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 №4 «ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ»

по предмету

«**ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ»**

Факультет: прикладной математики и информатики

Группа: ПМИ-12

Бригада: 7

Студенты: Курдюков И.Н.

Оменльницкая Е.И.

Преподаватели: Дворецкая В.К.

Новосибирск, 2023

**Цель работы**

Изучить реализацию на языке С++ отношений между классами и понятие интерфейса.

**Постановка задачи**

1. Ознакомиться с понятиями, связанными с отношениями между

классами (в том числе с механизмом позднего связывания), и понятием

интерфейса.

2. Модифицировать программу, разработанную в лабораторной работе

No3, так чтобы в ней были определены интерфейс распределения и интерфейс

персистентного объекта, а класс смеси был шаблоном.

2.1. Определить в интерфейсе распределения следующие чисто

виртуальные функции:

– функция для вычисления значений плотности распределения по

заданному аргументу;

– функция и/или функции для вычисления математического ожидания,

дисперсии, коэффициентов асимметрии и эксцесса;

– функция для моделирования случайной величины.

2.2. Определить в интерфейсе персистентного объекта чисто

виртуальные функции, обеспечивающие персистентность.

2.3. Классы, разработанные в работах No 2, 3, должны быть

реализациями интерфейсов, кроме того класс смеси необходимо сделать

шаблоном, параметризованным типами каждого из агрегированных объектов-

распределений (отношение инстанцирования).

2.4. В классе эмпирического распределения заменить все конструкторы,

получающие ссылки на объекты разных классов с целью инициализации

массива данных, единым конструктором, получающим ссылку на интерфейс

распределения (отношение зависимости класса от интерфейса).

2.5. Объекты всегда должны находиться в корректном состоянии. При

возникновении ошибок необходимо генерировать исключительные ситуации, а

их обработку осуществлять вне функций-членов классов. Диалог с

пользователем (если он предполагается) следует вести вне функций-членов

классов.

3. Разработать функцию (функции), тестирующую поведение классов, в

том числе механизм позднего связывания.

4. Защитить лабораторную работу, ответив на вопросы преподавателя., где ,  – гамма-функция.

**Алгоритм решения задачи**

Поскольку все задание было выполнено в фомате классов в лабораторной работе номер 3, то в данной работае нам нужно было ее модифицровать реализовав шаблон и интерфейсный класс.

**Работа с основным распределением**

Плотность стандартного распределения мы можем посчитать с помощью данной стандартной формулы. Однако мы получим неправильные результаты, если не учтем сдвиг-масштабные преобразования.



По данной формуле мы сможем посчитать плотность с учетом сдвига и масштаба.

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.

**Описание программы**

**Модуль Distributions.h**

Класс IDistribution – интерфейсный класс содержащий виртуальные методы.

Public часть:

double virtual density(const double x) const = 0; - расчет плотности

double virtual expected\_value() const = 0; - расчет мат.ожидания

double virtual dispersion() const = 0;- расчет дисперсии

double virtual excess() const = 0; - расчет эксцесса

double virtual asymmetry() const = 0;- расчет асимметрии

double virtual rand\_var() const = 0; - реализация случайной величины

std::vector<double> virtual generate\_selection(const int n) const = 0; - генерация выборки

std::vector<std::pair<double, double>> virtual generate\_graph\_selection(const std::vector<double>& selection, const int n) const = 0; - генерация пары вида <случайная величина, плотность в точке это величины>.

Класс IPresistend – Интерфейсный класс содержащий виртуальные методы для обеспечения персистентности объектов.

Public часть:

void virtual load\_from\_file(std::ifstream& file) = 0; - виртуальная функция для загрузки данных из файла

void virtual save\_in\_file(std::ofstream& file) = 0; - виртуальная функция для сохранения данных в файл.

**Модуль Standart\_distr.cpp**

Содержит в себе описание всех методов класса Std\_distr.

**Функции(методы), работающие с атрибутами**

get\_form() – возвращает параметр формы.

get\_shift() – возвращает параметр сдвига.

get\_ scale() – возвращает параметр масштаба.

set\_form() – задать новый параметр формы.

set\_shift() – задать новый параметр сдвига.

set\_scale() – задать новый параметр масштаба.

**Закрытые функции(методы) класса**

Standart(double x) – Нормализует х в соответствии со сдвигом и масштаом;

Randomizer() – генерирует случайное число в заданных пределах;

Random\_item12() – часть алгоритма реализации случайной величины для промежутка параметра формы [1;2);

Random\_item2() – часть алгоритма реализации случайной величины для промежутка параметра формы [2; inf);

Random\_item()–Функция, объединяющая все предшествующие вспомогательные функции для реализации случайной величины.

**Открытые функции(интерфейс класса)**

density(const double x)- Подсчитывает плотность для стандартной функции без сдвига и масштаба;

dispersion()- Подсчитывает дисперсию для стандартной функции без сдвига и масштаба;

double rand\_var() – Реализация случайной величины.

expected\_value()- Подсчитывает мат.ожидание для стандартной функции без сдвига и масштаба;

set\_shift(double shift) – задать значение сдвига;

set\_scale(double scale)- задать значение масштаба;

set\_form(double form)- задать значение формы;

get\_shift() – получить значение сдвига;

get\_scale() – задать значение масштаба;

get\_form()- задать значение формы;

asymmetry()- расчет асимметрии;

generate\_selection(const int n)– создание вектора случайных величин;

generate\_graph\_selection(const std::vector<double>& selection, const int n)- создание ветора пар вида: (случайная величина, плотность функции);

**Конструкторы класса**

Std\_distr( double form0,double shift0, double scale0):v(form0), u(shift0), l(scale0) {} – конструктор, принимающий на вход значения параметров и создающий объект с данными атрибутами.

Std\_distr(std::string File\_name)– конструктор, создающий объект класса с атрибутами считанными из файла. Формат файла: 3 числа в одной строке, разделенных пробелом (параметр формы, сдвига, масштаба).

Std\_distr() :u(0), l(1), v(1) {}– конструктор, создающий объект с параметрами по умолчанию( п. формы=1, п. сдвига=0, п.масштаба=1).

**Модуль Standart\_distr.h**

Содержит описание класса, его открытой и закрытой части, объявления методов, конструкторы, а также подключение всех необходимых библиотек.

**Модуль main.cpp**

Содержит код для коммуникации с пользователем, а так же операции работы с файлами

**Модуль Mixed\_distr.cpp**

Содержит в себе описание всех методов класса MixtureDistribution.

**Закрытые функции(методы класса)**

double random\_var()– метод для генерации случайной величины для смеси распределений.

**Открытые функции(интерфейс класса)**

double density(const double x) const override; - подсчет плотности

double expected\_value() const override; - подсчет мат.ожидания

double dispersion() const override; - подсчет дисперсии

double excess() const override; - подсчет эксцесса

double asymmetry() const override; - подсчет асимметрии

double rand\_var() const override; - реализация случайной величины

std::vector<double> generate\_selection(const int n) const override;

std::vector<std::pair<double, double>> generate\_graph\_selection(const std::vector<double>& selection, const int n) const override;

void load\_from\_file(std::ifstream& file) override;

void save\_in\_file(std::ofstream& file) override;

double get\_p() const; - метод для получения значения параметра смеси

void set\_p(const double p); - метод для присваивания значения параметру смеси

**Конструкторы класса**

MixtureDistribution(Distribution1& d1, Distribution2& d2, double p) : d1(d1), d2(d2), p(p) {}; – конструктор инициализирующий два объекта путем копирования их из входных параметров, а так же инициализирует параметр смеси распределений р.

**Модуль Empirical.cpp**

Содержит в себе описание всех методов класса Empiric\_distr.

**Закрытые функции(методы) класса**

double delta\_calc() const; - расчет длины шага в выборке

std::vector<double> create\_intervals(const double delta) const; - создание интервалов для эмпирического распределения.

double random\_var() const; - реализация случайной величины

std::vector<double> create\_empirical\_density(const double delta, std::vector<double> intervals) const; - расчет плостности эмпирического распределения.

double qumulative\_probability(const int i) const;

double top\_bound() const;

**Открытые функции(интерфейс класса)**

double density(const double x) const override;

double expected\_value() const override;

double dispersion() const override;

double excess() const override;

double asymmetry() const override;

double rand\_var() const override;

std::vector<double> generate\_selection(const int n) const override;

std::vector<std::pair<double,double>>generate\_graph\_selection(const std::vector<double>& selection, const int n) const override;

EmpiricalDistribution& operator=(const EmpiricalDistribution& ed);

void set\_intrevals\_number(int k);

std::vector<double> get\_selection() const;

std::vector<double> get\_empirical\_density() const;

int get\_size() const;

int get\_intrevals\_number() const;

void load\_from\_file(std::ifstream& file) override;

void save\_in\_file(std::ofstream& file) override;

**Конструкторы класса**

EmpiricalDistribution(const IDistribution& d, int n, int k); - конструктор эмпирического распределения по объекту класса смеси или стандартного распределения

EmpiricalDistribution(const EmpiricalDistribution& d); - конструктор копирования

EmpiricalDistribution(std::ifstream& file); - конструктор из файла

EmpiricalDistribution(const std::vector<double>& selection);

**Модуль Empiric.h**

Содержит описание класса, его открытой и закрытой части, объявления методов, конструкторы, а также подключение всех необходимых библиотек.

**Модуль Empirical.cpp**

Содержит в себе описание всех методов класса Empiric\_distr.

**Модуль distribution\_test.cpp**

Модуль для тестирования программы.

**Код программы**

**Модуль Distributions.h**

#pragma once

#include <vector>

#include <fstream>

#include <algorithm>

class IDistribution {

public:

double virtual density(const double x) const = 0;

double virtual expected\_value() const = 0;

double virtual dispersion() const = 0;

double virtual excess() const = 0;

double virtual asymmetry() const = 0;

double virtual rand\_var() const = 0;

std::vector<double> virtual generate\_selection(const int n) const = 0;

std::vector<std::pair<double, double>> virtual generate\_graph\_selection(const std::vector<double>& selection, const int n) const = 0;

};

class IPresistend {

public:

void virtual load\_from\_file(std::ifstream& file) = 0;

void virtual save\_in\_file(std::ofstream& file) = 0; };

**Модуль Standart\_distr.h**

#pragma once

#define \_USE\_MATH\_DEFINES

#include <iostream>

#include <math.h>

#include <vector>

#include <algorithm>

#include <time.h>

#include <string>

#include <fstream>

#include "Distributions.h"

class Std\_distr : public IDistribution, public IPresistend {

public:

Std\_distr() :u(0), l(1), v(1) {}; // конструктор по умолчанию

Std\_distr( double form0,double shift0, double scale0):v(form0), u(shift0), l(scale0) {}; // Конструктор лего

Std\_distr(std::string File\_name) { // конструктор из файла

double symbol;

double mass[3];

std::ifstream in(File\_name); // открываем файл для чтения

if (in.is\_open())

{

for (int i = 0; i < 3; i++)

{

in >> symbol;

while (in.fail())

{

in.clear();

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

std::cout << "Ошибка в данных из входного файла. Производится выход из программы. " << std::endl;

exit(0);

}

mass[i] = symbol;

}

}

else { std::cout << "Неверное имя файла"; exit(0); }

in.close(); // закрываем файл

Std\_distr::set\_form(mass[2]);

if (mass[2] >= 1) { Std\_distr::set\_form(mass[2]); }

else {

std::cout << "Ошибка в значении параметра формы. Производится выход из программы. ";

exit(0);

}

Std\_distr::set\_shift(mass[0]);

Std\_distr::set\_scale(mass[1]);

}

~Std\_distr() // деструктор

{

}

double density(const double x) const override;

double expected\_value() const override;

double dispersion() const override;

double excess() const override;

double asymmetry() const override;

double rand\_var() const override;

std::vector<double> generate\_selection(const int n) const override;

std::vector<std::pair<double, double>> generate\_graph\_selection(const std::vector<double>& selection, const int n) const override;

void set\_form(const double n);

void set\_shift(const double mu);

void set\_scale(const double lambda);

double get\_form() const;

double get\_shift() const;

double get\_scale() const;

void load\_from\_file(std::ifstream& file) override;

void save\_in\_file(std::ofstream& file) override;

private:

double v, u, l;

double Standart(double x)const;

double Randomizer()const;

double Random\_item12()const;

double Random\_item2()const;

};

**Модуль Standart\_distr.cpp**

#include "Standart\_distr.h"

void Std\_distr::set\_shift(double shift) {

Std\_distr::u = shift;

}

void Std\_distr::set\_scale(double scale) {

Std\_distr::l = scale;

}

void Std\_distr::set\_form(double form) {

Std\_distr::v = form;

}

double Std\_distr::get\_shift() const{

return Std\_distr::u;

}

double Std\_distr::get\_scale() const {

return Std\_distr::l;

}

double Std\_distr::get\_form() const {

return Std\_distr::v;

}

double Std\_distr::Standart(double x) const {

double xmod = ((x - Std\_distr::u) / Std\_distr::l);

return xmod;

}

double Std\_distr::density(double x) const{

double xmod = Standart(x);

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

return mod\_density;

}

double Std\_distr::expected\_value() const{

double mod\_expected\_value = Std\_distr::u;

return mod\_expected\_value;

}

double Std\_distr::dispersion() const{

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

return mod\_dispersion;

}

double Std\_distr::excess() const{

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

return excess;

}

double Std\_distr::asymmetry() const{ return 0; }

double Std\_distr::Randomizer() const{

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 Std\_distr::Random\_item12() const{

double r = Randomizer();

double a = (1 / Std\_distr::v) - 1;

double b = 1 / (pow(Std\_distr::v, 1 / Std\_distr::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(Std\_distr::v \* (-log(abs(x)))) + (abs(x) / b) + a)) { return x; }

else { return Random\_item12(); }

}

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

double Std\_distr::Random\_item2()const {

double a = (1 / Std\_distr::v) - 0.5;

double b = 1 / (pow(Std\_distr::v, 1 / Std\_distr::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(Std\_distr::v \* (-log(abs(x)))) + (pow(x, 2) / c) + a)) { return x; }

else { return Random\_item2(); }

}

// function for the whole algorithm use

double Std\_distr::rand\_var() const{

double result = 0;/////////////////////////////////

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

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

return ((result \* Std\_distr::l + Std\_distr::u));

}

std::vector<double> Std\_distr::generate\_selection(const int n) const{

std::vector<double> result;

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

result.push\_back(rand\_var());

}

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

return result;

}

std::vector<std::pair<double, double>> Std\_distr::generate\_graph\_selection(const std::vector<double> &vec, const int n) const{

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

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

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

}

return result;

}

void Std\_distr::save\_in\_file(std::ofstream& file) {

file << v << std::endl << u << std::endl << l << std::endl;

}

void Std\_distr::load\_from\_file(std::ifstream& file) {

double v, u, l;

if (!file.is\_open()) {

throw 0;

}

file >> v >>u >> l;

if (v <= 0 || l <= 0) {

throw 1;

}

this->v = v;

this->u = u;

this->l = l;

}

**Модуль Mixed\_distr.cpp**

#include "distributions.h"

template<class Distribution1, class Distribution2>

class MixtureDistribution : public IDistribution, public IPresistend {

public:

MixtureDistribution(Distribution1& d1, Distribution2& d2, double p) : d1(d1), d2(d2), p(p) {}; //конструктор

double density(const double x) const override;

double expected\_value() const override;

double dispersion() const override;

double excess() const override;

double asymmetry() const override;

double rand\_var() const override;

std::vector<double> generate\_selection(const int n) const override;

std::vector<std::pair<double, double>> generate\_graph\_selection(const std::vector<double>& selection, const int n) const override;

void load\_from\_file(std::ifstream& file) override;

void save\_in\_file(std::ofstream& file) override;

double get\_p() const;

void set\_p(const double p);

Distribution1& component1() { return d1; }

Distribution2& component2() { return d2; }

private:

Distribution1 d1;

Distribution2 d2;

double p;

double random\_var() const;

};

template<class Dist1, class Dist2>

double MixtureDistribution<Dist1, Dist2>::get\_p() const {

return p;

}

template<class Dist1, class Dist2>

void MixtureDistribution<Dist1, Dist2>::set\_p(const double p) {

if (p < 0 or p > 1) {

throw 1;

}

this->p = p;

}

template<class Dist1, class Dist2>

double MixtureDistribution<Dist1, Dist2>::density(const double x) const {

return (1 - p) \* d1.density(x) + p \* d2.density(x);

}

template<class Dist1, class Dist2>

double MixtureDistribution<Dist1, Dist2>::expected\_value() const {

return (1 - p) \* d1.expected\_value() + p \* d2.expected\_value();

}

template<class Dist1, class Dist2>

double MixtureDistribution<Dist1, Dist2>::dispersion() const {

return (1 - p) \* (pow(d1.expected\_value(), 2) + d1.dispersion()) +

p \* (pow(d2.expected\_value(), 2) + d2.dispersion()) -

pow(expected\_value(), 2);

}

template<class Dist1, class Dist2>

double MixtureDistribution<Dist1, Dist2>::asymmetry() const {

return ((1 - p) \* (pow((d1.expected\_value() - expected\_value()), 3) + 3 \* (d1.expected\_value() - expected\_value()) \* d1.dispersion() + pow(d1.dispersion(), 3 / 2) \* d1.asymmetry()) +

p \* (pow((d2.expected\_value() - expected\_value()), 3) + 3 \* (d2.expected\_value() - expected\_value()) \* d2.dispersion() + pow(d2.dispersion(), 3 / 2) \* d2.asymmetry())) /

pow(dispersion(), 3 / 2);

}

template<class Dist1, class Dist2>

double MixtureDistribution<Dist1, Dist2> ::excess() const {

return ((1 - p) \* (pow((d1.expected\_value() - expected\_value()), 4) + 6 \* d1.dispersion() \* pow((d1.expected\_value() - expected\_value()), 2) +

4 \* (d1.expected\_value() - expected\_value()) \* pow(d1.dispersion(), 3 / 2) \* d1.asymmetry() + pow(d1.dispersion(), 2) \* d1.excess()) +

p \* (pow((d2.expected\_value() - expected\_value()), 4) + 6 \* d2.dispersion() \* pow((d2.expected\_value() - expected\_value()), 2) +

4 \* (d2.expected\_value() - expected\_value()) \* pow(d2.dispersion(), 3 / 2) \* d2.asymmetry() + pow(d2.dispersion(), 2) \* d2.excess()) - 3) /

pow(dispersion(), 2);

}

template<class Dist1, class Dist2>

double MixtureDistribution<Dist1, Dist2>::random\_var() const {

double r;

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

return r;

}

template<class Dist1, class Dist2>

double MixtureDistribution<Dist1, Dist2>::rand\_var() const {

double r = random\_var();

if (r > p) {

return d1.rand\_var();

}

else {

return d2.rand\_var();

}

}

template<class Dist1, class Dist2>

std::vector<double> MixtureDistribution<Dist1, Dist2>::generate\_selection(const int n) const {

std::vector<double> sample;

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

sample.push\_back(rand\_var());

}

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

return sample;

}

template<class Dist1, class Dist2>

std::vector<std::pair<double, double>> MixtureDistribution<Dist1, Dist2>::generate\_graph\_selection(const std::vector<double>& selection, const int n) const {

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

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

result.push\_back(std::make\_pair(selection[i], density(selection[i])));

}

return result;

}

template<class Dist1, class Dist2>

void MixtureDistribution<Dist1, Dist2>::save\_in\_file(std::ofstream& file) {

file << p << std::endl;

component1().save\_in\_file(file);

component2().save\_in\_file(file);

}

template<class Dist1, class Dist2>

void MixtureDistribution<Dist1, Dist2>::load\_from\_file(std::ifstream& file) {

file >> p;

component1().load\_from\_file(file);

component2().load\_from\_file(file);

}

**Empiric.h**

#pragma once

#include "distributions.h"

class EmpiricalDistribution : public IDistribution, public IPresistend {

public:

EmpiricalDistribution(const IDistribution& d, int n, int k);

EmpiricalDistribution(const EmpiricalDistribution& d);

EmpiricalDistribution(std::ifstream& file);

EmpiricalDistribution(const std::vector<double>& selection);

~EmpiricalDistribution();

double density(const double x) const override;

double expected\_value() const override;

double dispersion() const override;

double excess() const override;

double asymmetry() const override;

double rand\_var() const override;

std::vector<double> generate\_selection(const int n) const override;

std::vector<std::pair<double, double>> generate\_graph\_selection(const std::vector<double>& selection, const int n) const override;

EmpiricalDistribution& operator=(const EmpiricalDistribution& ed);

void set\_intrevals\_number(int k);

std::vector<double> get\_selection() const;

std::vector<double> get\_empirical\_density() const;

int get\_size() const;

int get\_intrevals\_number() const;

void load\_from\_file(std::ifstream& file) override;

void save\_in\_file(std::ofstream& file) override;

private:

std::vector<double> selection;

std::vector<double> empirical\_density;

int size;

int k;

double delta\_calc() const;

std::vector<double> create\_intervals(const double delta) const;

double random\_var() const;

std::vector<double> create\_empirical\_density(const double delta, std::vector<double> intervals) const;

double qumulative\_probability(const int i) const;

double top\_bound() const;

};

**Empirical.cpp**

#include "Empiric.h"

EmpiricalDistribution::EmpiricalDistribution(const IDistribution& d, int n, int \_k) :

size(n > 1 ? n : throw 1), k(\_k > 2 ? \_k : (int)log2(size) + 1) {

selection = d.generate\_selection(size);

empirical\_density = create\_empirical\_density(delta\_calc(), create\_intervals(delta\_calc()));

}

EmpiricalDistribution::EmpiricalDistribution(const EmpiricalDistribution& ed) :

size(ed.size > 1 ? ed.size : throw 1), k(ed.k > 2 ? ed.k : ((int)log2(size) + 1)), selection(ed.selection), empirical\_density(ed.empirical\_density) {

empirical\_density = create\_empirical\_density(delta\_calc(), create\_intervals(delta\_calc()));

}

EmpiricalDistribution::EmpiricalDistribution(std::ifstream& file) {

file.open("eparams.txt");

if (!file.is\_open()) {

throw 0;

}

double x;

while (file >> x) {

selection.push\_back(x);

}

size = selection.size();

k = (int)log2(size) + 1;

empirical\_density = create\_empirical\_density(delta\_calc(), create\_intervals(delta\_calc()));

}

EmpiricalDistribution::EmpiricalDistribution(const std::vector<double>& selection) :

size(selection.size()), k((int)log2(size) + 1), selection(selection) {

empirical\_density = create\_empirical\_density(delta\_calc(), create\_intervals(delta\_calc()));

}

EmpiricalDistribution::~EmpiricalDistribution() {

selection.clear();

empirical\_density.clear();

}

std::vector<double> EmpiricalDistribution::create\_empirical\_density(const double delta, std::vector<double> intervals) const {

std::vector<double> result;

int j = 0;

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

int count = 0;

for (j; j < size; ++j) {

if (i + 1 == intervals.size() - 1) {

if (selection[j] <= intervals[i + 1]) {

++count;

}

else {

break;

}

}

else {

if (selection[j] < intervals[i + 1]) {

++count;

}

else {

break;

}

}

}

result.push\_back(count / (size \* delta));

}

return result;

}

double EmpiricalDistribution::random\_var() const {

double r;

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

return r;

}

double EmpiricalDistribution::qumulative\_probability(const int i) const {

double q = 0;

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

q += empirical\_density[j];

}

return q;

}

double EmpiricalDistribution::top\_bound() const {

double sum = 0;

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

sum += empirical\_density[i];

}

return sum;

}

double EmpiricalDistribution::rand\_var() const {

double r;

do r = (double)rand() / RAND\_MAX \* top\_bound();

while (r == 0 || r == top\_bound());

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

if (r > qumulative\_probability(i) and r < qumulative\_probability(i + 1)) {

do r = (double)rand() / RAND\_MAX \* ((selection[0] + delta\_calc() \* (i + 1)) - (selection[0] + delta\_calc() \* i)) + (selection[0] + delta\_calc() \* (i + 1));

while (r == (selection[0] + delta\_calc() \* i) || r == (selection[0] + delta\_calc() \* (i + 1)));

break;

}

}

return r;

}

EmpiricalDistribution& EmpiricalDistribution::operator = (const EmpiricalDistribution& ed) {

if (this == &ed) {

return \*this;

}

selection = ed.selection;

empirical\_density = ed.empirical\_density;

size = ed.size;

k = ed.k;

return \*this;

}

void EmpiricalDistribution::set\_intrevals\_number(const int k) {

if (k >= 2) {

this->k = k;

}

else {

this->k = (int)log2(size) + 1;

}

empirical\_density = create\_empirical\_density(delta\_calc(), create\_intervals(delta\_calc()));

}

std::vector<double> EmpiricalDistribution::get\_selection() const {

return selection;

}

std::vector<double> EmpiricalDistribution::get\_empirical\_density() const {

return empirical\_density;

}

int EmpiricalDistribution::get\_size() const {

return size;

}

int EmpiricalDistribution::get\_intrevals\_number() const {

return k;

}

double EmpiricalDistribution::delta\_calc() const {

return (selection[size - 1] - selection[0]) / (k);

}

std::vector<double> EmpiricalDistribution::create\_intervals(const double delta) const {

std::vector<double> intervals;

double slider = selection[0];

for (int i = 0; i < k + 1; ++i) {

intervals.push\_back(slider);

slider += delta;

}

return intervals;

}

double EmpiricalDistribution::density(const double x) const {

double slider = selection[0];

if (x < slider) {

return 0;

}

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

slider += delta\_calc();

if (i == k - 1) {

if (x <= slider) {

return empirical\_density[i];

}

}

else {

if (x < slider) {

return empirical\_density[i];

}

}

}

return 0;

}

double EmpiricalDistribution::expected\_value() const {

double sum = 0;

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

sum += selection[i];

}

return sum / size;

}

double EmpiricalDistribution::dispersion() const {

double sum = 0;

double exp\_val = expected\_value();

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

sum += pow(selection[i] - exp\_val, 2);

}

return sum / size;

}

double EmpiricalDistribution::asymmetry() const {

double sum = 0;

double exp\_val = expected\_value();

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

sum += pow(selection[i] - exp\_val, 3);

}

return sum / (size \* pow(dispersion(), 3 / 2));

}

double EmpiricalDistribution::excess() const {

double sum = 0;

double exp\_val = expected\_value();

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

sum += pow(selection[i] - exp\_val, 4);

}

return (sum / (size \* pow(dispersion(), 2))) - 3;

}

std::vector<double> EmpiricalDistribution::generate\_selection(const int n) const {

std::vector<double> result;

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

result.push\_back(rand\_var());

}

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

return result;

}

std::vector<std::pair<double, double>> EmpiricalDistribution::generate\_graph\_selection(const std::vector<double>& selection, const int n) const {

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

for (int j = 0; j < size; ++j) {

result.push\_back(std::make\_pair(selection[j], density(selection[j])));

}

return result;

}

void EmpiricalDistribution::save\_in\_file(std::ofstream& file) {

file.open("eparams.txt");

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

file << selection[i] << " " << std::endl;

}

file.close();

}

void EmpiricalDistribution::load\_from\_file(std::ifstream& file) {

file.open("emparams.txt");

if (!file.is\_open()) {

throw 0;

}

std::vector<double> result;

double x;

while (file >> x) {

result.push\_back(x);

}

selection = result;

size = result.size();

k = (int)log2(size) + 1;

empirical\_density = create\_empirical\_density(delta\_calc(), create\_intervals(delta\_calc()));

}

**Main.cpp**

#define CATCH\_CONFIG\_RUNNER

#include <iostream>

#include <fstream>

#include "Standart\_distr.h"

#include "Empiric.h"

#include "Mixed\_distr.cpp"

#include "catch.hpp"

using namespace std;

int run\_unit\_tests(int argc, char\*\* argv) {

int result = Catch::Session().run(argc, argv);

return result;

}

int main(int argc, char\*\* argv) {

setlocale(LC\_ALL, "ru");

srand((unsigned)time(0));

ofstream file1("sdt\_distr.txt");

ofstream file2("emp\_distr.txt");

try {

Std\_distr ld1(1, -6, 2);

Std\_distr ld2(1, -2, 1);

Std\_distr ld3(1, 2, 1);

Std\_distr ld4(1, 6, 2);

MixtureDistribution<Std\_distr, Std\_distr> md1(ld1, ld2, 0.5);

MixtureDistribution<Std\_distr, Std\_distr> md2(ld3, ld4, 0.5);

MixtureDistribution<MixtureDistribution<Std\_distr, Std\_distr>, MixtureDistribution<Std\_distr, Std\_distr>> md(md2, md1, 0.5);

auto selection = md.generate\_selection(1000);

auto graph1 = md.generate\_graph\_selection(selection,1000);

EmpiricalDistribution ed(selection);

auto graph2 = ed.generate\_graph\_selection(selection, 1000);

EmpiricalDistribution ed2(ed, 1000, 1);

auto graph3 = ed2.generate\_graph\_selection(selection, 1000);

cout << "параметры стандартных распределений:" << std::endl <<"n1 = "<< md.component1().component1().get\_form() << ", mu1 = " << md.component1().component1().get\_shift() << ", lambda1 = " << md.component1().component1().get\_scale() <<

"\nn2 = " << md.component1().component2().get\_form() << ", mu2 = " << md.component1().component2().get\_shift() << ", lambda2 = " << md.component1().component2().get\_scale() <<

"\nn3 = " << md.component2().component1().get\_form() << ", mu3 = " << md.component2().component1().get\_shift() << ", lambda3 = " << md.component2().component1().get\_scale() <<

"\nn4 = " << md.component2().component2().get\_form() << ", mu4 = " << md.component2().component2().get\_shift() << ", lambda4 = " << md.component2().component2().get\_scale() << ", p = " << md.get\_p() << endl << endl;;

cout << "смесь двух смесий:\nмат. ожидание:" << md.expected\_value() << "\nдисперсия: " << md.dispersion() << "\nассиметрия: " << md.asymmetry() << "\nэксцесс:" << md.excess() << endl << endl;

cout << "эмпирическое распределение для смеси1:\nмат. ожидание:" << ed.expected\_value() << "\nдисперсия: " << ed.dispersion() << "\nассиметрия: " << ed.asymmetry() << "\nэксцесс:" << ed.excess() << endl << endl;

cout << "2:\nмат. ожидание:" << ed2.expected\_value() << "\nдисперсия: " << ed2.dispersion() << "\nассиметрия: " << ed2.asymmetry() << "\nэксцесс:" << ed2.excess() << endl;

for (int i = 0; i < 1000; ++i) {

file1 << graph1[i].first << "\t" << graph1[i].second << endl;

file2 << graph2[i].first << "\t" << graph2[i].second << endl;

}

///

}

catch (const int error) {

if (error == 0) {

cout << "ошибка открытия params.txt!!!" << endl;

}

else if (error == 1) {

cout << "ошибка в данных в файле params.txt" << endl;

}

}

file1.close();

file2.close();

run\_unit\_tests(argc, argv);

}

**distribution\_test.cpp**

#include "catch.hpp"

#include "Standart\_distr.h"

#include "Mixed\_distr.cpp"

#include "Empiric.h"

bool equal(const double& x, const double& y) {

if (abs(x - y) <= 0.1) {

return true;

}

else return false;

}

TEST\_CASE("basic methods") {

Std\_distr distr;

CHECK(distr.get\_form() == 1);

CHECK(distr.get\_shift() == 0);

CHECK(distr.get\_scale() == 1);

}

TEST\_CASE("standart distribution") {

Std\_distr distr;

CHECK(distr.density(0) == 0.5);

CHECK(distr.expected\_value() == 0);

CHECK(distr.dispersion() == 2);

CHECK(distr.asymmetry() == 0);

CHECK(distr.excess() == 3);

}

TEST\_CASE("shift scale transformation") {

Std\_distr distr;

distr.set\_scale(2);

distr.set\_shift(2);

CHECK(equal(distr.density(0), 0.091) == true);

CHECK(distr.expected\_value() == 2);

CHECK(distr.dispersion() == 8);

CHECK(distr.asymmetry() == 0);

CHECK(distr.excess() == 3);

}

TEST\_CASE("mixture distribution") {

Std\_distr distr1;

Std\_distr distr2;

distr1.set\_scale(2);

distr2.set\_scale(2);

distr1.set\_shift(2);

distr2.set\_shift(2);

MixtureDistribution< Std\_distr, Std\_distr> mdistr(distr1, distr2, 0.5);

CHECK(equal(mdistr.density(0), 0.091) == true);

CHECK(mdistr.expected\_value() == 2);

CHECK(mdistr.dispersion() == 8);

CHECK(mdistr.asymmetry() == 0);

CHECK(equal(mdistr.excess(), 2.953) == true);

}

TEST\_CASE("mixture distribution expected") {

Std\_distr distr1;

Std\_distr distr2;

distr1.set\_scale(2);

distr2.set\_scale(2);

distr1.set\_shift(1);

distr2.set\_shift(2);

MixtureDistribution< Std\_distr, Std\_distr> mdistr(distr1, distr2, 0.5);

CHECK(mdistr.expected\_value() == 1.5);

}

TEST\_CASE("mixture distribution dispersion") {

Std\_distr distr1;

Std\_distr distr2;

distr1.set\_scale(1);

distr2.set\_scale(3);

MixtureDistribution< Std\_distr, Std\_distr> mdistr(distr1, distr2, 0.5);

CHECK(mdistr.dispersion() == 10);

}

TEST\_CASE("late binding mechanism") {

Std\_distr distr1;

Std\_distr distr2;

distr1.set\_scale(1);

distr2.set\_scale(3);

MixtureDistribution< Std\_distr, Std\_distr> mdistr(distr1, distr2, 0.5);

MixtureDistribution< Std\_distr, MixtureDistribution< Std\_distr, Std\_distr>> mdistr2(distr1, mdistr, 0.5);

CHECK(mdistr2.component1().get\_form() == 1);

CHECK(mdistr2.component1().get\_shift() == 0);

CHECK(mdistr2.component1().get\_scale() == 1);

CHECK(mdistr2.component2().component1().get\_form() == 1);

CHECK(mdistr2.component2().component1().get\_shift() == 0);

CHECK(mdistr2.component2().component1().get\_scale() == 1);

CHECK(mdistr2.component2().component2().get\_form() == 1);

CHECK(mdistr2.component2().component2().get\_shift() == 0);

CHECK(mdistr2.component2().component2().get\_scale() == 3);

}

TEST\_CASE("empirical distribution") {

Std\_distr distr1;

EmpiricalDistribution ed(distr1, 200, 1);

CHECK(ed.get\_size() == 200);

CHECK(ed.get\_intrevals\_number() == 8);

/\*CHECK(equal(ed.expected\_value(), 0.0) == true);\*/

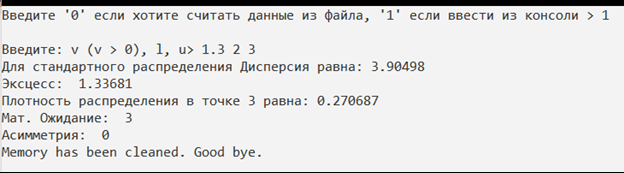
}

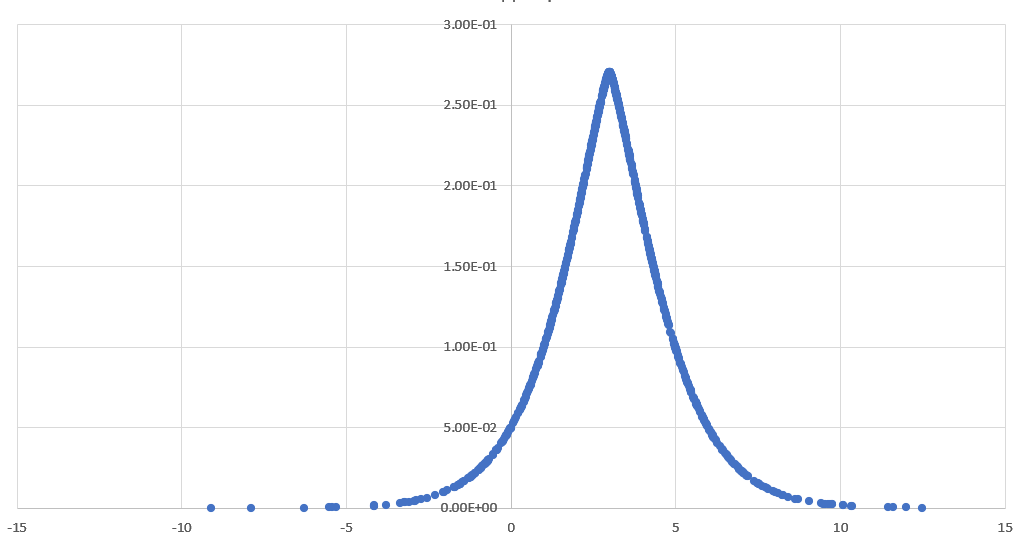
**Обработка исключений**

В программе предусмотрены юнит тесты для проверки программы, а так же обработка исключений с помощью try - catch – throw - конструкций.

**Тесты программы**

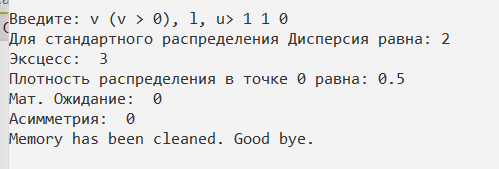
1.Тест для сдвиг масштабных преобразований (n=1000):

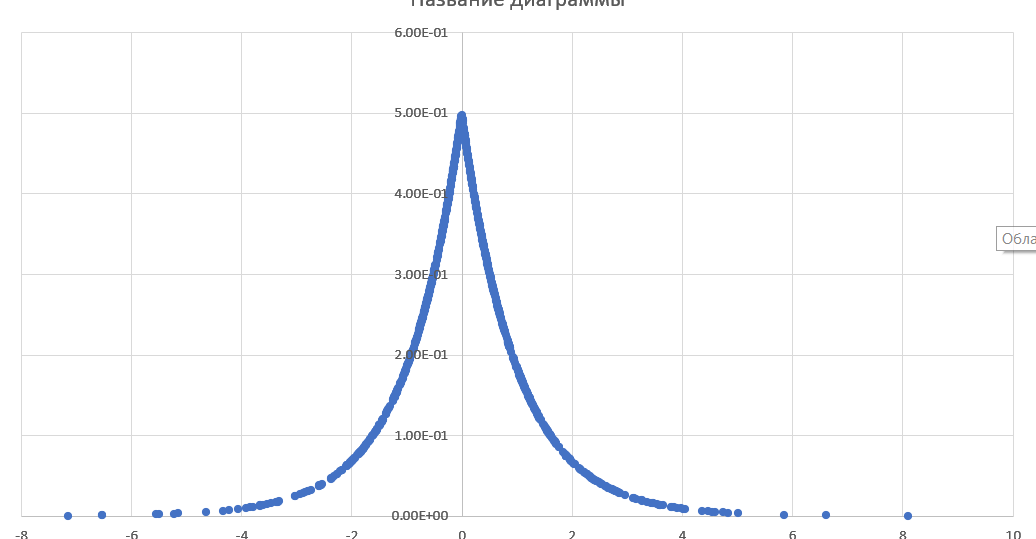


 Произошел сдвиг вправо на 3 единицы и масштаб изменился в два раза, как и ожидалось.

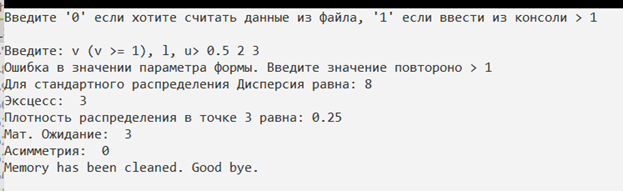
Как можем видеть, все значения посчитаны корректно, а график построен правильно.

2.Тест для стандартного распределения (n=1000):

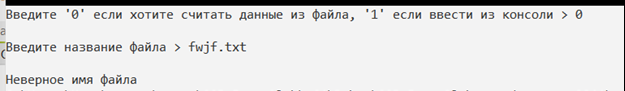




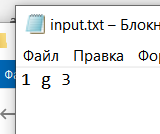
3. Попытка создать объект с некорректными входными данными(параметр формы меньше 1)

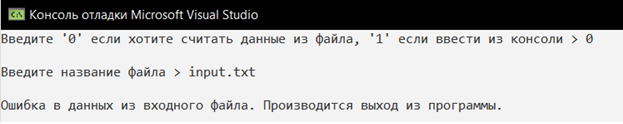


5.Попытка вызвать конструктор с несуществующим файлом

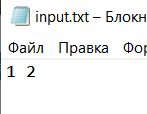


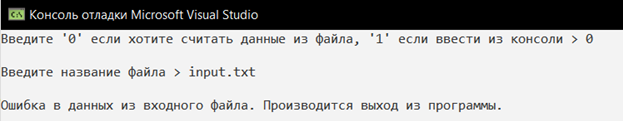
6. Во входном файле присутствуют символы





7.Во входном файле меньше 3 чисел



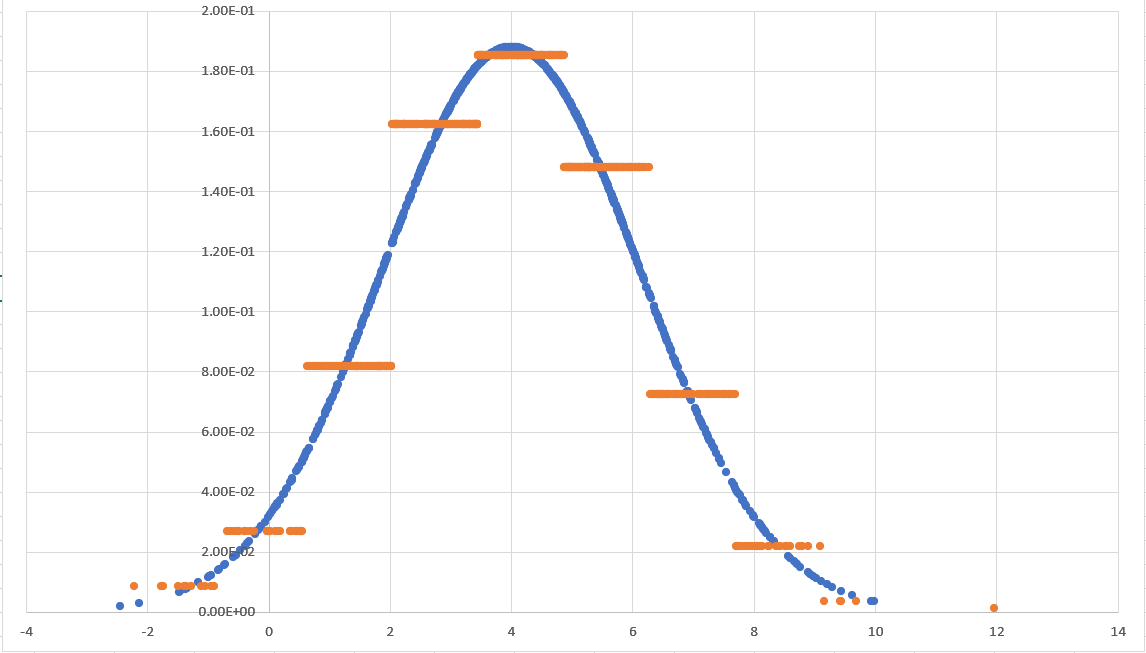


8. Сравнение полученных результатов с л.р. №1

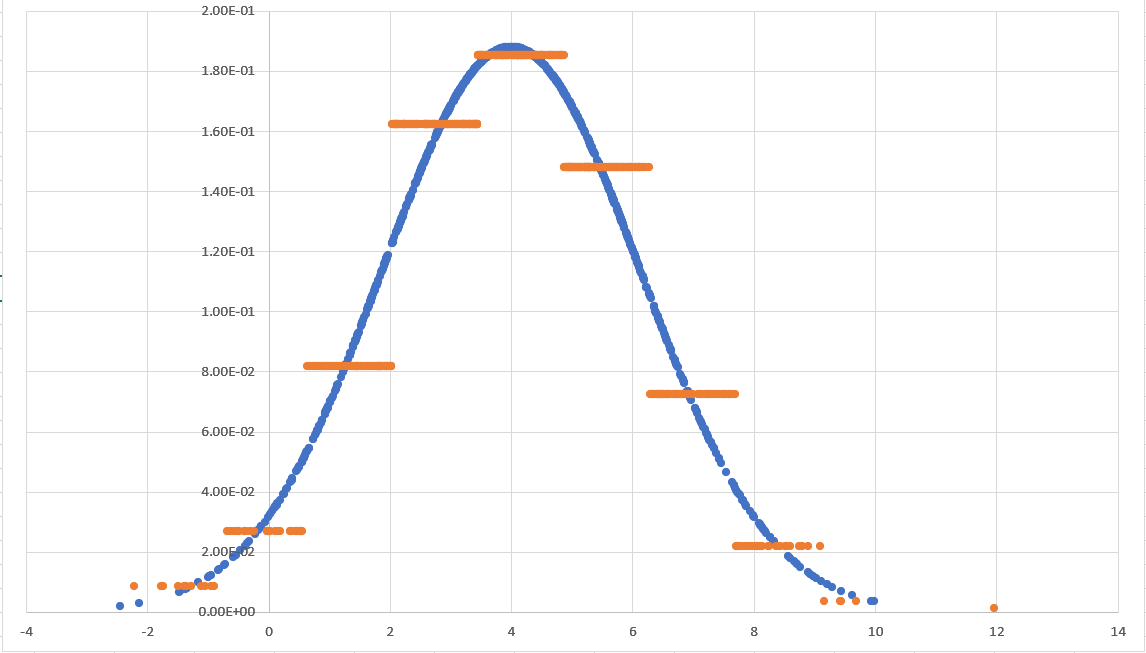
Размер выборки: 1000

Параметры:

Форма:2;сдвиг:4;масштаб:3



Л.р. №1



Л.р №4

Как мы видим, полученные результаты совпадают, что говорит о корректности работы программы.

**Выводы**

Используя методы, реализованные в ходе первой лабораторной работы, мы создали классы с методами, позволяющими вычислять характеристики стандартного, смешанного и эмпирического распределения, а также генерировать выборку данных и выводить ее для построения графиков. Была обеспечена персистентность объекта средствами сохранения и считывания атрибутов класса с файла, реализованы обработки исключений для разных ошибочных ситуаций, которые могут возникнуть при работе с классом.