Новосибирский государственный технический

университет

Кафедра ТПИ

Объектно-ориентированное программирование

Лабораторная работа №3

**Объекты**

Выполнили: Преподаватели:

Яшков Иван Дворецкая Виктория Константиновна

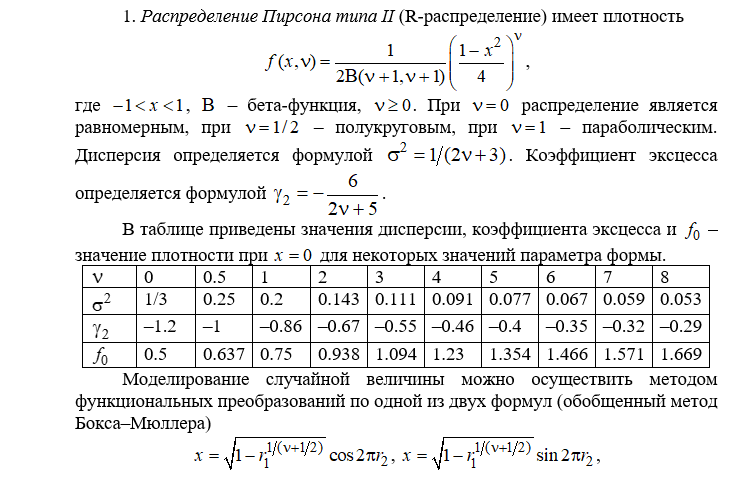
Гавриленко Андрей Лисицин Даниил Валерьевич

Группа ПМИ-13

Бригада 1

Новосибирск, 2023

**Задание (вариант 1):**

Были добавлены два класса: класс для распределения смеси MixtureDistribution и класс для эмпирического распределения EmpiricalDistribution.

**Описание разработанных классов:**

**Класс MixtureDistribution:**

**Атрибуты:**

Атрибутами класса два объекта класса PearsonDistributionType2, описывающего основное распределение, и параметр смеси p, типа double.

**Функции:**

1. **double get\_p()** и **void set\_p(double p0)** являются геттером и сеттером параметра смеси p. Функция **get\_p()** возвращает текущее значение p, **set\_p()** присваивает значение p0 значению p при условии 0 < p0 < 1.

2. **PearsonDistributionType2& getComponent1()** и **PearsonDistributionType2& getComponent2()** позволяют получить доступ к соответствующим компонентам смеси (к соответствующему объекту класса основного распределения). Получив доступ к компоненту, можно использовать все функции стандартного распределения (геттеры и сеттеры, вычисление числовых характеристик, моделирование и т. д.).

3. **double\* get\_moments()** вычисляет числовые характеристики распределения смеси и возвращает их в виде массива чисел с плавающей точкой.

4. **double get\_f(double x)** вычисляет функцию плотности для распределения смеси в точке x.

5. **double get\_model()** выполняет моделирование случайной величины, подчиняющейся закону текущего распределения смеси.

6. **void load(string p\_file, string pd1\_file, string pd2\_file)** и **void save(string p\_file, string pd1\_file, string pd2\_file)** нужны длязагрузки из файла и сохранения в файл распределения смеси. Функция **load()** принимает из первого файла значение p и вызывает функции load() компонентов основного распределения, которые получают свои параметры mu, lambda, nu из второго и третьего файлов. Функция **save()** загружает в первый файл значение p и вызывает функции save() компонентов основного распределения, которые заносят свои параметры mu, lambda, nu во второй и третий файлы.

7. **void generate\_distribution()** генерирует распределение с помощью большого количества вызовов функции get\_model() и получения значений функции плотности с помощью get\_f() для той или иной случайной величины. Значения случайных величин записываются в один файл ("x.txt"), значения функций плотности для каждой случайной величины записываются в другой файл ("fx.txt").

**Конструкторы:**

1. **MixtureDistribution(PearsonDistributionType2& pd01, PearsonDistributionType2& pd02, double p0 = 0.5)** - конструктор, использующий два объекта класса основного распределения и параметр смеси для задания распределения (по умолчанию 0.5). Используются инициализаторы конструктора, которые устанавливают оба объекта и автоматически проверяют вводимое p.

2. **MixtureDistribution(string p\_file, string pd1\_file, string pd2\_file)** - конструктор, считывающий параметры для двух основных распределений и параметр смеси p из файлов. Для получения всех параметров вызывает функцию load().

**Класс EmpiricalDistribution:**

**Атрибуты:**

1. n - количество элементов в выборке.

2. k - количество промежутков эмпирического распределения.

3. min - минимальное значение выборки. Необходимо для вычисления в некоторых функциях.

4. delta - длина промежутка.

5. data - массив элементов выборки.

6. counter - массив, хранящий количество элементов в том или ином промежутке.

7. frequency - массив относительных частот.

8. density - массив, содержащий значения функции плотности для каждого промежутка.

**Функции:**

1. **double\* get\_moments()** вычисляет числовые характеристики эмпирического распределения и возвращает их в виде массива чисел с плавающей точкой.

2. **double get\_f(double x)** вычисляет функцию плотности для эмпирического распределения в точке x.

3. **double get\_model()** выполняет моделирование случайной величины, подчиняющейся закону текущего эмпирического распределения.

4. **void load(string filename)** и **void save(string filename)** нужны длязагрузки из файла и сохранения в файл эмпирического распределения. Функция **load()** принимает из файла сначала количество элементов выборки n, а затем в цикле размера n начинает считывать сами элементы выборки. Функция **save()** загружает в файл сначала n, а затем последовательно загружает элементы выборки.

5. **void generate\_distribution()** генерирует распределение с помощью большого количества вызовов функции get\_model() и получения значений функции плотности с помощью get\_f() для той или иной случайной величины. Значения случайных величин записываются в один файл ("x.txt"), значения функций плотности для каждой случайной величины записываются в другой файл ("fx.txt").

**Конструкторы:**

1. **EmpiricalDistribution(int n0, PearsonDistributionType2& pd, int k0 = 1), EmpiricalDistribution(int n0, MixtureDistribution& pd, int k0 = 1), EmpiricalDistribution(int n0, EmpiricalDistribution& pd, int k0 = 1) -** конструкторы, поддерживающие создание эмпирического распределения для основного распределения, распределения смеси и эмпирического распределения соответственно. Конструкторы принимают количество элементов в выборке n0, объект соответствующего типа и количество промежутков k0. Используются инициализаторы конструктора, которые сразу проверяют вводимый n на правильность (n > 1), а также устанавливают k равным k0 в случае k0 > 1 и k равным если k0 <= 1.

2. **EmpiricalDistribution(string filename)** - конструктор, создающий эмпирическое распределение с помощью файла. Для создания в конструкторе вызывается функция load().

3. **EmpiricalDistribution(const EmpiricalDistribution& emp) -** конструктор копирования, позволяющий при создании объекта класса эмпирического распределения скопировать все атрибуты от уже существующего объекта класса эмпирического распределения.

**Деструктор:**

**~EmpiricalDistribution() -** деструктор, удаляющий все массивы (data, counter, frequency, density) для освобождения неиспользуемой памяти.

**Оператор присваивания "=":**

**EmpiricalDistribution& operator=(const EmpiricalDistribution& emp)** перегружает оператор присваивания. При попытке присвоить объект самому себе возвращается тот же объект, иначе проверяем равенства между количествами элементов выборки и между количествами промежутков. Если где-то нет равенства, значит размеры изменяются и необходимо у объекта, к которому присваивается другой объект, изменить размеры. После того как размеры стали равны происходит присваивание атрибутов одного объекта другому.

**Тестирование возможностей программы:**

В файл test.cpp было добавлено еще две функции: **void mix\_test()** и **void emp\_test()**.

mix\_test():

Тестирующая функция состоит из 6-ти блоков:

1. Проверка первого конструктора.

2. Проверка сеттера и геттера атрибута p.

3. Попытка вставить неправильные значения для p через конструктор и через сеттер.

4. Попытка обращения к геттерам и сеттерам объектов основного распределения через функции getComponent1() и getComponent2().

5. Получение функции плотности и моделирование случайной величины распределения смеси.

6. Проверка функций save() и load().

emp\_test():

Тестирующая функция состоит из 5-ти блоков:

1. Проверка первых трех конструкторов, k вычисляется по формуле по умолчанию.

2. Проверка одного из этих конструкторов, k задается пользователем.

3. Получение числовых характеристик эмпирического распределения, получение функции плотности в некоторых точках, моделирование случайной величины.

4. Проверка функций save() и load().

5. Проверка оператора присваивания и конструктора копирования.

**Текст программы:**

Далее будут отображены лишь те файлы, которые были изменены/добавлены в процессе выполнения данной лабораторной работы.

**Header.h:**

#pragma once

#include <iostream>

#include <fstream>

#include <string>

#include <cmath>

#include <time.h>

const double PI = std::acos(-1);

class PearsonDistributionType2 {

double mu;

double la;

double nu;

double\* moments = new double[4];

public:

PearsonDistributionType2(double mu0 = 0, double la0 = 1, double nu0 = 0);

PearsonDistributionType2(std::string name);

void setMU(double mu0);

double getMU();

void setLA(double la0);

double getLA();

void setNU(double nu0);

double getNU();

double\* get\_moments();

double get\_f(double x);

double get\_model();

void load(std::string name);

void save(std::string name);

void generate\_distribution();

private:

void set\_moments();

};

class MixtureDistribution {

PearsonDistributionType2 pd1, pd2;

double p;

public:

MixtureDistribution(PearsonDistributionType2& pd01, PearsonDistributionType2& pd02, double p0 = 0.5) : p(p0 > 0 && p0 < 1 ? p0 : throw "ERROR: Bad p. p should be: 0 < p < 1."), pd1(pd01), pd2(pd02) {}

MixtureDistribution(std::string p\_file, std::string pd1\_file, std::string pd2\_file);

PearsonDistributionType2& getComponent1() { return pd1; }

PearsonDistributionType2& getComponent2() { return pd2; }

double get\_p();

void set\_p(double p0);

double\* get\_moments();

double get\_f(double x);

double get\_model();

void load(std::string p\_file, std::string pd1\_file, std::string pd2\_file);

void save(std::string p\_file, std::string pd1\_file, std::string pd2\_file);

void generate\_distribution();

};

class EmpiricalDistribution {

int n, k;

double delta, min;

int\* counter;

double\* data, \*frequency, \*density;

public:

EmpiricalDistribution(int n0, PearsonDistributionType2& pd, int k0 = 1);

EmpiricalDistribution(int n0, MixtureDistribution& pd, int k0 = 1);

EmpiricalDistribution(int n0, EmpiricalDistribution& pd, int k0 = 1);

EmpiricalDistribution(std::string filename);

EmpiricalDistribution(const EmpiricalDistribution& emp);

EmpiricalDistribution& operator=(const EmpiricalDistribution& emp);

double get\_f(double x);

double get\_model();

double\* get\_moments();

void load(std::string filename);

void save(std::string filename);

void generate\_distribution();

~EmpiricalDistribution();

private:

void get\_min\_delta();

void get\_counter\_frequency\_density();

};

void test();

void mix\_test();

void emp\_test();

**mix.cpp:**

#include "Header.h"

MixtureDistribution::MixtureDistribution(std::string p\_file, std::string pd1\_file, std::string pd2\_file) {

load(p\_file, pd1\_file, pd2\_file);

}

double MixtureDistribution::get\_p() {

return p;

}

void MixtureDistribution::set\_p(double p0) {

p = p0 > 0 && p0 < 1 ? p0 : throw "ERROR: Bad p. p should be: 0 < p < 1.";

}

double\* MixtureDistribution::get\_moments() {

double\* moments = new double[4];

double\* moments1 = pd1.get\_moments();

double\* moments2 = pd2.get\_moments();

moments[0] = (1. - p) \* moments1[0] + p \* moments2[0];

moments[1] = (1. - p) \* (pow(moments1[0], 2) + moments1[1]) + p \* (pow(moments2[0], 2) + moments2[1]) - pow(moments[0], 2);

moments[2] = 1. / pow(moments[1], 3. / 2.) \* ((1 - p) \* (pow(moments1[0] - moments[0], 3) + 3 \* moments1[1] \* (moments1[0] - moments[0]) + pow(moments1[1], 3. / 2.) \* moments1[2]) + (1 - p) \* (pow(moments2[0] - moments[0], 3) + 3 \* moments2[1] \* (moments2[0] - moments[0]) + pow(moments2[1], 3. / 2.) \* moments2[2]));

moments[3] = 1. / pow(moments[1], 2) \* ((1 - p) \* (pow(moments1[0] - moments[0], 4) + 6 \* moments1[1] \* pow(moments1[0] - moments[0], 2) + 4 \* (moments1[0] - moments[0]) \* pow(moments1[1], 3. / 2.) \* moments1[2] + pow(moments1[1], 2) \* moments1[3]) + p \* (pow(moments2[0] - moments[0], 4) + 6 \* moments2[1] \* pow(moments2[0] - moments[0], 2) + 4 \* (moments2[0] - moments[0]) \* pow(moments2[1], 3. / 2.) \* moments2[2] + pow(moments2[1], 2) \* moments2[3])) - 3;

return moments;

}

double MixtureDistribution::get\_f(double x) {

return (1. - p) \* pd1.get\_f(x) + p \* pd2.get\_f(x);

}

double MixtureDistribution::get\_model() {

double r;

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

if (r > p) {

return pd1.get\_model();

}

return pd2.get\_model();

}

void MixtureDistribution::load(std::string p\_file, std::string pd1\_file, std::string pd2\_file) {

std::ifstream file;

std::string str\_p;

file.open(p\_file);

if (file.fail()) {

throw "ERROR: Bad file name";

}

if (!file.eof()) {

file >> str\_p;

}

else {

throw "ERROR: File should consist mix parameter";

}

file.close();

try {

p = stod(str\_p);

}

catch (std::invalid\_argument) {

throw "ERROR: p is not a number. Check your file and try again.";

}

pd1.load(pd1\_file);

pd2.load(pd2\_file);

}

void MixtureDistribution::save(std::string p\_file, std::string pd1\_file, std::string pd2\_file) {

std::ifstream check;

std::ofstream file;

check.open(p\_file);

if (check.fail()) {

throw "ERROR: Bad file name";

}

else {

check.close();

file.open(p\_file, std::ios::out);

file << p;

}

file.close();

pd1.save(pd1\_file);

pd2.save(pd2\_file);

}

void MixtureDistribution::generate\_distribution() {

std::ofstream file1, file2;

std::string s\_number;

double number;

file1.open("x.txt", std::ios::out);

file2.open("fx.txt", std::ios::out);

for (int i = 0; i < 10000; i++) {

number = get\_model();

s\_number = std::to\_string(number);

s\_number[s\_number.find(".")] = ',';

file1 << s\_number << "\n";

s\_number = std::to\_string(get\_f(number));

s\_number[s\_number.find(".")] = ',';

file2 << s\_number << "\n";

}

file1.close();

file2.close();

}

**emp.cpp:**

#include "Header.h"

EmpiricalDistribution::EmpiricalDistribution(int n0, PearsonDistributionType2& pd, int k0) : n(n0 > 1 ? n0 : throw "ERROR: Bad n. n should be > 1."), k(k0 > 1 ? k0 : int(log(double(n)) / log(double(2.))) + 1.), data(new double[n]), counter(new int[k]), frequency(new double[k]), density(new double[k]) {

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

data[i] = pd.get\_model();

}

get\_min\_delta();

get\_counter\_frequency\_density();

}

EmpiricalDistribution::EmpiricalDistribution(int n0, MixtureDistribution& mix, int k0) : n(n0 > 1 ? n0 : throw "ERROR: Bad n. n should be > 1."), k(k0 > 1 ? k0 : int(log(double(n)) / log(double(2.))) + 1.), data(new double[n]), counter(new int[k]), frequency(new double[k]), density(new double[k]) {

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

data[i] = mix.get\_model();

}

get\_min\_delta();

get\_counter\_frequency\_density();

}

EmpiricalDistribution::EmpiricalDistribution(int n0, EmpiricalDistribution& emp, int k0) : n(n0 > 1 ? n0 : throw "ERROR: Bad n. n should be > 1."), k(k0 > 1 ? k0 : int(log(double(n)) / log(double(2.))) + 1.), data(new double[n]), counter(new int[k]), frequency(new double[k]), density(new double[k]) {

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

data[i] = emp.get\_model();

}

get\_min\_delta();

get\_counter\_frequency\_density();

}

EmpiricalDistribution::EmpiricalDistribution(std::string filename) {

load(filename);

}

EmpiricalDistribution::EmpiricalDistribution(const EmpiricalDistribution& emp) {

n = emp.n;

k = emp.k;

min = emp.min;

delta = emp.delta;

data = new double[n];

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

data[i] = emp.data[i];

}

counter = new int[k];

frequency = new double[k];

density = new double[k];

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

counter[i] = emp.counter[i];

frequency[i] = emp.frequency[i];

density[i] = emp.density[i];

}

}

EmpiricalDistribution::~EmpiricalDistribution() {

delete[] data, counter, frequency, density;

}

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

if (this == &emp) return \*this;

if (n != emp.n) {

delete[] data;

n = emp.n;

data = new double[n];

}

if (k != emp.k) {

delete[] counter, frequency, density;

k = emp.k;

counter = new int[k];

frequency = new double[k];

density = new double[k];

}

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

data[i] = emp.data[i];

}

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

counter[i] = emp.counter[i];

frequency[i] = emp.frequency[i];

density[i] = emp.density[i];

}

get\_min\_delta();

return \*this;

}

void EmpiricalDistribution::get\_min\_delta() {

double max = data[0];

min = data[0];

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

if (data[i] > max) {

max = data[i];

}

if (data[i] < min) {

min = data[i];

}

}

delta = (max - min) / k;

}

void EmpiricalDistribution::get\_counter\_frequency\_density() {

bool flag;

counter = new int[k];

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

counter[i] = 0;

}

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

flag = false;

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

if (data[i] >= min + delta \* j && data[i] < min + delta \* (j + 1)) {

counter[j]++;

flag = true;

break;

}

}

if (!flag) {

counter[k - 1]++;

}

}

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

frequency[i] = counter[i] / (double)n;

density[i] = frequency[i] / delta;

}

}

double EmpiricalDistribution::get\_f(double x) {

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

if (x >= min + i \* delta && x < min + (i + 1) \* delta) {

return density[i];

}

}

return density[k - 1];

}

double EmpiricalDistribution::get\_model() {

double r1, r2, sum = 0;

do r1 = (double)rand() / RAND\_MAX; while (r1 == 0. || r1 == 1.);

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

sum += frequency[i];

if (sum > r1) {

if (i == k - 1) {

r2 = (double)rand() / RAND\_MAX;

}

else {

do r2 = (double)rand() / RAND\_MAX; while (r2 == 1.);

}

return min + delta \* i + r2 \* delta;

}

}

throw "ERROR: Something went wrong.";

}

double\* EmpiricalDistribution::get\_moments() {

double\* moments = new double[4];

for (int i = 0; i < 4; i++) {

moments[i] = 0;

}

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

moments[0] += data[i];

}

moments[0] /= (double)n;

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

moments[1] += pow(data[i] - moments[0], 2);

}

moments[1] /= (double)n;

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

moments[2] += pow(data[i] - moments[0], 3);

}

moments[2] /= ((double)n \* pow(moments[1], 3. / 2.));

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

moments[3] += pow(data[i] - moments[0], 4);

}

moments[3] = moments[3] / ((double)n \* pow(moments[1], 2)) - 3;

return moments;

}

void EmpiricalDistribution::load(std::string filename) {

std::ifstream file;

std::string temp;

int temp\_n;

double\* temp\_data;

file.open(filename);

if (file.fail()) {

throw "ERROR: Bad file name";

}

if (!file.eof()) {

file >> temp;

}

else {

throw "ERROR: File should consist n";

}

try {

temp\_n = stoi(temp);

}

catch (std::invalid\_argument) {

throw "ERROR: n is not a number, check your file and try again";

}

if (temp\_n <= 1) {

throw "ERROR: n should be > 1";

}

temp\_n = stoi(temp);

temp\_data = new double[temp\_n];

for (int i = 0; i < temp\_n; i++) {

if (file.eof()) {

throw "ERROR: Not enough numbers in data, check your file and try again";

}

file >> temp;

try {

temp\_data[i] = stod(temp);

}

catch (std::invalid\_argument) {

throw "ERROR: One of the numbers in data is not a number, check your file and try again";

}

temp\_data[i] = stod(temp);

}

file.close();

delete[] data;

delete[] counter;

delete[] frequency;

delete[] density;

n = temp\_n;

data = new double[n];

k = int(log(double(n)) / log(double(2.))) + 1.;

counter = new int[k];

frequency = new double[k];

density = new double[k];

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

data[i] = temp\_data[i];

}

get\_min\_delta();

get\_counter\_frequency\_density();

}

void EmpiricalDistribution::save(std::string filename) {

std::ifstream check;

std::ofstream file;

check.open(filename);

if (check.fail()) {

throw "ERROR: Bad file name";

}

check.close();

file.open(filename, std::ios::out);

file << n << std::endl;

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

file << data[i] << " ";

}

file.close();

}

void EmpiricalDistribution::generate\_distribution() {

std::ofstream file1, file2;

std::string s\_number;

double number;

file1.open("x.txt", std::ios::out);

file2.open("fx.txt", std::ios::out);

for (int i = 0; i < 10000; i++) {

number = get\_model();

s\_number = std::to\_string(number);

s\_number[s\_number.find(".")] = ',';

file1 << s\_number << "\n";

s\_number = std::to\_string(get\_f(number));

s\_number[s\_number.find(".")] = ',';

file2 << s\_number << "\n";

}

file1.close();

file2.close();

}

**test.cpp (без учета тестирующей функции основного распределения):**

#include "Header.h"

void mix\_test() {

PearsonDistributionType2 pd1(3., 5., 3.);

PearsonDistributionType2 pd2(-1., 4., 9.);

double\* moments;

double number;

std::cout << std::endl << std::endl << "\t\t\t\t----------MIX TEST----------" << std::endl;

//

std::cout << std::endl << std::endl << "\t\t----------test 1----------" << std::endl;

std::cout << "\t\t----------First constructor" << std::endl;

MixtureDistribution mix1(pd1, pd2);

if (mix1.get\_p() == 0.5) {

std::cout << ":D First constructor is correct." << std::endl;

std::cout << "First component: " << pd1.getMU() << " " << pd1.getLA() << " " << pd1.getNU() << std::endl;

std::cout << "Second component: " << pd2.getMU() << " " << pd2.getLA() << " " << pd2.getNU() << std::endl;

moments = mix1.get\_moments();

std::cout << "\tmoments: ";

for (int i = 0; i < 4; i++) {

std::cout << moments[i] << " ";

}

std::cout << std::endl;

}

else {

std::cout << ":C First constructor is incorrect." << std::endl;

}

//

std::cout << std::endl << std::endl << "\t\t----------test 2----------" << std::endl;

std::cout << "\t\t----------set\_p" << std::endl;

std::cout << "set p = 0.6" << std::endl;

mix1.set\_p(0.6);

std::cout << "p = " << mix1.get\_p() << std::endl;

//

std::cout << std::endl << std::endl << "\t\t----------test 3----------" << std::endl;

std::cout << "\t\t----------Enter wrong p (1.5) to constructor and \"set\_p()\"" << std::endl;

try {

MixtureDistribution mix2(pd1, pd2, 1.5);

}

catch (const char\* e) {

std::cout << e << std::endl;

}

try {

mix1.set\_p(1.5);

}

catch (const char\* e) {

std::cout << e << std::endl;

}

//

std::cout << std::endl << std::endl << "\t\t----------test 4----------" << std::endl;

std::cout << "\t\t----------Get and set mu, la, nu of Pearson distributions in mix distribution" << std::endl;

std::cout << "GET:" << std::endl;

std::cout << "mu of the first component: " << mix1.getComponent1().getMU() << std::endl;

std::cout << "la of the first component: " << mix1.getComponent1().getLA() << std::endl;

std::cout << "nu of the first component: " << mix1.getComponent1().getNU() << std::endl;

std::cout << "mu of the second component: " << mix1.getComponent2().getMU() << std::endl;

std::cout << "la of the second component: " << mix1.getComponent2().getLA() << std::endl;

std::cout << "nu of the second component: " << mix1.getComponent2().getNU() << std::endl << std::endl;

std::cout << "SET:" << std::endl;

mix1.getComponent1().setNU(2);

std::cout << "Set nu of the first component " << mix1.getComponent1().getNU() << std::endl;

mix1.getComponent1().setLA(8);

std::cout << "Set la of the first component " << mix1.getComponent1().getLA() << std::endl;

mix1.getComponent2().setMU(-4);

std::cout << "Set mu of the second component " << mix1.getComponent2().getMU() << std::endl;

mix1.getComponent2().setNU(6);

std::cout << "Set nu of the second component " << mix1.getComponent2().getNU() << std::endl << std::endl;

std::cout << "GET:" << std::endl;

std::cout << "mu of the first component: " << mix1.getComponent1().getMU() << std::endl;

std::cout << "la of the first component: " << mix1.getComponent1().getLA() << std::endl;

std::cout << "nu of the first component: " << mix1.getComponent1().getNU() << std::endl;

std::cout << "mu of the second component: " << mix1.getComponent2().getMU() << std::endl;

std::cout << "la of the second component: " << mix1.getComponent2().getLA() << std::endl;

std::cout << "nu of the second component: " << mix1.getComponent2().getNU() << std::endl << std::endl;

std::cout << "WROND SET:" << std::endl;

std::cout << "Try to set 100000 to mu of the first component:" << std::endl;

try {

mix1.getComponent1().setMU(100000);

}

catch (const char\* e) {

std::cout << e << std::endl;

}

std::cout << "Try to set -4 to la of the second component:" << std::endl;

try {

mix1.getComponent2().setLA(-4);

}

catch (const char\* e) {

std::cout << e << std::endl;

}

//

std::cout << std::endl << std::endl << "\t\t----------test 5----------" << std::endl;

std::cout << "\t\t----------get\_f, get\_model" << std::endl;

std::cout << "get\_f(" << 3 << ") = " << mix1.get\_f(3) << std::endl;

std::cout << "get\_f(" << 0 << ") = " << mix1.get\_f(0) << std::endl;

std::cout << "get\_f(" << 20 << ") = " << mix1.get\_f(20) << std::endl << std::endl;

for (int i = 0; i < 5; i++) {

number = mix1.get\_model();

if (number >= mix1.getComponent1().getMU() - mix1.getComponent1().getLA() && number <= mix1.getComponent1().getMU() + mix1.getComponent1().getLA() || number >= mix1.getComponent2().getMU() - mix1.getComponent2().getLA() && number <= mix1.getComponent2().getMU() + mix1.getComponent2().getLA()) {

std::cout << ":D get\_model(" << mix1.getComponent1().getMU() << ", " << mix1.getComponent1().getLA() << ", " << mix1.getComponent1().getNU() << ", " << mix1.getComponent2().getMU() << ", " << mix1.getComponent2().getLA() << ", " << mix1.getComponent2().getNU() << ", p = " << mix1.get\_p() << ") = " << number << " correct." << std::endl;

}

else {

std::cout << ":C get\_model(" << mix1.getComponent1().getMU() << ", " << mix1.getComponent1().getLA() << ", " << mix1.getComponent1().getNU() << ", " << mix1.getComponent2().getMU() << ", " << mix1.getComponent2().getLA() << ", " << mix1.getComponent2().getNU() << ", p = " << mix1.get\_p() << ") = " << number << " incorrect" << std::endl;

}

}

std::cout << std::endl;

//

std::cout << std::endl << std::endl << "\t\t----------test 6----------" << std::endl;

std::cout << "\t\t----------save, load" << std::endl;

std::cout << "SAVE: p - file1.txt, first component - file2.txt, second component - file3.txt" << std::endl << std::endl;

mix1.save("file1.txt", "file2.txt", "file3.txt");

MixtureDistribution mix2("file1.txt", "file2.txt", "file3.txt");

std::cout << "LOAD: p - file1.txt, first component - file2.txt, second component - file3.txt" << std::endl;

std::cout << "p: " << mix2.get\_p() << std::endl;

std::cout << "mu of the first component: " << mix2.getComponent1().getMU() << std::endl;

std::cout << "la of the first component: " << mix2.getComponent1().getLA() << std::endl;

std::cout << "nu of the first component: " << mix2.getComponent1().getNU() << std::endl;

std::cout << "mu of the second component: " << mix2.getComponent2().getMU() << std::endl;

std::cout << "la of the second component: " << mix2.getComponent2().getLA() << std::endl;

std::cout << "nu of the second component: " << mix2.getComponent2().getNU() << std::endl;

std::cout << std::endl << std::endl << "\t\t\t\t----------MIX TEST----------" << std::endl;

}

void emp\_test() {

PearsonDistributionType2 pd1(3., 5., 3.);

PearsonDistributionType2 pd2(-1., 4., 5.);

MixtureDistribution mix1(pd1, pd2, 0.6);

double number;

double\* moments;

std::cout << std::endl << std::endl << "\t\t\t\t----------EMP TEST----------" << std::endl;

//

std::cout << std::endl << std::endl << "\t\t----------test 1----------" << std::endl;

std::cout << "\t\t----------First constructor" << std::endl;

std::cout << "(Pearson distribution)" << std::endl;

EmpiricalDistribution emp1(10000, pd1);

//emp1.generate\_distribution();

//pd1.generate\_distribution();

std::cout << "Check emp\_test1\_1.png" << std::endl << std::endl;

std::cout << "(Mixture distribution)" << std::endl;

EmpiricalDistribution emp2(10000, mix1);

//emp2.generate\_distribution();

//mix1.generate\_distribution();

std::cout << "Check emp\_test1\_2.png" << std::endl << std::endl;

std::cout << "(Empirical distribution)" << std::endl;

EmpiricalDistribution emp3(10000, emp2);

//emp3.generate\_distribution();

//emp2.generate\_distribution();

std::cout << "Check emp\_test1\_3.png" << std::endl << std::endl;

//

std::cout << std::endl << std::endl << "\t\t----------test 2----------" << std::endl;

std::cout << "\t\t----------First constructor, set k = 3" << std::endl;

EmpiricalDistribution emp4(10000, pd2, 3);

//emp4.generate\_distribution();

//pd2.generate\_distribution();

std::cout << "Check emp\_test2\_1.png" << std::endl << std::endl;

//

std::cout << std::endl << std::endl << "\t\t----------test 3----------" << std::endl;

std::cout << "\t\t----------get\_moments, get\_f, get\_model" << std::endl;

std::cout << "mu of Pearson distribution: " << pd1.getMU() << std::endl;

std::cout << "la of Pearson distribution: " << pd1.getLA() << std::endl;

std::cout << "nu of Pearson distribution: " << pd1.getNU() << std::endl << std::endl;

std::cout << "GET\_MOMENTS" << std::endl;

std::cout << "Moments of Pearson distribution:" << std::endl;

std::cout << pd1.get\_moments()[0] << " " << pd1.get\_moments()[1] << " " << pd1.get\_moments()[2] << " " << pd1.get\_moments()[3] << std::endl;

moments = emp1.get\_moments();

std::cout << "Moments of copied version of Pearson distribution:" << std::endl;

std::cout << moments[0] << " " << moments[1] << " " << moments[2] << " " << moments[3] << std::endl << std::endl;

std::cout << "GET\_F" << std::endl;

std::cout << "get\_f of Pearson distribution:" << std::endl;

std::cout << "get\_f(" << 2 << ") = " << pd1.get\_f(2) << std::endl;

std::cout << "get\_f(" << -1 << ") = " << pd1.get\_f(-1) << std::endl;

std::cout << "get\_f(" << 10 << ") = " << pd1.get\_f(10) << std::endl;

std::cout << "get\_f of copied version of Pearson distribution:" << std::endl;

std::cout << "get\_f(" << 2 << ") = " << emp1.get\_f(2) << std::endl;

std::cout << "get\_f(" << -1 << ") = " << emp1.get\_f(-1) << std::endl;

std::cout << "get\_f(" << 10 << ") = " << emp1.get\_f(10) << std::endl << std::endl;

std::cout << "GET\_MODEL" << std::endl;

for (int i = 0; i < 5; i++) {

number = emp1.get\_model();

if (number >= pd1.getMU() - pd1.getLA() && number <= pd1.getMU() + pd1.getLA()) {

std::cout << ":D empirical get\_model of pearson distribution (" << pd1.getMU() << ", " << pd1.getLA() << ", " << pd1.getNU() << ") = " << number << " correct: " << pd1.getMU() - pd1.getLA() << " <= " << number << " <= " << pd1.getMU() + pd1.getLA() << std::endl;

}

else {

std::cout << ":C empirical get\_model of pearson distribution (" << pd1.getMU() << ", " << pd1.getLA() << ", " << pd1.getNU() << ") = " << number << " incorrect" << std::endl;

}

}

std::cout << std::endl;

//

std::cout << std::endl << std::endl << "\t\t----------test 4----------" << std::endl;

std::cout << "\t\t----------load, save" << std::endl;

std::cout << "SAVE IN file1.txt:" << std::endl;

emp1.save("file1.txt");

std::cout << "Check file1.txt." << std::endl << std::endl;

std::cout << "LOAD FROM file1.txt:" << std::endl;

emp2.load("file1.txt");

std::cout << "moments:" << std::endl;

moments = emp1.get\_moments();

std::cout << moments[0] << " " << moments[1] << " " << moments[2] << " " << moments[3] << std::endl << std::endl;

std::cout << "TRY TO LOAD FROM BAD FILES:" << std::endl;

std::string filenames[] = { "emp\_test\_1.txt", "emp\_test\_2.txt", "emp\_test\_3.txt", "emp\_test\_4.txt" };

for (int i = 0; i < 4; i++) {

try {

emp2.load(filenames[i]);

}

catch (const char\* e) {

std::cout << e << std::endl;

}

}

//

std::cout << std::endl << std::endl << "\t\t----------test 5----------" << std::endl;

std::cout << "\t\t----------\operator \"=\" and copy constructor" << std::endl;

std::cout << "OPERATOR \"=\":" << std::endl;

EmpiricalDistribution emp5(1000, pd2);

emp5 = emp1;

moments = emp1.get\_moments();

std::cout << "Moments of start empirical distribution:" << std::endl;

std::cout << moments[0] << " " << moments[1] << " " << moments[2] << " " << moments[3] << std::endl;

moments = emp5.get\_moments();

std::cout << "Moments of copied version of start empirical distribution:" << std::endl;

std::cout << moments[0] << " " << moments[1] << " " << moments[2] << " " << moments[3] << std::endl << std::endl;

std::cout << "COPY CONSTRUCTOR:" << std::endl;

EmpiricalDistribution emp6 = emp1;

moments = emp1.get\_moments();

std::cout << "Moments of start empirical distribution:" << std::endl;

std::cout << moments[0] << " " << moments[1] << " " << moments[2] << " " << moments[3] << std::endl;

moments = emp6.get\_moments();

std::cout << "Moments of copied version of start empirical distribution:" << std::endl;

std::cout << moments[0] << " " << moments[1] << " " << moments[2] << " " << moments[3] << std::endl << std::endl;

std::cout << std::endl << std::endl << "\t\t\t\t----------EMP TEST----------" << std::endl;

}

**main.cpp:**

#include "Header.h"

int main() {

srand(time(NULL));

try {

test();

mix\_test();

emp\_test();

}

catch (const char\* e) {

std::cout << e;

}

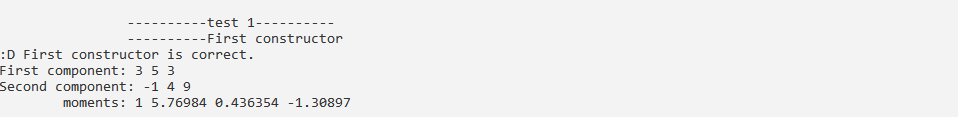
return 0;

}

**Тестирование программы:**

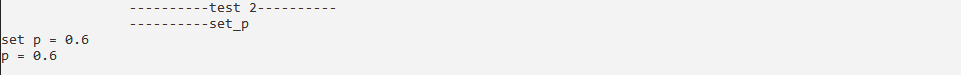
**Вызов функции mix\_test():**

**1. Проверка первого конструктора:**

****

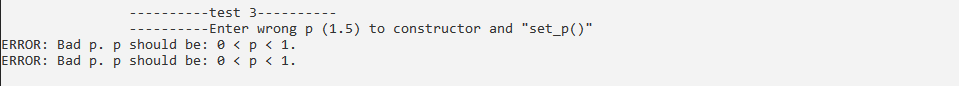
По умолчанию p должен быть 0.5. Проверяется значение атрибута p, если оно равно 0.5, значит конструктор работает правильно. Далее выводятся значения компонентов распределения, а также выводятся значения числовых характеристик для распределения смеси.

**2. Проверка сеттера и геттера атрибута p:**

****

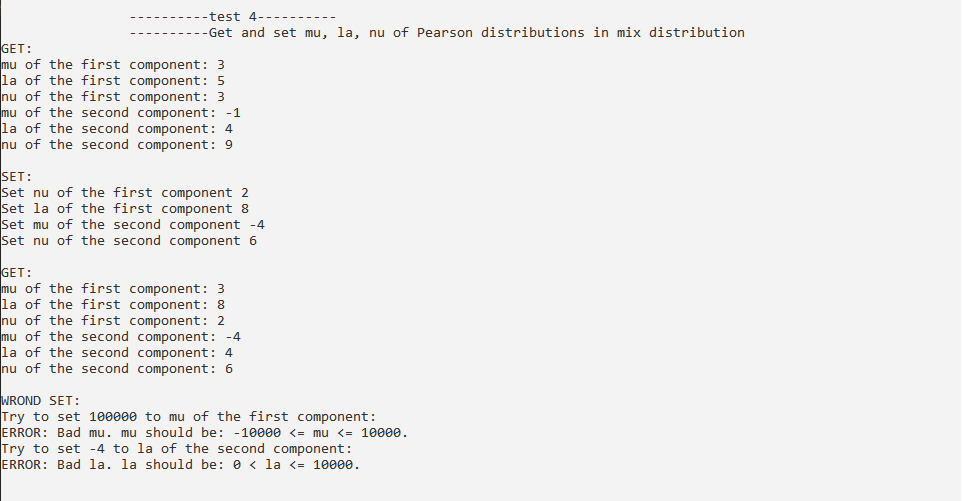
С помощью сеттера пытаемся сделать значение p равным 0.6, а затем с помощью геттера выводим значение p.

**3. Попытка вставить неправильные значения для p через конструктор и через сеттер:**

****

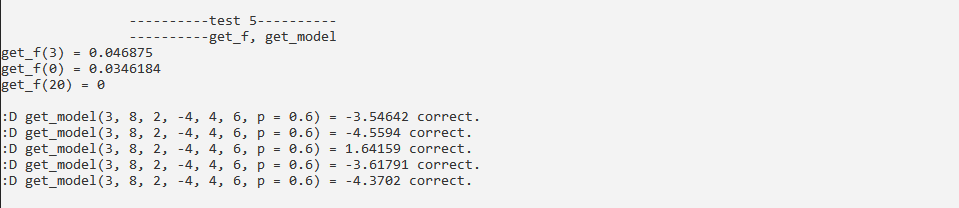
Пытаемся ввести неправильное значение для p (1.5) через конструктор и через сеттер. Получаем сообщения об ошибках.

**4. Попытка обращения к геттерам и сеттерам объектов основного распределения через функции getComponent1() и getComponent2():**

****

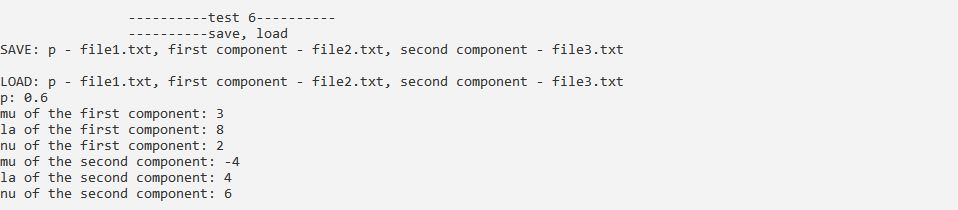
С помощью геттеров получаем значения атрибутов объектов основного распределения, затем меняем некоторые атрибуты с помощью сеттеров и снова получаем значения атрибутов объектов основного распределения с помощью геттеров. Затем с помощью сеттеров пытаемся поменять некоторые атрибуты на неправильные значения, получаем сообщения об ошибках.

**5. Получение функции плотности и моделирование случайной величины распределения смеси:**

****

Получаем значения функции плотности в некоторых точках, затем моделируем несколько случайных величин и проверяем, лежат ли они в пределах одного из двух основных распределений, если да, то тест пройден.

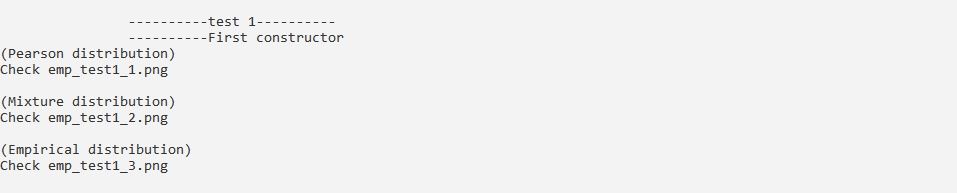
**6. Проверка функций save() и load():**



Пытаемся сохранить атрибуты распределения смеси в файл, затем из тех же файлов с помощью конструктора загрузки создаем новый объект класса распределения смеси, проверяем атрибуты нового объекта, они получились такие же, как и у первого объекта.

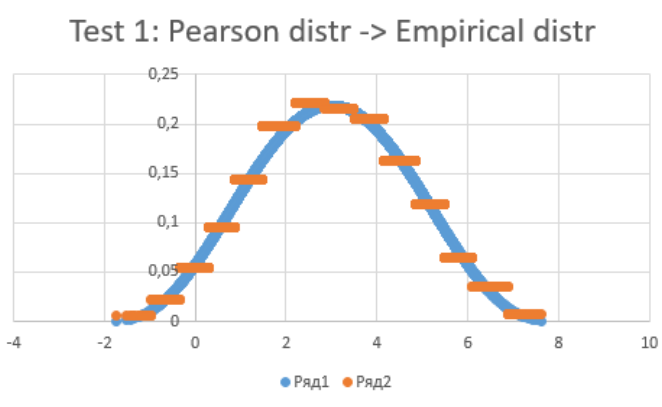
**Вызов функции emp\_test():**

**1. Проверка первых трех конструкторов, k вычисляется по формуле по умолчанию:**

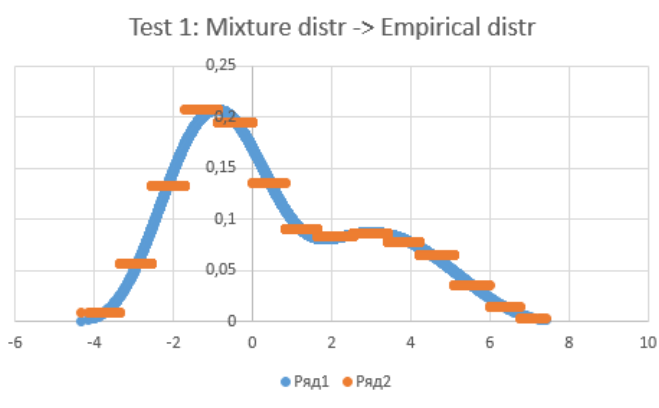


Происходит генерация эмпирического распределения на основе других распределений, с помощью Excel создаются графики полученного эмпирического распределения и того распределения, на основе которого генерировалось эмпирическое распределение. Данные графики в виде картинки можно посмотреть, открыв тот или иной файл.

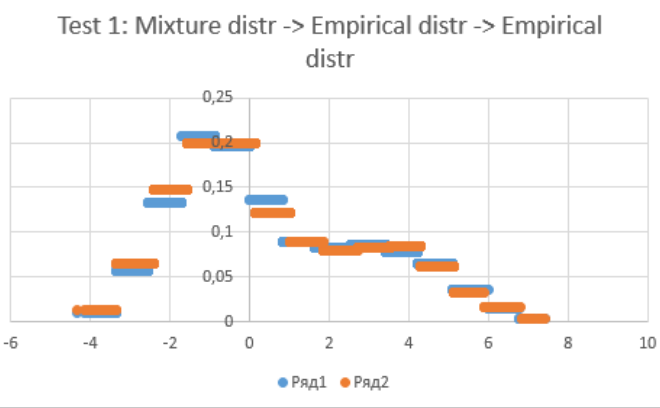
emp\_test1\_1.png - генерация эмпирического распределения на основе основного распределения:



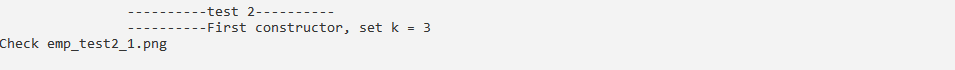
emp\_test1\_2.png - генерация эмпирического распределения на основе распределения смеси:



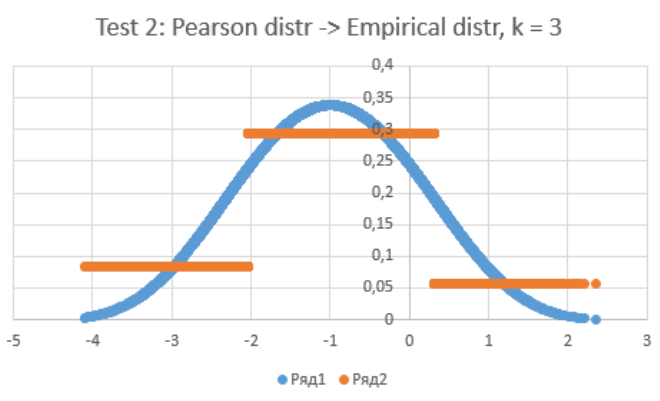
emp\_test1\_3.png - генерация эмпирического распределения на основе эмпирического распределения, сгенерированного на основе распределения смеси:



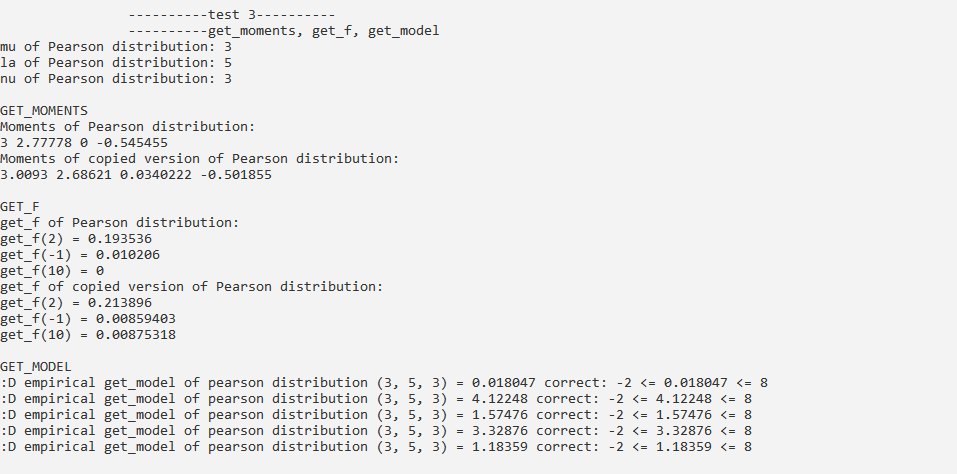
**2. Проверка одного из этих конструкторов, k задается пользователем:**

****

Происходит генерация эмпирического распределения на основе основного распределения, задается k = 3, полученные данные в Excel преобразуются в график. Полученный результат можно посмотреть, открыв файл emp\_test2\_1.png:

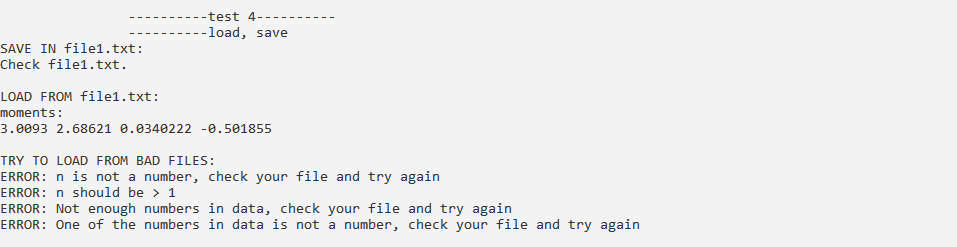


**3. Получение числовых характеристик эмпирического распределения, получение функции плотности в некоторых точках, моделирование случайной величины:**

****

Эмпирическое распределение генерируется на основе основного распределения. Сначала выводятся числовые характеристики основного распределения, затем числовые характеристики основного распределения. Как можно увидеть, значения сходятся. Затем выводятся значения функции плотности основного распределения в некоторых точках, а также значения функции плотности эмпирического распределения в этих же точках. Значения также достаточно похожи. В конце выводятся результаты моделирования функции. Происходит несколько вызовов функции моделирования, если случайная величина выходит за пределы стандартного распределения, значит есть какая-то ошибка, у нас такого не происходит, ошибок нет.

**4. Проверка функций save() и load():**



Сначала происходит сохранение эмпирического распределения в файл, затем происходит загрузка эмпирического распределения, числовые характеристики эмпирического распределения сохранились, что означает правильность работы функций загрузки и сохранения. После этого мы пытаемся загрузить данные о распределении из неправильно введенных файлов:

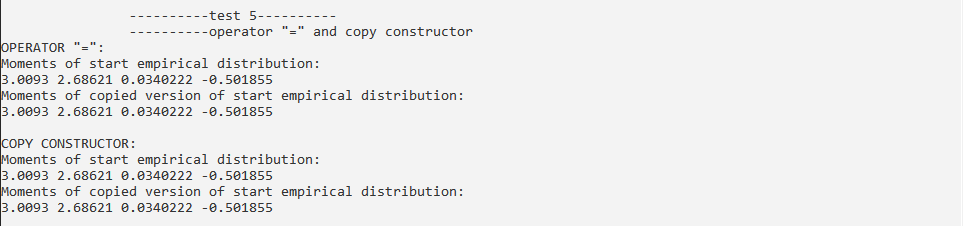
1. Файл "emp\_test\_1.txt" содержит строковое значение на месте, где должно быть значение числа элементов выборки n.

2. Файл "emp\_test\_2.txt" содержит число, которое меньше или равно единице, на месте, где должно быть значение числа элементов выборки n.

3. Файл "emp\_test\_3.txt" содержит недостаточно элементов выборки (n > реального количества элементов).

4. В файле "emp\_test\_4.txt" один из элементов выборки не является числом.

**5. Проверка оператора присваивания и конструктора копирования:**

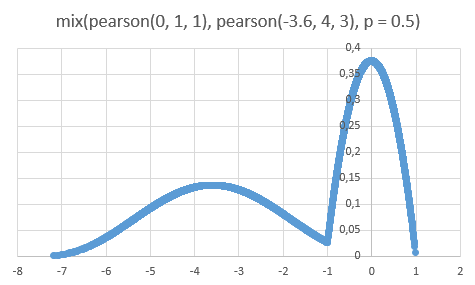


Сначала происходит проверка оператора присваивания. К уже существующему эмпирическому распределению присваивается другое распределение, после сравниваются числовые характеристики обоих распределений. Они равны.

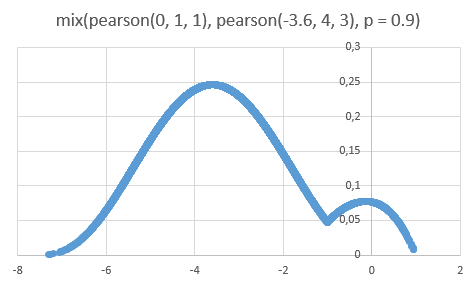
Затем происходит проверка конструктора копирования. Создается объект эмпирического распределения и к нему присваивается уже существующее распределение, после сравниваются числовые характеристики обоих распределений. Они равны.

**Графики:**

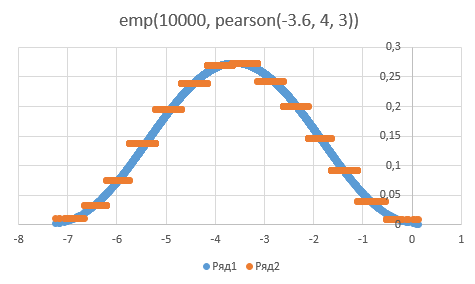
**1. Распределение смеси для распределений Пирсона с атрибутами (0, 1, 1) и (-3.6, 4, 3) и параметром смеси 0.5:**



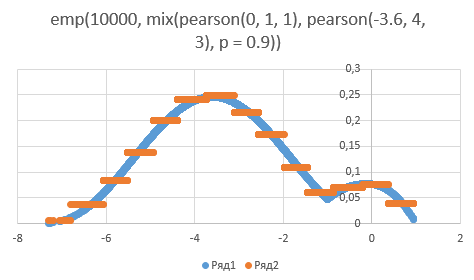
**2. Распределение смеси для распределений Пирсона с атрибутами (0, 1, 1) и (-3.6, 4, 3) и параметром смеси 0.9:**

****

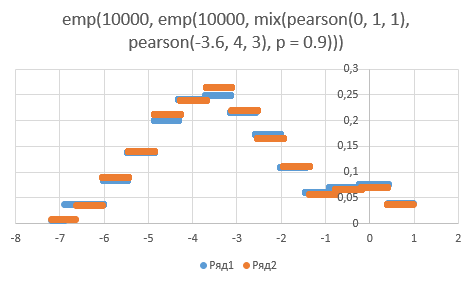
**3. Эмпирическое распределение для распределения Пирсона с атрибутами (-3.6, 4, 3):**

****

**4. Эмпирическое распределение для распределения смеси для распределений Пирсона с атрибутами (0, 1, 1) и (-3.6, 4, 3) и параметром смеси 0.9:**

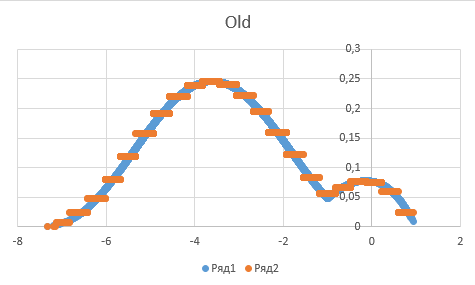
****

**5. Эмпирическое распределение для эмпирического распределения для распределения смеси для распределений Пирсона с атрибутами (0, 1, 1) и (-3.6, 4, 3) и параметром смеси 0.9:**

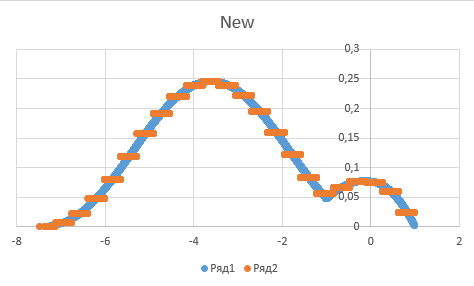
****

**6. Сравнение результатов текущей программы и программы, сделанной в лабораторной работе №1 (распределение смеси для распределений Пирсона с атрибутами (0, 1, 1) и (-3.6, 4, 3) и параметром смеси 0.9, размер выборки 5120000):**

Результаты старой программы:

****

Результаты новой программы:



**Результаты совпадают.**

**Контрольные вопросы проработаны.**