Новосибирский государственный технический университет

Кафедра ТПИ

Объектно-ориентированное программирование

Лабораторная работа №1

**Сложность программного обеспечения**

Выполнили: Преподаватели:

Яшков Иван Еланцева Ирина Леонидовна

Гавриленко Андрей Лисицин Даниил Валерьевич

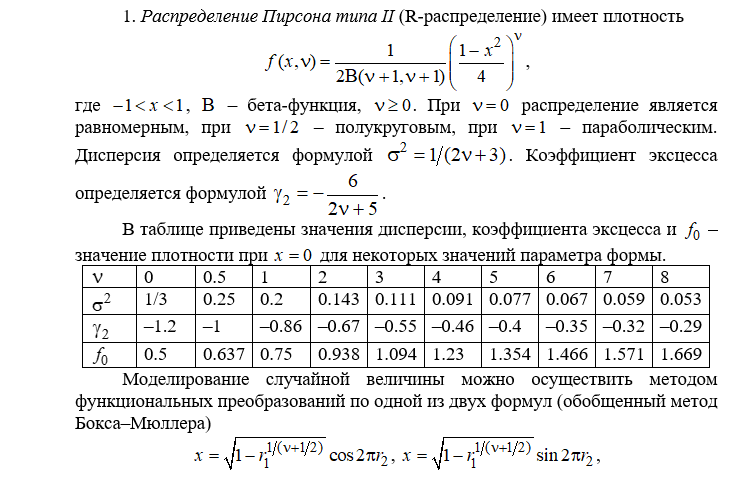
Группа:

ПМИ-13

Бригада 1

Новосибирск, 2023

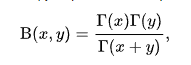
**Задание (вариант 1):**

В данной лабораторной работе необходимо реализовать возможность работы с тремя распределениями, алгоритм работы которых будет описан далее. Также в каждом распределении указываются дополнительные характеристики, такие как: μ (далее mu) - коэффициент сдвига, показывает, насколько сдвигается область значений распределения; λ (далее la) - коэффициент масштаба, показывает, во сколько раз увеличивается область значений распределения; ν (далее nu) - параметр формы.

**1) Обычное распределение:**

В функции плотности распределения Пирсона типа II используется бета-функция, которую было решено вычислять через гамма-функцию по следующей формуле:



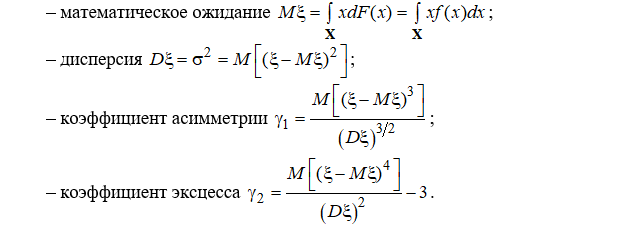
Гамма-функция подключается с помощью библиотеки "cmath". Для получения функции плотности, зависящей от коэффициентов сдвига и масштаба, воспользуемся формулой:



Таким образом, получение значения функции плотности от аргумента x, сдвига mu, масштаба la и формы nu можно описать функцией "get\_f()":

.

Числовые характеристики для распределения Пирсона типа II находим по формулам:



Для нахождения математического ожидания необходимо вычислить интеграл от функции плотности, которая содержит бета-функцию. Чтобы решить данную задачу, вычислим интеграл численно, использую квадратуру парабол.

Математическое ожидание и дисперсия, в отличие от коэффициента асимметрии и эксцесса, зависят от коэффициентов сдвига и масштаба, поэтому к математическому ожиданию необходимо добавлять mu, а дисперсию необходимо умножать на квадрат la.

Для моделирования случайной величины распределения Пирсона типа II воспользуемся методом функциональных преобразований по формуле обобщенного метода Бокса-Мюллера:





при этом, чтобы случайная величина зависела от mu и la, моделирование следует осуществлять сдвиг-масштабное преобразование:



Таким образом, результат моделирования (аргумент x), зависящий от mu, la, nu можно описать функцией get\_model():

.

**2) Распределение смеси:**

В смеси распределений есть особые параметры смеси p. В работе смесь производится на основе двух распределений Пирсона типа II, поэтому имеем два параметра смеси, , .

Для функции плотности распределения смеси используем формулу:

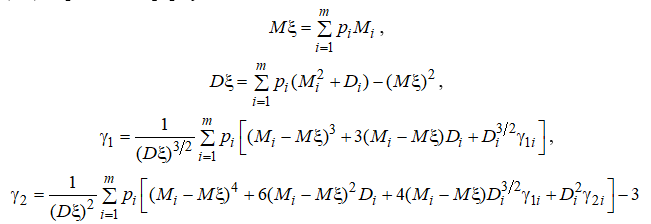


Зная, что параметров смеси всего два и они зависят от одного числа p, можно составить функцию плотности распределения смеси "get\_mix\_f()":

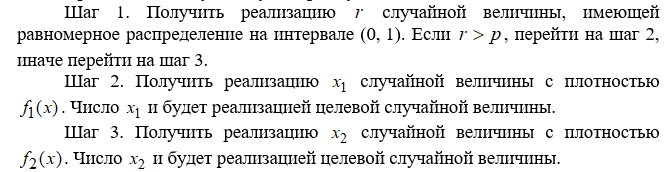
,

где args - массив из 6-ти элементов, хранящий коэффициенты mu, la, nu для обоих распределений (первые три элемента - коэффициенты для первого распределения, последние три элемента - коэффициенты для второго распределения).

Числовые характеристики для смеси распределения Пирсона типа II находим по формулам:



Для моделирования случайной величины смеси распределений Пирсона типа II также воспользуемся методом функциональных преобразований по формуле обобщенного метода Бокса-Мюллера, по следующему алгоритму:

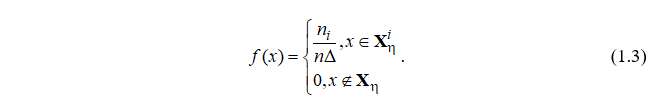


**3) Эмпирическое распределение:**

Функция плотности эмпирического распределения имеет особый вид - она представляется в виде гистограммы. Вся область определения разбивается на k промежутков, на каждом из которых функция сохраняет постоянное значение, равное количеству элементов в промежутке, деленное на общее количество элементов и на длину промежутка, которая находится по формуле:



Функция плотности эмпирического распределения равна:



Где - количество элементов в промежутке, - общее количество элементов, - дина отрезка.

Для нахождения числа k воспользуемся формулой Стёрджеса:

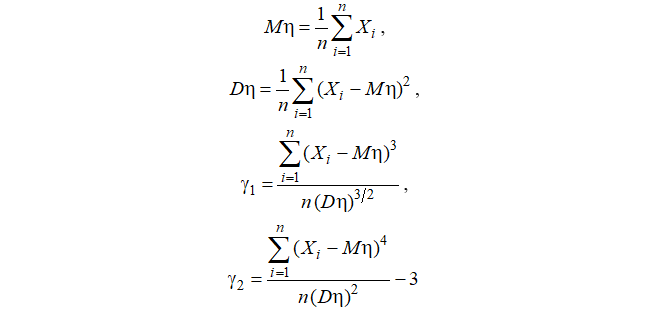
Назовем количество элементов в промежутке - counter, относительную частоту - frequency, плотность в конкретном промежутке - density.

Для получения функции плотности эмпирического распределения необходимо знать все плотности density, а также k, Δ и минимальное значение конкретного промежутка min.

Для получения плотностей на всех промежутках "density" необходимо знать количество элементов, входящих на каждый промежуток counter, количество всех элементов n, k и Δ.

Для получения количества элементов, входящих на каждый отрезок "counter", необходимо, зная количество промежутков и их длину, за каждый элемент, входящий в тот или иной промежуток, добавлять единицу в массиве counter.

Числовые характеристики для эмпирического распределения находим по формулам:



Для моделирования случайной величины эмпирического распределения будем генерировать случайную величину , в цикле начнем складывать относительные частоты frequency, и как только случайная величина станет меньше суммы некоторых относительных частот, проверим на какой итерации суммирования случайная величина стала меньше суммы, если на последней, то сгенерируем новую случайную величину , если не на последней, то сгенерируем случайную величину . Возвращать моделирующая функция будет значение, лежащее в том отрезке, на котором остановилось суммирование frequency, при этом позиция x в отрезке будет определяться величиной .

В программе также используются несколько вспомогательных функций, не относящиеся к распределениям:

check\_variables(mode) - специальная функция для ввода данных с клавиатуры, проверяет правильность ввода той или иной переменной, в случае неправильного ввода предлагает ввести значение еще раз. У функции есть параметр mode - модификация, задающая область допустимых значений для ввода разных параметров. Например, параметр mu ограничивается следующим образом: , la - и т. д. Также в этой же функции стоит проверка на ввод символьных значений вместо числовых.

write\_in\_file(number) - функция, которая преобразует числовое значение, полученное в каком-либо распределении, так, чтобы это значение Excel воспринимал как числовое. В C++ целая часть от дробной отделяется знаком '.', а в Excel - ',', поэтому для последующего построения графиков необходимо преобразовывать все дробные значения в соответствии со стандартами Excel.

Отдельно выделена функция test(), необходимая для тестирования сразу всех распределений. Функция последовательно вызывает срабатывание тестов из методических указаний.

**Текст программы:**

**Header.h** - заголовочный файл; **main.cpp** - файл, хранящий функцию main(), запускающую программу и содержащую пользовательские диалоги; **default.cpp** - файл, хранящий функции, связанные с обычным распределением,

**mix.cpp -** файл, хранящий функции, связанные с распределением смеси, **empirical.cpp -** файл, хранящий функции, связанные с эмпирическим распределением, **supporting\_functions.cpp -** файл, хранящий вспомогательные функции, **test.cpp -** файл, хранящий функцию test().

**Header.h:**

#pragma once

// Распределение Пирсона типа II (R-распределение)

// f(x, nu) = 1/(2B(nu+1,nu+1))\*((1-x^2)/4)^nu

// -1<=x<=1, nu>=0, v=0 равномерное, nu=0.5 полукруговое, nu=1 параболическое

// D=1/(2nu+3), y2=-6/(2nu+5)

#include <fstream>

#include <cmath>

#include <iostream>

#include <fstream>

#include <string>

#include <time.h>

const double PI = acos(-1);

const double EPS = pow(10, -2);

double check\_variables(int mode);

void enter(double\* args, int size);

std::string write\_in\_file(double num);

double get\_f(double x, double mu, double la, double nu);

double\* get\_moments(double mu, double la, double nu);

double get\_model(double mu, double la, double nu);

double check\_x(double mu, double la);

double get\_mix\_f(double x, double\* args, double mix\_param);

double\* get\_mix\_moments(double\* args, double mix\_param);

double get\_mix\_model(double\* args, double mix\_param);

double randomize\_X();

int get\_k(int n);

double get\_min(double\* X, int n);

double get\_delta(double\* X, int n, int k);

int\* get\_empirical\_counter(double\* X, double delta, int n, int k, double min);

double\* get\_empirical\_frequency(int\* counter, int n, int k);

double\* get\_empirical\_density(int\* counter, int n, int k, double delta);

double get\_empirical\_f(double x, double\* density, int k, double delta, double min);

double\* get\_empirical\_moments(double\* X, int n);

double get\_empirical\_model(double\* frequency, double delta, int k, double min);

void test();

**main.cpp:**

#include "Header.h"

int main() {

char mode = 'a';

//args: 0 - mu1, 1 - la1, 2 - nu1;

//args: 3 - mu2, 4 - la2, 5 - nu2;

double\* args, \*moments;

double x, \*X;

int n;

srand(time(NULL));

while (mode != 'd' && mode != 'm' && mode != 'e' && mode != 't') {

std::cout << "Enter one of the available modes (d - default, m - mix, e - emperical, t - test): ";

std::cin >> mode;

}

if (mode == 'd') {

args = new double[3];

enter(args, 3);

std::cout << "Enter x: ";

x = check\_x(args[0], args[1]);

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_f(x)\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

std::cout << "f(" << x << ") = " << get\_f(x, args[0], args[1], args[2]) << std::endl;

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_f(x)\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_M, D, y1, y2\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

moments = get\_moments(args[0], args[1], args[2]);

std::cout << "M = " << moments[0] << std::endl;

std::cout << "D = " << moments[1] << std::endl;

std::cout << "y1 = " << moments[2] << std::endl;

std::cout << "y2 = " << moments[3] << std::endl;

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_M, D, y1, y2\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_model\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

std::cout << "x(" << args[0] << ", " << args[1] << ", " << args[2] << ") = " << get\_model(args[0], args[1], args[2]) << std::endl;

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_model\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

std::cout << "Enter n: ";

std::cin >> n;

//X = new double[n];

std::ofstream x\_d\_data, d\_data;

x\_d\_data.open("x\_d\_data.txt");

d\_data.open("d\_data.txt");

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

x = get\_model(args[0], args[1], args[2]);

x\_d\_data << write\_in\_file(x) << std::endl;

d\_data << write\_in\_file(get\_f(x, args[0], args[1], args[2])) << std::endl;

}

x\_d\_data.close();

d\_data.close();

}

else if (mode == 'm') {

double param;

args = new double[6];

enter(args, 6);

std::cout << "Enter p: ";

param = check\_variables(4);

std::cout << "Enter x: ";

x = check\_variables(3);

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_mix f(x)\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

std::cout << "(p = " << param <<") mix\_f(" << x << ") = " << get\_mix\_f(x, args, param) << std::endl;

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_mix f(x)\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_mix M, D, y1, y2\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

moments = get\_mix\_moments(args, param);

std::cout << "(p = " << param << ") mix M = " << moments[0] << std::endl;

std::cout << "(p = " << param << ") mix D = " << moments[1] << std::endl;

std::cout << "(p = " << param << ") mix y1 = " << moments[2] << std::endl;

std::cout << "(p = " << param << ") mix y2 = " << moments[3] << std::endl;

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_mix M, D, y1, y2\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_mix model\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

std::cout << "(p = " << param << ") mix x(" << args[0] << ", " << args[1] << ", " << args[2] << ", " << args[3] << ", " << args[4] << ", " << args[5] << ") = " << get\_mix\_model(args, param) << std::endl;

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_mix model\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

}

else if (mode == 'e') {

int\* counter, k;

double delta, \* frequency, \* density, min, max, mu, la, nu;

std::cout << "Enter n: ";

n = check\_variables(5);

while (mode != 'r' && mode != 'd') {

std::cout << "Enter one of the available modes (r - random, d - default): ";

std::cin >> mode;

}

//args[i] - X[i]

args = new double[n];

if (mode == 'r') {

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

args[i] = randomize\_X();

}

}

else {

std::cout << "Enter mu, lambda, nu: ";

std::cin >> mu >> la >> nu;

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

args[i] = get\_model(mu, la, nu);

}

}

k = get\_k(n);

std::cout << "k = " << k << std::endl;

min = get\_min(args, n);

delta = get\_delta(args, n, k);

counter = get\_empirical\_counter(args, delta, n, k, min);

frequency = get\_empirical\_frequency(counter, n, k);

density = get\_empirical\_density(counter, n, k, delta);

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

std::cout << counter[i] << " " << frequency[i] << " " << density[i] << std::endl;

}

std::cout << "Enter x: ";

x = check\_variables(3);

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_empirical f(x)\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

std::cout << "empirical\_f(" << x << ") = " << get\_empirical\_f(x, density, k, delta, min) << std::endl;

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_empirical f(x)\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_empirical M, D, y1, y2\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

moments = get\_empirical\_moments(args, n);

std::cout << "empirical M = " << moments[0] << std::endl;

std::cout << "empirical D = " << moments[1] << std::endl;

std::cout << "empirical y1 = " << moments[2] << std::endl;

std::cout << "empirical y2 = " << moments[3] << std::endl;

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_empirical M, D, y1, y2\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_empirical model\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

std::cout << "(delta = " << delta << ", k = " << k <<", min = " << min << ") empirical x = " << get\_empirical\_model(frequency, delta, k, min) << std::endl;

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_empirical model\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

}

else if (mode == 't') {

test();

}

return 0;

}

**default.cpp:**

#include "Header.h"

double get\_f(double x, double mu, double la, double nu) {

if (x > -1. \* la + mu && x < 1. \* la + mu) {

return pow((1 - pow((x - mu) / la, 2)) / 4., nu) / (2 \* std::tgamma(nu + 1) \* std::tgamma(nu + 1) / std::tgamma(2 \* nu + 2)) / la;

}

return 0;

}

double\* get\_moments(double mu, double la, double nu) {

double\* moments;

moments = new double[4];

moments[0] = 1. / 3. \* (-1 \* get\_f(-1, 0, 1, nu) + 1 \* get\_f(1, 0, 1, nu)) + mu;

moments[1] = pow(la, 2) / (2 \* nu + 3);

moments[2] = 1 / pow(moments[1], 3. / 2.) \* 1 / 3 \* (-1 \* get\_f(1, 0, 1, nu) + 1 \* get\_f(1, 0, 1, nu));

moments[3] = -6. / (2 \* nu + 5);

return moments;

}

double get\_model(double mu, double la, double nu) {

double r1, r2;

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

do r2 = (double)rand() / RAND\_MAX; while (r2 == 0. || r2 == 1.);

return sqrt(1 - pow(r1, 1 / (nu + 1. / 2.))) \* cos(2 \* PI \* r2) \* la + mu;

}

double check\_x(double mu, double la) {

std::string temp;

double result;

while (true) {

std::cin >> temp;

try {

result = stod(temp);

}

catch (std::invalid\_argument) {

std::cout << "You should enter only number and \'.\' for real numbers ";

continue;

}

if (!(result >= -1. \* la + mu && result <= 1. \* la + mu)) {

std::cout << "x should be more or equal to " << -1. \* la + mu << " and less or equal to " << 1. \* la + mu << ". Try again: ";

continue;

}

break;

}

return result;

}

**mix.cpp:**

#include "Header.h"

double get\_mix\_f(double x, double\* args, double mix\_param) {

return (1. - mix\_param) \* get\_f(x, args[0], args[1], args[2]) + mix\_param \* get\_f(x, args[3], args[4], args[5]);

}

double\* get\_mix\_moments(double\* args, double mix\_param) {

double\* mix\_moments, \*moments1, \* moments2;

mix\_moments = new double[4];

moments1 = new double[4];

moments2 = new double[4];

mix\_moments = get\_moments(args[0], args[1], args[2]);

moments1[0] = mix\_moments[0]; moments1[1] = mix\_moments[1]; moments1[2] = mix\_moments[2]; moments1[3] = mix\_moments[3];

mix\_moments = get\_moments(args[3], args[4], args[5]);

moments2[0] = mix\_moments[0]; moments2[1] = mix\_moments[1]; moments2[2] = mix\_moments[2]; moments2[3] = mix\_moments[3];

mix\_moments[0] = (1. - mix\_param) \* moments1[0] + mix\_param \* moments2[0];

mix\_moments[1] = (1. - mix\_param) \* (pow(moments1[0], 2) + moments1[1]) + mix\_param \* (pow(moments2[0], 2) + moments2[1]) - pow(mix\_moments[0], 2);

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

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

return mix\_moments;

}

double get\_mix\_model(double\* args, double mix\_param) {

double r;

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

if (r > mix\_param) {

return get\_model(args[0], args[1], args[2]);

}

return get\_model(args[3], args[4], args[5]);

}

**empirical.cpp:**

#include "Header.h"

double randomize\_X() {

double X;

X = (rand() - 16384.) / 16384.;

std::cout << X << std::endl;

return X;

}

int get\_k(int n) {

return (int) (log(n) / log(2)) + 1;

}

double get\_min(double\* X, int n) {

double min = X[0];

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

if (X[i] < min) {

min = X[i];

}

}

return min;

}

double get\_delta(double\* X, int n, int k) {

double max = X[0];

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

if (X[i] > max) {

max = X[i];

}

}

return (max - get\_min(X, n)) / k;

}

int\* get\_empirical\_counter(double\* X, double delta, int n, int k, double min) {

int\* counter;

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

counter[j]++;

flag = true;

break;

}

}

if (!flag) {

counter[k - 1]++;

}

}

return counter;

}

double\* get\_empirical\_frequency(int\* counter, int n, int k) {

double\* frequency = new double[k];

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

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

}

return frequency;

}

double\* get\_empirical\_density(int\* counter, int n, int k, double delta) {

double\* density;

density = new double[k];

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

density[i] = counter[i] / (double)n / delta;

}

return density;

}

double get\_empirical\_f(double x, double\* density, int k, double delta, double min) {

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\* get\_empirical\_moments(double\* X, int n) {

double\* moments;

moments = new double[4];

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

moments[i] = 0;

}

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

moments[0] += X[i];

}

moments[0] /= (double)n;

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

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

}

moments[1] /= (double)n;

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

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

}

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

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

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

}

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

return moments;

}

double get\_empirical\_model(double\* frequency, double delta, int k, double min) {

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) {

//return frequency[i] / delta;

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;

}

}

return 10;

}

**supporting\_functions.cpp:**

#include "Header.h"

double check\_variables(int mode) {

std::string temp;

double result;

while (true) {

std::cin >> temp;

try {

result = stod(temp);

}

catch (std::invalid\_argument) {

std::cout << "You should enter only number and \'.\' for real numbers ";

continue;

}

if (mode == 0) {

if (result < -10000 || result > 10000) {

std::cout << "Mu should be more or equal to " << -10000 << " and less or equal to " << 10000 << ". Try again: ";

continue;

}

}

else if (mode == 1) {

if (result > 10000 || result < 0) {

std::cout << "Lambda should be more or equal to " << 0 << " and less or equal to " << 10000 << ". Try again: ";

continue;

}

}

else if (mode == 2) {

if (result < 0 || result > 10000) {

std::cout << "Nu should be more or equal to " << 0 << " and less or equal to " << 10000 << ". Try again: ";

continue;

}

}

else if (mode == 4) {

if (result <= 0 || result >= 1) {

std::cout << "p should be more then " << 0 << " and less then " << 1 << ". Try again: ";

continue;

}

}

else if (mode == 5) {

if (result <= 0) {

std::cout << "n should be more then " << 0 << ". Try again: ";

continue;

}

}

break;

}

return result;

}

void enter(double\* args, int size) {

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

std::cout << "Enter " << i + 1 << " argument: ";

args[i] = check\_variables(i % 3);

}

}

std::string write\_in\_file(double num) {

std::string s;

s = std::to\_string(num);

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

if (s[i] == '.') {

s[i] = ',';

break;

}

}

return s;

}

**test.cpp:**

#include "Header.h"

void test() {

double temp, \* moments;

bool pass;

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_default\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

double

nu[10]{ 0., 0.5, 1., 2., 3., 4., 5., 6., 7., 8. },

f0[10]{ 0.5, 0.637, 0.75, 0.938, 1.094, 1.23, 1.354, 1.466, 1.571, 1.669 },

D[10]{ 1. / 3., 0.25, 0.2, 0.143, 0.111, 0.091, 0.077, 0.067, 0.059, 0.053 },

y2[10]{ -1.2, -1., -0.86, -0.67, -0.55, -0.46, -0.4, -0.35, -0.32, -0.29 };;

pass = true;

//default 1

for (int i = 0; i < 10; i++) {

temp = get\_f(0, 0, 1, nu[i]);

moments = get\_moments(0, 1, nu[i]);

if (abs(temp - f0[i]) < EPS && abs(moments[1] - D[i]) < EPS && abs(moments[3] - y2[i]) < EPS) {

std::cout << ":D Test default 1." << i + 1 << " passed" << std::endl;

}

else {

std::cout << ":C Test default 1." << i + 1 << " failed" << std::endl;

std::cout << " |expected: f0 = " << f0[i] << ", D = " << D[i] << ", y2 = " << y2[i] << std::endl;

std::cout << " |got: f0 = " << temp << ", D = " << moments[1] << ", y2 = " << moments[3] << std::endl;

pass = false;

}

}

if (pass) {

std::cout << "------ :D Test default 1 passed ------" << std::endl << std::endl;

}

else {

std::cout << "------ :C Test default 1 failed ------" << std::endl << std::endl;

pass = true;

}

//default 2

for (int i = 0; i < 10; i++) {

temp = get\_f(0, -3.5, 2, nu[i]);

moments = get\_moments(-3.5, 2, nu[i]);

if (abs(moments[1] - D[i] \* 4) < EPS && abs(moments[3] - y2[i]) < EPS) {

std::cout << ":D Test default 2." << i + 1 << " passed" << std::endl;

}

else {

std::cout << ":C Test default 2." << i + 1 << " failed" << std::endl;

std::cout << " |expected: f0 = " << f0[i] << ", D = " << D[i] << ", y2 = " << y2[i] << std::endl;

std::cout << " |got: f0 = " << temp << ", D = " << moments[1] << ", y2 = " << moments[3] << std::endl;

pass = false;

}

}

if (pass) {

std::cout << "------ :D Test default 2 passed ------" << std::endl << std::endl;

}

else {

std::cout << "------ :C Test default 2 failed ------" << std::endl << std::endl;

pass = true;

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_default\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_mix\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

double args[6]{ 7.1, 2, 0, 7.1, 2, 0 };

double param = 0.75;

double mix\_D, mix\_y2;

double temp2;

//mix 1

for (int i = 0; i < 10; i++) {

args[2] = nu[i];

args[5] = nu[i];

temp = get\_mix\_f(7.1, args, param);

temp2 = get\_f(7.1, args[0], args[1], args[2]);

if (abs(temp - temp2) < EPS) {

std::cout << ":D Test mix 1." << i + 1 << " passed" << std::endl;

}

else {

std::cout << ":C Test mix 1." << i + 1 << " failed" << std::endl;

std::cout << " |expected: f0 = " << temp2 << std::endl;

std::cout << " |got: f0 = " << temp << std::endl;

pass = false;

}

}

if (pass) {

std::cout << "------ :D Test mix 1 passed ------" << std::endl << std::endl;

}

else {

std::cout << "------ :C Test mix 1 failed ------" << std::endl << std::endl;

pass = true;

}

//mix 2

args[0] = -3.;

args[1] = 1.;

args[3] = 2.5;

args[4] = 1.9;

param = 0.5;

for (int i = 0; i < 10; i++) {

args[2] = nu[i];

args[5] = nu[i];

moments = get\_moments(args[0], args[1], args[2]);

temp = moments[0];

moments = get\_moments(args[3], args[4], args[5]);

temp2 = moments[0];

moments = get\_mix\_moments(args, param);

if (abs(moments[0] - ((temp + temp2) / 2)) < EPS) {

std::cout << ":D Test mix 2." << i + 1 << " passed" << std::endl;

}

else {

std::cout << ":C Test mix 2." << i + 1 << " failed" << std::endl;

std::cout << " |expected: M = " << (temp + temp2) / 2 << std::endl;

std::cout << " |got: M = " << moments[0] << std::endl;

pass = false;

}

}

if (pass) {

std::cout << "------ :D Test mix 2 passed ------" << std::endl << std::endl;

}

else {

std::cout << "------ :C Test mix 2 failed ------" << std::endl << std::endl;

pass = true;

}

//mix 3

args[0] = 0.;

args[1] = 1.;

args[3] = 0.;

args[4] = 3.;

param = 0.5;

for (int i = 0; i < 10; i++) {

args[2] = nu[i];

args[5] = nu[i];

moments = get\_moments(args[0], args[1], args[2]);

temp = moments[1];

moments = get\_moments(args[3], args[4], args[5]);

temp2 = moments[1];

moments = get\_mix\_moments(args, param);

if (abs(moments[1] - (temp \* param + temp2 \* (1 - param))) < EPS) {

std::cout << ":D Test mix 3." << i + 1 << " passed" << std::endl;

}

else {

std::cout << ":C Test mix 3." << i + 1 << " failed" << std::endl;

std::cout << " |expected: M = " << (temp \* param + temp2 \* (1 - param)) << std::endl;

std::cout << " |got: M = " << moments[1] << std::endl;

pass = false;

}

}

if (pass) {

std::cout << "------ :D Test mix 3 passed ------" << std::endl << std::endl;

}

else {

std::cout << "------ :C Test mix 3 failed ------" << std::endl << std::endl;

pass = true;

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_mix\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_empirical\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

double

mu[10]{ 0., 5., -3., 9., 3.1, -9.4, 0.5, -0.23, 12.5, -10. },

la[10]{ 1., 2., 8., 1.7, 1.3, 5.1, 10.3, 5.9, 0.1, 9.9 },

mu2[10]{ 6., -2.1, 3.7, 2.9, -1.6, 8.1, -2.9, -5.9, 0.5, -0.4 },

la2[10]{ 5., 9., 3.8, 5.2, 6.3, 0.5, 2.5, 2.1, 0.7, 8.2 },

nu2[10]{ 8., 2.6, 1.1, 5., 1.2, 9.1, 2.9, 2.1, 8.1, 3. },

x[10];

double\* X, \* expected\_moments = new double[4];

double min, delta, \* density;

int z = 100, k, \* counter;

X = new double[100 \* z \* pow(2, 9)];

param = 0.62;

//empirical mix

for (int i = 0; i < 10; i++) {

args[0] = mu[i];

args[1] = la[i];

args[2] = nu[i];

args[3] = mu2[i];

args[4] = la2[i];

args[5] = nu2[i];

for (int j = 0; j < 100 \* z; j++) {

X[j] = get\_mix\_model(args, param);

}

for (int j = 0; j < 10; j++) {

x[j] = X[j];

}

min = get\_min(X, 100 \* z);

k = get\_k(100 \* z);

delta = get\_delta(X, 100 \* z, k);

counter = get\_empirical\_counter(X, delta, 100 \* z, k, min);

density = get\_empirical\_density(counter, 100 \* z, k, delta);

moments = get\_empirical\_moments(X, 100 \* z);

expected\_moments = get\_mix\_moments(args, param);

for (int q = 0; q < 10; q++) {

if (abs(moments[1] - expected\_moments[1]) < EPS && abs(moments[3] - expected\_moments[3]) < EPS && abs(get\_empirical\_f(x[q], density, k, delta, min) - get\_mix\_f(x[q], args, param) < EPS)) {

std::cout << ":D Test empirical 1.1." << i + 1 << "." << q + 1 << " passed" << std::endl;

}

else {

temp = get\_mix\_f(x[q], args, param);

std::cout << ":C Test empirical 1.1." << i + 1 << "." << q + 1 << " failed" << std::endl;

std::cout << " |expected: f = " << temp << std::endl;

std::cout << " |got: f = " << get\_empirical\_f(x[q], density, k, delta, min) << std::endl;

pass = false;

}

}

if (pass) {

std::cout << "------ :D Test empirical 1.1." << i + 1 << " (n = " << z \* 100 << ") passed ------" << std::endl << std::endl;

}

else {

std::cout << " |expected: D = " << expected\_moments[1] << ", y2 = " << expected\_moments[3] << std::endl;

std::cout << " |got: D = " << moments[1] << ", y2 = " << moments[3] << std::endl;

std::cout << "------ :C Test empirical 1.1." << i + 1 << " (n = " << z \* 100 << ") failed------" << std::endl << std::endl;

pass = true;

}

z \*= 2;

}

std::ofstream dm\_data, e\_data, x\_data;

dm\_data.open("m\_data.txt");

e\_data.open("e\_m\_data.txt");

x\_data.open("x\_m\_data.txt");

for (int i = 0; i < z / 2; i++) {

x\_data << write\_in\_file(X[i]) << std::endl;

dm\_data << write\_in\_file(get\_mix\_f(X[i], args, param)) << std::endl;

e\_data << write\_in\_file(get\_empirical\_f(X[i], density, k, delta, min)) << std::endl;

}

x\_data.close();

e\_data.close();

dm\_data.close();

//empirical default

z = 100;

for (int i = 0; i < 10; i++) {

for (int j = 0; j < 100 \* z; j++) {

X[j] = get\_model(mu[i], la[i], nu[i]);

}

for (int j = 0; j < 10; j++) {

x[j] = X[j];

}

min = get\_min(X, 100 \* z);

k = get\_k(100 \* z);

delta = get\_delta(X, 100 \* z, k);

counter = get\_empirical\_counter(X, delta, 100 \* z, k, min);

density = get\_empirical\_density(counter, 100 \* z, k, delta);

moments = get\_empirical\_moments(X, 100 \* z);

expected\_moments = get\_moments(mu[i], la[i], nu[i]);

for (int q = 0; q < 10; q++) {

if (abs(moments[1] - expected\_moments[1]) < EPS && abs(moments[3] - expected\_moments[3]) < EPS && abs(get\_empirical\_f(x[q], density, k, delta, min) - get\_f(x[q], mu[i], la[i], nu[i]) < EPS)) {

std::cout << ":D Test empirical 1.2." << i + 1 << "." << q + 1 << " passed" << std::endl;

}

else {

temp = get\_f(x[q], mu[i], la[i], nu[i]);

std::cout << ":C Test empirical 1.2." << i + 1 << "." << q + 1 << " failed" << std::endl;

std::cout << " |expected: f = " << temp << std::endl;

std::cout << " |got: f = " << get\_empirical\_f(x[q], density, k, delta, min) << std::endl;

pass = false;

}

}

if (pass) {

std::cout << "------ :D Test empirical 1.2." << i + 1 << " (n = " << z \* 100 << ") passed ------" << std::endl << std::endl;

}

else {

std::cout << " |expected: D = " << expected\_moments[1] << ", y2 = " << expected\_moments[3] << std::endl;

std::cout << " |got: D = " << moments[1] << ", y2 = " << moments[3] << std::endl;

std::cout << "------ :C Test empirical 1.2." << i + 1 << " (n = " << z \* 100 << ") failed------" << std::endl << std::endl;

pass = true;

}

z \*= 2;

}

dm\_data.open("d\_data.txt");

e\_data.open("e\_d\_data.txt");

x\_data.open("x\_d\_data.txt");

for (int i = 0; i < z / 2; i++) {

x\_data << write\_in\_file(X[i]) << std::endl;

dm\_data << write\_in\_file(get\_f(X[i], mu[9], la[9], nu[9])) << std::endl;

e\_data << write\_in\_file(get\_empirical\_f(X[i], density, k, delta, min)) << std::endl;

}

x\_data.close();

e\_data.close();

dm\_data.close();

//empirical 2

double\* Y, \* empirical\_moments;

double e\_delta, e\_min;

z = z / 2;

Y = new double[z];

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

Y[i] = get\_empirical\_model(get\_empirical\_frequency(counter, z, k), delta, k, min);

}

e\_min = get\_min(Y, z);

e\_delta = get\_delta(Y, z, k);

empirical\_moments = get\_empirical\_moments(Y, z);

for (int q = 0; q < 10; q++) {

if (abs(moments[1] - empirical\_moments[1]) < EPS && abs(moments[3] - empirical\_moments[3]) < EPS && abs(get\_empirical\_f(x[q], density, k, delta, min) - get\_empirical\_f(x[q], get\_empirical\_density(counter, z, k, e\_delta), k, e\_delta, e\_min) < EPS)) {

std::cout << ":D Test empirical 2.1." << q + 1 << " passed" << std::endl;

}

else {

temp = get\_f(x[q], mu[9], la[9], nu[9]);

std::cout << ":C Test empirical 2.1." << q + 1 << " failed" << std::endl;

std::cout << " |expected: f = " << get\_empirical\_f(x[q], density, k, delta, min) << std::endl;

std::cout << " |got: f = " << get\_empirical\_f(x[q], get\_empirical\_density(counter, z, k, e\_delta), k, e\_delta, e\_min) << std::endl;

std::cout << " |expected: D = " << expected\_moments[1] << ", y2 = " << expected\_moments[3] << std::endl;

std::cout << " |got: D = " << moments[1] << ", y2 = " << moments[3] << std::endl;

pass = false;

}

}

if (pass) {

std::cout << "------ :D Test empirical 2.1" << " (n = " << z \* 100 << ") passed ------" << std::endl << std::endl;

}

else {

std::cout << "------ :C Test empirical 2.1" << "(n = " << z \* 100 << ") failed------" << std::endl << std::endl;

pass = true;

}

for (int q = 0; q < 10; q++) {

if (abs(expected\_moments[1] - empirical\_moments[1]) < EPS && abs(expected\_moments[3] - empirical\_moments[3]) < EPS && abs(temp - get\_empirical\_f(x[q], get\_empirical\_density(counter, z, k, e\_delta), k, e\_delta, e\_min) < EPS)) {

std::cout << ":D Test empirical 2.2." << q + 1 << " passed" << std::endl;

}

else {

temp = get\_f(x[q], mu[9], la[9], nu[9]);

std::cout << ":C Test empirical 2.2." << q + 1 << " failed" << std::endl;

std::cout << " |expected: f = " << temp << std::endl;

std::cout << " |got: f = " << get\_empirical\_f(x[q], get\_empirical\_density(counter, z, k, e\_delta), k, e\_delta, e\_min) << std::endl;

std::cout << " |expected: D = " << expected\_moments[1] << ", y2 = " << expected\_moments[3] << std::endl;

std::cout << " |got: D = " << empirical\_moments[1] << ", y2 = " << empirical\_moments[3] << std::endl;

pass = false;

}

}

if (pass) {

std::cout << "------ :D Test empirical 2.2" << " (n = " << z \* 100 << ") passed ------" << std::endl << std::endl;

}

else {

std::cout << "------ :C Test empirical 2.2" << "(n = " << z \* 100 << ") failed------" << std::endl << std::endl;

pass = true;

}

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_empirical\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

}

В программе используется несколько **текстовых файлов** сбора информации о распределениях и последующего создания графиков в Excel. Вот эти файлы:

x\_d\_data.txt - файл, собирающий значения x в тесте 3.3.1 с обычным распределением;

d\_data.txt - файл, содержащий значения функции плотности обычного распределения, аргументами в которой выступают значения из x\_d\_data.txt;

e\_d\_data.txt - файл, содержащий значения эмпирической функции плотности, сгенерированной относительно значений из d\_data.txt (эмпирически сгенерированное обычное распределение);

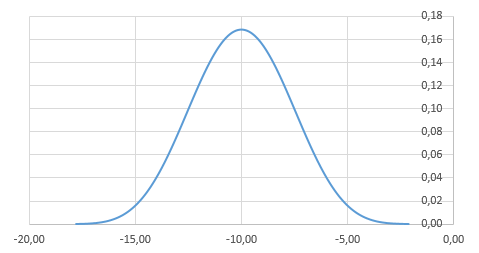
x\_m\_data.txt - файл, собирающий значения x в тесте 3.3.1 с распределением смеси;

m\_data.txt - файл, содержащий значения функции плотности смеси, аргументами в которой выступают значения из x\_m\_data.txt;

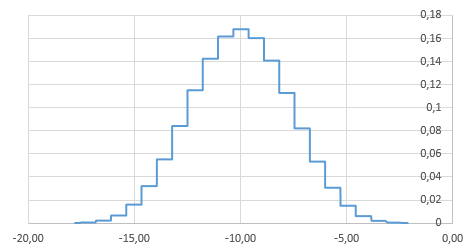
e\_m\_data.txt - файл, содержащий значения эмпирической функции плотности, сгенерированной относительно значений из m\_data.txt (эмпирически сгенерированное распределение смеси).

На основе текстовых данных было построено несколько **графиков** в программе Excel:

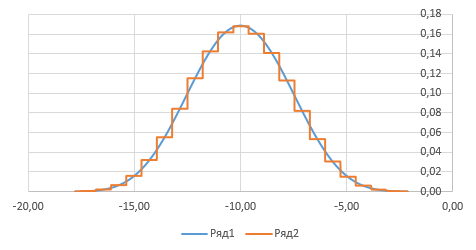
Обычное распределение (mu = -10, la = 9.9, nu = 8):



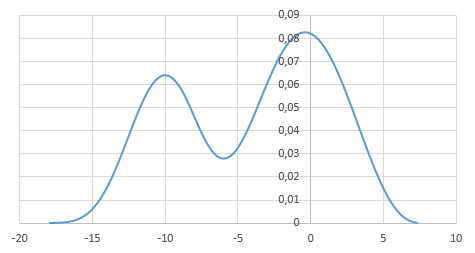
Эмпирическое распределение, сгенерированное относительно обычного распределения:



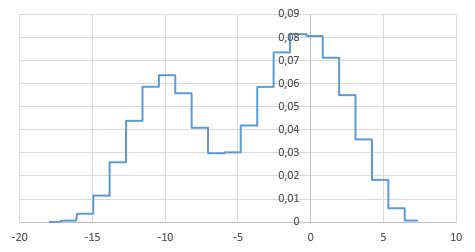
Наложение двух графиков:



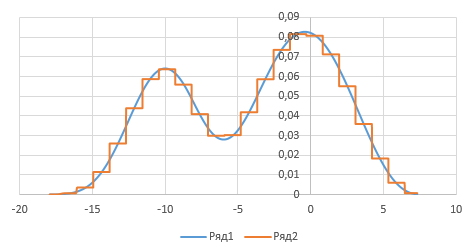
Распределение смеси (= -10, = 9.9, = 8, = -0.4, = 8.2, = 3):



Эмпирическое распределение, сгенерированное относительно распределения смеси:



Наложение двух графиков:



Контрольные вопросы проработаны.