Министерство науки и высшего образования

Российской Федерации

Федеральное государственное бюджетное образовательное

учреждение высшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра Теоретической и Прикладной Информатики

Лабораторная работа №1 «СЛОЖНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ»

по предмету

«**ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ»**

Факультет: прикладной математики и информатики

Группа: ПМИ-12

Бригада: 7

Студенты: Курдюков И. Н.

Омельницкая Е. И.

Преподаватели: Еланцева И.Л.

Новосибирск, 2023

**Цель работы**

Изучить предметную область, связанную с вероятностно-статистическим моделированием.

**Постановка задачи**

Дляобобщенное гауссовского распределения*,* имеющего плотность

, где ,  – гамма-функция.

написать программу, с помощью которой пользователь будет иметь возможность работы с тремя распределениями:

– распределением, заданным в варианте и имеющим, помимо параметра формы n, параметры сдвига μ и масштаба λ (данное распределение будет называться **основным**);

– распределением в виде **смеси** двух основных распределений с параметрами (μ1, λ1, n1) и (μ2, λ2, n2) и параметром смеси *p*;

– **эмпирическим** распределением, строящимся по выборке.

Для каждого из распределений необходимо реализовать в виде набора функций следующие действия:

– вычисление значений плотности распределения по заданному аргументу (плотность для эмпирического распределения необходимо предварительно сформировать);

– вычисление математического ожидания, дисперсии, коэффициентов асимметрии и эксцесса;

– моделирование случайной величины.

**Алгоритм решения задачи**

**Основное распределение**

Одним из основных величин, необходимых для расчётов в работе является плотность распределения в точке. Она может быть посчитана по исходной формуле. Однако необходимо учитывать сдвиг-масштабные преобразования. Для этого необходимо в алгоритм вычисления встроить следующую формулу:

.

Также необходимо производить моделирование случайной величины, распределенной по нашему закону распределения. Для этого воспользуемся данным алгоритмом:

Шаг 0. Вычислить константы ,  (один раз для каждого значения ).

Шаг 1. Получить методом обратной функции реализацию  лапласовской с масштабом  случайной величины. Для этого получить реализацию  случайной величины, равномерно распределенной на интервале (0, 1), и вычислить

.

Шаг 2. Получить реализацию  случайной величины, равномерно распределенной на интервале (0, 1).

Шаг 3. Если , то  – реализация целевой случайной величины, иначе перейти на шаг 1.

Для моделирования случайной величины, имеющей распределение с параметром формы , можно использовать следующий алгоритм, реализующий метод исключения.

Шаг 0. Вычислить константы , ,  (один раз для каждого значения ).

Шаг 1. Получить реализацию  нормальной с масштабом *b* случайной величины.Для этого получить реализации ,  случайной величины, равномерно распределенной на интервале (0, 1), и вычислить  по одной из двух формул

, .

Шаг 2. Получить реализацию  случайной величины, равномерно распределенной на интервале (0, 1).

Шаг 3. Если , то  – реализация целевой случайной величины, иначе перейти на шаг 1.

где -реализация случайной величины равномерно распределенной на интервале (0;1).

Как и в случае с плотностью случайную величину также необходимо пронормировать по формуле:

.

Исходя из того, что распределения из списка вариантов являются симметричными унимодальными, а стандартные распределения при этом имеют нулевые математическое ожидание и коэффициент асимметрии. По определению, математическое ожидание распределения в сдвиг-масштабном преобразовании с заданной плотностью, если оно существует, равно математическому ожиданию стандартного распределения, увеличенному на величину . А значит, мы можем говорить о том, что всякое математического ожидание для стандартного обобщенного распределения Гаусса будет равно величине . Также по определению коэффициент асимметрии при сдвиг-масштабном преобразовании не изменяется, а значит мы всегда можем возвращать ноль в случае стандартного обобщенного распределения Гаусса. Дисперсия стандартного распределения определяется формулой , где  – гамма-функция, а дисперсия при сдвиг-масштабном преобразовании равна дисперсии стандартного распределения, увеличенной в  раз. Коэффициент эксцесса определяется формулой , и при сдвиг-масштабном преобразовании не изменяется.

**Смесь распределений**

Функция плотности для смеси распределений имеет следующий вид:



Где p – параметр смеси.

Моделирование случайной величины происходить по следующему алгоритму:

Шаг 1. Получить реализацию  случайной величины, имеющей равномерное распределение на интервале (0, 1). Если , перейти на шаг 2, иначе перейти на шаг 3.

Шаг 2. Получить реализацию  случайной величины с плотностью . Число  и будет реализацией целевой случайной величины.

Шаг 3. Получить реализацию  случайной величины с плотностью . Число  и будет реализацией целевой случайной величины.

Математическое ожидание вычисляется по формуле:



Дисперсия вычисляется по формуле:



Коэффициента асимметрии вычисляется по формуле:

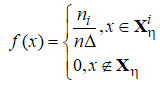


Коэффициент эксцесса:



**Эмпирическое распределение**

Эмпирической плотностью распределения называют функцию

.

Плотность формируется на основе выборки, следовательно, необходимо сгенерировать выбору случайной величины, распределенной по непрерывному закону. Затем сортируем выборку получившуюся выборку.

Далее необходимо определить число промежутков *k*, которое мы высчитываем по формуле Стёрджесса, округляя в большую сторону,

.

На основании предыдущих действий рассчитывается значение длины каждого промежутка:

.

После мы создаем массив, хранящий границы интервалов . Так как для реализации функции плотности эмпирического распределения необходимо не только длина каждого промежутка и количество элементов выборки, но и количество точек в интервале. Для получения этой информации мы проходимся по массиву границ и проверяем к какому интервалу принадлежит наш *х*. Затем ищем границы этого интервала и считаем в нем количество элементов выборки.

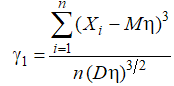
Математическое ожидание для эмпирического распределения:

D:\Study\Лабораторные работы\Прога\5 семестр\ЛР№1\эмп. мат ожидание.png,

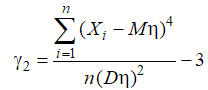
Дисперсия для эмпирического распределения:

,

Коэффициент асимметрии для эмпирического распределения:

,

Коэффициент эксцесса для эмпирического распределения:

.

**Описание программы**

В модуле **Main.cpp** реализован пользовательский диалог в виде меню, в котором можно выбрать моделирование интересующего распределения и по желанию ввести его начальные параметры. Формирование эмпирического распределения происходит вместе с теоретическим. Также в нем предусмотрены проверки входных данных и вывод данных в файл для дальнейшего изображения графиков.

Модуль **standart\_distribution.cpp** содержит функции, моделирующие характеристики основного распределения.

Функции, присутствующие в модуле **standart\_distribution.cpp**:

Standart () – нормирует случайную величину с учетом сдвиг-масштабных преобразований;

Random\_item() – реализация равномерно распределенной случайной величины;

Random\_item12()- реализация случайной величины на промежутке v от 1 до 2

Random\_item2() - реализация случайной величины на промежутке v от 2 до бесконечности

Randomizer() – генерация случайного значения распределеного по закону Лапласа

Density\_calc () – вычисления плотности распределения в точке, для стандартного распределения;

Modified\_Density\_calc() - вычисления плотности распределения в точке, для сдвиг - масштабного распределения

Expected\_value() – вычисление мат ожидания для стандартного распределения

Modified\_Expected\_value\_calc() - вычисление мат ожидания для свдиг – масштабного распределения

Dispersion\_calc() – вычисление дисперсии для стандартного распределения

Modified\_Dispersion\_calc() - вычисление дисперсии для сдвиг - масштабного распределения

Modified\_Func\_calc() – Функция, упаковывающая все параметры распределения в один вектор

Asymmetry\_calc()– вычисление коэффициента асимметрии;

Excess\_calc()– вычисление коэффициента эксцесса;

Create\_std\_set()– формирование выборки случайных величин, распределенных по закону Лапласа;

Create\_std\_graph()– формирование таблицы, значений принимаемых случайными величинами, содержащимися в выборке.

Модуль **Mixed Function distribution.cpp** содержит функции, моделирующие характеристики смеси распределений.

Функции, присутствующие в модуле **Mixed Function distribution.cpp**:

Mixed\_Func\_calc() – вычисляет все характеристики смеси распределений, такие как: плотность, мат ожидание, дисперсию, асимметрию, эксцесс.

Mixed\_Random\_value()– моделирование случайной величины;

Create\_mixed\_set () – формирование выборки случайных величин, распределенных по закону смеси распределений;

Generate\_mixed\_graph()- формирование таблицы, значений принимаемых случайными величинами, содержащимися в выборке.

Модуль **empirical \_distribution.cpp** содержит функции, формирующие и моделирующие характеристики эмпирического распределения.

Функции, присутствующие в модуле **empirical \_distribution.cpp**:

H\_calc () – вычисление длины промежутка;

Intervals\_creation () – формирование интервалов;

Get\_interval\_index () – нахождение индекса необходимого интервала;

Empiric\_Density\_calc () – вычисление плотности эмпирического распределения в точке;

Empiric\_Expected\_value\_calc () – вычисление математического ожидания для эмпирического распределения;

Empiric\_Dispersion\_calc () - вычисление дисперсии для эмпирического распределения;

Empiric\_Asymmetry\_calc () - вычисление коэффициента асимметрии для эмпирического распределения;

Empiric\_Excess\_calc () – вычисление коэффициента эксцесса для эмпирического распределения;

Generate\_empric\_graph () - формирование таблицы, значений принимаемых случайными величинами, содержащимися в выборке.

**Код программы на языке c++**

**Модуль main.cpp**

#include "Header.h"

#include <fstream>

int main() {

setlocale(LC\_ALL, "ru");

srand((unsigned)time(0));

double v, l1, l2, u1, u2, p, v2;

std::cout << "Введите: " << std::endl << "1, для стандартного распределения" << std::endl << "2, для смеси распределений" << std::endl << "> ";

int flag;

std::cin >> flag;

switch (flag) {

case 1:

{

std::cout << "Введите: v (v > 0), l, u> ";

std::cin >> v >> l1>> u1;

while (std::cin.fail()) { // проверка вводимых данных

std::cin.clear();

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

std::cout << "Ошибка ввода. Введите значения повторно > ";

std::cin >> v >> l1>> u1;

}

while (v <= 0) {

std::cout << "Ошибка ввода v. Введите значение повтороно > ";

std::cin >> v;

}

auto vec = Create\_std\_set(1000, v,u1,l1);

auto func = Create\_std\_graph(vec, v, u1, l1, 1000);

std::ofstream file1("standart\_distr.txt");

for (int i = 0; i < func.size(); ++i) {

file1 << func[i].first << "\t" << func[i].second << std::endl;

}

auto func2 = Generate\_empric\_graph(vec, 1000);

std::ofstream file2("empiric\_distr.txt");

for (int i = 0; i < func2.size(); ++i) {

file2 << func2[i].first << "\t" << func2[i].second << std::endl;

}

// standart function print

std::cout << "Для стандартного распределения ";

std::cout << "Дисперсия равна: " << Modified\_Dispersion\_calc(v,l1) << std::endl << "Эксцесс: " << Excess\_calc(v) << std::endl

<< "Плотность распределения в точке " << u1 << " равна: " << Modified\_Density\_calc(u1, v, u1, l1) << std::endl << "Мат. Ожидание: "

<<Modified\_Expected\_value\_calc(u1) << std::endl << "Асимметрия: " <<Asymmetry\_calc() << std::endl;

std::cout << "Для эмпирического распределения "<<std::endl;

std::cout << "Мат. Ожидание: " << Empiric\_Expected\_value\_calc(vec) << std::endl << "Дисперсия: "

<< Empiric\_Dispersion\_calc(vec) << std::endl << "Эксцесс: " << Empiric\_Excess\_calc(vec) << std::endl

<< "Плотность распределения в точке "<< u1<<" равна: " << Empiric\_Density\_calc(vec, u1) << std::endl << "Асимметрия : "

<< Empiric\_Asymmetry\_calc(vec) << std::endl;

break;

}

case 2:

{

std::cout << "Введите v (v > 0), p (0 < p < 1), v2 (v2 > 0), l1, l2, u1, u2 > ";

std::cin >> v >> p >> v2 >> l1 >> l2 >> u1 >> u2;

while (std::cin.fail()) { // проверка вводимых данных

std::cin.clear();

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

std::cout << "Ошибка ввода. Введите значения повторно > ";

std::cin >> p >> v2 >> l1 >> l2 >> u1 >> u2;

}

while (v <= 0) {

std::cout << "Ошибка ввода v. Введите значение повтороно > ";

std::cin >> v;

}

while (v2 <= 0) {

std::cout << "Ошибка в v2. Введите значение больше нуля > ";

std::cin >> v2;

}

while ((p >= 1) || (p <= 0)) {

std::cout << "Ошибка в p. Введите значение повтороно > ";

std::cin >> p;

}

auto mixed\_vec = Create\_mixed\_set(v, v2, 1000, p, u1, l1, u2, l2);

auto func3 = Generate\_mixed\_graph(mixed\_vec, p, v, u1, l1, v2, u2, l2, 1000);

std::ofstream file3("mixed\_distr.txt");

for (int i = 0; i < func3.size(); ++i) {

file3 << func3[i].first << "\t" << func3[i].second << std::endl;

}

auto func4 = Generate\_empric\_graph(mixed\_vec, 1000);

std::ofstream file4("empiric\_mixed\_distr.txt");

for (int i = 0; i < func4.size(); ++i) {

file4 << func4[i].first << "\t" << func4[i].second << std::endl;

}

std::cout << "Для смеси распределений ";

std::vector<double>vec1 = Mixed\_Func\_calc(p, 0, v, u1, l1, v2, u2, l2);

std::cout << std::endl << "Параметры смеси распределений : " << std::endl;

std::cout << "Дисперсия: " << vec1[0] << std::endl << "Эксцесс: " << vec1[1] << std::endl << "Плотность в точке 0: " << vec1[2] << std::endl << "Мат. ожидание " << vec1[3] << std::endl << "Асимметрия: " << vec1[4] << std::endl;

std::cout << "Для эмпирического распределения ";

std::cout << "Дисперсия равна " << Empiric\_Dispersion\_calc(mixed\_vec) << std::endl << "Эксцесс: " << Empiric\_Excess\_calc(mixed\_vec) << std::endl

<< "Плотность распределения в точке 0 равна: " << Empiric\_Density\_calc(mixed\_vec, 0) << std::endl << "Мат. Ожидание: "

<< Empiric\_Expected\_value\_calc(mixed\_vec) << std::endl << "Асимметрия: " << Empiric\_Asymmetry\_calc(mixed\_vec) << std::endl;

break;

}

default:

std::cout << "bad input" << std::endl;

break;

}

return 0;

}

**Модуль Header.h**

#pragma once

#define \_USE\_MATH\_DEFINES

#include <iostream>

#include <math.h>

#include <vector>

#include <algorithm>

#include <time.h>

double Density\_calc(double x, double v);

double Modified\_Density\_calc(double x, double v, double u, double l);

double Expected\_value();

double Modified\_Expected\_value\_calc(double u);

double Dispersion\_calc(double v);

double Modified\_Dispersion\_calc(double v, double l);

double Asymmetry\_calc();

double Excess\_calc(double v);

std::vector<double> Modified\_Func\_calc(double x, double v, double u, double l);

std::vector<double> Mixed\_Func\_calc(double p, double x, double v1, double u1, double l1, double v2, double u2, double l2);

double Randomizer();

double Random\_item12(double v);

double Random\_item2(double v);

double Random\_item(double v, double u, double l);

double Mixed\_Random\_value(double p, double v1, double v2, double u1, double l1, double u2, double l2);

std::vector<double>Create\_mixed\_set(double v1, double v2, double n, double p, double u1, double l1, double u2, double l2);

std::vector<std::pair<double, double>> Generate\_mixed\_graph(std::vector<double> vec, double p, double v1, double u1, double l1, double v2, double u2, double l2, const int n);

double H\_calc(double N, double min, double max);

std::vector<double> Intervals\_creation(double min, double max, double h);

int Get\_interval\_index(const std::vector<double> intervals, const double x);

double Empiric\_Expected\_value\_calc(std::vector<double> vec);

double Empiric\_Dispersion\_calc(std::vector<double> vec);

double Empiric\_Asymmetry\_calc(std::vector<double> vec);

double Empiric\_Excess\_calc(std::vector<double> vec);

double Empiric\_Density\_calc(std::vector<double> vec, double x);

std::vector<std::pair<double, double>> Generate\_empric\_graph(std::vector<double> vec, const int n);

std::vector<double> Create\_std\_set(const int n, double v, double u, double l);

std::vector<std::pair<double, double>> Create\_std\_graph(std::vector<double> vec, double v, double u, double l, const int n);

double Standart(double x, double l, double u);

**Модуль Standart and modified distribution.cpp**

#include "Header.h"

double Density\_calc(double x, double v) {

double density = (v / (2 \* tgamma(1 / v)) \* exp(-pow(abs(x), v)));

/\*double density = (v / (2 \* tgamma(1 / v)) \* exp(exp(v \* (-log(abs(x))))));\*/

return density;

}

double Standart(double x, double l, double u) {

double xmod = ((x - u) / l);

return xmod;

}

double Modified\_Density\_calc(double x, double v, double u, double l) {

double xmod = Standart(x, l, u);

double mod\_density = (v / (2 \* tgamma(1 / v)) \* exp(-pow(abs(xmod), v))) / l;

//double mod\_density = (v / (2 \* tgamma(1 / v)) \* exp(exp(v\*(-log(abs(x)))))) / l;

return mod\_density;

}

double Expected\_value() { return 0; }

double Modified\_Expected\_value\_calc(double u) {

double mod\_expected\_value = Expected\_value() + u;

return mod\_expected\_value;

}

double Dispersion\_calc(double v) {

double dispersion = (tgamma(3 / v) / tgamma(1 / v));

return dispersion;

}

double Modified\_Dispersion\_calc(double v, double l) {

double mod\_dispersion = (tgamma(3 / v) / tgamma(1 / v)) \* pow(l, 2);

return mod\_dispersion;

}

double Excess\_calc(double v) {

double excess = ((tgamma(5 / v) \* tgamma(1 / v) / pow(tgamma(3 / v), 2)) - 3);

return excess;

}

double Asymmetry\_calc() { return 0; }

std::vector<double> Modified\_Func\_calc(double x, double v, double u, double l) { //подсчет всех параметров для модифицированной функции

std::vector<double> vec = { Modified\_Dispersion\_calc(v, l), Excess\_calc(v), Modified\_Density\_calc(x,v,u,l), Modified\_Expected\_value\_calc(v) };

return vec;

}

double Randomizer() {

double r;

do r = (double)rand() / RAND\_MAX; while (r == 0 || r == 1);

return r;

}

// part of an algorithm for v in range [1;2)

double Random\_item12(double v) {

double r = Randomizer();

double a = (1 / v) - 1;

//double b = 1 / (exp((1 / v) \* log(v)));

double b = 1 / (pow(v, 1/v));

double x = 0;

if (r <= 0.5) {

x = b \* log(2 \* r);

}

else if (r > 0.5) {

x = -b \* log(2 \* (1 - r));

}

double r2 = Randomizer();

if (log(r2) <= (exp(v \* (-log(abs(x)))) + (abs(x) / b) + a)) { return x; }

//if (log(r2) <= (-pow(abs(x), v) + (abs(x) / b) + a)) { return x; }

else {return Random\_item12(v); }

}

// part of an algorithm for v in range [2; inf)

double Random\_item2(double v) {

double a = (1 / v) - 0.5;

//double b = 1 / (exp((1 / v) \* log(v)));

double b = 1 / (pow(v, 1 / v));

double c = 2\* pow(b,2);

double r = Randomizer();

double r2 = Randomizer();

double x = b \* sqrt(-2 \* log(r))\* cos(2\*M\_PI\*r2);

double r3 = Randomizer();

if (log(r3) <=( exp(v \* ( - log(abs(x)))) + (pow(x, 2) / c) + a)) { return x; }

/\*if (log(r3) <= (-pow(abs(x), v) + (pow(x, 2) / c) + a)) { return x; }\*/

else {return Random\_item2(v); }

}

// function for the whole algorithm use

double Random\_item(double v, double u, double l) {

double result;

if ((1 <= v)&&(v<2)) {result= Random\_item12(v); }

else if (v >= 2) {result= Random\_item2(v); }

return ((result \*l +u));

}

std::vector<double> Create\_std\_set(const int n, double v, double u, double l) {

std::vector<double> result;

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

result.push\_back(Random\_item(v, u, l));

}

sort(result.begin(), result.end());

return result;

}

std::vector<std::pair<double, double>> Create\_std\_graph(std::vector<double> vec, double v, double u, double l,const int n) {

std::vector<std::pair<double, double>> result;

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

result.push\_back(std::make\_pair(vec[i], Modified\_Density\_calc(vec[i], v, u, l)));

}

return result;

}

**Модуль Mixed Function distribution.cpp**

#include "Header.h"

std::vector<double> Mixed\_Func\_calc(double p, double x, double v1, double u1, double l1, double v2, double u2, double l2) {

std::vector<double>results1 = Modified\_Func\_calc(x, v1, u1, l1);

std::vector<double>results2 = Modified\_Func\_calc(x, v2, u2, l2);

double mixed\_density = ((1 - p) \* results1[2]) + (p \* results2[2]);

double mixed\_expected\_value = ((1 - p) \* results1[3]) + (p \* results2[3]); /\*((1 - p) \* Modified\_Density\_calc(x, v1, u1, l1) + p \* Modified\_Density\_calc(x, v2, u2, l2));\*/

double mixed\_dispersion = (1 - p) \* (pow(results1[3], 2) + results1[0]) - pow(mixed\_expected\_value, 2) + (p \* (pow(results2[3], 2) + results2[0]) - pow(mixed\_expected\_value, 2));

double asymmetry = (1 / (pow(mixed\_dispersion, 3 / 2))) \* ((1 - p) \* (pow(results1[3] - mixed\_expected\_value, 3) + 3 \* (results1[3] - mixed\_expected\_value) \* results1[0] + pow(results1[0], 3 / 2) \* Asymmetry\_calc()) + p \* (pow(results2[3] - mixed\_expected\_value, 3) + 3 \* (results2[3] - mixed\_expected\_value) \* results2[0] + pow(results2[0], 3 / 2) \* Asymmetry\_calc()));

double excess = (1 / pow(mixed\_dispersion, 3)) \* ((1 - p) \* (pow(results1[3] - mixed\_expected\_value, 4) + 6 \* pow(results1[3] - mixed\_expected\_value, 2) \* results1[0] + 4 \* (results1[3] - mixed\_expected\_value) \* pow(results1[0], 3 / 2) \* Asymmetry\_calc() + pow(results1[0], 2) \* Excess\_calc(v1)) - 3 + p \* ((pow(results2[3] - mixed\_expected\_value, 4) + 6 \* pow(results2[3] - mixed\_expected\_value, 2) \* results2[0] + 4 \* (results2[3] - mixed\_expected\_value) \* pow(results2[0], 3 / 2) \* Asymmetry\_calc() + pow(results2[0], 2) \* Excess\_calc(v2))) - 3);

std::vector<double> vec = { mixed\_dispersion, excess, mixed\_density, mixed\_expected\_value, asymmetry };

return vec;

}

double Mixed\_Random\_value(double p, double v1, double v2, double u1, double l1, double u2, double l2) {

double r = Randomizer();

if (r > p) {

return Random\_item(v1, u1, l1);

}

else { return Random\_item(v2, u2, l2); }

}

std::vector<double>Create\_mixed\_set(double v1, double v2, double n,double p, double u1, double l1, double u2, double l2) {

std::vector<double>vec;

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

vec.push\_back(Mixed\_Random\_value(p,v1,v2,u1,l1,u2,l2));

}

sort(vec.begin(), vec.end());

return vec;

}

std::vector<std::pair<double, double>> Generate\_mixed\_graph(std::vector<double> vec, double p, double v1, double u1, double l1, double v2, double u2, double l2, const int n) {

std::vector<std::pair<double, double>> result;

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

auto mixed\_density = Mixed\_Func\_calc(p, vec[i], v1, u1, l1, v2, u2, l2)[2];

result.push\_back(std::make\_pair(vec[i],mixed\_density));

}

return result;

}

**Модуль Empirical distribution.cpp**

#include "Header.h"

double H\_calc(double N, double min, double max) {

return ((max - min) / ((int)(1 + log2(N))));

}

std::vector<double> Intervals\_creation(double min, double max, double h ) {

std::vector<double> vec;

double buffer = min;

vec.push\_back(buffer);

while (buffer < max) {

vec.push\_back(buffer+h);

buffer += h;

}

return vec;

}

int Get\_interval\_index(const std::vector<double> intervals, const double x) {// всегда возвращает левую границу интервала

if (intervals.size() < 2) { // always returns the left side of the interval

return 0;

}

if (x >= intervals[intervals.size() - 2] && x <= intervals[intervals.size() - 1]) {

return intervals.size() - 2;

}

for (int i = 0; i < intervals.size() - 1; ++i) {

if (x >= intervals[i] && x < intervals[i + 1]) {

return i;

}

}

}

double Empiric\_Expected\_value\_calc(std::vector<double> vec) {

double summ = 0;

for (int i = 0; i < vec.size(); i++) {

summ += vec[i];

}

return summ / vec.size();

}

double Empiric\_Dispersion\_calc(std::vector<double> vec) {

double summ = 0;

double M = Empiric\_Expected\_value\_calc(vec);

for (int i = 0; i < vec.size(); i++) {

summ += pow((vec[i] - M),2);

}

return summ / vec.size();

}

double Empiric\_Asymmetry\_calc(std::vector<double> vec) {

double summ = 0;

double M = Empiric\_Expected\_value\_calc(vec);

double D = Empiric\_Dispersion\_calc(vec);

for (int i = 0; i < vec.size(); i++) {

summ += pow((vec[i] - M), 3);

}

return (summ / (vec.size()\* pow(D, 3/2)));

}

double Empiric\_Excess\_calc(std::vector<double> vec) {

double summ = 0;

double M = Empiric\_Expected\_value\_calc(vec);

double D = Empiric\_Dispersion\_calc(vec);

for (int i = 0; i < vec.size(); i++) {

summ += pow((vec[i] - M), 4);

}

return ((summ / (vec.size() \* pow(D, 2))) - 3);

}

double Empiric\_Density\_calc(std::vector<double> vec, double x) {

double h = H\_calc(vec.size(),vec[0],vec[vec.size()-1]);

auto intervals = Intervals\_creation(vec[0], vec[vec.size() - 1], h);

int index = Get\_interval\_index(intervals, x);

int left = 0;

int right = vec.size() - 1;

if (index == 0) { // еслм х лежит в первом интервале

right = 0;

while (vec[right] < intervals[1]) {

++right;

}

} // в результате в right получаем правую границу первого интервала (тоетсь длину интервала - 1)

else if (index == intervals.size() - 2) { // если х лежит в последнем интервале

left = vec.size() - 1;

while (vec[left] > intervals[intervals.size() - 2]) {

--left;

}

}// в результате в left получаем левую границу последнего интервала (тоетсь длину интервала - 1)

else {

while (vec[left] < intervals[index]) { //в left значение левой границы интервала в котором х

++left;

}

right = left;

while (vec[right] < intervals[index + 1]) {//в right значение левой границы интервала следующего после того, в котором х

++right; // и соответственно правой границы х интервала

}

}

return (right - left + 1) / (vec.size() \* h); // делим длину интервала в котором лежит х на всю длину прямой

}

std::vector<std::pair<double, double>> Generate\_empric\_graph(std::vector<double> vec, const int n) {

std::vector<std::pair<double, double>> result;

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

result.push\_back(std::make\_pair(vec[i], Empiric\_Density\_calc(vec, vec[i])));

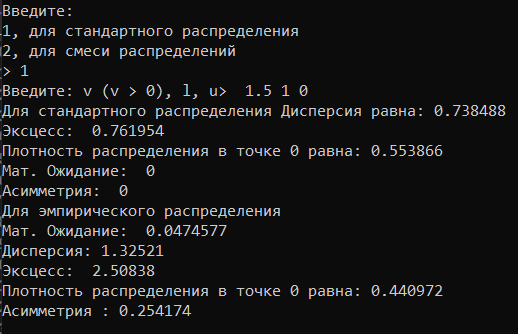
}

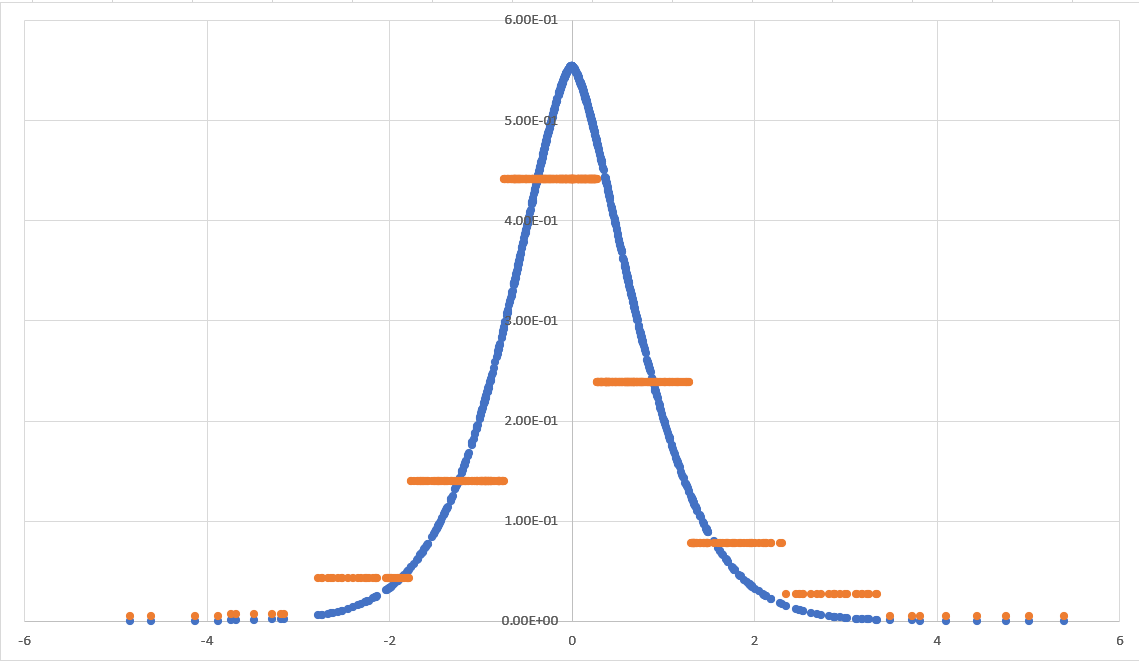
return result;

}

**Результаты работы программы**

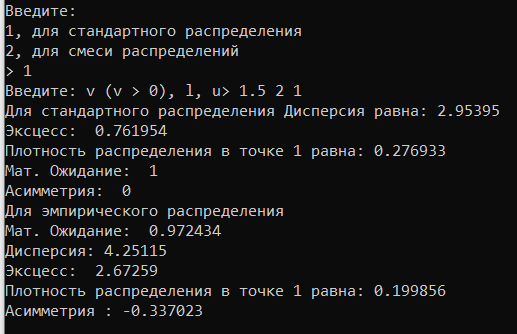
1. Тест для стандартного распределения (выборка из 1000 значений):

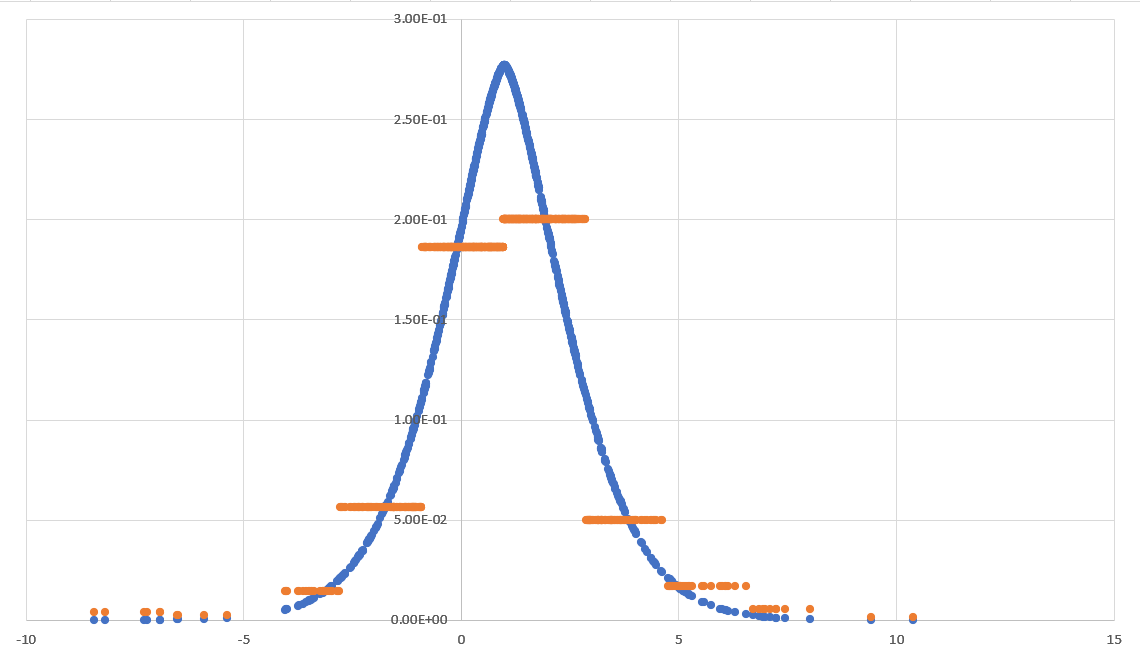




Исходя из данного теста можно сделать вывод о правильности вычислений и о примерных соответствиях теоретических и эмпирических характеристиках распределения с начальными параметрами по умолчанию.

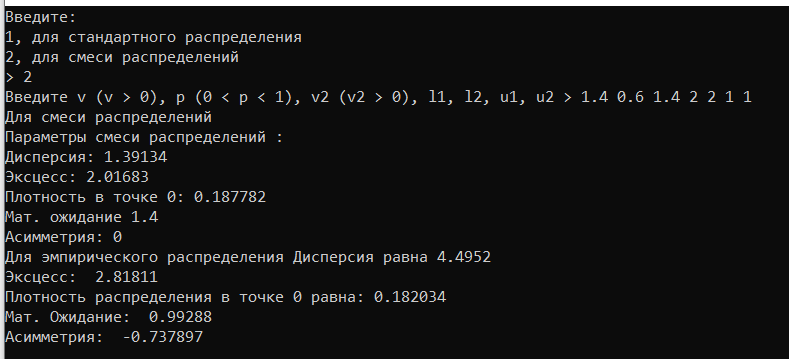
2) Тест для сдвиг масштабных преобразований (выборка из 1000 значений):

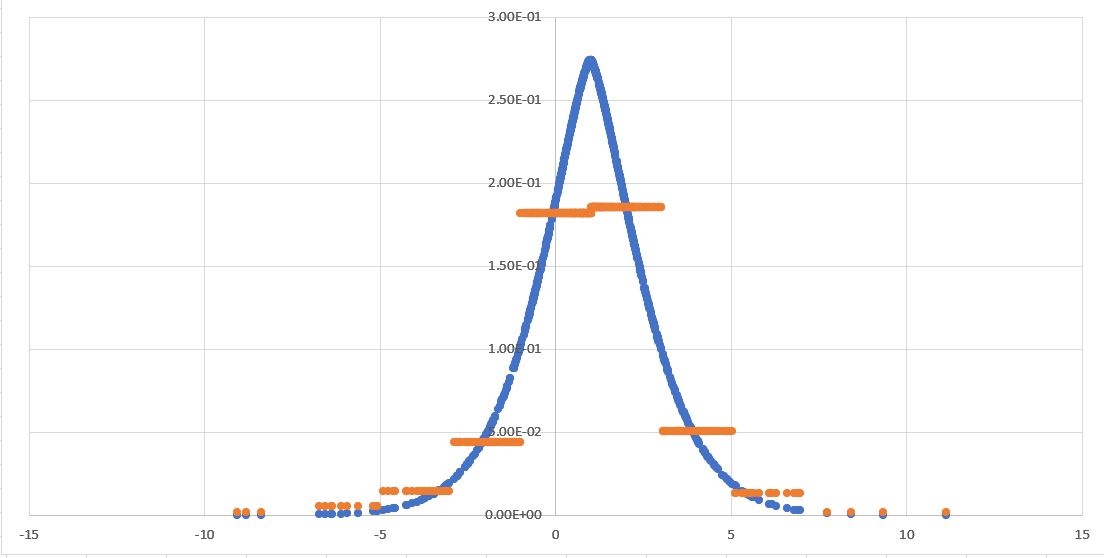




Как видно, результаты вычислений теоретических и эмпирических характеристик близки. Также подтверждается то, что при увеличении n распределение стремится к нормальному с неограниченным ростом дисперсии.

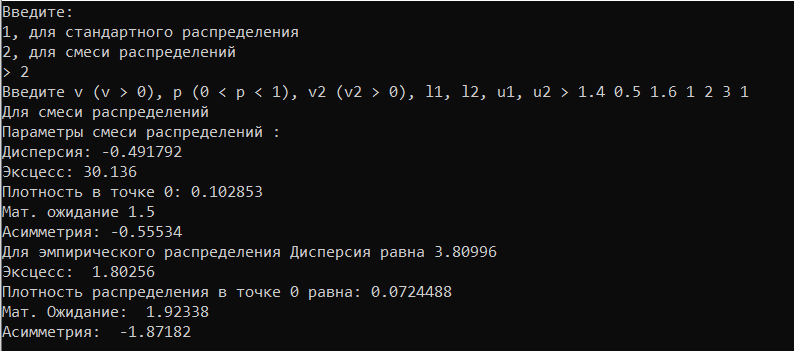
3) Тест для тривиального случая смеси распределений (выборка из 1000 значений):

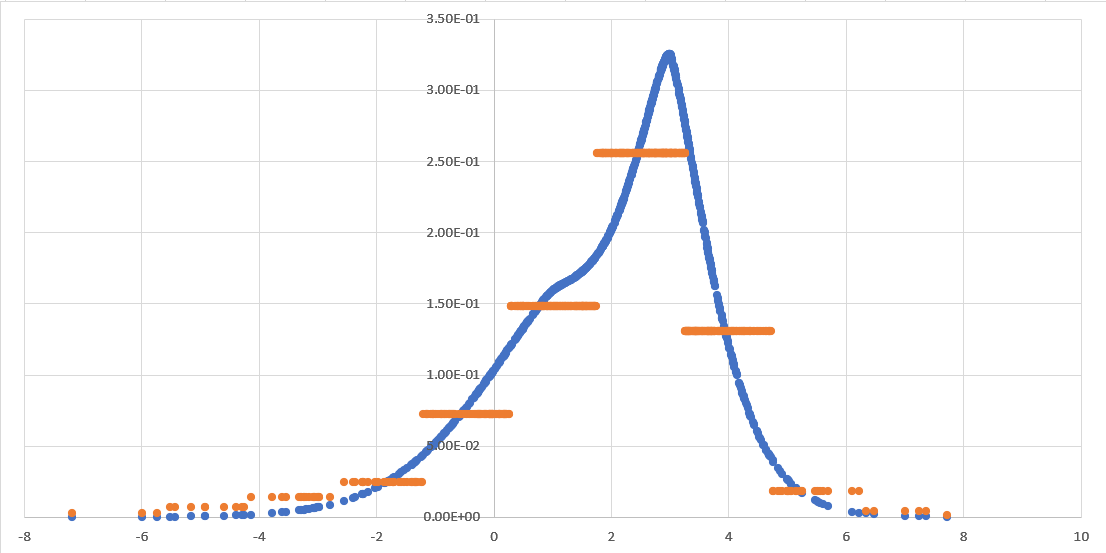




Данный тест также показывает верность вычислений и соответствие теоретических и эмпирических характеристик.

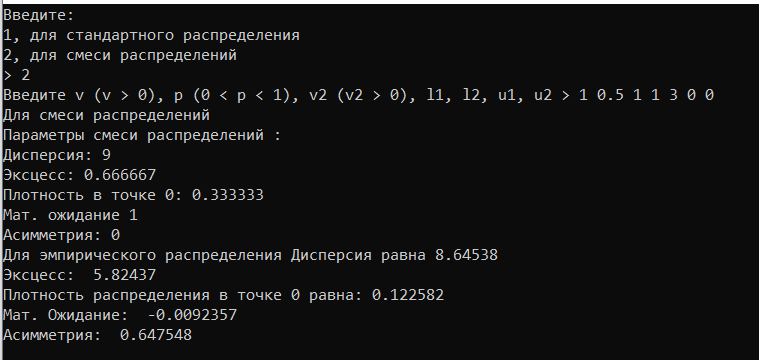
4) Простой тест для математического ожидания смеси распределений (выборка из 1000 значений):

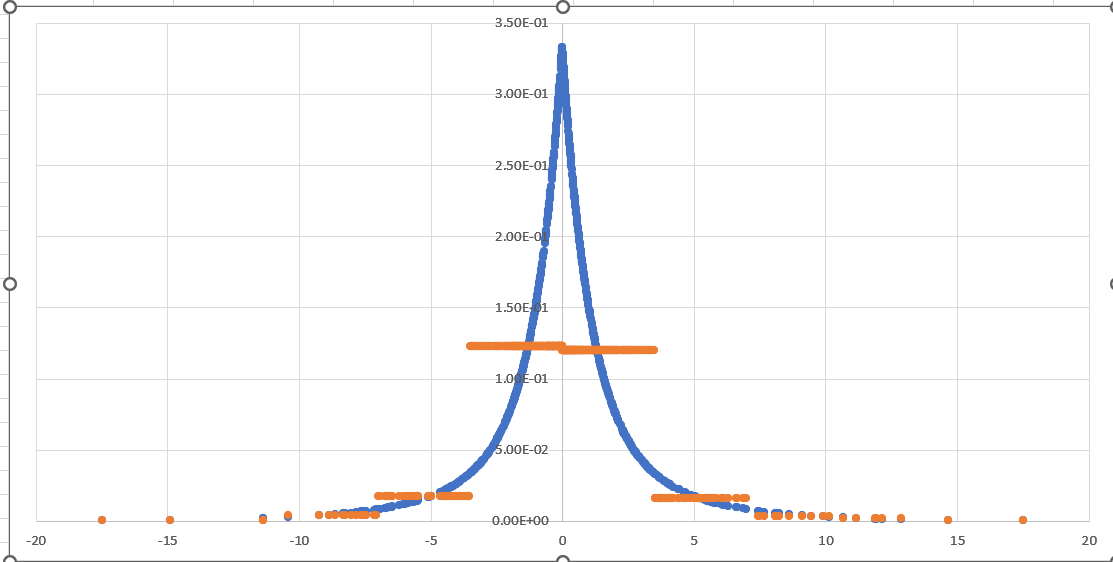




Вычисления верны, а теоретические и эмпирические характеристики соответствуют.

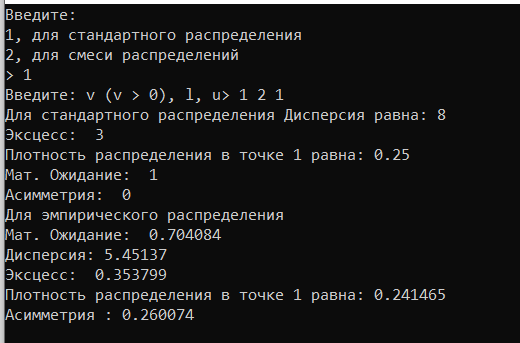
5) Простой тест для дисперсии смеси распределений (выборка из 1000 значений):

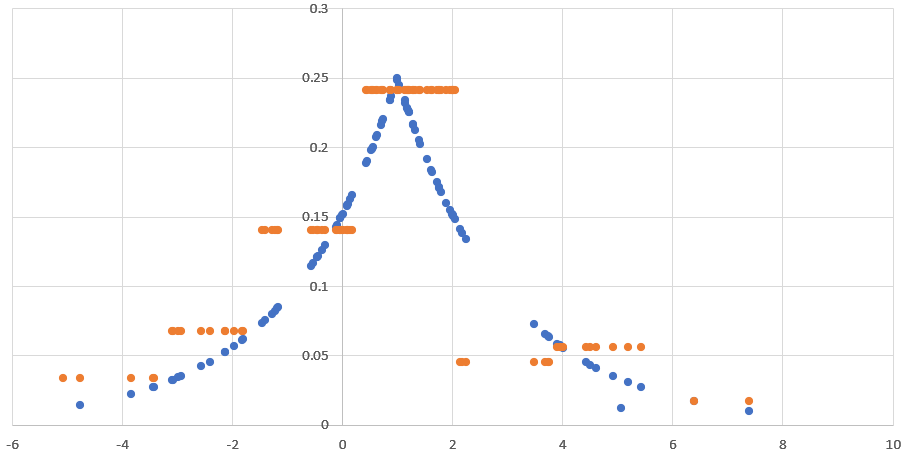




Вычисления верны, а теоретические и эмпирические характеристики соответствуют.

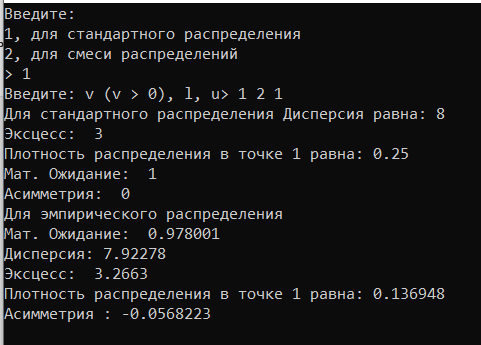
6) Сравнительно небольшая выборка (выборка из 100 значений):

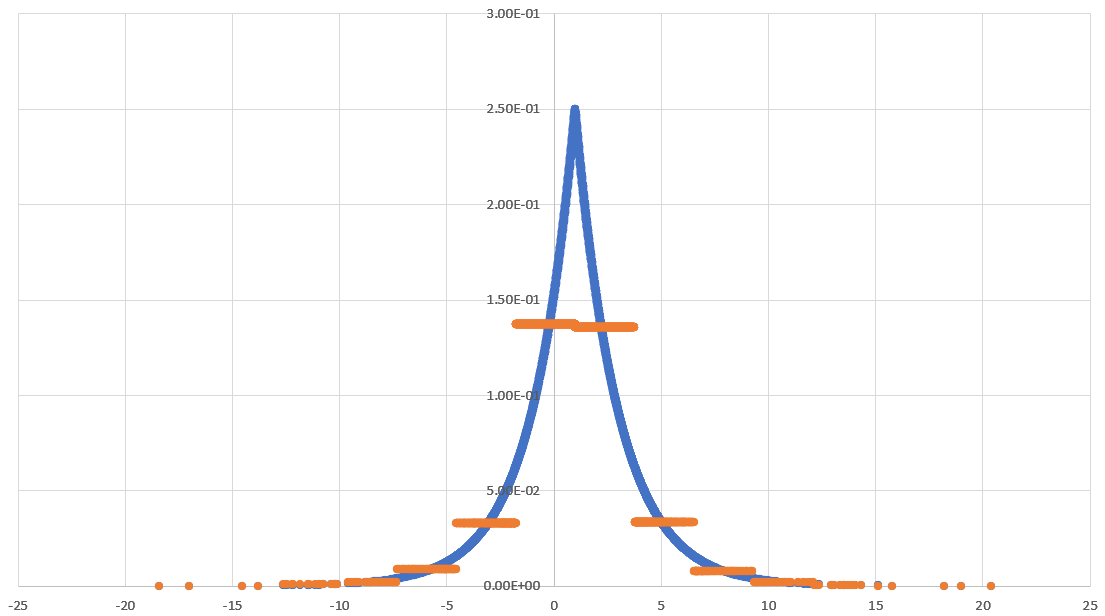




Отдаленно прослеживается соответствие эмпирических и теоретических характеристик.

7) Большая выборка (выборка из 100000 значений):





Между эмпирическими и теоретическими характеристиками прослеживается тесное соответствие, граничащее с равенством до сотых.

**Выводы по работе**

В ходе работы были изучены методы статистического моделирования, реализованы алгоритмы, осуществляющие данные методы.