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

Институт компьютерных наук и технологий

Лабораторная работа №2

"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

1) Практическое освоение методов получения случайных величин, имеющих дискретный характер распределения.

2) Разработка программных датчиков дискретных случайных величин.

3) Исследование характеристик моделируемых датчиков:

   (a) Оценка точности моделирования: вычисление математического ожидания и дисперсии, сравнение полученных оценок с соответствующими теоретическими значениями.

4) Графическое представление функции плотности распределения и интегральной функции распределения.

2 Ход работы

1) Написать и отладить подпрограммы получения дискретных псевдослучайных чисел в соответствии с алгоритмами, приведенными в описании.

2) Осуществить проверку точности моделирования полученных датчиков псевдослучайных чисел.

3) Отлаженные подпрограммы собрать в единый пакет "Дискретные распределения"и создать в головной программе "меню"для всего пакета.

3 Равномерное распределение

Тестирование проводилось на выборке объемом $n = 1000$ значений и параметрами равномерного закона распределения

$$a = 0$$
$$b = 1$$

N	Expectation experimental	Expectation theoretical	Expectation error	Dispersion experimental	Dispersion theoretical	Dispersion error
1000	0,513843	0,5	0,013843	0,080637	0,083333333	-0,002696333
1000	0,507503	0,5	0,007503	0,083225	0,083333333	-0,000108333
1000	0,500015	0,5	0,000015	0,082952	0,083333333	-0,000381333

Рис. 1: Результаты для равномерного распределения

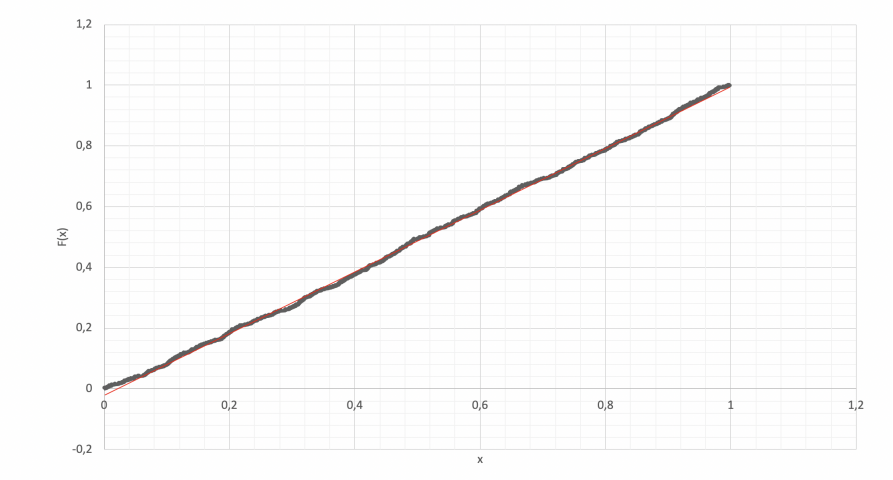

Рис. 2: Функция распределения

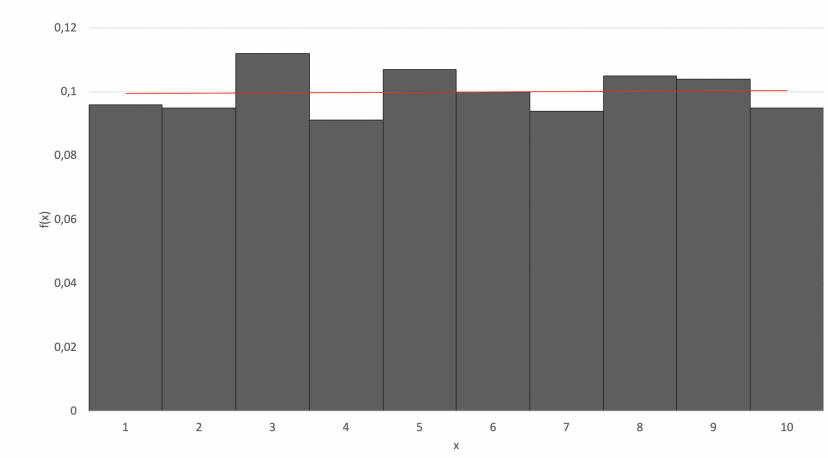

Рис. 3: Плотность вероятности

4 Биномиальное распределение

Тестирование проводилось на выборке объемом $n = 1000$ значений и параметрами биномиального закона распределения

$$k = 40$$
$$p = 0.5$$

N	Expectation experimental	Expectation theoretical	Expectation error	Dispersion experimental	Dispersion theoretical	Dispersion error
1000	19,994	20	0,006	10,017964	10	-0,017964
1000	20,046	20	-0,046	9,753884	10	0,246116
1000	20,079	20	-0,079	10,612759	10	-0,612759

Рис. 4: Результаты для равномерного распределения

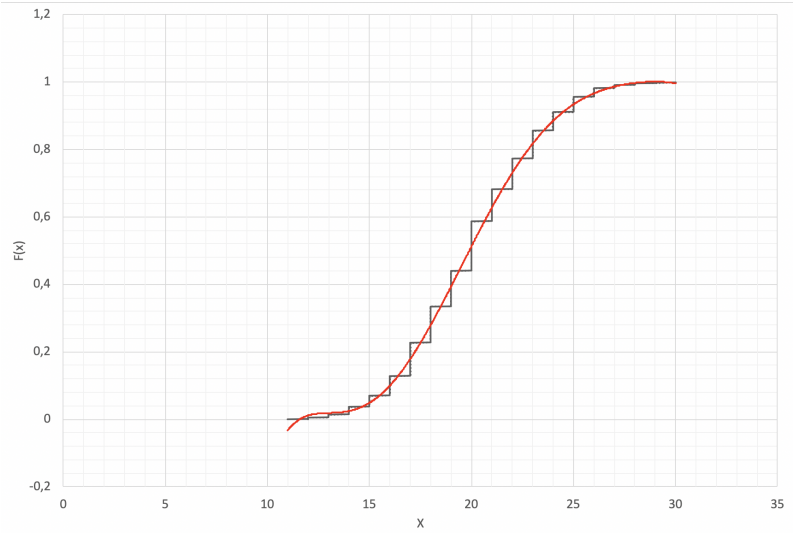

Рис. 5: Функция распределения

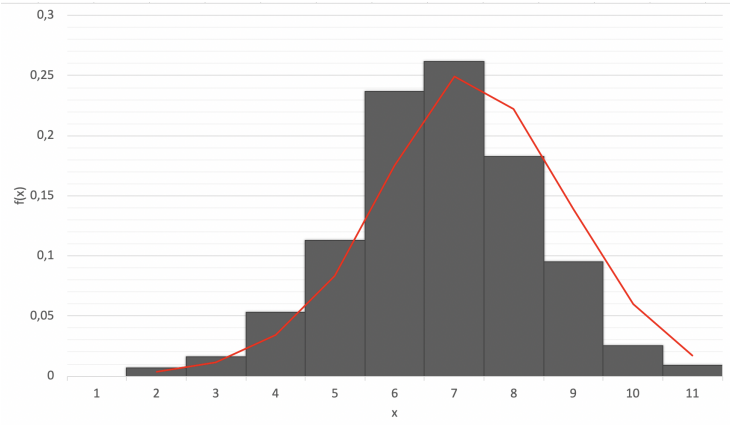

Рис. 6: Плотность вероятности

5 Геометрическое распределение

Тестирование проводилось на выборке объемом $n = 1000$ значений и параметрами геометрического закона распределения

$$p = 0.5$$

N	Expectation experimental	Expectation theoretical	Expectation error	Dispersion experimental	Dispersion theoretical	Dispersion error
1000	1,04	1	-0,04	2,1784	2	0,1784
1000	0,945	1	0,055	1,807975	2	-0,192025
1000	1,043	1	-0,043	2,095151	2	0,095151

Рис. 7: Результаты для геометрического распределения

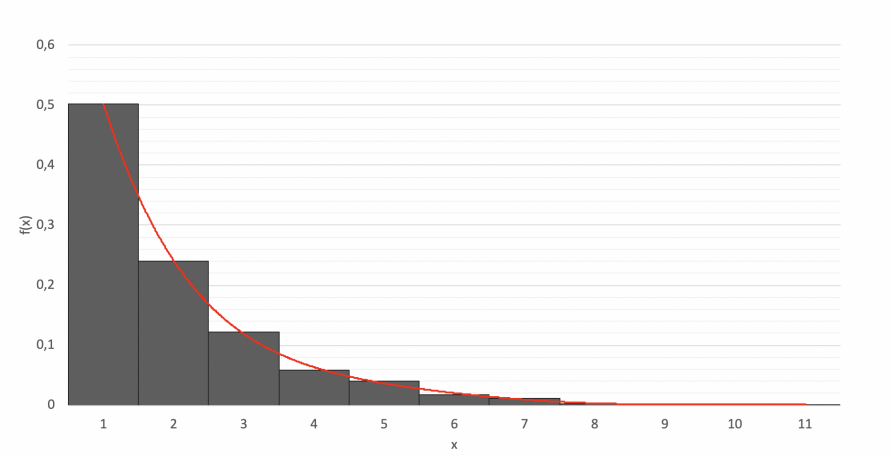

Рис. 8: Функция распределения

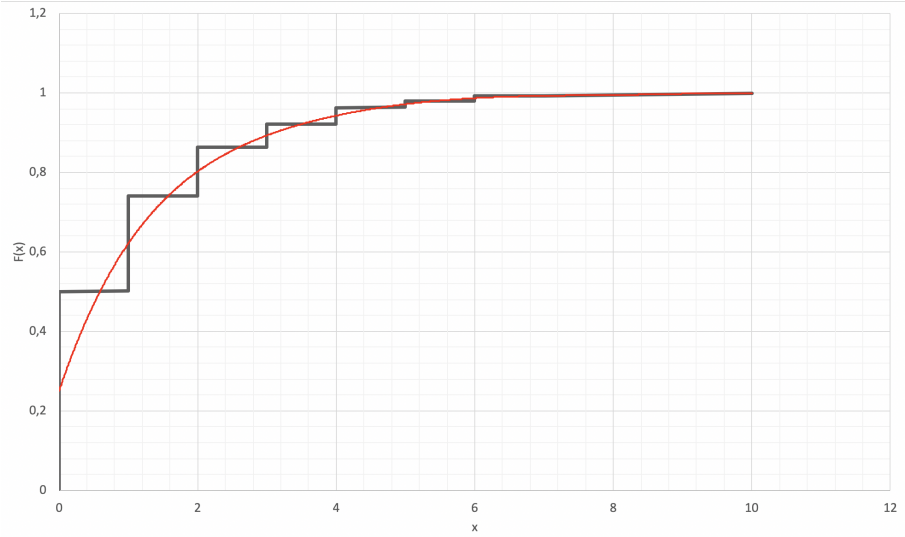

Рис. 9: Плотность вероятности

6 Пуассоновское распределение

Тестирование проводилось на выборке объемом $n = 1000$ значений и параметрами Пуссоновского закона распределения

$$\lambda = 0.5$$

N	Expectation experimental	Expectation theoretical	Expecation error	Dispersion experimental	Dispersion theoretical	Dipercion error
1000	4,038	4	-0,038	4,170556	4	0,170556
1000	3,991	4	0,009	4,250919	4	0,250919
1000	4,011	4	-0,011	4,208879	4	0,208879

Рис. 10: Результаты для Пуассоновского распределения

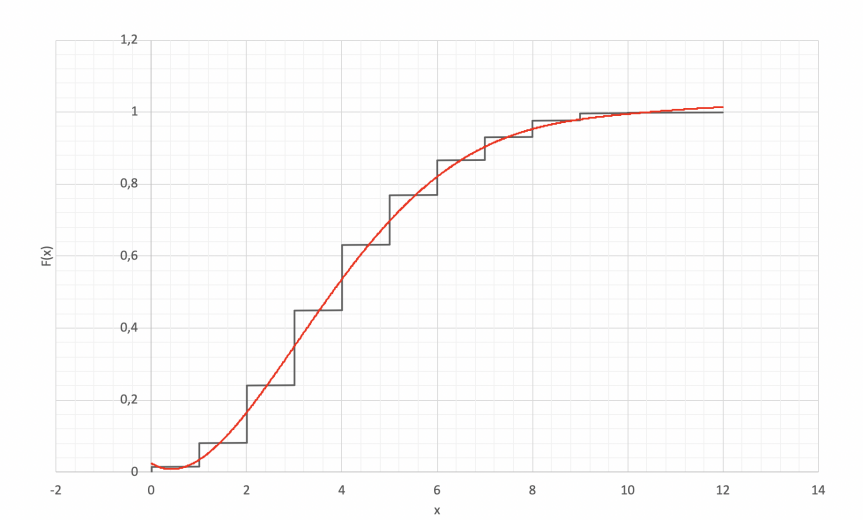

Рис. 11: Функция распределения

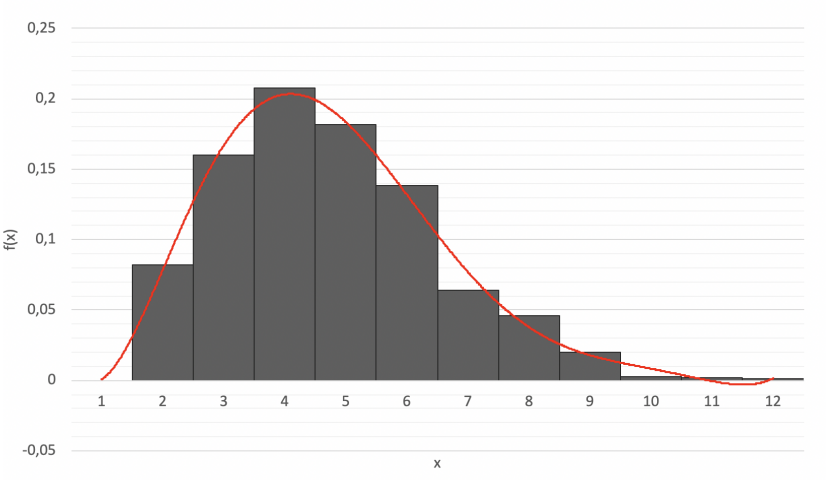

Рис. 12: Плотность вероятности

7 Вывод

По представленным результатам математического ожидания и дисперсии можно сказать что экспериментальные значения примерно соответствуют теоретическим значениям. Так же, представлены графикам дискретных распределений можно сказать, что они достаточно близки теоретически построенным графикам соответствующих распределений, по форме и принимаемым значениям. Из этого можно сделать вывод, что разработанные программные датчики дискретных случайных величин достаточно точны и подходят для генерации дискретной случайной величины.

8 Приложение

8.1 Код на языке C++

Листинг 1: writer.hpp

```
#ifndef model_hpp
#define model_hpp

#include <iostream>
#include <numeric>
#include <array>
#include <cmath>
#include <vector>
#include <random>

#define MAX_VAL 1
#define MIN_VAL 0

template <class TDistr, class... Args>
class RandomGenerator
{
public:
    RandomGenerator(const Args &... args);
    auto getValue();

private:
    std::random_device rd_;
    std::mt19937 generator_;
    TDistr distr_;
};

template <class G, std::size_t S>
class Model_t
{
public:
    template <class... Args>
    Model_t(const Args &... args);

    void generate();
    double getExpectation();
    double getDispersion();
    const std::array<double, S> &getCorrelation();
    const std::array<double, S> &getData();
    const std::array<double, S> getSortedData();
    const std::vector<std::pair<double, double>> &getProbDensity();

private:
    void countExpectation();
    void countDispersion();
    void countCorrelation();
    void countProbDensity(const int partitions);

    G randGenerator_;
    std::array<double, S> value_arr_;
    std::array<double, S> correlation_coef_arr_;
    std::vector<std::pair<double, double>> prob_density_;
    double expectation_;
    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_,
        functor) / S;
159 }
160
161 template <class G, std::size_t S>
162 void Model_t<G, S>::countCorrelation()
163 {
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;
            h += step)
192     {
193         int count = 0;
194         for (auto &el : value_arr_)
195         {
196             if (el >= h - half_step && el < h + half_step)
197             {
198                 count++;
199             }
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 ExelWriter
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 dispercion);
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 dispercion)
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() = dispercion;
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) +
            " [STEP]");
112     wks.Cell(1, wks.ColumnCount() + 1).Value() = std::string("N = " + std::to_string(n) +
            " [VALUE]");
113     for (auto i = 0; i < data.size(); i++)
114     {
115         wks.Cell(i + 2, wks.ColumnCount() - 1).Value() = data.at(i).first;
116         wks.Cell(i + 2, wks.ColumnCount()).Value() = data.at(i).second;
117         wks.Cell(i + 2, 1).Value() = i;
118     }
119 }
120
121 void ExelWriter::initTables()
122 {
123     doc_.Workbook().AddWorksheet("Summary");
124     auto wks = doc_.Workbook().Worksheet("Summary");
125     wks.Cell("A1").Value() = "N";
126     wks.Cell("B1").Value() = "Expectation experimental";
127     wks.Cell("C1").Value() = "Expectation theoretical";
128     wks.Cell("D1").Value() = "Dispersion experimental";
129     wks.Cell("E1").Value() = "Dispersion theoretical";
130
131     doc_.Workbook().AddWorksheet("Data");
132     doc_.Workbook().AddWorksheet("Sorted Data");
133     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  }
```