Федеральное государственное бюджетное образовательное учреждение
высшего образования «Сибирский государственный университет
телекоммуникаций и информатики»

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №1

по дисциплине «Моделирование»

Выполнила:
Студент гр. ИВ-622
Корягина В.А.

Проверила:
Ассистент Кафедры ВС
Петухова Я.В.

Новосибирск 2020

СОДЕРЖАНИЕ

Постановка задачи

Реализовать:

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

Теоретические сведения

1. Непрерывное распределение

В некоторых случаях требуется точное соответствие заданному закону распределения при отсутствии эффективных методов генерации. В такой ситуации для ограниченных с.в. можно использовать метод «Отбраковки». Функция плотности распределения вероятностей с.в. $f\eta(x)$ вписывается в прямоугольник $(a, b) \times (0, c)$, такой, что a и b соответствуют границам диапазона изменения с.в. η, а c – максимальному значению функции плотности её распределения. Тогда очередная реализация с.в. определяется по следующему алгоритму:

Шаг 1. Получить два независимых случайных числа ξ_1 и ξ_2.

Шаг 2. Если $f\eta(a + (b - a)\ \xi_1) > c\,\xi_2$ то выдать $a + (b - a)\ \xi_1$ в качестве результата. Иначе повторить Шаг 1.

Неэффективность алгоритма для распределений с длинными “хвостами” очевидна, поскольку в этом случае часты повторные испытания.

2. Дискретное распределение

Случайная величина ξ называется дискретной, если она может принимать дискретное множество значений ($x1$, $x2$, $x3$ … , xn). Дискретная случайная величина ξ определяется таблицей (распределением случайной величины ξ) или аналитически в виде формулы:

$$\xi = (x_1\ x_2\ x_3 \ldots x_n\ \ p_1\ p_2\ p_3 \ldots p_n\ ) \text{ , где:}$$

- $(x_1 , x_2\ \ x_3 , \ldots x_n)$ – возможные значения величины ξ;
- $(p , p_2\ \ p_3 , \ldots p_n)$ – соответствующие им вероятности:

$$P(\xi = x_i) = p_i$$

Числа $(x_1 , x_2\ \ x_3 , \ldots x_n)$ могут быть любыми, а вероятности $(p , p_2$ $p_3 , \ldots p_n)$ удовлетворяют условиям: $p_i > 0$ и $\sum_{i=1}^{n} p_i = 1$.

Функция распределения выглядит так:

$$F(x) = \sum_{x_k<x} P(X = x_k)$$

Наиболее общий случай построения генератора дискретных случайных величин основывается на следующем алгоритме. Пусть имеется таблица пар (x_i , p_i). Тогда кумулятивную сумму можно представить полуинтервалом [0, 1), разбитым на полуинтервалы $[v_{i-1} , v_i)$, $v_0 = 0$, $v_n = 1$ длины p_i . Случайная величина ξ с неизбежностью принадлежит одному из этих полуинтервалов, тем самым определяя индекс дискретного значения. Номер полуинтервала, очевидно, определяется как:

$$min\{i \mid \xi < v_i\} = min\{i \mid \xi < \sum_{j=1}^{i} p_j\}$$

При дискретном распределении без возврата есть n случайных величин с одинаковой вероятностью (при следующих выборках вероятность распределяется поровну между величинами), мы выбираем $3 \cdot n / 4$ следующих величин без повторений, проделываем это большое количество раз и считаем частоты этих значений.

Ход работы

Определим функцию плотности распределения:

$$f(x) = \begin{cases} 0, \quad x \le 0 \\ \dfrac{a * x^3}{3}, \quad 0 < x \le 3 \\ 0, \quad x > 3 \end{cases}$$

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

$$\int_{-\infty}^{\infty} f(x)dx = 1$$

Тогда найдем а:

$$\int_{-\infty}^{0} 0\,dx + \int_{0}^{3} \frac{a * x^3}{3}\,dx + \int_{3}^{\infty} 0\,dx = \frac{a * x^3}{3}\Big|_{0}^{3} = 1$$

$$a = 1.3334$$

Таким образом мы получаем плотность распределения:

$$f(x) = \begin{cases} 0, \quad x \le 0 \\ 0.444 * x^3, \quad 0 < x \le 3 \\ 0, \quad x > 3 \end{cases}$$

```
        | 0, x ≤ 0
f(x) = | (a * x^3) / 3, 0 < x ≤ 3
        | 0, x > 3
Коэффициент: 1.3333333333333899
```

Рисунок 1 - Вывод расчета коэффициента а

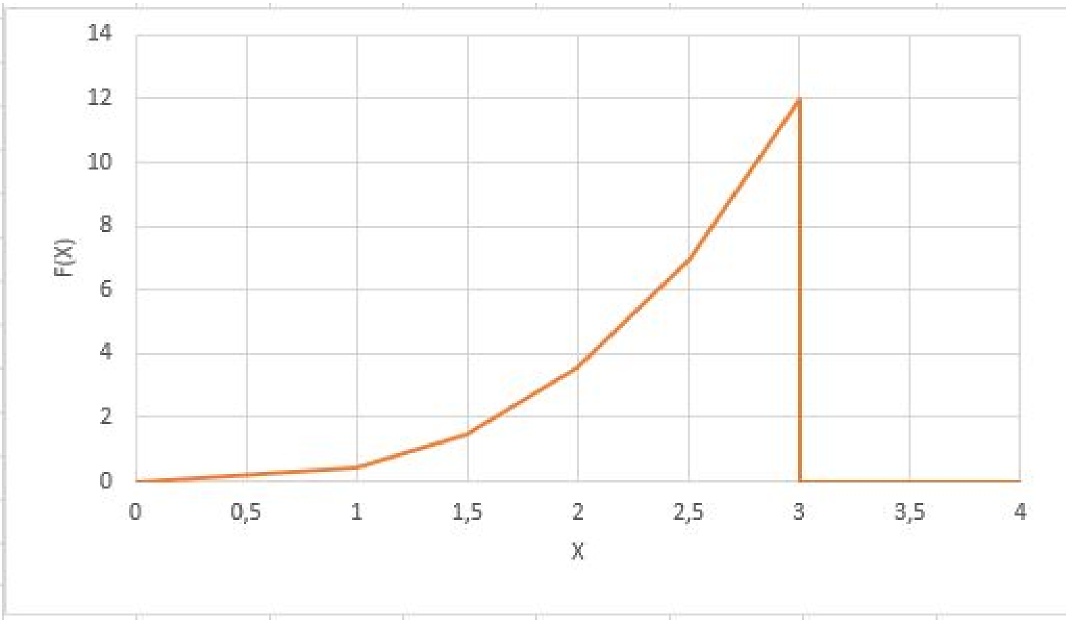

Рисунок 1 – График функции плотности

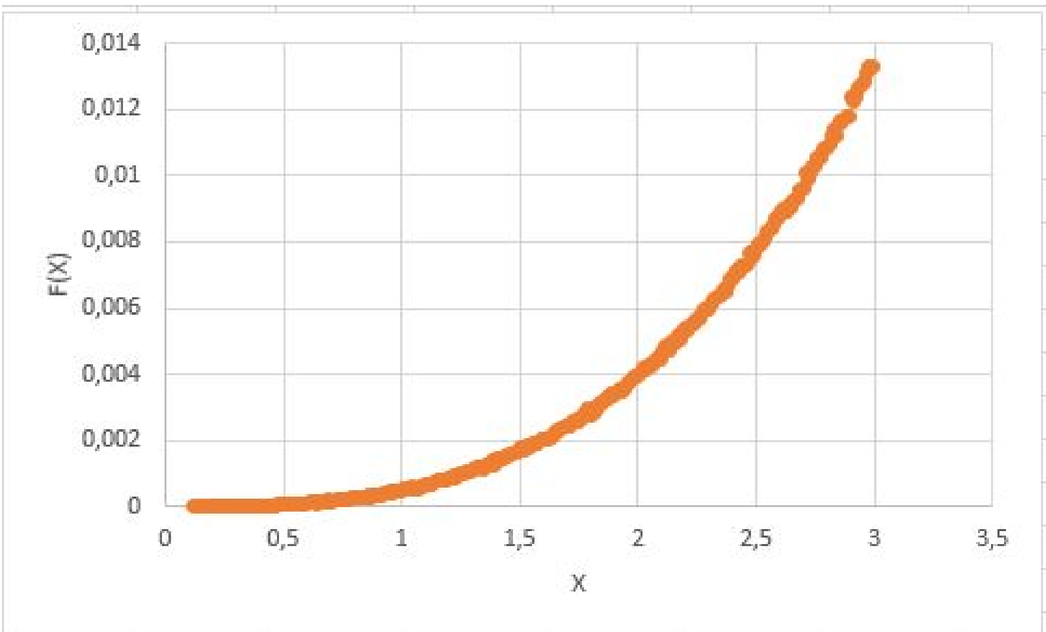

Рисунок 2 – Результаты работы метода отбраковки

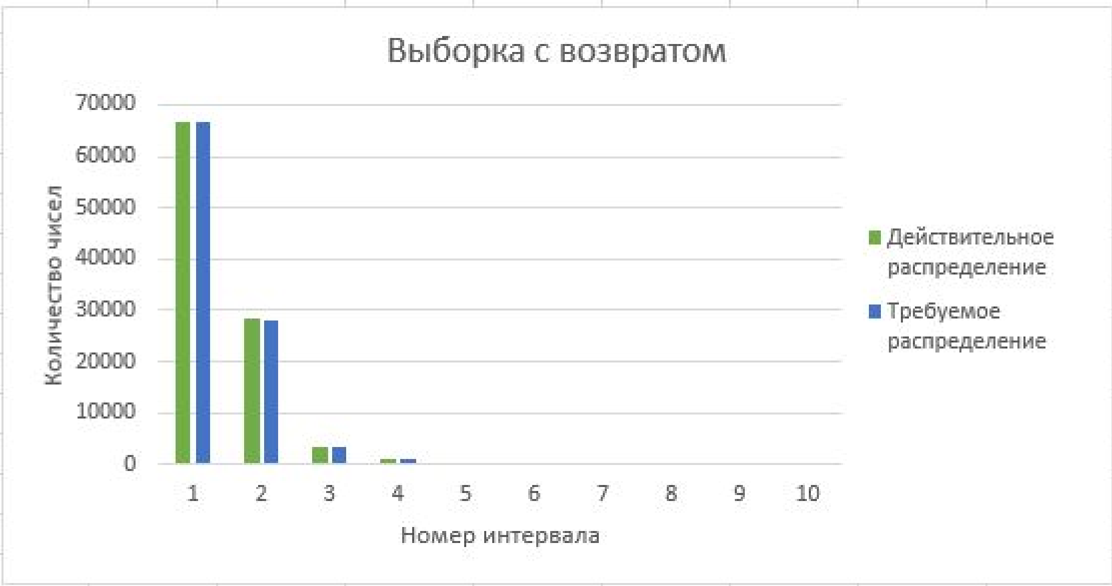

Рисунок 3 - Моделирование дискретной случайной величины с возвратом

Рисунок 4 - Моделирование дискретной случайной величины без возврата

Вывод

В ходе работы были изучены: непрерывное распределение методом отбраковки, дискретное распределение с возвратом и дискретное распределение без возврата. В качестве генератора случайных чисел был выбран стандартный генератор языка Java (java.random.Random), на котором были написаны исследуемые алгоритмы.

При исследовании работы метода отбраковки можно сделать следующие выводы: текущую функцию плотности метод смог смоделировать достаточно точно, но при этом есть ряд недостатков:

1. Точки, которые не попадали на кривую распределения плотности вероятности, отбрасываются, на генерацию которых затрачивается время. Это сказывается на общее время работы алгоритма;
2. Метод применим только для аналитических функций. Алгоритма слабо эффективен для распределений с длинными «хвостами», поскольку в этом случае повышается частота повторных испытаний.

По результатам моделирования дискретных случайных величин с реализацией выборки с возвратом и без возврата можно следующие выводы: исследуемый генератор случайных чисел выдает распределение близкое к равномерному, так как действительное распределение приближенно к требуемому.

Листинг программы

```
class DistributionDensity(val a: Double, val b: Double) {

    private val random: Random by lazy { Random() }

    private val xiJavaRandomDouble: Double
        get() = random.nextDouble()

    init {
       println("""
                  | 0, x ≤ 0
            f(x) = | (a * x^3) / 3, 0 < x ≤ 3
                  | 0, x > 3
            К о э ф ф и ц и е н т: ${calcCof()}

        """.trimIndent())
        calcRejection()
        repeat(n = 100000, m = 10)
        notRepeat(n = 100000, m = 10)
    }

    private fun f(x: Double) = when {
        (x > 0.0) && (x <= 3.0) -> x.pow(3) * 0,444
        else -> 0.0
    }

    private fun calcRejection() {
        var otb = mutableMapOf<Double, Double>()
        val n = 1_000_000
        repeat(n) {
            val x = rejectionMethod()
            val index = floor(x * 100) / 100
            otb[index] = otb[index]?.plus(1) ?: 0.0
        }
        val fileX = File("x.txt")
        val fileY = File("y.txt")
        otb = otb.toSortedMap()
        val textX = otb.map { (x, _) -> x.double() }.reduce { i, j -> "$i\n$j" }
        fileX.writeText(textX)
        val textY = otb.map { (_, y) -> (y/n).minDouble() }.reduce { i, j -> "$i\n$j"
}
        fileY.writeText(textY)
    }

    private fun rejectionMethod(): Double {
        val c = 1.0
        var x: Double
        var y: Double
        var leftHandSide: Double
        var rightHandSide: Double
```

```
    do {
        x = xiJavaRandomDouble * 10
        y = xiJavaRandomDouble * 10
        leftHandSide = f(a + (b - a) * x)
        rightHandSide = c * y
    } while (leftHandSide <= rightHandSide)
    return a + (b - a) * x
}

private fun calcCof(): Double {
    val n = 10000000
    val h = (b - a) / n
    var result = 0.0
    for (i in 0..n) {
        val x = a + h * (i + 0.5)
        result += f(x)
    }
    result *= h
    return 1.0 / result
}

private fun repeat(n: Int, m: Int) {
    var list = (1..m).map { 0.0 }.toMutableList()
    val hits = (1..m).map { 0.0 }.toMutableList()
    var residueChance = 1.0
    list = list.map {
        val randomValue = xiJavaRandomDouble
        val probability = abs(randomValue.rem(residueChance))
        residueChance -= probability
        probability
    }.toMutableList()
    list[list.size - 1] = residueChance
    for (i in 0..n) {
        val randomValue = xiJavaRandomDouble
        var sum = 0.0
        for (j in 0 until m) {
            sum += list[j]
            if (randomValue < sum) {
                hits[j]++
                break
            }
        }
    }
    hits.forEach { println(it.double()) }
    list.forEach { println((it * n).double()) }
}

private fun notRepeat(n: Int, m: Int) {
    var nums: MutableSet<Int>
    val hits = (1..m + 1).map { 0 }.toMutableList()
    var k: Int = 3 * m / 4
    val path = n / k + 1
    for (j in 0 until path) {
```

```
            nums = (1..m).toMutableSet()
            if (j == path - 1) {
                k = n % k
            }
            for (i in 0 until k) {
                val index = kotlin.random.Random.nextInt(0, nums.size)
                val element = nums.elementAt(index)
                hits[element]++
                nums.remove(element)
            }
        }
        val randomNambers = (1..n).map {
            kotlin.random.Random.nextInt(0, m)
        }
        val counts = (1..m).map { 0 }.toMutableList()
        randomNambers.forEach {
            counts[it]++
        }
        for (i in 1..m) {
            println(hits[i])
        }
        println()
        counts.forEach { println(it) }
    }
}
```