-01)=-2.89e-04
f(   7.387695312500000e-01)= 5.28e-04, f(   7.392578125000000e-01)=-2.89e-04
f(   7.390136718750000e-01)= 1.20e-04, f(   7.392578125000000e-01)=-2.89e-04
f(   7.390136718750000e-01)= 1.20e-04, f(   7.391357421875000e-01)=-8.47e-05
f(   7.390747070312500e-01)= 1.74e-05, f(   7.391357421875000e-01)=-8.47e-05
f(   7.390747070312500e-01)= 1.74e-05, f(   7.391052246093750e-01)=-3.36e-05
f(   7.390747070312500e-01)= 1.74e-05, f(   7.390899658203125e-01)=-8.09e-06
f(   7.390823364257812e-01)= 4.68e-06, f(   7.390899658203125e-01)=-8.09e-06
f(   7.390823364257812e-01)= 4.68e-06, f(   7.390861511230469e-01)=-1.70e-06
f(   7.390842437744141e-01)= 1.49e-06, f(   7.390861511230469e-01)=-1.70e-06
f(   7.390842437744141e-01)= 1.49e-06, f(   7.390851974487305e-01)=-1.08e-07
f(   7.390847206115723e-01)= 6.91e-07, f(   7.390851974487305e-01)=-1.08e-07
f(   7.390849590301514e-01)= 2.92e-07, f(   7.390851974487305e-01)=-1.08e-07
f(   7.390850782394409e-01)= 9.20e-08, f(   7.390851974487305e-01)=-1.08e-07
f(   7.390850782394409e-01)= 9.20e-08, f(   7.390851378440857e-01)=-7.75e-09
f(   7.390851080417633e-01)= 4.21e-08, f(   7.390851378440857e-01)=-7.75e-09
f(   7.390851229429245e-01)= 1.72e-08, f(   7.390851378440857e-01)=-7.75e-09
f(   7.390851303935051e-01)= 4.72e-09, f(   7.390851378440857e-01)=-7.75e-09
f(   7.390851303935051e-01)= 4.72e-09, f(   7.390851341187954e-01)=-1.51e-09
f(   7.390851322561502e-01)= 1.61e-09, f(   7.390851341187954e-01)=-1.51e-09
f(   7.390851331874728e-01)= 4.63e-11, f(   7.390851341187954e-01)=-1.51e-09
f(   7.390851331874728e-01)= 4.63e-11, f(   7.390851336531341e-01)=-7.33e-10
f(   7.390851331874728e-01)= 4.63e-11, f(   7.390851334203035e-01)=-3.43e-10
f(   7.390851331874728e-01)= 4.63e-11, f(   7.390851333038881e-01)=-1.48e-10
f(   7.390851331874728e-01)= 4.63e-11, f(   7.390851332456805e-01)=-5.11e-11
f(   7.390851331874728e-01)= 4.63e-11, f(   7.390851332165767e-01)=-2.37e-12
f(   7.390851332020247e-01)= 2.20e-11, f(   7.390851332165767e-01)=-2.37e-12
f(   7.390851332093007e-01)= 9.81e-12, f(   7.390851332165767e-01)=-2.37e-12
f(   7.390851332129387e-01)= 3.72e-12, f(   7.390851332165767e-01)=-2.37e-12
f(   7.390851332147577e-01)= 6.74e-13, f(   7.390851332165767e-01)=-2.37e-12
f(   7.390851332147577e-01)= 6.74e-13, f(   7.390851332156672e-01)=-8.48e-13
f(   7.390851332147577e-01)= 6.74e-13, f(   7.390851332152124e-01)=-8.66e-14
f(   7.390851332149850e-01)= 2.94e-13, f(   7.390851332152124e-01)=-8.66e-14
f(   7.390851332150987e-01)= 1.04e-13, f(   7.390851332152124e-01)=-8.66e-14
f(   7.390851332151556e-01)= 8.55e-15, f(   7.390851332152124e-01)=-8.66e-14
f(   7.390851332151556e-01)= 8.55e-15, f(   7.390851332151840e-01)=-3.91e-14
f(   7.390851332151556e-01)= 8.55e-15, f(   7.390851332151698e-01)=-1.53e-14
f(   7.390851332151556e-01)= 8.55e-15, f(   7.390851332151627e-01)=-3.44e-15
f(   7.390851332151627e-01)=-3.44e-15
oyabun%
```

2.7 簡単なファイル入出力

実はあまり自信がないが、C 言語の「簡単なファイル入出力」[5] の C++ バージョンを作ってみた。

fileio.cpp

```
/*
 * fileio.cpp --- prog13check.c の C++ バージョン
 *   http://nalab.mind.meiji.ac.jp/~mk/labo/text/cminimum/node18.html
 *   http://nalab.mind.meiji.ac.jp/~mk/labo/text/cminimum/node19.html
 *   input.data が "2 3" という内容だとして
 *    c++ fileio.cpp
 *    ./a.out
 *   とすると output.data
 *
 */

#include <iostream>
#include <fstream>
using namespace std;

int main(void)
{
  int a, b, sum;
  ifstream ifs("input.data");
  if (!ifs) {
    cerr <<  "input.data を読むために開こうとして失敗しました。" << endl;
    exit(1);
  }
  ifs >> a >> b;
  ifs.close();

  sum = a + b;
  printf("%d と %d の和は %d\n", a, b, sum);

  ofstream ofs("output.data");
  if (!ofs) {
    cerr << "output.data を書くために開こうとして失敗しました。" << endl;
    exit(1);
  }
  ofs << sum << endl;
  ofs.close();

  return 0;
}
```

2.8 配列 — Gauss の消去法による連立 1 次方程式の解法

(準備中…いつになるやら)

2.9 行列・ベクトル演算

(準備中)
Eigen あるいは Boost を使うという手がある。個人的には Eigen[6] がおすすめ。

[5] http://nalab.mind.meiji.ac.jp/~mk/labo/text/cminimum/node18.html
[6] https://eigen.tuxfamily.org/

2.9.1 Eigen

WWW サイトから、Stable Release (2022/10/17 現在、version 3.4.0) のファイルを取ってきて、インストールするのは簡単である。

eigen-3.4.0/INSTALL というファイルを読むと、二つの方法が書かれている。

方法 1 は単純で、マニュアルでコピーする感じ。eigen-3.4.0/Eigen というディレクトリィをどこか適当な場所 (インクルード・ファイルを置くことにしてあるディレクトリィ) にコピーする、というものである。私はその手のディレクトリィは、通常 /usr/local/include にコピーすることにしているにしているので、それを踏襲すると次のようになる。

方法 1: ターミナルで実行

```
curl -O https://gitlab.com/libeigen/eigen/-/archive/3.4.0/eigen-3.4.0.tar.gz
tar xzf eigen-3.4.0.tar.gz
cd eigen-3.4.0
sudo mkdir -p /usr/local/include
sudo cp -pr Eigen /usr/local/include
```

(注: 古いバージョンをインストールしていた場合は、それをどけておいてからコピーするとか考える。)

こうしてインストールした場合はコンパイル時に、-I /usr/local/include というオプションを使うことになる。

方法 2 は cmake を使うものである。

方法 2: ターミナルで実行

```
curl -O https://gitlab.com/libeigen/eigen/-/archive/3.4.0/eigen-3.4.0.tar.gz
tar xzf eigen-3.4.0.tar.gz
mkdir eigen-build
cd eigen-build
cmake ../eigen-3.4.0
sudo make install
```

こうすると /usr/local/include/eigen3/Eigen にインストールされる。コンパイル時に -I /usr/local/include/eigen3 というオプションを使うことになる。

ball.cpp[7]

```
/*
 * ball.cpp --- はねるボール
 *   http://nalab.mind.meiji.ac.jp/~mk/program/ode_prog/ball.cpp
 *   cc -I /usr/local/include ball.cpp
 *   ./a.out > ball.data
 *   gnuplot で plot "ball.data" with lp
 */

#include <iostream>
#include <math.h>
#include <Eigen/Dense>
using namespace Eigen;

double m, g, Gamma, e;

VectorXd f(double t, VectorXd x)
{
  VectorXd y(4);
  y(0) = x(2);
  y(1) = x(3);
  y(2) = - Gamma / m * x(2);
  y(3) = - g - Gamma / m * x(3);
  return y;
}

int main(void)
{
  int n, N;
  double tau, Tmax, t,pi;
  VectorXd x(4),k1(4),k2(4),k3(4),k4(4);

  pi = 4 * atan(1.0);
  m = 100;
  g = 9.8;
  Gamma = 1.0;
  e = 1.0;

  Tmax = 20;
  N = 1000;
  tau = Tmax / N;
  x << 0,0,50*cos(pi*50/180),50*sin(pi*50/180);
  for (n = 0; n < N; n++) {
    t = n * tau;
    k1 = tau * f(t, x);
    k2 = tau * f(t+tau/2, x+k1/2);
    k3 = tau * f(t+tau/2, x+k2/2);
    k4 = tau * f(t+tau, x+k3);
    x = x + (k1 + 2 * k2 + 2 * k3 + k4) / 6;
```

入手して実行する

```
curl -O https://m-katsurada.sakura.ne.jp/program/ode_prog/ball.cpp
c++ -I/usr/local/include -o ball ball.cpp
./ball > ball.data
gnuplot
```

gnuplot が起動して gnuplot>　というプロンプトが表示される。

```
gnuplot> plot "ball.data" with lp
```

これでボールの軌跡が描かれる。以下は画像ファイルへの保存。

```
gnuplot> set term png
gnuplot> set output "ball.png"
gnuplot> replot
gnuplot> quit
```

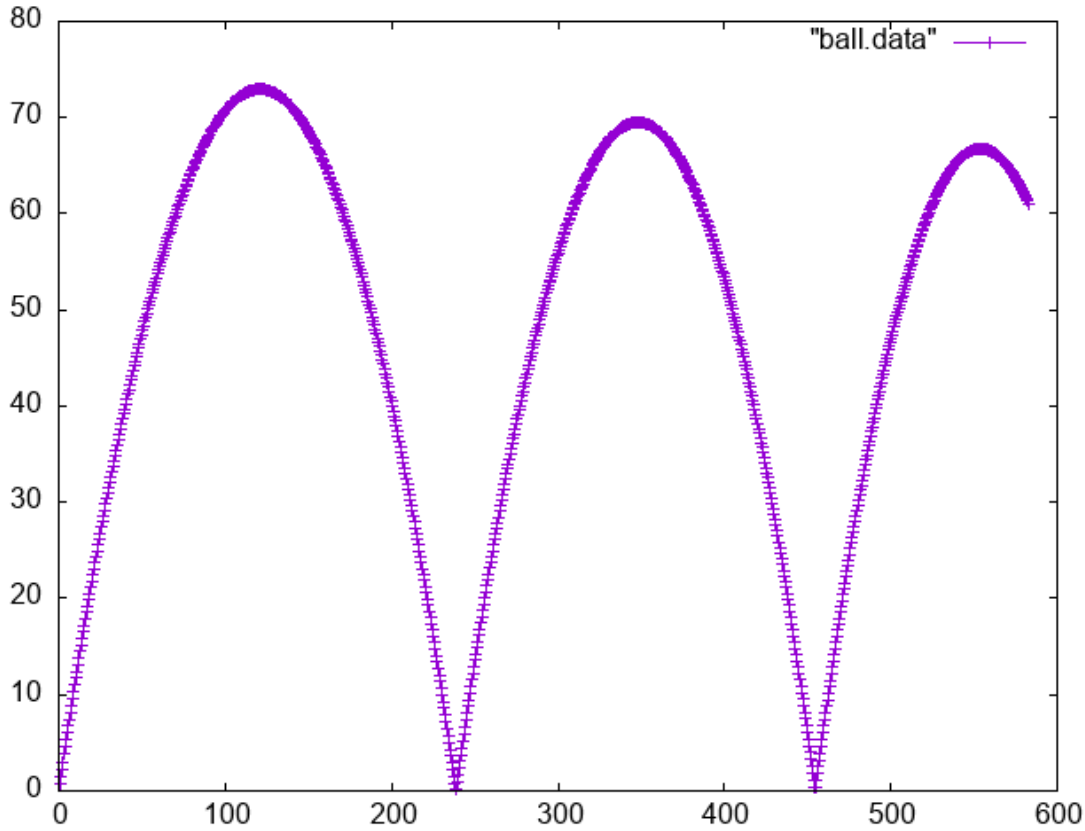

図 1: はずむボール, 空気抵抗あり, 完全弾性衝突

2.10　入出力の書式 (とりあえずの例)

- `std::setw`(w)
- `std::setprecision`(n)
- 形式の選択

- %f 相当
  std::cout << std::fixed あるいは cout.setf(ios::fixed, ios::floatfield);
- %e 相当
  std::cout << std::scientific あるいは cout.setf(ios::scientific, ios::floatfield)
- %g 相当
  std:cout << std::defaultfloat あるいは cout << resetiosflags(ios::floatfield);

```
#include <iostream>

int main(void)
{
  double x=1234.5678;
  std::cout << x << std::endl;
  std::cout << std::scientific << x << std::endl;
  std::cout << std::fixed << x << std::endl;
  std::cout << std::defaultfloat << x << std::endl;
  return 0;
}
```

二分法のプログラム、元々の C プログラムでは、固定形式も使っていたのだった。

```
/*
 * bisection.c -- 二分法 (bisection method) で方程式 f(x)=0 を解く
 *   コンパイル: gcc -o bisection bisection.c -lm
 *   実行: ./bisection
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(void)
{
  int i, maxitr = 100;
  double alpha, beta, a, b, c, eps;
  double fa, fb, fc;
  double f(double);

  printf(" 探す区間の左端α, 右端β, 許容精度ε =");
  scanf("%lf %lf %lf", &alpha, &beta, &eps);

  a = alpha;
  b = beta;

  fa = f(a);
  fb = f(b);
  if (fa * fb > 0.0) {
    printf(" f(α) f(β) > 0 なのであきらめます。\n");
    exit(0);
  }
  else {
    printf("f(%20.15f)=%9.2e, f(%20.15f)=%9.2e\n", a, fa, b, fb);
    for (i = 0; i < maxitr; i++) {
      c = (a + b) / 2;
      fc = f(c);
```

```
      if (fc == 0.0)
        break;
      else if (fa * fc <= 0.0) {
        /* 左側 [a,c] に根がある */
        b = c;
        fb = fc;
      }
      else {
        /* 左側 [a,c] には根がないかもしれない。[c,b] にあるはず */
        a = c;
        fa = fc;
      }
      printf ("f(%20.15f)=%9.2e, f(%20.15f)=%9.2e\n", a, fa, b, fb);
      if ((b - a) <= eps)
        break;
    }
    printf ("f(%20.15f)=%e\n", c, fc);
  }
  return 0;
}

double f(double x)
{
  return cos(x) - x;
}
```

これを C++ に直すと次のようになる。std:: つけると、ゲップが出そう。

```
/*
 * bisection.cpp -- 二分法 (bisection method) で方程式 f(x)=0 を解く
 *   コンパイル: gcc -o bisection bisection.c -lm
 *   実行: ./bisection
 */

#include <iostream>
#include <iomanip>
#include <cmath>

int main(void)
{
  int i, maxitr = 100;
  double alpha, beta, a, b, c, eps;
  double fa, fb, fc;
  double f(double);

  std::ios_base::fmtflags original_flags = std::cout.flags();
  std::cout << " 探す区間の左端α, 右端β, 許容精度ε =";
  std::cin >> alpha >> beta >> eps;

  a = alpha;
  b = beta;

  fa = f(a);
  fb = f(b);
  if (fa * fb > 0.0) {
    std::cout << " f(α) f(β) > 0 なのであきらめます。" << std::endl;
    exit(0);
  }
  else {
    std::cout << "f("
              << std::fixed << std::setw(20) << std::setprecision(15) << a
              << ")="
              << std::scientific << std::setw(9) << std::setprecision(2) << fa
              << ", f("
```

```
            << std::fixed << std::setw(20) << std::setprecision(15) << b
            << ")="
            << std::scientific << std::setw(9) << std::setprecision(2) << fb << std::endl;
    for (i = 0; i < maxitr; i++) {
      c = (a + b) / 2;
      fc = f(c);
      if (fc == 0.0)
        break;
      else if (fa * fc <= 0.0) {
        /* 左側 [a,c] に根がある */
        b = c;
        fb = fc;
      }
      else {
        /* 左側 [a,c] には根がないかもしれない。[c,b] にあるはず */
        a = c;
        fa = fc;
      }
      std::cout << "f("
                << std::fixed << std::setw(20) << std::setprecision(15) << a
                << ")="
                << std::scientific << std::setw(9) << std::setprecision(2) << fa
                << ", f("
                << std::fixed << std::setw(20) << std::setprecision(15) << b
                << ")="
                << std::scientific << std::setw(9) << std::setprecision(2) << fb
                << std::endl;
      if ((b - a) <= eps)
        break;
    }
    std::cout << "f("
              << std::fixed << std::setw(20) << std::setprecision(15) << c
              << ")=";
    std::cout.flags(original_flags);
    std::cout << std::scientific << std::setprecision(6) << fc << std::endl;
  }
  return 0;
}

double f(double x)
{
  return cos(x) - x;
}
```

3 複素数

std::complex というクラス・ライブラリィがある。

「C, C++ で複素数」[8] と言うのを書いてあって、とりあえずはそちらを見て下さい (特に http://nalab.mind.meiji.ac.jp/~mk/labo/text/complex-c/node6.html)。

(C にも複素数が導入されて、それも使うことが出来るのかな？面倒なことは必要が生じない限り考えないようにしよう…)

[8] http://nalab.mind.meiji.ac.jp/~mk/labo/text/complex-c/

4　これまで使ったことのあるクラス・ライブラリィ

PROFIL　「BIAS/Profil を動かす」[9] (2017)
　以下は内輪向けだけど、

- 「区間演算用ソフトウェア BIAS/Profil の紹介」[10],
- 「BIAS/PROFIL ノート」[11]
- 「Profil を使ったプログラム例」[12]
- 「BIAS, Profil のパッチ当てメモ」[13]

kv　「kv を試してみる」[14]

Eigen　「Eigen – 行列演算用 C++ クラスライブラリィ」[15]

多倍長計算関係　色々あるが、「多倍長計算ノート」[16] を見て下さい。

A　Boost で特殊関数

　Boost[17] と言う有名で巨大な (サグラダファミリアのような？) クラスライブラリィがあるらしいけれど、その中に特殊関数が含まれている。
　Chapter 6. Special Functions[18]
　楕円関数の計算で使ったことがある。

A.1　easy_elliptic.hpp

```
/*
 * easy_elliptic.hpp --- boost を簡単に使って、K(k),sn(u;k),cn(u;k),dn(u;k) 計算
 *  よく考えてみたら、agm.cpp の内容を移せば、boost に依存しないように出来る。
 *  つまり全く自前の計算で済ませられる。これは驚いた。
 */

#include <boost/math/special_functions.hpp>

static double K(double k)
{
  return boost::math::ellint_1(k);
}

static double F(double k, double phi)
{
  return boost::math::ellint_1(k, phi);
}

static double sn(double k, double x)
```

[9] http://nalab.mind.meiji.ac.jp/~mk/knowhow-2017/node1.html
[10] http://nalab.mind.meiji.ac.jp/~mk/labo/members/validating/intro-bias-profil/
[11] http://nalab.mind.meiji.ac.jp/~mk/labo/members/validating/bias-profil/
[12] http://nalab.mind.meiji.ac.jp/~mk/labo/members/validating/how-to-program-in-Profil/
[13] http://nalab.mind.meiji.ac.jp/~mk/labo/members/validating/How-to-patch-BIAS-Profil/
[14] http://nalab.mind.meiji.ac.jp/~mk/labo/text/studying-kv/
[15] http://nalab.mind.meiji.ac.jp/~mk/knowhow-2013/node32.html
[16] http://nalab.mind.meiji.ac.jp/~mk/labo/text/on-multiprecision/
[17] https://www.boost.org/
[18] https://www.boost.org/doc/libs/1_65_0/libs/math/doc/html/special.html

```
{
  return boost::math::jacobi_sn(k, x);
}

static double cn(double k, double x)
{
  return boost::math::jacobi_cn(k, x);
}

static double dn(double k, double x)
{
  return boost::math::jacobi_dn(k, x);
}

// 複素関数として sn()
static std::complex<double> jacobi_sn(double k, std::complex<double> z)
{
  double kp,m,x,y,s,c,d,s1,c1,d1,den;
  x = z.real(); y = z.imag(); kp = sqrt(1 - k * k); m = k * k;
#ifdef OLD
  s  = boost::math::jacobi_sn(k,  x);
  c  = boost::math::jacobi_cn(k,  x);
  d  = boost::math::jacobi_dn(k,  x);
  s1 = boost::math::jacobi_sn(kp, y);
  c1 = boost::math::jacobi_cn(kp, y);
  d1 = boost::math::jacobi_dn(kp, y);
#else
  s =  boost::math::jacobi_elliptic(k, x, &c, &d);
  s1 = boost::math::jacobi_elliptic(kp, y, &c1, &d1);
#endif
  den = c1 * c1 + m * s * s * s1 * s1; // denominator: 分母
  return std::complex<double>(s * d1 / den, c * d * s1 * c1 / den);
}

// 複素数版 Jacobi の sn(), cn(), dn()
static std::complex<double> jacobi_elliptic(double k,
    std::complex<double> z,
    std::complex<double> *cn,
    std::complex<double> *dn)
{
  double kp,m,x,y,s,c,d,s1,c1,d1,den;
  std::complex<double> sn;
  x = z.real(); y = z.imag(); kp = sqrt(1 - k * k); m = k * k;
#ifdef OLD
  s  = boost::math::jacobi_sn(k, x);
  c  = boost::math::jacobi_cn(k, x);
  d  = boost::math::jacobi_dn(k, x);
  s1 = boost::math::jacobi_sn(kp, y);
  c1 = boost::math::jacobi_cn(kp, y);
  d1 = boost::math::jacobi_dn(kp, y);
#else
  s =  boost::math::jacobi_elliptic(k, x, &c, &d);
  s1 = boost::math::jacobi_elliptic(kp, y, &c1, &d1);
#endif
  den = c1 * c1 + m * s * s * s1 * s1; // denominator: 分母
  sn  = std::complex<double>(s * d1 / den, c * d * s1 * c1 / den);
  *cn = std::complex<double>(c * c1 / den, - s * d * s1 * d1 / den);
  *dn = std::complex<double>(d * c1 * d1 / den, - m * s * c * s1 / den);
  return sn;
}

// まだ十分テストしていないけれど
static double my_jacobi_elliptic(double k, double u, double *cn, double *dn)
```

```
{
  int n, maxn = 10, N, success;
  double kp, a[maxn+1], b[maxn+1], c[maxn+1], phi[maxn+1], eps, myK, pi;

  success = 0;
  pi = 4.0 * atan(1.0);
  // AGM
  eps = 1e-15;
  kp = sqrt(1 - k * k);
  a[0] = 1.0; b[0] = kp; c[0] = k;
  for (n = 1; n <= maxn; n++) {
    a[n] = (a[n-1] + b[n-1]) / 2;
    b[n] = sqrt(a[n-1] * b[n-1]);
    c[n] = (a[n-1] - b[n-1]) / 2;
    if (c[n] < eps * a[n]) {
      success = 1;
      break;
    }
  }
  if (!success) {
    std::cerr << "収束しませんでした。" << std::endl;
  }
  N = n;
  // 完全楕円積分
  myK = pi / (2 * a[N]); // これは返さない...
  // 楕円関数の amplitude の計算
  phi[N] = (1 << n) * a[N] * u; // φ N
  for (n = N; n >= 1; n--)
    phi[n-1] = (phi[n] + asin(c[n] * sin(phi[n]) / a[n])) / 2;
  //
  *cn = cos(phi[0]);
  *dn = cos(phi[0]) / cos(phi[1]-phi[0]);
  return sin(phi[0]);
}
```