MOwNiT Lab7 - sprawozdanie

Autor: Michał Flak

Polecenie

Tematem zadania będzie obliczanie metodami Monte Carlo całki funkcji x^2 oraz 1/sqrt(x) w przedziale (0,1) Proszę dla obydwu funkcji:

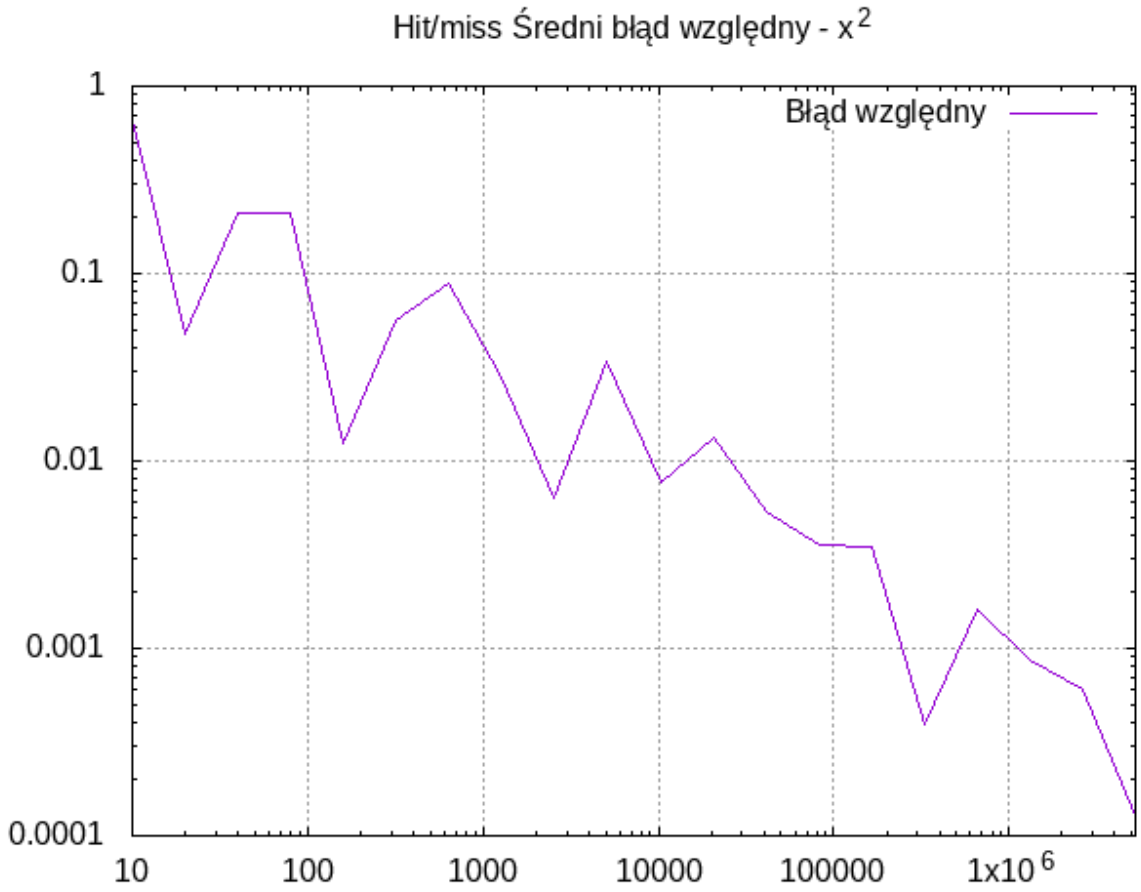
1. Napisać funkcję liczącą całkę metodą "hit-and-miss". Czy będzie ona dobrze działać dla funkcji 1/sqrt(x)?
2. Policzyć całkę przy użyciu napisanej funkcji. Jak zmienia się błąd wraz ze wzrostem liczby podprzedziałów? Narysować wykres tej zależności przy pomocy Gnuplota. Przydatne będzie skala logarytmiczna.
3. Policzyć wartość całki korzystając z funkcji Monte Carlo z GSL. Narysować wykres zależności błędu od ilości wywołań funkcji dla różnych metod (PLAIN, MISER, VEGAS).

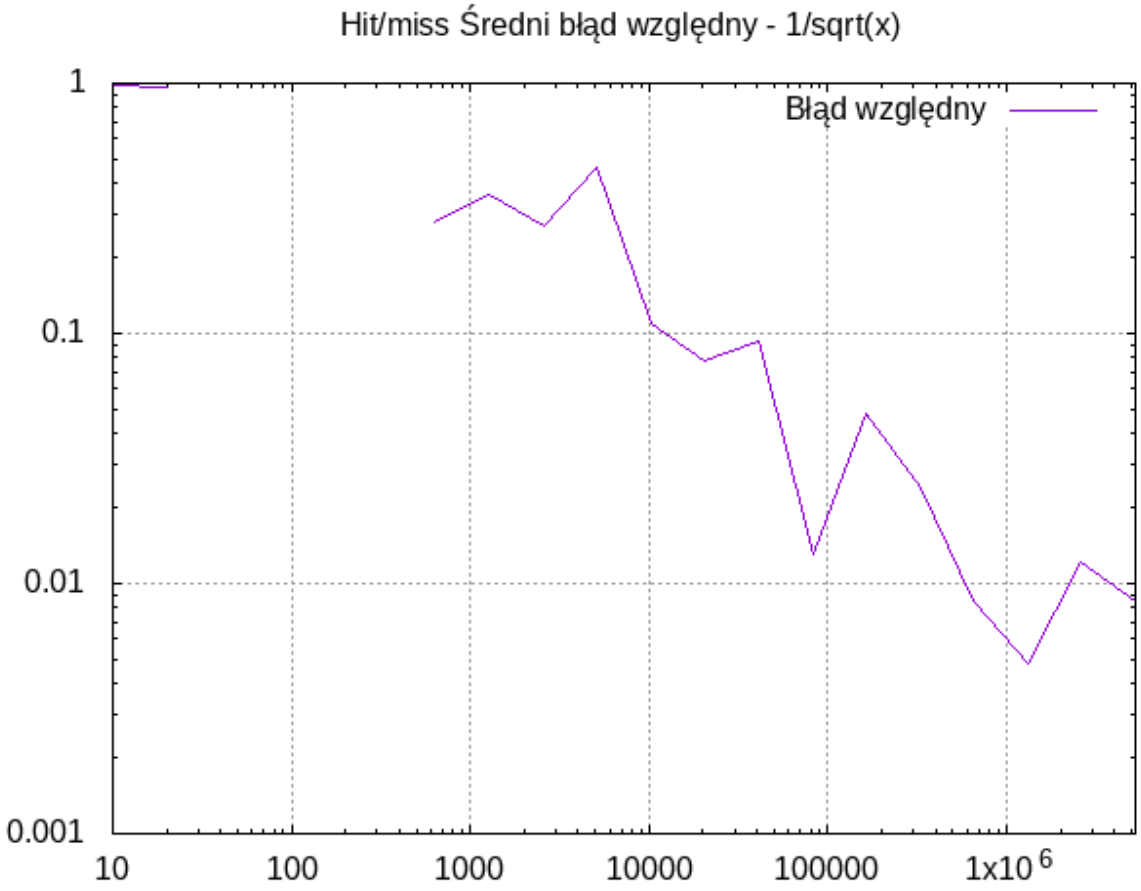
Rozwiązanie

Napisano program, który dla określonej argumentem funkcji (1=x^2, 2=1/sqrt(x)) całkuje ją na przedziale (0,1) w pętli z posuwem logarytmicznym ilości punktów / wywołań w przedziale (10, 10000000). Na wyjście standardowe podawany jest wynik całkowania oraz błąd względny wobec wartości policzonej analitycznie.

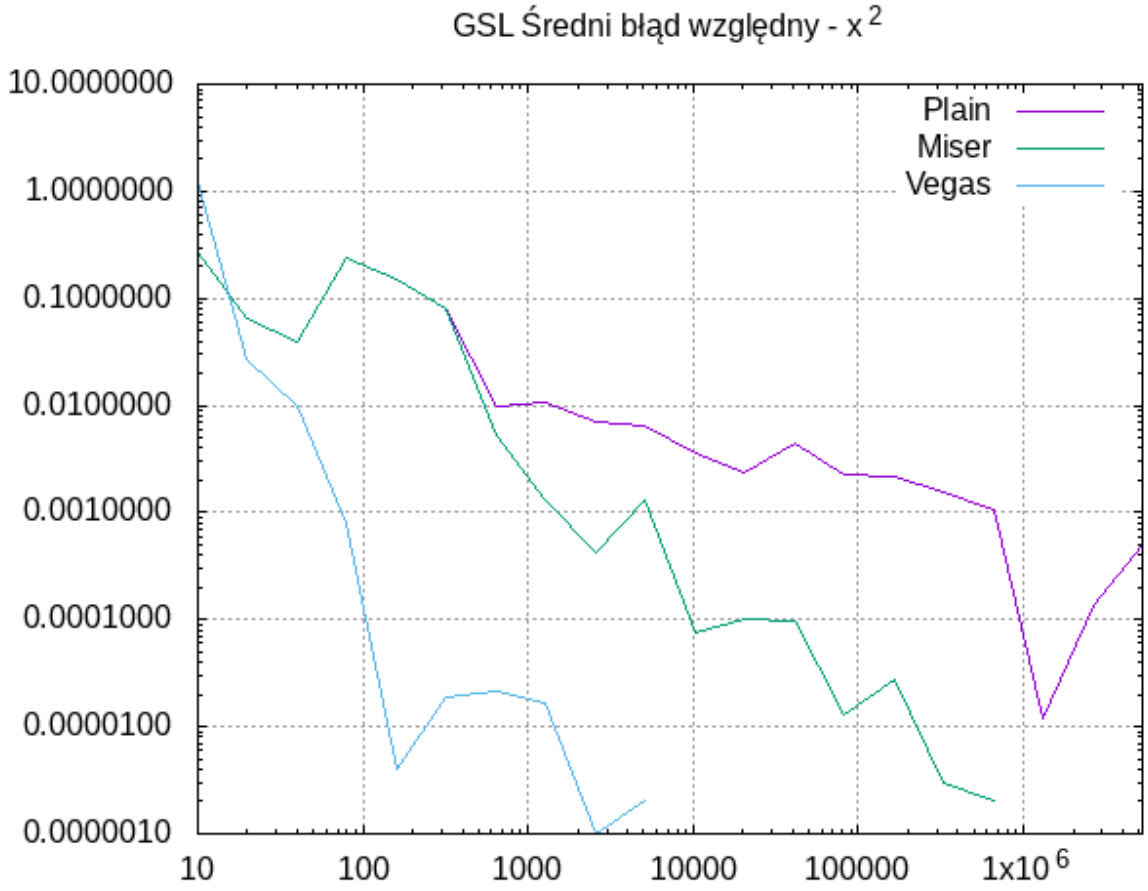
Wyniki

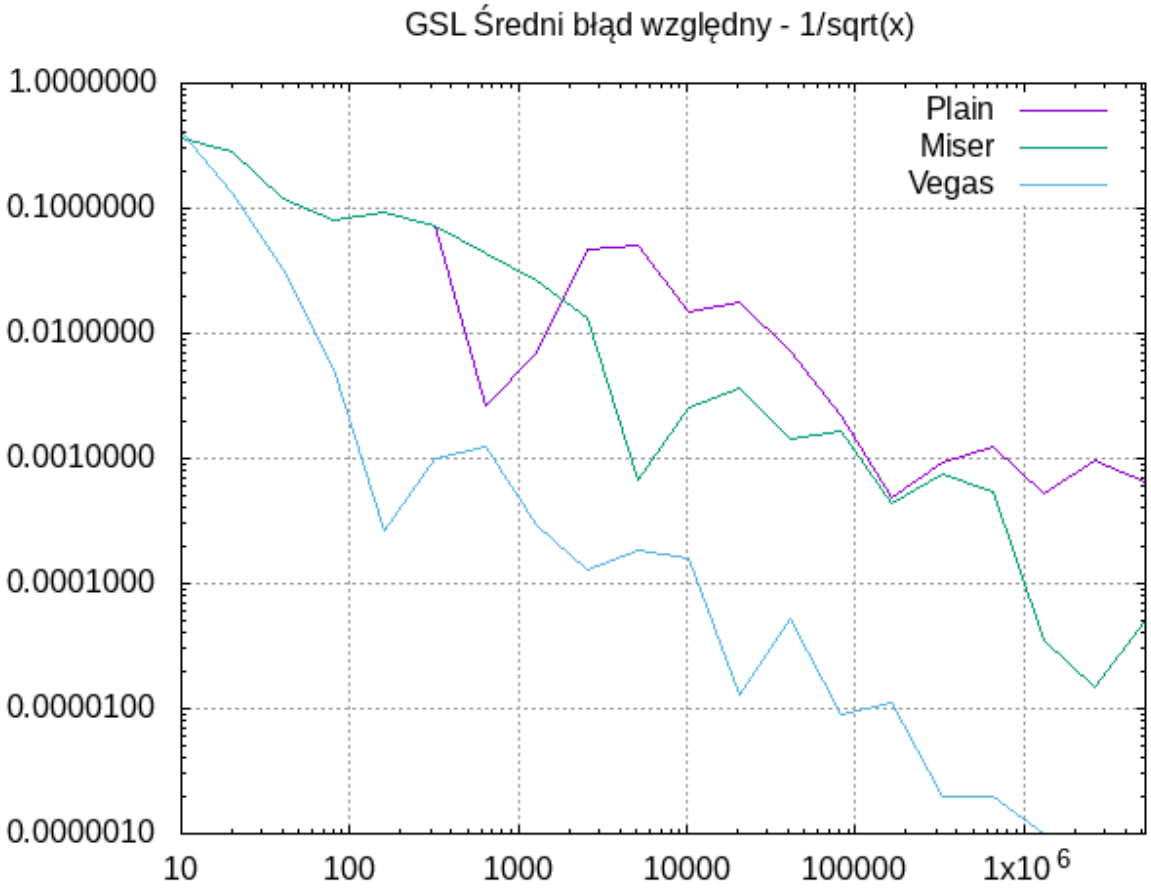
Hit or miss




Hit/miss Średni błąd względny - 1/sqrt(x)
Błąd względny
1
0.1
0.01
0.001
10
100
1000
10000
100000
1x10^6

GSL


GSL Średni błąd względny - x 2
10.0000000
1.0000000
0.1000000
0.0100000
0.0010000
0.0001000
0.0000100
0.0000010
10
100
1000
10000
100000
1x10 6
Plain
Miser
Vegas



Wnioski

We wszystkich metodach zwiększenie ilości punktów / wywołań proporcjonalnie zmniejsza błąd względny.

Hit or miss

Metoda ta niezbyt nadaje się do estymowania całek funkcji które swoje granice mają w nieskończoności - jak właśnie 1/sqrt(x). Nie da się jednoznacznie określić, jaka powinna być wysokość prostokąta, w którym umieszczamy punkty. Ja przyjąłem wartość 1000. Jednak przy niskich liczbach punktów zdarzało się, że żaden z punktów nie trafił pod wykres. Obserwujemy to jako przerwę w wykresie powyżej - błąd względny wynosił tam nieskończoność, co możemy zauważyć w danych wynikowych.

GSL

Poza najmniejszymi liczbami wywołań błąd względny metod plasuje się następująco:

1. Vegas - najmniejszy
2. Miser
3. Plain - największy

Wyniki - surowe dane

Funkcja 1

`n, hm, ehm, p, ep, m, em, v, ev`

10, 0.200000, 0.666667, 0.460188, 0.275658, 0.460188, 0.275658, 0.149168, 1.234610
20, 0.350000, 0.047619, 0.356649, 0.065376, 0.356649, 0.065376, 0.324624, 0.026828
40, 0.275000, 0.212121, 0.320847, 0.038915, 0.320847, 0.038915, 0.330080, 0.009855
80, 0.275000, 0.212121, 0.268871, 0.239752, 0.268871, 0.239752, 0.333074, 0.000780
160, 0.337500, 0.012346, 0.289883, 0.149888, 0.289883, 0.149888, 0.333332, 0.000004
320, 0.353125, 0.056047, 0.309055, 0.078556, 0.309055, 0.078556, 0.333327, 0.000019
640, 0.365625, 0.088319, 0.330025, 0.010023, 0.335084, 0.005225, 0.333340, 0.000021
1280, 0.342969, 0.028094, 0.329815, 0.010667, 0.333779, 0.001335, 0.333339, 0.000017
2560, 0.331250, 0.006289, 0.331055, 0.006881, 0.333192, 0.000424, 0.333333, 0.000001
5120, 0.322266, 0.034343, 0.335474, 0.006381, 0.332900, 0.001302, 0.333333, 0.000002
10240, 0.335938, 0.007752, 0.334547, 0.003627, 0.333358, 0.000075, 0.333333, 0.000000
20480, 0.337891, 0.013487, 0.332540, 0.002386, 0.333368, 0.000103, 0.333333, 0.000000
40960, 0.335132, 0.005367, 0.331863, 0.004430, 0.333366, 0.000098, 0.333333, 0.000000
81920, 0.334534, 0.003588, 0.332587, 0.002244, 0.333329, 0.000013, 0.333333, 0.000000
163840, 0.334503, 0.003497, 0.332610, 0.002175, 0.333324, 0.000027, 0.333333, 0.000000
327680, 0.333203, 0.000391, 0.332823, 0.001534, 0.333334, 0.000003, 0.333333, 0.000000
655360, 0.332790, 0.001634, 0.332986, 0.001042, 0.333334, 0.000002, 0.333333, 0.000000
1310720, 0.333048, 0.000856, 0.333337, 0.000012, 0.333333, 0.000000, 0.333333, 0.000000
2621440, 0.333535, 0.000606, 0.333379, 0.000136, 0.333333, 0.000000, 0.333333, 0.000000
5242880, 0.333376, 0.000128, 0.333503, 0.000508, 0.333333, 0.000000, 0.333333, 0.000000

Funkcja 2

n, hm, ehm, p, ep, m, em, v, ev
10, 100.000000, 0.980000, 1.460423, 0.369466, 1.460423, 0.369466, 1.428227, 0.400337
20, 50.000000, 0.960000, 1.551885, 0.288755, 1.551885, 0.288755, 1.762795, 0.134562
40, 0.000000, inf, 1.787553, 0.118848, 1.787553, 0.118848, 1.937376, 0.032324
80, 0.000000, inf, 1.851802, 0.080029, 1.851802, 0.080029, 1.989552, 0.005251
160, 0.000000, inf, 1.832153, 0.091612, 1.832153, 0.091612, 2.000534, 0.000267
320, 0.000000, inf, 2.155188, 0.072007, 2.155188, 0.072007, 1.998017, 0.000993
640, 1.562500, 0.280000, 2.005278, 0.002632, 2.091117, 0.043573, 1.997546, 0.001229
1280, 3.125000, 0.360000, 1.986093, 0.007002, 1.947881, 0.026757, 1.999420, 0.000290
2560, 2.734375, 0.268571, 2.098686, 0.047023, 1.973676, 0.013338, 1.999741, 0.000130
5120, 1.367188, 0.462857, 2.108129, 0.051291, 2.001361, 0.000680, 1.999634, 0.000183
10240, 2.246094, 0.109565, 2.030472, 0.015008, 1.994908, 0.002552, 1.999680, 0.000160
20480, 1.855469, 0.077895, 2.036364, 0.017857, 1.992758, 0.003634, 1.999974, 0.000013
40960, 1.831055, 0.092267, 2.014832, 0.007361, 2.002878, 0.001437, 1.999896, 0.000052
81920, 2.026367, 0.013012, 1.995557, 0.002227, 2.003260, 0.001628, 1.999983, 0.000009
163840, 2.099609, 0.047442, 1.999026, 0.000487, 1.999138, 0.000431, 1.999977, 0.000011
327680, 1.953125, 0.024000, 1.998126, 0.000938, 2.001511, 0.000755, 1.999997, 0.000002
655360, 2.017212, 0.008533, 1.997507, 0.001248, 2.001094, 0.000547, 1.999997, 0.000002
1310720, 1.990509, 0.004768, 1.998942, 0.000529, 1.999930, 0.000035, 1.999998, 0.000001
2621440, 1.976013, 0.012139, 1.998054, 0.000974, 1.999971, 0.000015, 1.999999, 0.000001
5242880, 1.983261, 0.008440, 1.998685, 0.000658, 1.999900, 0.000050, 1.999998, 0.000001

Kod źródłowy

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_monte.h>
#include <gsl/gsl_monte_plain.h>
```

```
#include <gsl/gsl_monte_miser.h>
#include <gsl/gsl_monte_vegas.h>

const double MIN = 0.0;
const double MAX = 1.0;

double drand(double lo, double hi) {
    return ((double)rand() * (hi-lo))/(double)RAND_MAX+lo;
}

typedef struct {
    double val;
    double err;
} IntegrationResult;

typedef enum {
    PLAIN,
    MISER,
    VEGAS
} GslIntegrationType;

double fun1 (double x){
    return x*x;
}

double fun2 (double x) {
    return 1.0/sqrt(x);
}

double fun1wrapped (
    double x[],
    size_t dim,
    void * p
    )
{
   return  fun1(x[0]);
}

double fun2wrapped (
    double x[],
    size_t dim,
    void * p
    )
{
   return  fun2(x[0]);
}

IntegrationResult integrateHitAndMiss(
    long sampleCount,
    double maxY,
    double (*fun)(double x),
    double expected
    )
```

```
{
    IntegrationResult result;



    int correctSamples = 0;
    for (long i = 0; i < sampleCount; ++i){
        double x = drand(MIN, MAX);
        double y = drand(0, maxY);

        if (y <= fun(x)){
            correctSamples++;
        }
    }

    result.val = ((double)correctSamples / (double) sampleCount)*maxY;
    result.err = fabs(result.val - expected)/result.val;
    return result;
}


IntegrationResult integrateGSL(
    long sampleCount,
    double (*fun)(double x[], size_t dim, void * p),
    double expected,
    GslIntegrationType type
    )
{

    IntegrationResult result;

    double res = 0.0;
    double err = 0.0;
    double xl[1] = {MIN};
    double xu[1] = {MAX};

    const gsl_rng_type *T;
    gsl_rng *r;

    const size_t dim = 1;

    gsl_monte_function G;
    G.dim = dim;
    G.f = fun;
    G.params = 0;

    gsl_rng_env_setup();

    T = gsl_rng_default;
    r = gsl_rng_alloc(T);

    switch(type){
        case PLAIN:
```

```
        {
            gsl_monte_plain_state *sp = gsl_monte_plain_alloc(dim);

            gsl_monte_plain_integrate(&G, xl, xu, dim, sampleCount, r, sp, &res, &err);
            gsl_monte_plain_free(sp);
            break;
        }
        case MISER:
        {
            gsl_monte_miser_state* sm = gsl_monte_miser_alloc(dim);

            gsl_monte_miser_integrate(&G, xl, xu, dim, sampleCount, r, sm, &res, &err);
            gsl_monte_miser_free(sm);
            break;
        }
        case VEGAS:
        {
            gsl_monte_vegas_state* sv = gsl_monte_vegas_alloc(dim);

            gsl_monte_vegas_integrate(&G, xl, xu, dim, sampleCount, r, sv, &res, &err);
            gsl_monte_vegas_free(sv);
            break;
        }
    }
    gsl_rng_free(r);

    result.val = res;
    result.err = fabs(result.val - expected)/res;
    return result;
}

int main(int argc, char** argv)
{
    if(argc != 2) return -1;
    srand(2139);

    int fchoice = atoi(argv[1]);
    IntegrationResult ihm, ip, im, iv;
    double expected;
    printf("n, hm, ehm, p, ep, m, em, v, ev\n");

    for(long sampleCount = 10; sampleCount<10000000; sampleCount*=2){
        switch (fchoice) {
            case 1:
            {
                expected = 1.0/3.0;
                ihm = integrateHitAndMiss(sampleCount, 1, fun1, expected);
                ip = integrateGSL(sampleCount, fun1wrapped, expected, PLAIN);
                im = integrateGSL(sampleCount, fun1wrapped, expected, MISER);
                iv = integrateGSL(sampleCount, fun1wrapped, expected, VEGAS);
                break;
            }
            case 2:
```

```
            {
                expected = 2;
                ihm = integrateHitAndMiss(sampleCount, 1000, fun2, expected);
                ip = integrateGSL(sampleCount, fun2wrapped, expected, PLAIN);
                im = integrateGSL(sampleCount, fun2wrapped, expected, MISER);
                iv = integrateGSL(sampleCount, fun2wrapped, expected, VEGAS);
                break;
            }
        }
        printf("%ld, %lf, %lf, %lf, %lf, %lf, %lf, %lf, %lf\n",
            sampleCount,
            ihm.val, ihm.err,
            ip.val, ip.err,
            im.val, im.err,
            iv.val, iv.err
            );
    }
    return 0;
}
```