

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

ИНСТИТУТ КОМПЬЮТЕРНЫХ НАУК И ТЕХНОЛОГИЙ

## Лабораторная работа

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

Выполнил:

Каргалов Л.А.

Проверил:

Чуркин В. В.

3 марта 2020 г.

Санкт-Петербург  
2020

## Содержание

1	Цель работы	3
2	Ход работы	4
3	Равномерное распределение	5
4	Биномиальное распределение	6
5	Геометрическое распределение	7
6	Пуассоновское распределение	8
7	Вывод	9
8	Приложение	10
8.1	Код на языке C++ . . . . .	10

## 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:
  - (а) 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 параметра 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

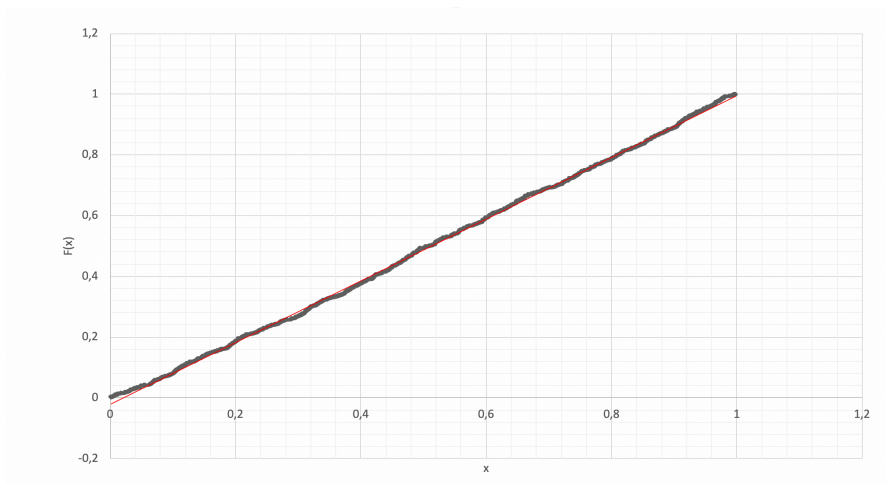


Рис. 2: Функция распределения

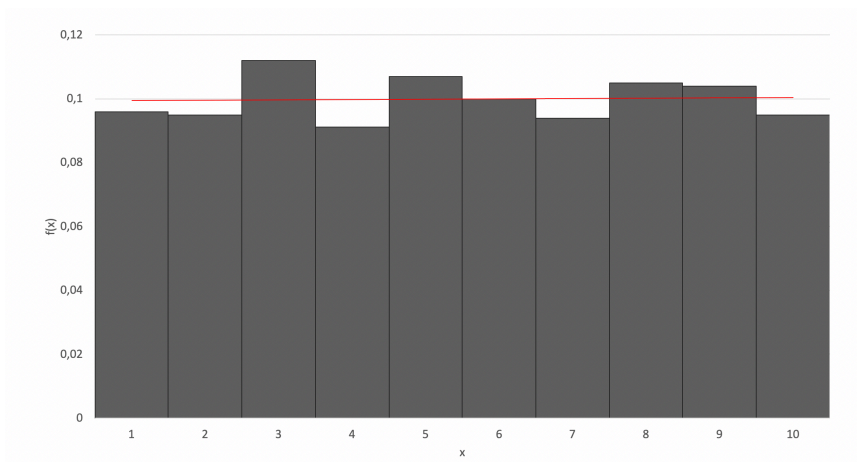


Рис. 3: Плотность вероятности

## 4 Биномиальное распределение

Тестирование проводилось на выборке объемом  $n = 1000$  значений параметры 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

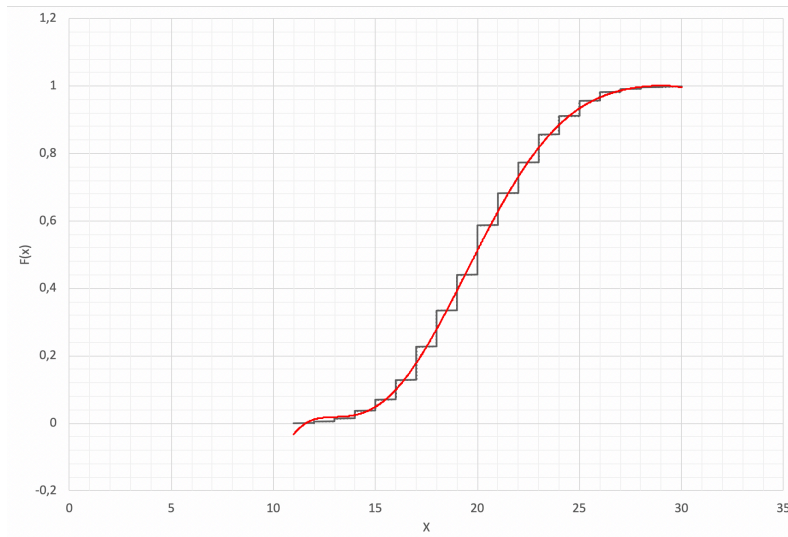


Рис. 5: Функция распределения

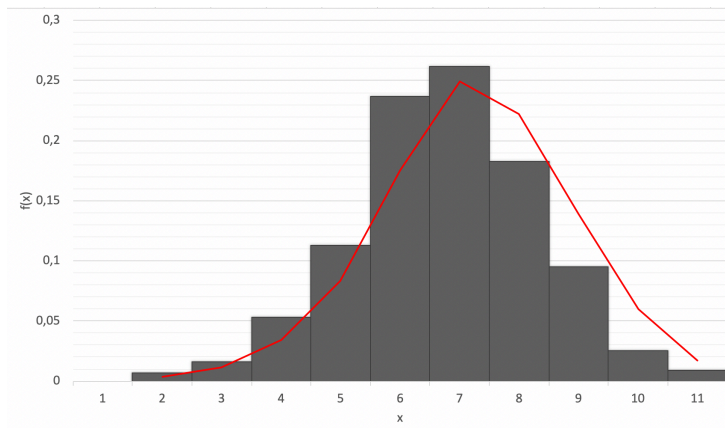


Рис. 6: Плотность вероятности

## 5 Геометрическое распределение

Тестирование проводилось на выборке объемом  $n = 1000$  значений параметра 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

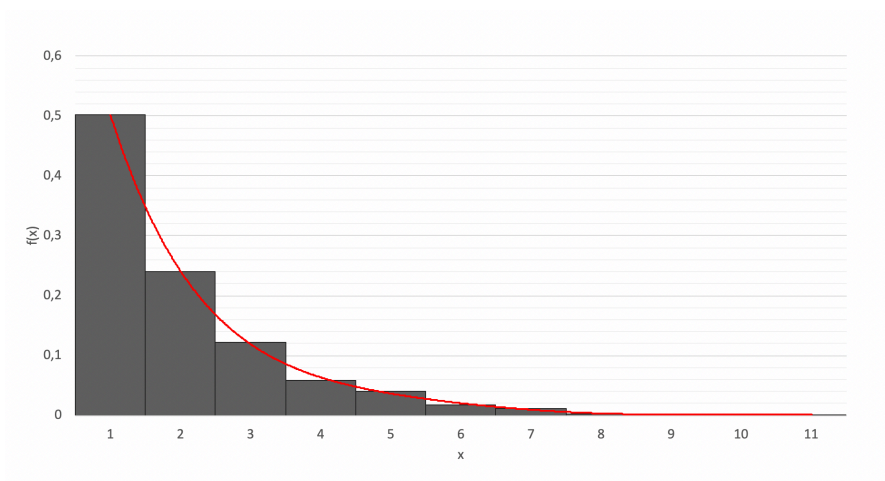


Рис. 8: Функция распределения

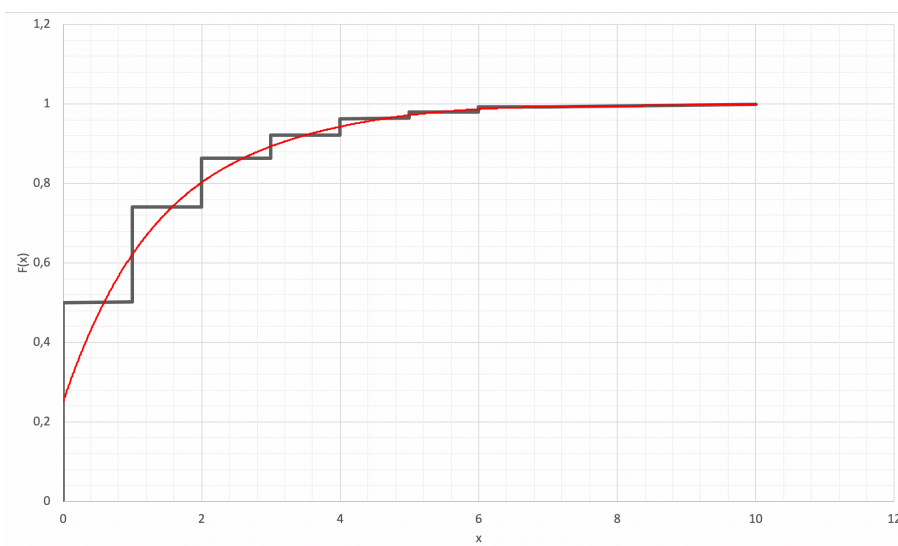


Рис. 9: Плотность вероятности

## 6 Пуассоновское распределение

Тестирование проводилось на выборке объемом  $n = 1000$  значений параметра пуассоновского закона распределения

$$\lambda = 0.5$$

N	Expectation experimental	Expectation theoretical	Expection 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

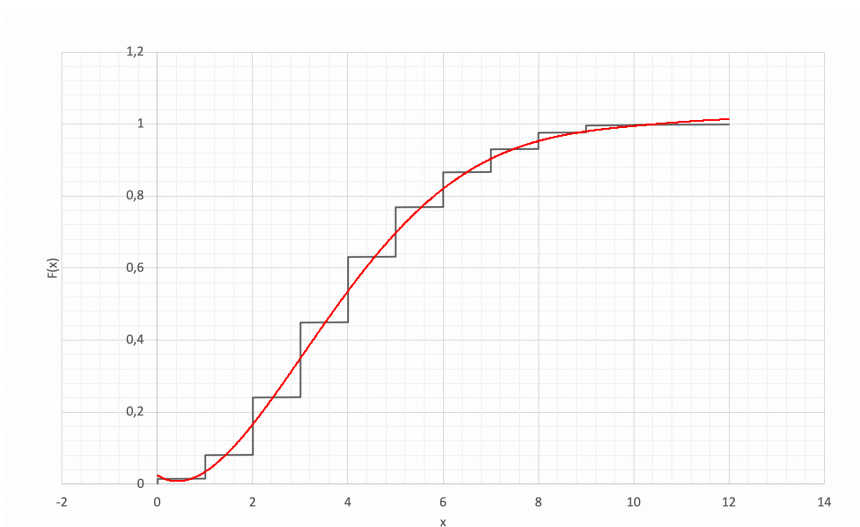


Рис. 11: Функция распределения

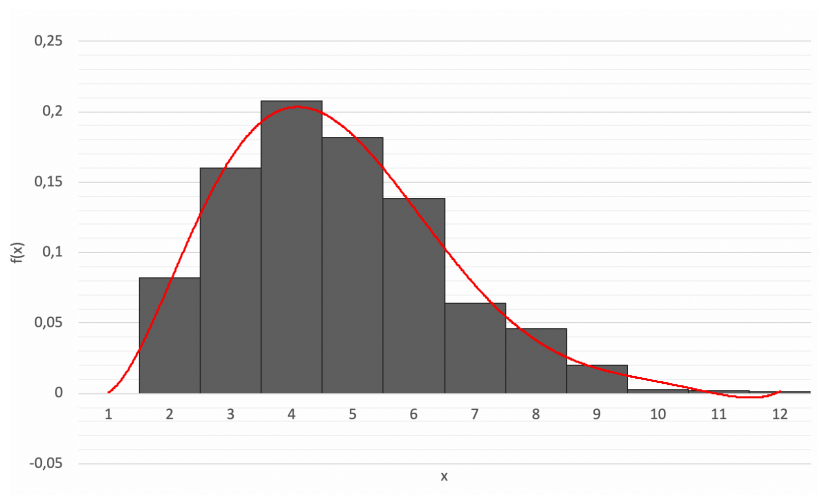


Рис. 12: Плотность вероятности



## 7 Вывод

В ходе работы были разработаны программные датчики дискретных случайных величин. Исследование характеристик моделируемых датчиков показало, что точность моделирования для всех датчиков достаточно высока.

## 8 Приложение

### 8.1 Код на языке C++

Листинг 1: writer.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 //===== Implementation =====
63
64 template <class TDistr, class... Args>
65 RandomGenerator<TDistr, Args...>::RandomGenerator(const Args &... args)
66     : rd_(),
67       generator_(rd_()),
68       distr_(args...)
69 {
70 }
71
72 template <class TDistr, class... Args>
73 auto RandomGenerator<TDistr, Args...>::getValue()
74 {
75     return distr_(generator_);
76 }
77
78 template <class G, std::size_t S>
79 template <class... Args>
80 Model_t<G, S>::Model_t(const Args &... args)
81     : randGenerator_(args...)
82 {
83 }
84
85 template <class G, std::size_t S>
86 void Model_t<G, S>::generate()
87 {
88     std::srand(unsigned(std::time(0)));
89     for (std::size_t i = 0; i < S; i++)
90     {
91         value_arr_.at(i) = randGenerator_.getValue();
92     }
93
94     countExpectation();
95     countDispersion();
96     countCorrelation();
97     countProbDensity(10);
98 }
99
100 template <class G, std::size_t S>
101 double Model_t<G, S>::getExpectation()
102 {
103     return expectation_;
104 }
105
106 template <class G, std::size_t S>
107 double Model_t<G, S>::getDispersion()
108 {
109     return dispersion_;
110 }
111

```

```

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

```

```

168         for (std::size_t i = 0; i < S; i++)
169         {
170             v2 += pow(value_arr_.at(i) - expectation_, 2);
171         }
172
173         correlation_coef_arr_.at(f) = v1 / v2;
174     }
175 }
176
177
178 template <class G, std::size_t S>
179 void Model_t<G, S>::countProbDensity(const int partitions)
180 {
181     auto sorted_data = this->getSortedData();
182
183     double step = (sorted_data.back() - sorted_data.front()) / partitions;
184     double half_step = step / 2.0000;
185
186     for (double h = sorted_data.front() - half_step; h <= sorted_data.back() + half_step;
187          h += step)
188     {
189         int count = 0;
190         for (auto &el : value_arr_)
191         {
192             if (el >= h - half_step && el < h + half_step)
193             {
194                 count++;
195             }
196         }
197         prob_density_.push_back({h, (count / static_cast<double>(S))});
198     }
199 }
200 #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     ExcelWriter(const std::string &filepath);
20     ~ExcelWriter();
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");
136     doc_.Workbook().AddWorksheet("Prob Density");
137 }
138
139 template <class G>
140 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 <OpenXLXS/OpenXLXS.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<class Gen, class ... Args>
27 void gen_tester(std::string file_path, const Args& ... args)
28 {
29     const std::size_t S = 1000;
30     Model_t<Gen, S> m1(args ...);
31     Model_t<Gen, S> m2(args ...);
32     Model_t<Gen, S> m3(args ...);
33
34     m1.generate();
35     m2.generate();
36     m3.generate();
37
38     ExelWriter exel(file_path);
39     exel << m1 << m2 << m3;
40
41 }
42
43 void lab2()
44 {
45     gen_tester<UniformGenerator>("./lab2_uniform_gen.xlsx", 0, 1);
46     gen_tester<BinomialGenerator>("./lab2_binomial_gen.xlsx", 40, 0.5);
47     gen_tester<GeometricGenerator>("./lab2_geometric_gen.xlsx", 0.5);
48     gen_tester<PoissonGenerator>("./lab2_poisson_gen.xlsx", 4);
49 }
50

```



```
51 int main(int argc, const char *argv[])
52 {
53
54     lab2();
55     return 0;
56 }
```