Министерство образования Республики Беларусь

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

Факультет прикладной математики и информатики

Веренич Владислав Николаевич

Отчет по лабораторным работам по курсу

“Имитационное и статистическое моделирование”

студента 2 курса 13 группы

|  |  |
| --- | --- |
| Работа сдана 2022г. | **Преподаватель** |
| зачтена \_\_\_\_\_\_\_\_\_\_ 2022 г.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись преподавателя) | *Лобач Виктор Иванович*  доцент кафедры ММАД,  канд. физ.-мат. наук |
|  |  |

*Минск 2022*

**Лабораторная работа №1. Моделирование БСВ.**

**(Крайний срок сдачи до )**

Используя метод Маклерена-Марсальи построить датчик БСВ (1 датчик должен быть мультипликативно конгруентный, второй – на выбор). Исследовать точность построенной БСВ.

1. Осуществить моделирование *n* = 1000 реализаций БСВ с помощью мультипликативного конгруэнтного метода (МКМ) с параметрами *a*0, β, *M* = 231 .
2. Осуществить моделирование *n* = 1000 реализаций БСВ с помощью метода Макларена-Марсальи (один датчик должен быть мультипликативно конгруентный (п. 1), второй – на выбор).*K* – объем вспомогательной таблицы.
3. Проверить точность моделирования обоих датчиков (п. 1 и п. 2) с помощью критерия согласия Колмогорова и χ2-критерия Пирсона с уровнем значимости ε = 0.05.

**ВАРИАНТ:**

4) *a*0 = β = 78 125, K = 256

**Теория:**

**Мультипликативный конгруэнтный метод:**

Псевдослучайная последовательность  строится по следующим рекуррентным формулам:

где  - параметры датчика:  - множитель (*<M*), *M* – модуль,  - стартовое значение (нечетное число).

В данной работе брались значения: *M*=2147483648, ==65539.

**Метод Маклорена-Марсальи:**

Пусть  - псевдослучайные последовательности, порожденные независимо работающими датчиками;  - результирующая псевдослучайная последовательность реализация БСВ;

*V={V(0), V(1), …,V(K-1)}* – вспомогательная таблица *K* чисел.

Процесс вычисления  включает следующие этапы:

- первоначальное заполнение таблицы

*V*: 

- случайный выбор из таблицы:



-обновление табличных значений:

.

В данной работе в качестве  бралась последовательность (из 100 элементов), полученная мультипликативным конгруэнтным методом, описанным выше. В качестве , бралась последовательности (из 10000) элементов, полученная аналогичным способом с тем же M и . *K*=100.

** - критерий согласия Пирсона:**

Область возможных значений случайной величины разбивается на интервалы .

Рассматривается следующая статистика,

,

*n* – объем выборки,

 - количество элементов выборки, попавших в *k*-ый интервал,

 - вероятность попадания случайной величины в *k*-ый интервал.

Проверяется условие , где , *G* функция распределения распределения**,**  - уровень значимости (обычно =0.05).

В данной работе отрезок [0;1] разбивался на 10 интервалов.

**Критерий согласия Колмогорова:**

Рассматривается статистика:



где

,



Проверяется условие , где , *K* - функция распределенияраспределения Колмогорова**,**  - уровень значимости.

**Код программы:**

#include <iostream>

#include <cmath>

#include <fstream>

#include <algorithm>

double KolmogorovTest(double\* a, int n)

{

std::sort(a, a + n);

int i = 0;

double D = 0;

for (i = 0; i < n; i++)

{

if (D < fabs(((double)(i + 1)) / n - a[i]))

D = fabs(((double)(i + 1)) / n - a[i]);

}

std::cout << "Kolmogorov test result: D = " << D << " < 1.36\n";

return D;

}

double PirsonTest(double\* a, int n, int k)

{

std::sort(a, a + n);

int i = 0;

int count = 0, j = 0;

double xi2 = 0;

for (j = 1; j <= k; j++)

{

count = 0;

while ((a[i] < ((double)j) / k) & (i <= n))

{

i++;

count++;

}

xi2 += (count - ((double)n) / k) / (((double)n) \* k);

}

std::cout << "Pirson test result: xi2 = " << xi2 << " < 16.9\n";

return xi2;

}

int main()

{

std::ofstream foutMC("outputMC.txt");

std::ofstream foutMM("outputMM.txt");

constexpr unsigned long M{ 2'147'483'648 };

constexpr int n{ 1'000 };

constexpr long NN{ 1'000'000 };

constexpr unsigned long beta{ 79'507 };

constexpr unsigned K{ 64 };

//Generation of the first sample by the multiplicative congruent method.

unsigned long\* x = new unsigned long[NN];

x[0] = beta;

double\* b = new double[n];

b[0] = ((double)x[0]) / M;

long i;

long j;

foutMC << "Mult-Congr:\n";

for (i = 0; i < n; i++)

{

x[i] = (beta \* x[i - 1]) % M;

b[i] = ((double)x[i]) / M;

foutMC << b[i] << "\n";

}

std::cout << "Multi-Congruent sensor\n";

std::cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

KolmogorovTest(b, n);

PirsonTest(b, n, K);

std::cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

//Generation of the second sample by the multiplicative congruent method.

unsigned long beta2 = 4 \* beta + 1;

x[0] = beta2;

double\* c = new double[NN];

c[0] = ((double)x[0]) / M;

for (i = 0; i < NN; i++)

{

x[i] = (beta2 \* x[i - 1]) % M;

c[i] = ((double)x[i]) / M;

}

//Sample generation by the Maclaurin-Marsaglia method

double\* a = new double[n];

foutMM << "Makl-Mars:\n";

for (i = 0; i < n; i++)

{

a[i] = c[((long)(b[i] \* n)) + n \* i];

foutMM << a[i] << "\n";

}

std::cout << "\n\nMcLaren-Marsali sensor\n";

std::cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

KolmogorovTest(a, n);

PirsonTest(a, n, 10);

std::cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

int m;

std::cin >> m;

delete[] b;

delete[] c;

delete[] x;

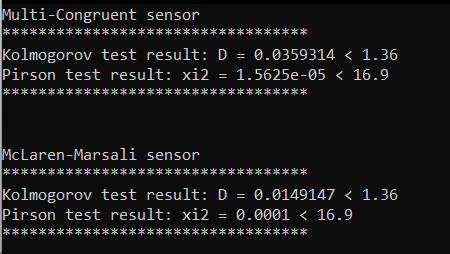
foutMC.close();

foutMM.close();

return 1;

}

**Результаты:**



**Лабораторная работа №2. Моделирование дискретных СВ.**

**(Крайний срок сдачи до )**

Смоделировать дискретную случайную величину (задания на стр. 18-22). Исследовать точность моделирования.

1. Осуществить моделирование *n* = 1000 реализаций СВ из заданных дискретных распределений.
2. Вывести на экран несмещенные оценки математического ожидания и дисперсии, сравнить их с истинными значениями.
3. Для каждой из случайных величин построить свой χ2-критерием Пирсона с уровнем значимость ε=0.05. Проверить, что вероятность ошибки I рода стремится к 0.05.
4. Осуществить проверку каждой из сгенерированных выборок каждым из построенных критериев.

**ВАРИАНТ:**

Бернулли – Bi(1,*p*), *p* = 0.2; Геометрическое – G(*p*), p = 0.6;

**Теория:**

**Распределение Пуассона (с параметром ):**

Случайная величина  принимает только целые неотрицательные значения, причем 

В данной работе, сначала моделировалась последовательность БСВ, а потом по каждой БСВ строился соответствующий элемент выборки распределения Пуассона: отрезок [0;1] разбивался на интервалы длин  проверялось, в какой интервал попадает элемент последовательности БСВ.

**Код программы:**

#include <iostream>

#include <fstream>

#include <limits.h>

#include <math.h>

#include <time.h>

#define printcnt 10000

#define tests 5000

int a0 = 262147;

int b = a0;

unsigned int M = 2147483648;

unsigned int curr\_rand = a0;

double rand1() {

return (double)rand() / RAND\_MAX;

}

int geom\_rand(double p) {

return ceil(log(rand1()) / log(1 - p)) - 1;

}

/\*\*

int poisson\_rand(double param) {

int k = 0;

double prod = rand1();

double expparam = exp(-param);

while (prod > expparam) {

prod \*= rand1();

++k;

}

return k;

}

\*/

int binomial\_rand(double p) {

return (p - rand1()) >= 0 ? 1 :0;

}

double E(int(\*f)(double), double param, int n) {

double summ = 0;

for (int i = 0; i < n; ++i)

summ += f(param);

return summ / n;

}

double D(int(\*f)(double), double param, int n) {

double summ = 0;

double \_E = E(f, param, n);

for (int i = 0; i < n; ++i)

summ += pow(f(param) - \_E, 2);

return summ / (n - 1);

}

double pirson\_criterium(int(\*f)(double), double param, int n, double\* P, int L) {

int\* v = new int[L];

double hi = 0;

for (int i = 0; i < L; ++i)

v[i] = 0;

for (int i = 0; i < n; ++i) {

int rd = f(param);

if (rd >= 0) {

v[rd < L ? rd : L - 1] += 1;

}

}

for (int i = 0; i < L; ++i)

hi += pow((v[i] - P[i] \* n), 2) / (n \* P[i]);

return hi;

}

/\*\*

double\* generate\_P\_poisson(double param, int L) {

double\* P = new double[L];

P[0] = 1;

P[L - 1] = 1;

for (int i = 1; i < L - 1; ++i)

P[i] = param;

for (int i = 2; i < L - 1; ++i)

P[i] \*= P[i - 1] / i;

for (int i = 0; i < L - 1; ++i)

P[i] \*= exp(-param);

for (int i = 0; i < L - 1; ++i)

P[L - 1] -= P[i];

return P;

}

\*/

double\* generate\_P\_geom(double param, int L) {

double\* P = new double[L];

P[0] = param;

P[L - 1] = 1;

for (int i = 1; i < L - 1; ++i)

P[i] = 1 - param;

for (int i = 2; i < L - 1; ++i)

P[i] \*= P[i - 1];

for (int i = 1; i < L - 1; ++i)

P[i] \*= param;

for (int i = 0; i < L - 1; ++i)

P[L - 1] -= P[i];

return P;

}

//1; n / 1; n\* (n - 1)/(1\*2)

double\* generate\_P\_binomial(double p, int L) {

double\* P = new double[L];

P[0] = 1;

P[L - 1] = 1;

for (size\_t i{ 1 }; i < L-1; ++i) {

P[i] = P[i - 1] \* (L - i - 1) / (i);

}

for (size\_t i{ 0 }; i < L - 1; ++i)

{

P[L - 1] -= P[i];

}

return P;

}

int main() {

srand(time(NULL));

int L = 4;

int n = 5000;

double geom\_param = 0.6;

//double poisson\_param = 0.7;

double binomial\_param = 0.2;

double C1 = 21.5689;

double C2 = 7.8147;

int geom\_count = 0;

//int poisson\_count = 0;

int binomial\_count = 0;

for (int i = 0; i < tests; ++i) {

if (pirson\_criterium(geom\_rand, geom\_param, n, generate\_P\_geom(geom\_param, L), L) < C1)

geom\_count++;

if (pirson\_criterium(binomial\_rand, binomial\_param, n, generate\_P\_binomial(binomial\_param, L), L) < C2)

binomial\_count++;

}

std::cout << "Geom Pirson criterium: " << geom\_count << "/" << tests;

std::cout << "\nBinomial Pirson criterium: " << binomial\_count << "/" << tests;

std::cout << "\n\nGeom E = " << E(geom\_rand, geom\_param, n) << " | Real E = " << (1 - geom\_param) / geom\_param;

std::cout << "\nBinomial E = " << E(binomial\_rand, binomial\_param, n) << " | Real E = " << binomial\_param;

std::cout << "\n\nGeom D = " << D(geom\_rand, geom\_param, n) << " | Real D = " << (1 - geom\_param) / (geom\_param \* geom\_param);

std::cout << "\nBinomial D = " << D(binomial\_rand, binomial\_param, n) << " | Real D = " << binomial\_param << std::endl;

std::ofstream binomial\_out("binomial.txt");

std::ofstream geom\_out("geom.txt");

for (int i = 0; i < printcnt; ++i) {

binomial\_out << binomial\_rand(binomial\_param) << std::endl;

geom\_out << binomial\_rand(binomial\_param) << std::endl;

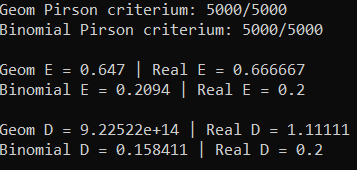
}

binomial\_out.close();

geom\_out.close();

}

Результыт:



**Лабораторная работа №3. Моделирование непрерывных СВ.**

**(Крайний срок сдачи )**

Смоделировать непрерывную случайную величину (задания на стр. 25-47). Исследовать точность моделирования.

1. Осуществить моделирование *n* = 1000 реализаций СВ из нормального закона распределения *N*(*m*, *s*2) с заданными параметрами. Вычислить несмещенные оценки математического ожидания и дисперсии, сравнить их с истинными.
2. Смоделировать *n* = 1000 СВ из заданных абсолютно непрерывных распределений. Вычислить несмещенные оценки математического ожидания и дисперсии, сравнить их с истинными значениями (если это возможно).
3. Для каждой из случайных величин построить свой критерий Колмогорова с уровнем значимость ε=0.05. Проверить, что вероятность ошибки I рода стремится к 0.05.
4. Для каждой из случайных величин построить свой χ2-критерий Пирсона с уровнем значимость ε=0.05. Проверить, что вероятность ошибки I рода стремится к 0.05.
5. Осуществить проверку каждой из сгенерированных выборок каждым из построенных критериев.

**ВАРИАНТ:**

4) *m* = 0, *s*2 = 1; Логистическое *LG*(*a*,*b*), *a* = 2, *b* = 3; Лапласа *L*(*a*), *a* = 2.

**Теория:**

**Распределение Вейбулла-Гнеденко (с параметрами** *k,* **):**

Распределение имеет плотность



Распределение может быть смоделировано методом обратной функции:

,

*U* – БСВ.

**Код программы:**

#define \_USE\_MATH\_DEFINES

#include <iostream>

#include <fstream>

#include <limits.h>

#include <algorithm>

#include <math.h>

#include <time.h>

#define ep 2.0

#define sqr(x) (x)\*(x)

using namespace std;

double rd() {

int r = rand();

while (r == 0 || r == RAND\_MAX)

r = rand();

return (double)r / RAND\_MAX;

}

/\*

#define gm -5.0

#define gs 5

double gaussian() {

return gm + gs \* sqrt(-2 \* log(rd())) \* cos(2 \* M\_PI \* rd());

}

#define lambda 0.5

#define hi\_n 10

#define k (hi\_n/2)

double hi() {

////

return -logistic\_a + (-logistic\_b) \* log(1 / rd() - 1);

}

#define ep 2.0

double exponential() {

return -log(rd()) / ep;

}

\*/

#define lp 2.0

double laplace() {

double y = rd();

if (y < 0.5)

return log(2 \* y) / lp;

return -log(2 \* (1 - y)) / lp;

}

#define logistic\_a 2.0

#define logistic\_b 3.0

double logistic() {

return -logistic\_a + (-logistic\_b) \* log(1 / rd() - 1);

}

double E(double(\*f)(), int n) {

double summ = 0;

for (int i = 0; i < n; ++i)

summ += f();

return summ / n;

}

double D(double(\*f)(), int n) {

double summ = 0;

double \_E = E(f, n);

for (int i = 0; i < n; ++i)

summ += pow(f() - \_E, 2);

return summ / (n - 1);

}

void printED(int n) {

//cout << "\nhi^2 distribution:\n\t E = " << E(hi, n) << '(' << -2.0 << ")\n\t D = " << D(hi, n) << '(' << 30.5 << ')';

//cout << "\ngaussian distribution:\n\t E = " << E(gaussian, n) << '(' << gm << ")\n\t D = " << D(gaussian, n) << '(' << gs \* gs << ')';

//cout << "\nexponential distribution:\n\t E = " << E(exponential, n) << '(' << 1 / ep << ")\n\t D = " << D(exponential, n) << '(' << 1 / (ep \* ep) << ')';

cout << "\nlogistic distribution:\n\t E = " << E(logistic, n) << '(' << -2.0 << ")\n\t D = " << D(logistic, n) << '(' << 29.4 << ')';

cout << "\nlaplace distribution:\n\t E = " << E(laplace, n) << "(0.00)\n\t D = " << D(laplace, n) << '(' << 0.5 << ')';

}

/\*\*

double Fgaussian(double x) {

return (1 + erf((x - gm) / sqrt(2 \* sqr(gs)))) / 2;

}

double Fexponential(double x) {

return 1 - exp(-ep \* x);

}

// gamma function with param k = 10 / 2

double gammaF(double x) {

return pow(log(x), 4);

}

double Fhi(double x) {

if (x < 0) {

return 0;

}

else {

return pow(lambda, k) \* pow(x, k - 1) \* exp(-lambda \* x) / gammaF(x);

}

}

\*/

double Flaplace(double x) {

if (x < 0)

return exp(lp \* x) / 2;

return 1 - exp(-lp \* x) / 2;

}

double Flogistic(double x) {

return 1 / (1 + exp(-(x - logistic\_a) / logistic\_b));

}

double komogorov(double(\*f)(), double(\*F)(double), int n, double\* arr)

{

for (int i = 0; i < n; ++i)

arr[i] = f();

sort(arr, arr + n);

double max = 0;

for (int i = 0; i < n; ++i) {

double b = abs(F(arr[i]) - (2 \* (double)i + 1) / (2 \* n));

if (max < b)

max = b;

}

return sqrt(n) \* (max + 1 / (2 \* n));

}

int map(int n, double x, double range\_start, double range\_end) {

if (x < range\_start)

return 0;

if (x > range\_end)

return n - 1;

return (int)((n - 2) \* (x - range\_start) / (range\_end - range\_start)) + 1;

}

double pearson(double(\*f)(), int\* v, int L, int N, double range\_start, double range\_end, double\* P) {

for (int i = 0; i < L; ++i)

v[i] = 0;

for (int i = 0; i < N; ++i)

v[map(L, f(), range\_start, range\_end)] += 1;

double hi = 0;

for (int i = 0; i < L; ++i)

hi += sqr(v[i] - N \* P[i]) / (N \* P[i]);

return hi;

}

void generateP(double(\*F)(double), int L, double range\_start, double range\_end, double\* P) {

P[0] = F(range\_start);

for (int i = 1; i < L - 1; ++i)

P[i] = F(range\_start + (range\_end - range\_start) \* i / (L - 2)) - F(range\_start + (range\_end - range\_start) \* (i - 1) / (L - 2));

P[L - 1] = 1;

for (int i = 0; i < L - 1; ++i)

P[L - 1] -= P[i];

}

void printKolmogorovCheck(int n, int testcnt) {

int laplaceKolmogorovCnt = 0;

/\*\*

int gaussianKolmogorovCnt = 0;

int exponentialKolmogorovCnt = 0;

int hiKolmogorovCnt = 0;

\*/

int logisticKolmogorovCnt = 0;

double C = 0.24;

double\* arr = new double[n];

for (int i = 0; i < testcnt; ++i) {

/\*\*

if (C < komogorov(gaussian, Fgaussian, n, arr))

++gaussianKolmogorovCnt;

if (C < komogorov(hi, Fhi, n, arr))

++hiKolmogorovCnt;

if (C < komogorov(exponential, Fexponential, n, arr))

++exponentialKolmogorovCnt;

\*/

if (C < komogorov(logistic, Flogistic, n, arr))

++logisticKolmogorovCnt;

if (C < komogorov(laplace, Flaplace, n, arr))

++laplaceKolmogorovCnt;

}

/\*

cout << "\nKolmogorov test gaussian: " << gaussianKolmogorovCnt << '/' << testcnt;

cout << "\nKolmogorov test exponential: " << exponentialKolmogorovCnt << '/' << testcnt;

cout << "\nKolmogorov test hi: " << hiKolmogorovCnt << '/' << testcnt;

\*/

cout << "\nKolmogorov test logistic: " << logisticKolmogorovCnt << '/' << testcnt;

cout << "\nKolmogorov test laplace: " << laplaceKolmogorovCnt << '/' << testcnt;

}

void printPearsonCheck(int n, int testcnt) {

/\*\*

int Lgaussian = 20;

double Cgaussian = 5.1435;

double\* Pgaussian = new double[Lgaussian];

int\* Vgaussian = new int[Lgaussian];

double range\_start\_gaussian = gm - 2 \* gs;

double range\_end\_gaussian = gm + 2 \* gs;

generateP(Fgaussian, Lgaussian, range\_start\_gaussian, range\_end\_gaussian, Pgaussian);

int gaussianPearsonCnt = 0;

for (int i = 0; i < testcnt; ++i)

if (pearson(gaussian, Vgaussian, Lgaussian, n, range\_start\_gaussian, range\_end\_gaussian, Pgaussian) > Cgaussian)

++gaussianPearsonCnt;

int Lexponential = 20;

double Cexponential = 5.1435;

double\* Pexponential = new double[Lexponential];

int\* Vexponential = new int[Lexponential];

double range\_start\_exponential = 0.001;

double range\_end\_exponential = log(ep);

generateP(Fexponential, Lexponential, range\_start\_exponential, range\_end\_exponential, Pexponential);

int exponentialPearsonCnt = 0;

for (int i = 0; i < testcnt; ++i)

if (pearson(exponential, Vexponential, Lexponential, n, range\_start\_exponential, range\_end\_exponential, Pexponential) > Cexponential)

++exponentialPearsonCnt;

int Lhi = 20;

double Chi = 30.1435;

double\* Phi = new double[Lhi];

int\* Vhi = new int[Lhi];

double range\_start\_hi = 0.001;

double range\_end\_hi = log(ep);

generateP(Fhi, Lhi, range\_start\_hi, range\_end\_hi, Phi);

int hiPearsonCnt = 0;

for (int i = 0; i < testcnt; ++i)

if (pearson(hi, Vhi, Lhi, n, range\_start\_hi, range\_end\_hi, Phi) > Chi)

++hiPearsonCnt;

\*/

int Llaplace = 20;

double Claplace = 0.89;

double\* Plaplace = new double[Llaplace];

int\* Vlaplace = new int[Llaplace];

double range\_start\_laplace = -5 \* lp;

double range\_end\_laplace = 5 \* lp;

generateP(Flaplace, Llaplace, range\_start\_laplace, range\_end\_laplace, Plaplace);

int laplacePearsonCnt = 0;

for (int i = 0; i < testcnt; ++i)

if (pearson(laplace, Vlaplace, Llaplace, n, range\_start\_laplace, range\_end\_laplace, Plaplace) > Claplace)

++laplacePearsonCnt;

int Llogistic = 20;

double Clogistic = 10.1435;

double\* Plogistic = new double[Llogistic];

int\* Vlogistic = new int[Llogistic];

double range\_start\_logistic = 0.001;

double range\_end\_logistic = log(ep);

generateP(Flogistic, Llogistic, range\_start\_logistic, range\_end\_logistic, Plogistic);

int logisticPearsonCnt = 0;

for (int i = 0; i < testcnt; ++i)

if (pearson(logistic, Vlogistic, Llogistic, n, range\_start\_logistic, range\_end\_logistic, Plogistic) > Clogistic)

++logisticPearsonCnt;

/\*\*

cout << "\nPearson test gaussian: " << gaussianPearsonCnt << '/' << testcnt;

cout << "\nPearson test exponential: " << exponentialPearsonCnt << '/' << testcnt;

cout << "\nPearson test hi: " << hiPearsonCnt << '/' << testcnt;

\*/

cout << "\nPearson test logistic: " << logisticPearsonCnt << '/' << testcnt;

cout << "\nPearson test laplace: " << laplacePearsonCnt << '/' << testcnt;

}

void print(double(\*f)(), char\* file, int cnt, int param) {

ofstream fout(file);

for (int i = 0; i < cnt; ++i)

fout << f() << endl;

fout.close();

/\*\*

char\* run = new char[100];

strcpy(run, "python graph.py ");

strcat(run, file);

int pos = strlen(run);

run[pos] = ' ';

itoa(param, run + pos + 1, 10);

system(run);

\*/

}

int main() {

srand(time(0));

printED(10000);

cout << "\n\n";

printPearsonCheck(1000, 1000);

cout << "\n\n";

printKolmogorovCheck(1000, 1000);

//print(gaussian, "gaussian.txt", 10000, 50);

//print(exponential, "exponential.txt", 10000, 20);

print(laplace, "laplace.txt", 10000, 20);

print(logistic, "logistic.txt", 10000, 20);

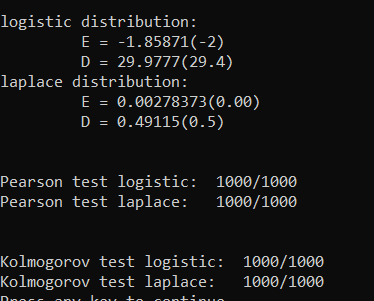
//print(hi, "hi.txt", 10000, 20);

cout << "\n";

system("pause");

}

**Результат:**

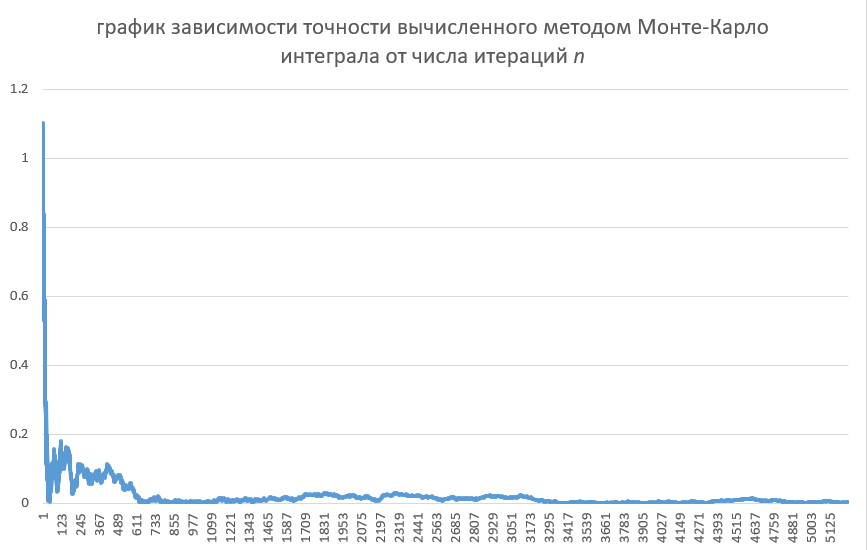
****

**Лабораторная работа №4. Метод Монте-Карло.**

**(Срок сдачи до )**

Вычислить значение интеграла, используя метод Монте-Карло. Оценить точность.

1. По методу Монте-Карло вычислить приближенное значения интегралов.
2. Сравнить полученное значение либо с точным значением (если его получится вычислить), либо с приближенным, полученным в каком-либо математическом пакете (например, в mathematica). Для этого построить график зависимости точности вычисленного методом Монте-Карло интеграла от числа итераций *n*.



**ВАРИАНТ:**

|  |  |  |
| --- | --- | --- |
| 4 |  |  |

**Теория:**

**Метод Монте-Карло приближенного вычисления интеграла:**

Необходимо вычислить .

Пусть  - произвольная случайная величина с плотностью распределения  имеющая конечный момент второго порядка.

Пусть  Тогда 

В качестве приближенного значения *a* можно взять



В данной работе в качестве  бралась случайная величина, равномерно распределенная на [0;1].

**Код программы(1-я часть):**

#define \_USE\_MATH\_DEFINES

#include <iostream>

#include <cmath>

#include <functional>

#include <fstream>

// Real value for the definite integral [0, 5\*pi / 7] of the following function

const double realValueVar4{ -0.485736 };

double funcVar4(double x)

{

return cos(x + sin(x));

}

double getRandZeroToOne()

{

return rand() / (RAND\_MAX + 1.0);

}

double getXi(const double a, const double b)

{

return a + getRandZeroToOne() \* (b - a);

}

double calculateDefiniteIntegral(const std::function<double(double)>& func,

const double a, const double b, const size\_t N,

std::ofstream &fout)

{

double result{ 0 };

for (size\_t i{ 0 }; i < N; ++i)

{

result += func(getXi(a, b));

fout << abs((b - a) \* result / N - realValueVar4) << "\n";

}

return (b - a) \* result / N;

}

void doVar4()

{

std::ofstream fout("delta.txt");

const double a{ 0 };

const double b{ 5 \* M\_PI / 7 };

const size\_t N{ 100 };

std::cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

std::cout << "Results for var 4:\n";

std::cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

std::cout << "Calculated value: " << calculateDefiniteIntegral(funcVar4, a, b, N, fout) << "\n";

std::cout << "Real value: " << realValueVar4 << "\n";

std::cout << "Iteration count: " << N << "\n";

std::cout << "Accuracy: " << 1.0 / sqrt(N) << "\n";

std::cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

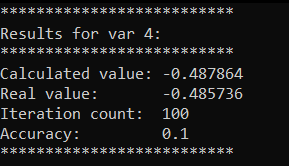
}

int main() {

doVar4();

}

Результат:



**Код программы(2-я часть):**

#define \_USE\_MATH\_DEFINES

#include <iostream>

#include <cmath>

#include <functional>

#include <fstream>

const double realValueVar4{ 3.21588 };

const double sigmaVar4{ 2 \* M\_PI };

double funcVar4(double x, double y)

{

return 1 / (std::pow(x, 2) + std::pow(y, 4));

}

bool isPointBelongToIntegrationAreaVar4(double x, double y)

{

return 1 <= std::pow(x, 2) + std::pow(y, 2) && std::pow(x, 2) + std::pow(y, 2) < 3;

}

double getRandZeroToOne()

{

return rand() / (RAND\_MAX + 1.0);

}

double getX(double z)

{

const double a{ -std::sqrt(3) };

const double A{ std::sqrt(3) };

return a + (A - a) \* z;

}

double getY(double z)

{

const double a{ -std::sqrt(3) };

const double A{ std::sqrt(3) };

return a + (A - a) \* z;

}

double calculateDefiniteIntegral(const std::function<double(double, double)> &func, const size\_t N,

std::ofstream &fout)

{

double sum{ 0 };

size\_t m{ 0 };

for (size\_t i{ 0 }; i < N; ++i)

{

const double x{ getX(getRandZeroToOne()) };

const double y{ getY(getRandZeroToOne()) };

if (isPointBelongToIntegrationAreaVar4(x, y))

{

++m;

sum += funcVar4(x, y);

fout << abs(sum \* sigmaVar4 / m - realValueVar4) << "\n";

}

}

return sum \* sigmaVar4 / m;

}

void doVar4()

{

std::ofstream fout("delta.txt");

const size\_t N{ 10000 };

std::cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

std::cout << "Results for var 4:\n";

std::cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

std::cout << "Calculated value: " << calculateDefiniteIntegral(funcVar4, N, fout) << "\n";

std::cout << "Real value: " << realValueVar4 << "\n";

std::cout << "Iteration count: " << N << "\n";

std::cout << "Accuracy: " << 1.0 / sqrt(N) << "\n";

std::cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

fout.close();

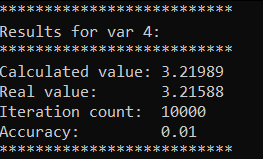
}

int main() {

doVar4();

}

Результат:



**Лабораторная работа №5. Метод Монте-Карло.**

**(Срок сдачи до )**

Решить систему линейных уравнений, используя метод Монте-Карло.

1. Решить систему линейных алгебраических уравнений  методом Монте-Карло.
2. Сравнить с решением данного уравнения, полученным в произвольном математическом пакете.
3. Построить график зависимости точности решения от длины цепи маркова и числа смоделированных цепей маркова.

**ВАРИАНТ:**

4) ;

**Теория:**

**Метод Монте-Карло приближенного решения системы линейных алгебраических уравнений:**

Необходимо решить систему, представленную в виде , где , собственные значения *A* по модулю меньше 1.

Наша цель – вычислить скалярное произведение вектора решения  с некоторым вектором .

Рассмотрим цепь Маркова с параметрами  такими что





 если 

 если 

Положим



Выберем некоторое натуральное *N* и рассмотрим случайную величину



Где 🡪🡪…🡪 - траекторая цепи Маркова.

*Qm* опряделяется как:



Тогда скалярное произведение вектором *h* и *x* приблизительно равно .

Можем найти *x*, скалярно умножая его на векторы *h* у которых в одной позиции стоит 1, а в остьльных – 0.

В данной работе выбиралось 

**Код программы:**

import numpy as np

import random

import matplotlib

import matplotlib.pyplot as plt

def get\_next\_markov(currentI, P):

rand = random.random()

i = 0

k = 0

while i < rand:

i += P[currentI][k]

k += 1

return k - 1

def count\_hi(l, A, f, h\_vector, pi, p):

rand = random.random()

i = 0

k = 0

while i < rand:

i += pi[k]

k += 1

current = k - 1

if h\_vector[current] == 0:

return 0

Q = h\_vector[current] / pi[current]

hi = Q \* f[current]

for i in range(l):

next\_markov = get\_next\_markov(current, p)

Q \*= A[current][next\_markov] / p[current][next\_markov]

hi += Q \* f[next\_markov]

current = next\_markov

return hi

def monte\_carlo\_solving(markov\_len, markov\_number, A, f):

size = len(A)

p = [[1 / size] \* size for \_ in range(size)]

pi = [1 / size] \* size

x = np.zeros(size)

h = np.identity(size)

for j in range(size):

x[j] = sum(count\_hi(markov\_len, A, f, h[:, j], pi, p) for \_ in range(markov\_number)) / markov\_number

return x

def norm(a, b):

return np.linalg.norm(a - b)

def get\_numbers():

return (2 \*\* x for x in range(13))

def calculate\_iteration\_number(func, rez, A, f):

return [norm(func(1000, i, A, f), rez) for i in get\_numbers()]

def calculate\_markov\_len(func, rez, A, f):

return [norm(func(i, 1000, A, f), rez) for i in get\_numbers()]

def calculate\_iteration\_number\_and\_\_markov\_len(func, rez, A, f):

return [norm(func(i, i, A, f), rez) for i in get\_numbers()]

def main():

A = [[1.1, -0.1, 0.2],

[0.1, 0.5, 0.3],

[-0.3, -0.1, 1.3]]

f = [-3, 1, 4]

B = np.eye(3) - A

rez = np.linalg.solve(A, f)

print("Точное решение:", rez)

print("Решение методом Монте-Карло:", monte\_carlo\_solving(1000, 1000, B.copy(), f))

offset\_markov\_len = calculate\_markov\_len(monte\_carlo\_solving, rez, B.copy(), f)

offset\_number\_iteration = calculate\_iteration\_number(monte\_carlo\_solving, rez, B.copy(), f)

print("Press enter button to show plots")

input()

draw(offset\_markov\_len, "markov chains length")

draw(offset\_number\_iteration, "markov chains count")

if \_\_name\_\_ == '\_\_main\_\_':

main()

**Результат:**

****

*График зависимости точности решения от длины цепи маркова и числа смоделированных цепей маркова*

