

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧЕРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«Санкт-Петербургский политехнический университет Петра Великого»

ИНСТИТУТ КОМПЬЮТЕРНЫХ НАУК И ТЕХНОЛОГИЙ

Лабораторная работа №1

“Получение базовой последовательности псевдослучайных чисел и тестовые проверки его работы”

Выполнил:

Каргалов Л.А.

Проверил:

Чуркин В. В.

4 апреля 2020 г.

Санкт-Петербург
2020

Содержание

1	Цель работы	3
2	Ход работы	4
2.1	Результаты	4
3	Выводы	8
4	Приложение	9
4.1	Исходный код на языке C++	9

1 Цель работы

- 1) Получение на ЭВМ с помощью программного датчика базовой последовательности псевдослучайных чисел, имеющих равномерное распределение.
- 2) Освоение методов статистической оценки полученного распределения: вычисление эмпирических значений для математического ожидания и дисперсии.
- 3) Освоение методов оценки статистики связи: вычисление значений автокорреляционной функции и построение коррелограммы.
- 4) Освоение методов графического представления законов распределения: построение функции плотности распределения и интегральной функции распределения.

2 Ход работы

- 1) С помощью программного датчика получены псевдослучайные числа $u[1], u[2], \dots, u[n]$, имеющие равномерный характер распределения. Поставим задачу простейшей оценки качества полученного датчика путем вычисления так называемых эмпирических точечных оценок распределения, в частности, математического ожидания и дисперсии, и сравнения полученных результатов с известными теоретическими значениями.
- 2) Оценка степени связанности псевдослучайных чисел определена с помощью корреляционной (или "автокорреляционной") функции $K(f)$ (построены кореллограммы), которая представляет собой последовательность коэффициентов корреляции, зависящих от величины сдвига f , как от аргумента.
- 3) Приведено графическое представление законов распределения: построение эмпирической функции плотности распределения и эмпирической интегральной функции распределения и сравнение с соответствующими теоретическими кривыми.

2.1 Результаты

- 1) Таблица точечных оценок

N	Expectation experimental	Expectation theoretical	Dispersion experimental	Dispersion theoretical
10	0,497839	0,5	0,01659	0,083333333
100	0,513515	0,5	0,091357	0,083333333
1000	0,498484	0,5	0,081804	0,083333333
10000	0,500286	0,5	0,084356	0,083333333

- 2) График функции распределения

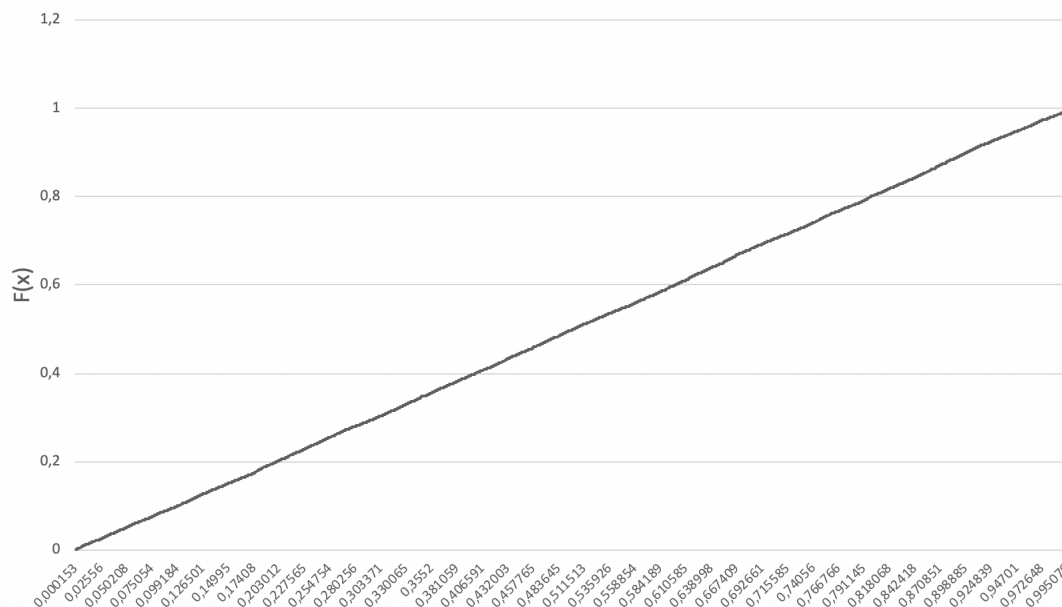


Рис. 1: Для $n = 10000$

- 3) График функции плотности распределения

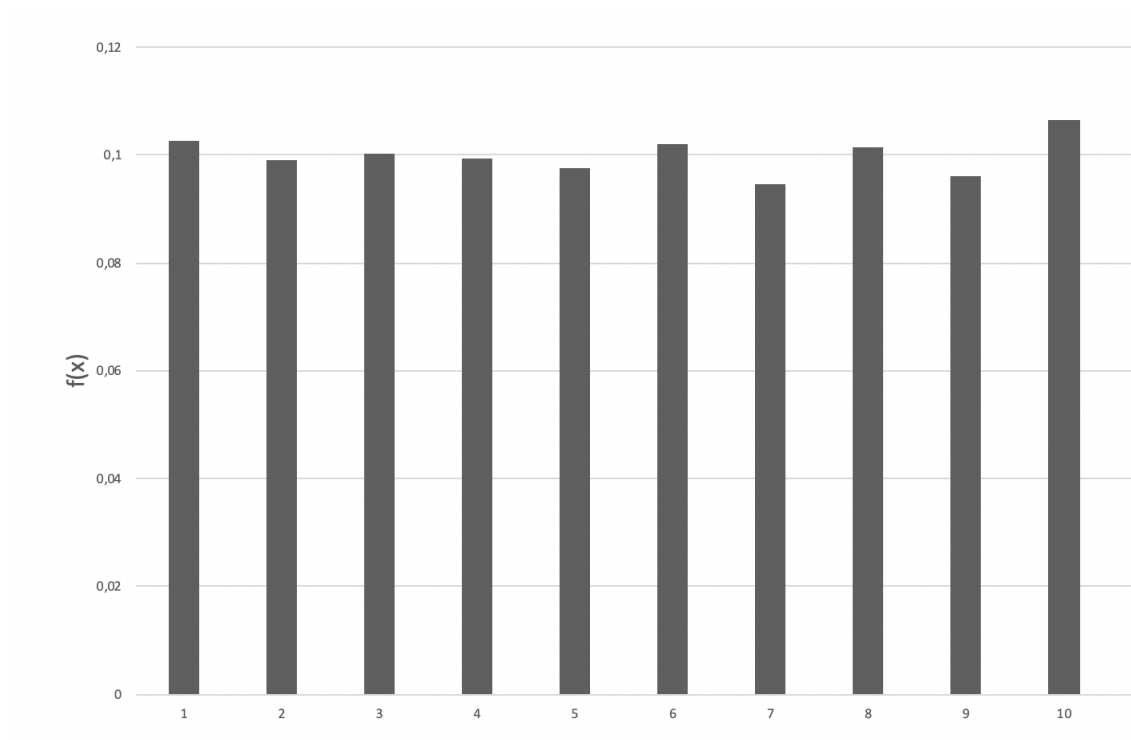


Рис. 2: Для $n = 10000$

4) Корелограммы



Рис. 3: Для $n = 10$

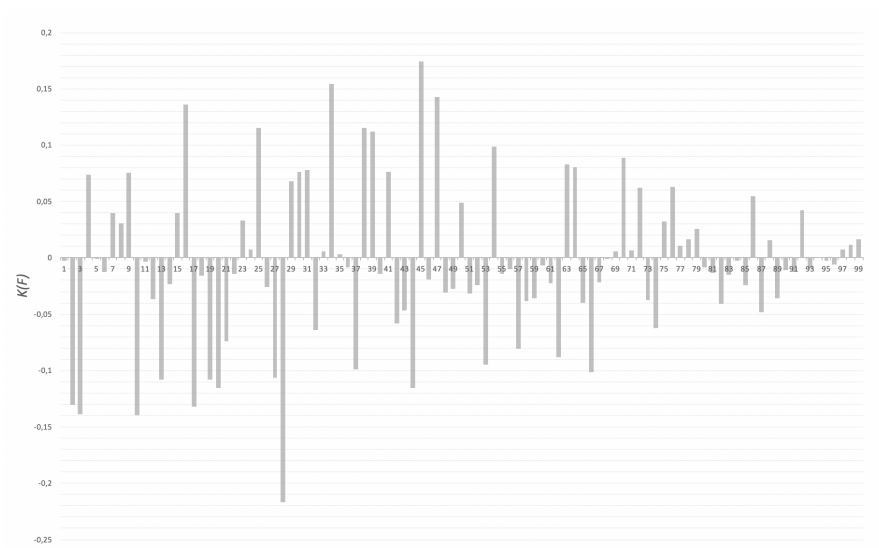


Рис. 4: Для $n = 100$

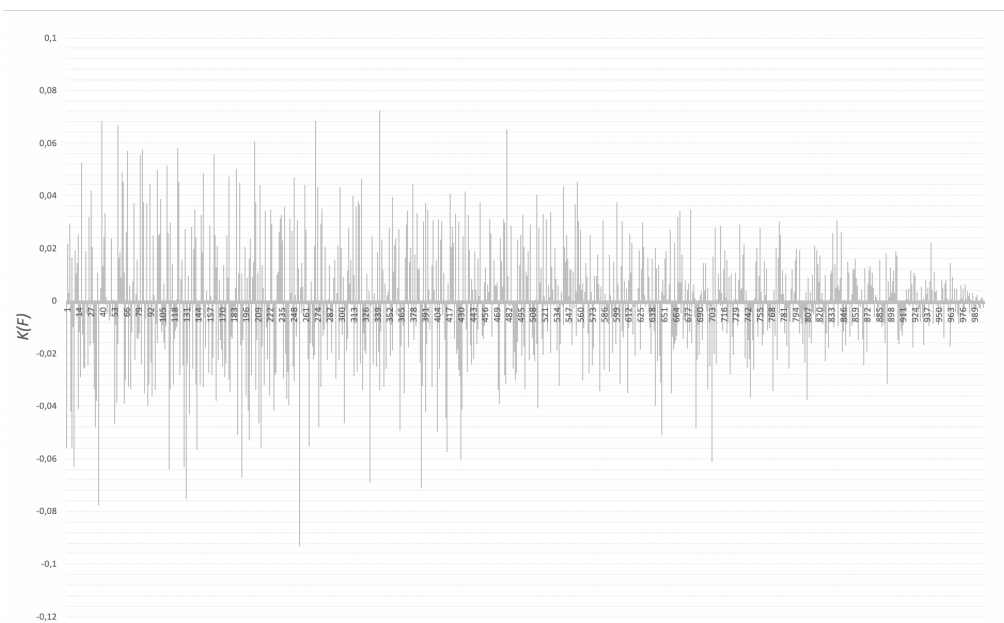


Рис. 5: Для $n = 1000$

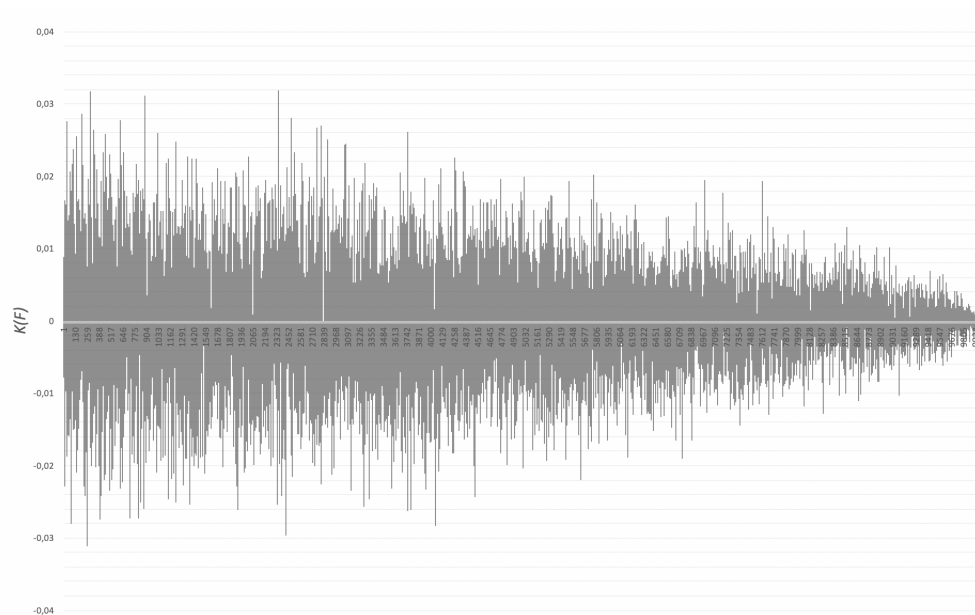


Рис. 6: Для $n = 10000$

3 Выводы

Исходя из полученных точечных оценок, можно сказать, что при достаточно больших $n > 100$ значения математического ожидания и дисперсии принимают значения соответствующие теоретическим, с некоторой погрешностью, это объясняется тем что с ростом размера выборки полученные эмпирическим путем данные все лучше описывают теоретические.

Так же, основываясь на графиках АКФ, можно сказать, что между элементами случайной последовательности существуют некоторые зависимости, т.к. в точках не кратных n значения АКФ не стремятся к нулю, еще можно заметить что график АКФ не имеет периодически повторяющихся паттернов, что свидетельствует об отсутствии периодичности в исходной функции генератора.

Поэтому встроенный класс `std::uniform_real_distribution<>` можно использовать в качестве базового для получения псевдослучайных величин с равномерным законом распределения.

4 Приложение

4.1 Исходный код на языке C++

Листинг 1: model.hpp

```
1  #ifndef model_hpp
2  #define model_hpp
3
4  #include <iostream>
5  #include <numeric>
6  #include <array>
7  #include <cmath>
8  #include <vector>
9  #include <random>
10
11 #define MAX_VAL 1
12 #define MIN_VAL 0
13
14 template <class TDistr, class... Args>
15 class RandomGenerator
16 {
17 public:
18     RandomGenerator(const Args &... args);
19     auto getValue();
20
21 private:
22     std::random_device rd_;
23     std::mt19937 generator_;
24     TDistr distr_;
25 };
26
27 template <class G, std::size_t S>
28 class Model_t
29 {
30 public:
31     template <class... Args>
32     Model_t(const Args &... args);
33
34     void generate();
35     double getExpectation();
36     double getDispersion();
37     const std::array<double, S> &getCorrelation();
38     const std::array<double, S> &getData();
39     const std::array<double, S> &getSortedData();
40     const std::vector<std::pair<double, double>> &getProbDensity();
41
42 private:
43     void countExpectation();
44     void countDispersion();
45     void countCorrelation();
46     void countProbDensity(const int partitions);
47
48     G randGenerator_;
49     std::array<double, S> value_arr_;
50     std::array<double, S> correlation_coef_arr_;
51     std::vector<std::pair<double, double>> prob_density_;
52     double expectation_;
53     double dispersion_;
```

```

54 };
55
56 typedef RandomGenerator<std::uniform_real_distribution<>, int, int> UniformGenerator;
57 typedef RandomGenerator<std::binomial_distribution<>, int, double> BinomialGenerator;
58 typedef RandomGenerator<std::geometric_distribution<>, double> GeometricGenerator;
59 typedef RandomGenerator<std::poisson_distribution<>, double> PoissonGenerator;
60 typedef RandomGenerator<std::lognormal_distribution<>, double, double> LogonormalGenerator
    ;
61
62 typedef RandomGenerator<std::normal_distribution<>, double, double> NormalGenerator;
63 typedef RandomGenerator<std::exponential_distribution<>, double> ExponentialGenerator;
64 typedef RandomGenerator<std::chi_squared_distribution<>, double> ChiSquareGenerator;
65 typedef RandomGenerator<std::student_t_distribution<>, double> StudentGenerator;
66
67 //===== Implementation =====
68
69 template <class TDistr, class... Args>
70 RandomGenerator<TDistr, Args...>::RandomGenerator(const Args &... args)
71     : rd_(),
72       generator_(rd_()),
73       distr_(args...)
74 {
75 }
76
77 template <class TDistr, class... Args>
78 auto RandomGenerator<TDistr, Args...>::getValue()
79 {
80     return distr_(generator_);
81 }
82
83 template <class G, std::size_t S>
84 template <class... Args>
85 Model_t<G, S>::Model_t(const Args &... args)
86     : randGenerator_(args...)
87 {
88 }
89
90 template <class G, std::size_t S>
91 void Model_t<G, S>::generate()
92 {
93     std::srand(unsigned(std::time(0)));
94     for (std::size_t i = 0; i < S; i++)
95     {
96         value_arr_.at(i) = randGenerator_.getValue();
97     }
98
99     countExpectation();
100    countDispersion();
101    countCorrelation();
102    countProbDensity(10);
103 }
104
105 template <class G, std::size_t S>
106 double Model_t<G, S>::getExpectation()
107 {
108     return expectation_;
109 }
110
111 template <class G, std::size_t S>

```

```

112 double Model_t<G, S>::getDispersion()
113 {
114     return dispersion_;
115 }
116
117 template <class G, std::size_t S>
118 const std::array<double, S> &Model_t<G, S>::getCorrelation()
119 {
120     return correlation_coef_arr_;
121 }
122
123 template <class G, std::size_t S>
124 const std::array<double, S> &Model_t<G, S>::getData()
125 {
126     return value_arr_;
127 }
128
129 template <class G, std::size_t S>
130 const std::array<double, S> Model_t<G, S>::getSortedData()
131 {
132     auto arr = value_arr_;
133     std::sort(arr.begin(), arr.end());
134     return arr;
135 }
136
137 template <class G, std::size_t S>
138 const std::vector<std::pair<double, double>> &Model_t<G, S>::getProbDensity()
139 {
140     return prob_density_;
141 }
142
143 template <class G, std::size_t S>
144 void Model_t<G, S>::countExpectation()
145 {
146     expectation_ = 0;
147     expectation_ = std::accumulate(value_arr_.begin(), value_arr_.end(), expectation_) / S;
148 }
149
150 template <class G, std::size_t S>
151 void Model_t<G, S>::countDispersion()
152 {
153     auto functor = [this](double res, double val) {
154         return res + std::pow(val - this->expectation_, 2);
155     };
156
157     dispersion_ = 0;
158     dispersion_ = std::accumulate(value_arr_.begin(), value_arr_.end(), dispersion_,
159                                   functor) / S;
160 }
161
162 template <class G, std::size_t S>
163 void Model_t<G, S>::countCorrelation()
164 {
165     for (std::size_t f = 0; f < S; f++)
166     {
167         double v1 = 0;
168         double v2 = 0;

```

```

169         for (std::size_t i = 0; i < S - f; i++)
170         {
171             v1 += (value_arr_.at(i) - expectation_) * (value_arr_.at(i + f) - expectation_);
172         }
173
174         for (std::size_t i = 0; i < S; i++)
175         {
176             v2 += pow(value_arr_.at(i) - expectation_, 2);
177         }
178
179         correlation_coef_arr_.at(f) = v1 / v2;
180     }
181 }
182
183 template <class G, std::size_t S>
184 void Model_t<G, S>::countProbDensity(const int partitions)
185 {
186     auto sorted_data = this->getSortedData();
187
188     double step = (sorted_data.back() - sorted_data.front()) / partitions;
189     double half_step = step / 2.0000;
190
191     for (double h = sorted_data.front() - half_step; h <= sorted_data.back() + half_step;
192          h += step)
193     {
194         int count = 0;
195         for (auto &el : value_arr_)
196         {
197             if (el >= h - half_step && el < h + half_step)
198             {
199                 count++;
200             }
201             prob_density_.push_back({h, (count / static_cast<double>(S))});
202         }
203     }
204
205 #endif /* model_hpp */

```

Листинг 2: writer.hpp

```

1  #ifndef writer_hpp
2  #define writer_hpp
3
4  #include <iostream>
5  #include <numeric>
6  #include <array>
7  #include <cmath>
8  #include <iomanip>
9  #include <OpenXLSX/OpenXLSX.h>
10
11 #include "model.hpp"
12
13 using namespace std;
14 using namespace OpenXLSX;
15
16 class ExcelWriter
17 {

```

```

18 public:
19     ExelWriter(const std::string &filepath);
20     ~ExelWriter();
21
22     template <std::size_t S, class T, template <class, std::size_t> class G>
23     void addResult(G<T, S> &mod);
24     void addSummary(int n, double expectation, double dispersion);
25
26     template <std::size_t S>
27     void addSequence(int n, const std::array<double, S> &data);
28
29     template <std::size_t S>
30     void addSortedSequence(int n, const std::array<double, S> &data);
31
32     template <std::size_t S>
33     void addCorrelation(int n, const std::array<double, S> &data);
34     void addProbDensity(int n, const std::vector<std::pair<double, double>> &data);
35
36 private:
37     void initTables();
38     XLDocument doc_;
39 };
40
41 //===== Implementation =====
42
43 ExelWriter::ExelWriter(const std::string &filepath)
44 {
45     doc_.CreateDocument(filepath);
46     initTables();
47 }
48
49 ExelWriter::~ExelWriter()
50 {
51     doc_.SaveDocument();
52 }
53
54 template <std::size_t S, class T, template <class, std::size_t> class G>
55 void ExelWriter::addResult(G<T, S> &mod)
56 {
57     addSummary(S, mod.getExpectation(), mod.getDispersion());
58     addSequence(S, mod.getData());
59     addSortedSequence(S, mod.getSortedData());
60     addCorrelation(S, mod.getCorrelation());
61     addProbDensity(S, mod.getProbDensity());
62 }
63
64 void ExelWriter::addSummary(int n, double expectation, double dispersion)
65 {
66     auto wks = doc_.Workbook().Worksheet("Summary");
67     wks.Cell(wks.RowCount() + 1, 1).Value() = n;
68     wks.Cell(wks.RowCount(), 2).Value() = expectation;
69     wks.Cell(wks.RowCount(), 4).Value() = dispersion;
70 }
71
72 template <std::size_t S>
73 void ExelWriter::addSequence(int n, const std::array<double, S> &data)
74 {
75     auto wks = doc_.Workbook().Worksheet("Data");
76     wks.Cell(1, wks.ColumnCount() + 1).Value() = std::string("N = " + std::to_string(S));

```

```

77     for (auto i = 0; i < data.size(); i++)
78     {
79         wks.Cell(i + 2, wks.ColumnCount()).Value() = data.at(i);
80         wks.Cell(i + 2, 1).Value() = i;
81     }
82 }
83
84 template <std::size_t S>
85 void ExelWriter::addSortedSequence(int n, const std::array<double, S> &data)
86 {
87     auto wks = doc_.Workbook().Worksheet("Sorted Data");
88     wks.Cell(1, wks.ColumnCount() + 1).Value() = std::string("N = " + std::to_string(S));
89     for (auto i = 0; i < data.size(); i++)
90     {
91         wks.Cell(i + 2, wks.ColumnCount()).Value() = data.at(i);
92         wks.Cell(i + 2, 1).Value() = i;
93     }
94 }
95
96 template <std::size_t S>
97 void ExelWriter::addCorrelation(int n, const std::array<double, S> &data)
98 {
99     auto wks = doc_.Workbook().Worksheet("Correlation");
100    wks.Cell(1, wks.ColumnCount() + 1).Value() = std::string("N = " + std::to_string(S));
101    for (auto i = 0; i < data.size(); i++)
102    {
103        wks.Cell(i + 2, wks.ColumnCount()).Value() = data.at(i);
104        wks.Cell(i + 2, 1).Value() = i;
105    }
106 }
107
108 void ExelWriter::addProbDensity(int n, const std::vector<std::pair<double, double>> &data)
109 {
110     auto wks = doc_.Workbook().Worksheet("Prob Density");
111     wks.Cell(1, wks.ColumnCount() + 1).Value() = std::string("N = " + std::to_string(n) +
112         " [STEP]");
113     wks.Cell(1, wks.ColumnCount() + 1).Value() = std::string("N = " + std::to_string(n) +
114         " [VALUE]");
115     for (auto i = 0; i < data.size(); i++)
116     {
117         wks.Cell(i + 2, wks.ColumnCount() - 1).Value() = data.at(i).first;
118         wks.Cell(i + 2, wks.ColumnCount()).Value() = data.at(i).second;
119         wks.Cell(i + 2, 1).Value() = i;
120     }
121 }
122
123 void ExelWriter::initTables()
124 {
125     doc_.Workbook().AddWorksheet("Summary");
126     auto wks = doc_.Workbook().Worksheet("Summary");
127     wks.Cell("A1").Value() = "N";
128     wks.Cell("B1").Value() = "Expectation experimental";
129     wks.Cell("C1").Value() = "Expectation theoretical";
130     wks.Cell("D1").Value() = "Dispersion experimental";
131     wks.Cell("E1").Value() = "Dispersion theoretical";
132
133     doc_.Workbook().AddWorksheet("Data");
134     doc_.Workbook().AddWorksheet("Sorted Data");
135     doc_.Workbook().AddWorksheet("Correlation");

```

```

134     doc_.Workbook().AddWorksheet("Prob Density");
135 }
136
137 template <class G>
138 ExelWriter &operator<<(ExelWriter &wr, G &mod)
139 {
140     wr.addResult(mod);
141     return wr;
142 }
143
144 #endif /* writer_hpp */

```

Листинг 3: main.cpp

```

1  #include <iostream>
2  #include <numeric>
3  #include <array>
4  #include <cmath>
5  #include <iomanip>
6  #include <OpenXLSX/OpenXLSX.h>
7
8  #include "writer.hpp"
9
10 void lab1()
11 {
12     Model_t<UniformGenerator, 10> m1(0, 1);
13     Model_t<UniformGenerator, 100> m2(0, 1);
14     Model_t<UniformGenerator, 1000> m3(0, 1);
15     Model_t<UniformGenerator, 10000> m4(0, 1);
16
17     m1.generate();
18     m2.generate();
19     m3.generate();
20     m4.generate();
21
22     ExelWriter exel("./lab1.xlsx");
23     exel << m1 << m2 << m3 << m4;
24 }
25
26 template<std::size_t S, class Gen, class ... Args>
27 void gen_tester(std::string file_path, const Args& ... args)
28 {
29     Model_t<Gen, S> m1(args ...);
30     Model_t<Gen, S> m2(args ...);
31     Model_t<Gen, S> m3(args ...);
32
33     m1.generate();
34     m2.generate();
35     m3.generate();
36
37     ExelWriter exel(file_path);
38     exel << m1 << m2 << m3;
39 }
40
41 void lab2()
42 {
43     gen_tester<1000, UniformGenerator>("./lab2_uniform_gen.xlsx", 0, 1);
44     gen_tester<1000, BinomialGenerator>("./lab2_binomial_gen.xlsx", 40, 0.5);
45     gen_tester<1000, GeometricGenerator>("./lab2_geometric_gen.xlsx", 0.5);

```

```

46     gen_tester <1000, PoissonGenerator>("./lab2_poisson_gen.xlsx", 4);
47 }
48
49 void lab3()
50 {
51     gen_tester <10000, UniformGenerator>("./lab3_uniform_gen.xlsx", 2, 10);
52     gen_tester <10000, NormalGenerator>("./lab3_normal_gen.xlsx", -2.0, 2.5);
53     gen_tester <10000, ExponentialGenerator>("./lab3_exponential_gen.xlsx", 1.5);
54     gen_tester <10000, ChiSquareGenerator>("./lab3_chi_square_gen.xlsx", 3.0);
55     gen_tester <10000, StudentGenerator>("./lab3_student_gen.xlsx", 5.0);
56 }
57
58
59 int main(int argc, const char *argv[])
60 {
61
62     lab3();
63     return 0;
64 }

```