

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ  
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к первой лабораторной работе

«Генерация случайных чисел с заданным распределением»

по дисциплине «Моделирование»

Выполнил студент \_\_\_\_\_ Симонова Валерия Владимировна  
Ф.И.О.

Группы \_\_\_\_\_ ИВ – 621

Работу принял \_\_\_\_\_ ассистент кафедры ВС  
Я. В. Петухова  
подпись

Новосибирск – 2020

# Содержание

Постановка задачи.....	3
Теоретические сведения .....	3
Результаты экспериментов .....	4
Выводы .....	7
Приложение.....	8

## Постановка задачи

В рамках лабораторной работы необходимо смоделировать генерацию независимых случайных величин:

1. Непрерывное распределение случайных величин методом отбраковки.
2. Дискретное распределение случайных величин с возвратом.
3. Дискретное распределение случайных величин без возврата.

## Теоретические сведения

### Метод отбраковки

Один из методов моделирования непрерывной случайной величины – метод отбраковки. Он используется, когда функция задана аналитически. График функции вписывают в прямоугольник. На ось  $Y$  и на ось  $X$  подают по случайному равномерно распределенному числу из ГСЧ. Если точка в пересечении этих двух координат лежит ниже кривой плотности вероятности, то событие  $X$  произошло, иначе нет.

Недостаток метода – точки, оказавшиеся выше кривой распределения плотности вероятности, отбрасываются как ненужные, и время на их вычисление оказывается напрасным. Метод применим только для аналитических функций плотности вероятности.

Алгоритм:

- В цикле генерируется два случайных числа от 0 до 1.
- Числа масштабируются в шкалу  $X$  и  $Y$ , и проверяется попадание точки со сгенерированными координатами под график заданной функции  $Y = f(X)$ .
- Если точка находится под графиком функции, то событие  $X$  произошло с вероятностью  $Y$ , иначе точка отбрасывается.

### Дискретное распределение с возвратом

Если  $x$  – дискретная случайная величина, принимающая значения  $x_1 < x_2 < \dots < x_i < \dots$  с вероятностями  $p_1 < \dots < p_i < \dots$ , то таблица вида

$x_1$	$x_2$	$\dots$	$x_i$
$p_1$	$p_2$	$\dots$	$p_i$

называется распределением дискретной случайной величины.

Функция распределения случайной величины с таким распределением имеет вид

$$F(x) = \begin{cases} 0, & \text{при } x < x_1 \\ p_1, & \text{при } x_1 \leq x < x_2 \\ p_1 + p_2, & \text{при } x_2 \leq x < x_3 \\ \dots & \dots \\ p_1 + p_2 + \dots + p_{n-1}, & \text{при } x_{n-1} \leq x < x_n \\ 1, & \text{при } x \geq x_n \end{cases}$$

### Дискретное распределение без возврата

Есть  $n$  случайных величин с одинаковой вероятностью (при следующих выборках вероятность распределяется поровну между величинами), следует выбрать  $3/4$   $n$  следующих величин без повторений, проделать это большое количество раз и посчитать частоты этих величин.

## Результаты экспериментов

### Метод отбраковки

Пусть случайная величина  $X$  задана плотностью вероятности:

$$f(x) = \begin{cases} 0, & \text{если } x < 1 \\ a(x + 1), & \text{если } 1 \leq x \leq 3 \\ 0, & \text{если } x > 3 \end{cases}$$

Известно, что несобственный интеграл от плотности вероятности есть вероятность достоверного события (условие нормировки):

$$\int_{-\infty}^{\infty} f(x) dx = P\{-\infty < X < \infty\} = P\{\Omega\} = 1$$

Исходя из этого свойства, можем найти параметр  $a$ :

$$\int_a^b f(x)dx = \int_1^3 a(x+1)dx = 1$$

$$\int_{-\infty}^1 0dx + \int_1^3 a(x+1)dx + \int_3^{\infty} 0dx = 0 + \int_1^3 a(x+1)dx + 0 = 1$$

$$a \frac{x^2}{2} + ax \Big|_1^3 = a \frac{3^2}{2} + 3a - a \frac{1^2}{2} - a = 4,5a + 3a - 0,5a - a = 1$$

$$6a = 1$$

$$a = \frac{1}{6}$$

Итоговая функция распределения:

$$F(x) = \begin{cases} 0, & \text{если } x < 1 \\ \frac{1}{6}x^3, & \text{если } 1 \leq x \leq 3 \\ 0, & \text{если } x > 3 \end{cases}$$

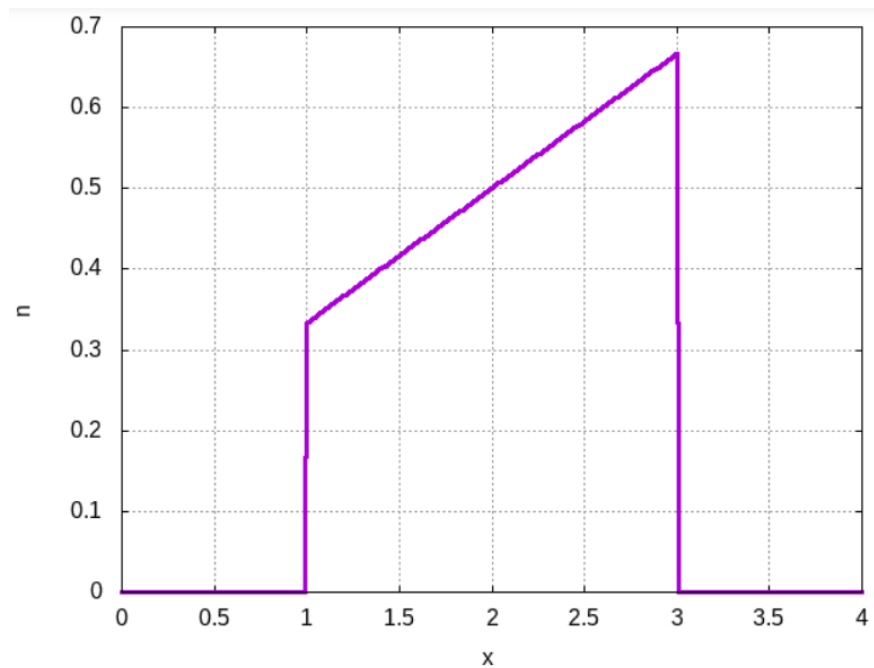


Рисунок 1. График функции распределения.

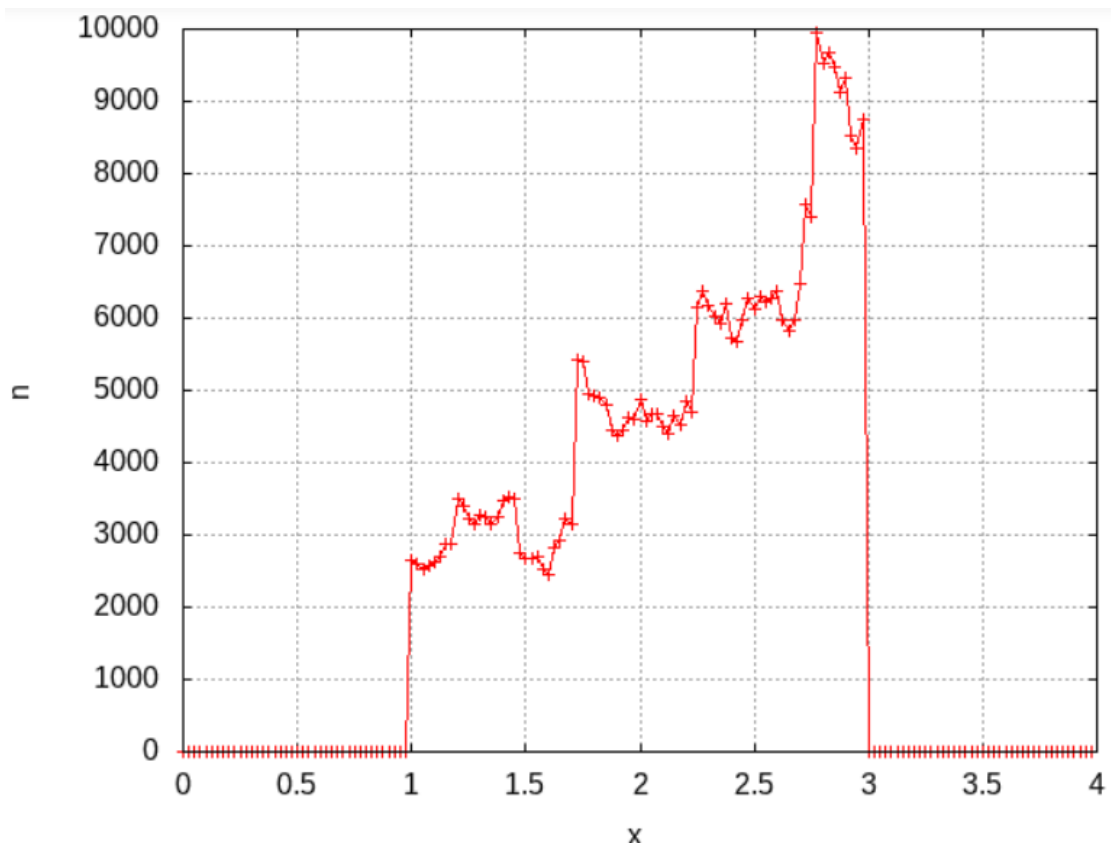


Рисунок 2. Результаты работы метода отбраковки.

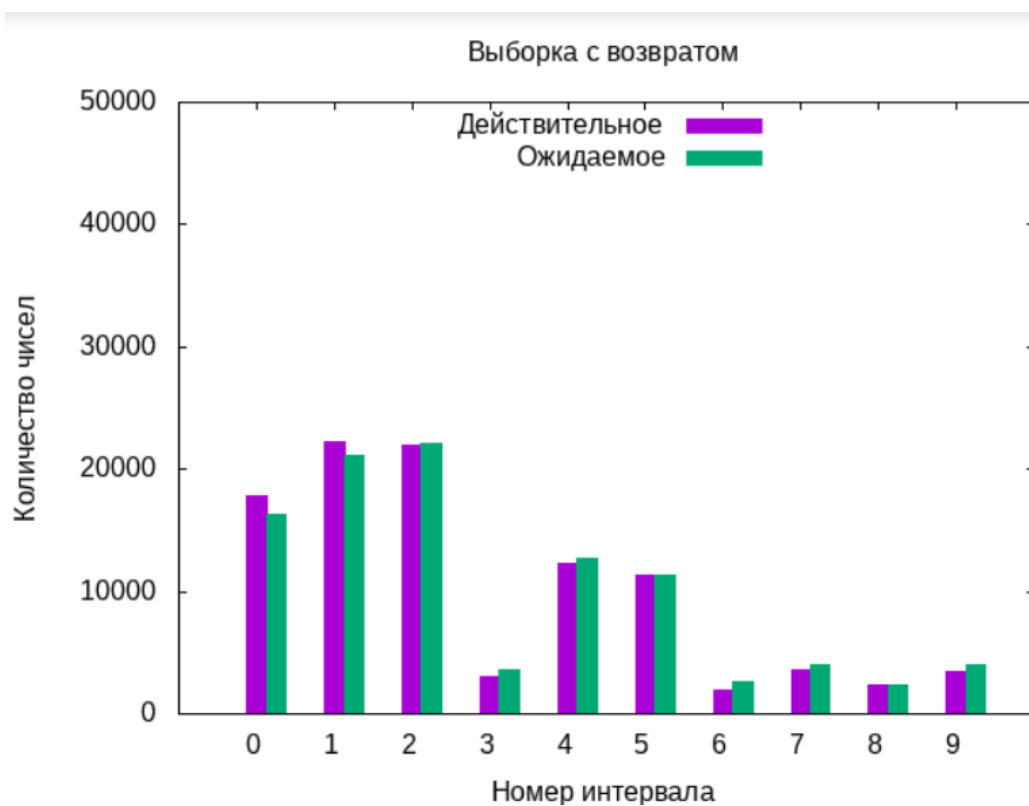


Рисунок 3. Результаты работы моделирования дискретной с.в с возвратом.

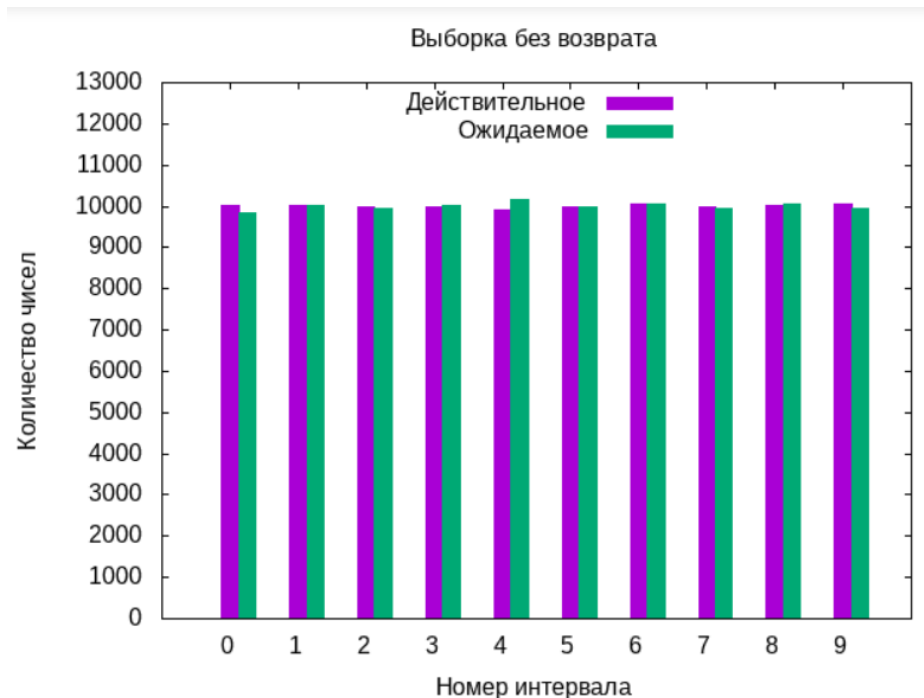


Рисунок 4. Результаты работы моделирования дискретной с.в без возврата.

## Выводы

В ходе работы были изучены методы моделирования случайных величин, реализованы соответствующие алгоритмы. Результаты экспериментов представлены на графиках и гистограммах.

Метод отбраковки достаточно эффективен – числа были сгенерированы в соответствии с заданным законом распределения: по графикам можно заметить, что экспериментально полученный схож с теоретическим графиком плотности распределения данной функции. Однако у данного метода существуют недостатки. Во-первых, точки, оказавшиеся выше кривой распределения плотности вероятности, отбрасываются как ненужные, и это может заметно увеличить время работы алгоритма. Во-вторых, метод применим только для аналитических функций и неэффективен для распределений с длинными «хвостами», т.к. будут часты повторные испытания.

По результатам моделирования дискретных случайных величин с реализацией выборки с возвратом и без возврата можно заметить, что среднее отклонение требуемого распределения от действительного достаточно мало, из чего можно сделать вывод, что используемый генератор псевдослучайных чисел имеет равномерное распределение с малой долей погрешности.

# Приложение

## Листинг

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <time.h>
#include <cmath>
#include <string>
#include <string.h>
#include <fstream>
#include <vector>
#include <chrono>
#include <random>

using namespace std;
#define DEF 1
#define MAX_EPOCH 100000

double getRand(double a, double b) {
    long seed = chrono::system_clock::now().time_since_epoch().count();

    default_random_engine rand_generator(seed);
    uniform_real_distribution<double> distribution(a,b);

    return distribution(rand_generator);
}

double get_rand_congruent() {
    return ((double) rand()) / RAND_MAX;
}

int get_rand_congruent_int() {
    return rand();
}

double get_rand_mersenne() {
    static random_device rd;
    static mt19937_64 mersenne(rd());
    return ((double) mersenne()) / RAND_MAX;
}

double f(double x) {
    double y, a = 1.0 / 4.0;

    if (1.0 <= x && x <= 3.0) {
        y = a * (x + 1);
    } else {
        y = 0.0;
    }

    return y;
}
```



```

double rejection_method (double a, double b, double c, double (*f)(double)) {
    double x1, x2;

    do {
        #if DEF == 1
            x1 = getRand(0.0, 1.0);
            x2 = getRand(0.0, 1.0);
        #elif CONG
            x1 = get_rand_congruent();
            x2 = get_rand_congruent();
        #elif MER
            x1 = get_rand_mersenne();
            x2 = get_rand_mersenne();
        #endif
    } while (f(a + (b - a) * x1) <= c * x2);

    return a + (b - a) * x1;
}

vector<double> get_started_distribution(uint64_t size) {
    vector<double> distribution(size);
    double residue_chance = 1, rand_num;

    for (uint64_t i = 0; i < size - 1; i++) {
        #if DEF == 1
            rand_num = getRand(0.0, 1.0);
        #elif CONG
            rand_num = get_rand_congruent();
        #elif MER
            rand_num = get_rand_mersenne();
        #endif
        double probability = abs(remainder(rand_num, residue_chance));
        distribution[i] = probability;
        residue_chance -= probability;
    }

    distribution[size - 1] = residue_chance;

    double sum = 0.0;
    for (int i = 0; i < size; i++)
        sum += distribution[i];

    return distribution;
}

void repeat(int generated_numbers) {
    vector<double> distribution = get_started_distribution(generated_numbers);
    double chance;
    vector<int> hit(generated_numbers);

    for (int i = 0; i < MAX_EPOCH; i++) {
        #if DEF == 1
            chance = getRand(0.0, 1.0);
        #elif CONG
            chance = get_rand_congruent();

```

```

        #elif MER
            chance = get_rand_mersenne();
        #endif
        double sum = 0;

        for (int j = 0; j < generated_numbers; j++) {
            sum += distribution[j];

            if (chance < sum) {
                hit[j]++;
                break;
            }
        }

        }

    ofstream output("repeat");
    output << "Number\tДействительное\tОжидаемое" << endl;
    float step = ((float) MAX_EPOCH) / generated_numbers;
    float sum = 0.0;
    for (int i = 0; i < generated_numbers; i++)
        output << i << "\t" << hit[i] << "\t" << distribution[i] * MAX_EPOCH
<< endl;
    output.close();
    system("gnuplot repeat.plt");
}

vector<int> get_array_random_numbers(int N, int max) {
    vector<int> array(N);
    for (int i = 0; i < N; i++)
        array[i] = get_rand_congruent_int() % max;

    return array;
}

vector<int> get_counts_unique(int N, int max) {
    vector<int> counts(max);
    vector<int> random_numbers = get_array_random_numbers(N, max);
    for (int elemet : random_numbers)
        counts[elemet]++;

    return counts;
}

vector<double> get_probablity(int N, int size) {
    vector<double> probablity(size);
    vector<int> counts = get_counts_unique(N, size);
    for (int i = 0; i < size; i++)
        probablity[i] = ((double) counts[i]) / N;

    return probablity;
}

void no_repeat(int generated_numbers) {
    vector<int> nums, temp, hit(generated_numbers);

```

```

int k = 3 * generated_numbers / 4;
int part = MAX_EPOCH / k + 1;
for (int i = 0; i < generated_numbers; i++)
    temp.push_back(i);
for (int j = 0; j < part; j++) {
    nums = temp;
    if (j == part - 1)
        k = MAX_EPOCH % k;
    for (int i = 0; i < k; i++) {
        double distribution_unit = 1.0 / (generated_numbers - i);
        double probability = get_rand_congruent();
        double ratio = probability / distribution_unit;
        int it = static_cast<int>(trunc1(ratio));
        hit[nums[it]]++;
        nums.erase(nums.begin() + it);
    }
}

ofstream output("no_repeat");
float step = ((float) MAX_EPOCH) / generated_numbers;
vector<int> counts = get_counts_unique(MAX_EPOCH, 10);
output << "Number\tДействительное\tОжидаемое" << endl;
for (int i = 0; i < generated_numbers; i++)
    output << i << "\t" << hit[i] << "\t" << counts[i] << endl;
output.close();
system("gnuplot no_repeat.plt");
}

void _discrete_distribution() {
    repeat(10);
    no_repeat(10);
}

void continuous_distribution() {
    int size = 4000;
    vector<uint64_t> hit(size);
    for (int i = 0; i < MAX_EPOCH * 100; i++) {
        double p = rejection_method(0.0, 4.0, 1.0, f);
        auto it = static_cast<uint64_t>(trunc1(p * 1000));
        hit[it]++;
    }

    ofstream output("reject");
    for (int i = 0; i < size; i += 25) {
        output << static_cast<double>(i) / 1000.0 << "\t" << hit[i] << endl;
    }

    output.close();
    system("gnuplot reject.plt");
}

void distribution_density() {
    ofstream output("density");

```

```

    for (double i = 0.0; i < 4.0; i += 0.01)
        output << i << " " << f(i) << endl;
    output.close();
    system("gnuplot density.plt");
}

int main(int argc, char const *argv[]) {
    #if CONG
        srand(time(0));
    #endif
    distribution_density();
    continuous_distribution();
    _discrete_distribution();

    return 0;
}

```