Министерство образования Республики Беларусь

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

Факультет прикладной математики и информатики

**Кафедра вычислительной математики**

Стефанович Константин Андреевич

Отчет по лабораторным работам по курсу

“Имитационное и статистическое моделирование”

студента 4 курса 5 группы

|  |  |
| --- | --- |
| Работа сдана 2018г. | **Преподаватель** |
| зачтена \_\_\_\_\_\_\_\_\_\_ 2018 г.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись преподавателя) | *Лобач Виктор Иванович*  доцент кафедры ММАД,  канд. физ.-мат. наук |
|  |  |

*Минск 2018*

**Лабораторная работа 1.**

**Условие:**

Используя метод Маклерена-Марсальи построить датчик БСВ (1 датчик должен быть мультипликативно конгруентный, второй – на выбор). Исследовать точность построенной БСВ.

1. Осуществить моделирование *n* = 1000 реализаций БСВ с помощью мультипликативного конгруэнтного метода (МКМ) с параметрами *a*0, β, *M* = 231 .
2. Осуществить моделирование *n* = 1000 реализаций БСВ с помощью метода Макларена-Марсальи (один датчик должен быть мультипликативно конгруентный (п. 1), второй – на выбор).*K* – объем вспомогательной таблицы.
3. Проверить точность моделирования обоих датчиков (п. 1 и п. 2) с помощью критерия согласия Колмогорова и χ2-критерия Пирсона с уровнем значимости ε = 0.05.

*a*0 = β = 50 653, K = 64

**Теория:**

**Мультипликативный конгруэнтный метод:**

Псевдослучайная последовательность  строится по следующим рекуррентным формулам:

  

где  - параметры датчика:  - множитель (*<M*), *M* – модуль,  - стартовое значение (нечетное число).

В данной работе брались значения: *M*=2147483648, ==65539.

**Метод Маклорена-Марсальи:**

Пусть  - псевдослучайные последовательности, порожденные независимо работающими датчиками;  - результирующая псевдослучайная последовательность реализация БСВ;

*V={V(0), V(1), …,V(K-1)}* – вспомогательная таблица *K* чисел.

Процесс вычисления  включает следующие этапы:

- первоначальное заполнение таблицы

*V*: 

- случайный выбор из таблицы:



-обновление табличных значений:

.

В данной работе в качестве  бралась последовательность (из 100 элементов), полученная мультипликативным конгруэнтным методом, описанным выше. В качестве , бралась последовательности (из 10000) элементов, полученная аналогичным способом с тем же M и . *K*=100.

** - критерий согласия Пирсона:**

Область возможных значений случайной величины разбивается на интервалы .

Рассматривается следующая статистика,

,

*n* – объем выборки,

 - количество элементов выборки, попавших в *k*-ый интервал,

 - вероятность попадания случайной величины в *k*-ый интервал.

Проверяется условие , где , *G* функция распределения распределения**,**  - уровень значимости (обычно =0.05).

В данной работе отрезок [0;1] разбивался на 10 интервалов.

**Критерий согласия Колмогорова:**

Рассматривается статистика:



где

,

Проверяется условие , где , *K* - функция распределенияраспределения Колмогорова**,**  - уровень значимости.

**Код программы:**

#include **<iostream>**#include **<math.h>**#include **<time.h>**#include **<algorithm>**#include **<array>**#include **<fstream>  
using namespace** std;  
  
**const int** K = 64;  
**const int** β = 50653;  
**unsigned long long** a = 50653;  
**const unsigned long long** M = pow(2, 31);  
**const int** n = 1000;  
  
**class** IsInRange {  
 **int** index;  
**public**:  
 IsInRange(**int** p) : index(p) {}  
  
 **bool operator**()(**double** i) {  
 **return** (i >= index / 10.0 && i < (index + 1.0) / 10.0);  
 }  
};  
  
**int** main() {  
 array<**double**, n> d1;  
 d1[0] = **double**(a) / M;  
 **for**(**int** i = 1; i < n; i += 1) {  
 a = (β \* a) % M;  
 d1[i] = **double**(a) / M;  
 }  
  
 srand(time(NULL));  
 array<**double**, K> V;  
 array<**double**, n> d3;  
 a = 50653;  
 V[0] = **double**(a) / M;  
 **for**(**int** i = 1; i < n; i += 1) {  
 a = (β \* a) % M;  
 V[i] = **double**(a) / M;  
 }  
 **for**(**int** i = 0; i < n; i += 1) {  
 **int** s = floor(rand()/**float**(RAND\_MAX) \* K);  
 d3[i] = V[s];  
 a = (β \* a) % M;  
 V[s] = **double**(a) / M;  
 }  
  
 *// write arrays to file* ofstream f (**"../numbers.txt"**);  
 **for**(**int** i = 0; i < n; i += 1) {  
 f << d1[i] << **"\n"**;  
 }  
 **for**(**int** i = 0; i < n; i += 1) {  
 f << d3[i] << **"\n"**;  
 }  
 f.close();  
  
 *// Pearson's chi-squared test  
 // https://ru.wikipedia.org/wiki/Квантили\_распределения\_хи-квадрат  
 // for pearson test normal result should be less than 16.92* **double** pearson1 = 0, pearson3 = 0;  
 **double** middle = **double**(n) / 10;  
 **for**(**int** i = 0; i < 10; i +=1 ){  
 pearson1 += pow((count\_if(d1.begin(), d1.end(), IsInRange(i)) - middle), 2) / middle;  
 pearson3 += pow((count\_if(d3.begin(), d3.end(), IsInRange(i)) - middle), 2) / middle;  
 }  
 cout << **"Pirson test for D1: "** << pearson1 << endl;  
 cout << **"Pirson test for D3: "** << pearson3 << endl;  
  
 *// for ks test normal result should be less than 1.36 ???? 0.04294689290274677* sort(d1.begin(), d1.end());  
 sort(d3.begin(), d3.end());  
 **double** max1 = 0, max3 = 0;  
 **for**(**int** i = 1; i < n; i += 1) {  
 **double** v1 = **double**(i) / n - d1[i - 1];  
 **double** v3 = **double**(i) / n - d3[i - 1];  
 **if**(v1 > max1) max1 = v1;  
 **if**(v3 > max3) max3 = v3;  
  
 v1 = d1[i - 1] - ((**double**(i) - 1) / n);  
 v3 = d3[i - 1] - ((**double**(i) - 1) / n);  
 **if**(v1 > max1) max1 = v1;  
 **if**(v3 > max3) max3 = v3;  
 }  
 cout << **"Kolmogorov-Smirnov test for D1: "** << max1 << endl;  
 cout << **"Kolmogorov-Smirnov test for D3: "** << max3 << endl;  
  
 **return** 0;  
}

**Результаты:**

****

**Лабораторная работа 2.**

**Условие:**

Смоделировать дискретную случайную величину. Исследовать точность моделирования.

1. Осуществить моделирование *n* = 1000 реализаций СВ из заданных дискретных распределений.
2. Вывести на экран несмещенные оценки математического ожидания и дисперсии, сравнить их с истинными значениями.
3. Для каждой из случайных величин построить свой χ2-критерием Пирсона с уровнем значимость ε=0.05. Проверить, что вероятность ошибки I рода стремится к 0.05.
4. Осуществить проверку каждой из сгенерированных выборок каждым из построенных критериев.

Биномиальное – Bi(*m*,*p*), *m* = 5, *p* = 0.6; Пуассона – П(λ), λ = 2;

**Теория:**

**Распределение Пуассона (с параметром ):**

Случайная величина  принимает только целые неотрицательные значения, причем 

В данной работе, сначала моделировалась последовательность БСВ, а потом по каждой БСВ строился соответствующий элемент выборки распределения Пуассона: отрезок [0;1] разбивался на интервалы длин  проверялось, в какой интервал попадает элемент последовательности БСВ.

#### Распределение Бернулли

ДСВ имеет распределение Бернулли *Bi(1,p)*, если:   где *p*  (0,1) – параметр распределения.

Характеристики распределения Бернулли (*x* {0,1}):

* функция распределения:



* функция вероятности:



* среднее значение: 
* дисперсия: 

#### Биномиальное распределение

ДСВ имеет биномиальное распределение *Bi(m,p)*, если:  

Параметры распределения: *m* – натуральное число; *p* (0,1).

Характеристики распределения *Bi(m,p)* (*x*  {0,1,…,m}):

* функция распределения:



* функция вероятности:



* среднее значение: 

дисперсия: 

#### Геометрическое распределение

ДСВ имеет геометрическое распределение *G*(*p*), если: {1,2,…}, *P*{ = *i*} = *p*(1*-p*)*i*-1 , *i* {1,2,…}, где *p* (0,1) – параметр распределения.

Характеристики распределения *G*(*p*) (*x* {1,2,…}):

* функция распределения:



* функция вероятности:



* среднее значение: 
* дисперсия: 

#### Отрицательное биномиальное распределение

ДСВ имеет отрицательное биномиальное распределение(*m,p*), если: {0,1,…}, *P*{ = *i*} = **Параметры распределения: *m* – натуральное число, *p* (0,1).

Характеристики распределения  (*x* {0,1,…}):

Описываемый тест используется для проверки гипотезы Н0 о равномерности двухмерного распределения векторов  и представляет собой следующее решающее правило:

принимается  (11)

где в случае истинной гипотезы H0 и  статистика

 (12)

имеет *x*2 – распределение с *k* – 1 степенями свободы, а порог Δ определяется как квантиль этого распределения: , где ε – заданный уровень значимости.

В пакете СТАТМОД предполагается использование эквивалентной формы правила (11):

принимается 

где Р - значение вычисляется по формуле: *P=1-F(x*2*)*, здесь *F(****.****)*- функция распределения статистики (12).

**Код программы:**

#include **<iostream>**#include **<fstream>**#include **<iomanip>**#include **<math.h>**#include **<algorithm>**#include **<array>**#include **<random>**#include **<ctime>  
  
using namespace** std;  
  
**const int** n = 1000;  
  
**template** <**class** T = **unsigned long**>  
T C(**unsigned long** k, **unsigned long** n) {  
 **unsigned long** i;  
 T b;  
 **if** (0 == k || n == k) {  
 **return** 1;  
 }  
 **if** (k > n) {  
 **return** 0;  
 }  
 **if** (k > (n - k)) {  
 k = n - k;  
 }  
 **if** (1 == k) {  
 **return** n;  
 }  
 b = 1;  
 **for** (i = 1; i <= k; ++i) {  
 b \*= (n - (k - i));  
 **if** (b < 0) **return** -1;  
 b /= i;  
 }  
 **return** b;  
}  
  
**long long** factorial(**long long** n) {  
 **return** (n == 1 || n == 0) ? 1 : factorial(n - 1) \* n;  
}  
  
**template** <size\_t N>  
**double** expectedValue(array<**double**, N> a) {  
 **return** accumulate(a.begin(), a.end(), 0.0, plus<>()) / N ;  
}  
  
**template** <size\_t N>  
**double** variance(array<**double**, N> a) {  
 **double** middle = expectedValue(a);  
 **return** accumulate(a.begin(), a.end(), 0.0, [&middle](**double** accumulator, **double** e) {  
 **return** accumulator + (e - middle) \* (e - middle);  
 }) / N;  
}  
  
**template** <size\_t N>  
**double** pearson\_bernoulli(array<**double**, N> a, **double** p) {  
 **return** (  
 pow(**double**(count\_if(a.begin(), a.end(), [](**int** i) {**return** i == 1;}))/ N - p, 2) / p +  
 pow(**double**(count\_if(a.begin(), a.end(), [](**int** i) {**return** i == 0;})) / N - (1 - p), 2) / (1 - p)  
 ) \* N;  
}  
  
**template** <size\_t N>  
**double** pearson\_binomial(array<**double**, N> a, **double** p, **int** m) {  
 **double** s = 0;  
 **for**(**int** i = 0; i < m; i += 1) {  
 **double** v = C(i, m) \* pow(p, i) \* pow(1 - p, m - i);  
 s += pow(**double**(count\_if(a.begin(), a.end(), [&i](**int** j) {**return** j == i;}))/ N - v, 2) / v;  
 }  
 **return** s \* N;  
}  
  
**template** <size\_t N>  
**double** pearson\_geometric(array<**double**, N> a, **double** p) {  
 **double** s = 0;  
 **for**(**int** i = 1; i < 20; i += 1) {  
 **double** v = p \* pow(1 - p, i - 1);  
 s += pow(**double**(count\_if(a.begin(), a.end(), [&i](**int** j) {**return** j == i;}))/ N - v, 2) / v;  
 }  
 **return** s \* N;  
}  
  
**template** <size\_t N>  
**double** pearson\_negative\_binomial(array<**double**, N> a, **double** p, **int** r) {  
 **double** s = 0;  
 **for**(**int** i = 0; i < 100; i += 1) {  
 **double** v = C(i, i + r - 1) \* pow(p, r) \* pow(1 - p, i);  
 s += pow(**double**(count\_if(a.begin(), a.end(), [&i](**int** j) {**return** j == i;}))/ N - v, 2) / v;  
 }  
 **return** s \* N;  
}  
  
**template** <size\_t N>  
**double** pearson\_poisson(array<**double**, N> a, **int** λ) {  
 **double** s = 0;  
 **for**(**int** i = 0; i < 15; i += 1) {  
 **double** v = exp(-λ) \* pow(λ, i) / factorial(i);  
 s += pow(**double**(count\_if(a.begin(), a.end(), [&i](**int** j) {**return** j == i;}))/ N - v, 2) / v;  
 }  
 **return** s \* N;  
}  
  
**int** main() {  
 **double** p;  
 **int** m, r, λ;  
 mt19937 gen;  
 uniform\_real\_distribution<> urd(0, 1);  
 gen.seed(time(0));  
  
 *// Bernoulli* p = 0.7;  
 array<**double**, n> be;  
 **for**(**int** i = 0; i < n; i += 1) {  
 **double** e = urd(gen);  
 be[i] = p >= e ? 1 : 0;  
 }  
 cout << **"Bernoulli"** << endl;  
 cout << **"Expected value: "** << expectedValue(be) << endl;  
 cout << **"True expected value: "**<< p << endl;  
 cout << **"Variance: "** << variance(be) << endl;  
 cout << **"True variance: "**<< p \* (1 - p) << endl;  
 cout << **"Pearson: "** << pearson\_bernoulli(be, p) << endl;  
 cout << **"Pearson critical: "**<< 3.841458820694124 << endl;  
  
 *// Binomial* p = 0.6;  
 m = 5;  
 array<**double**, n> bi;  
 **for**(**int** i = 0; i < n; i += 1) {  
 **int** x = 0;  
 **for**(**int** j = 0; j < m; j += 1) {  
 **double** e = p - urd(gen);  
 **if**(e > 0) x += 1;  
 }  
 bi[i] = x;  
 }  
 cout << **"\nBinomial"** << endl;  
 cout << **"Expected value: "** << expectedValue(bi) << endl;  
 cout << **"True expected value: "**<< m \* p << endl;  
 cout << **"Variance: "** << variance(bi) << endl;  
 cout << **"True variance: "**<< m \* p \* (1 - p) << endl;  
 cout << **"Pearson: "** << pearson\_binomial(bi, p, m) << endl;  
 cout << **"Pearson critical: "**<< 11.070497693516351 << endl;  
  
 *// Geometric* p = 0.6;  
 array<**double**, n> ge;  
 **for**(**int** i = 0; i < n; i += 1) {  
 ge[i] = ceil(log2(urd(gen)) \* log2(p));  
 }  
 cout << **"\nGeometric"** << endl;  
 cout << **"Expected value: "** << expectedValue(ge) << endl;  
 cout << **"True expected value: "**<< 1 / p << endl;  
 cout << **"Variance: "** << variance(ge) << endl;  
 cout << **"True variance: "**<< (1 - p) / (p \* p) << endl;  
 cout << **"Pearson: "** << pearson\_geometric(ge, p) << endl;  
 cout << **"Pearson critical: "**<< 30.14352720564616 << endl;  
  
 *// Negative binomial* p = 0.25;  
 r = 5;  
 array<**double**, n> nbi;  
 **for**(**int** i = 0; i < n; i += 1) {  
 **int** x = 0, y = 0;  
 **while**(x < r) {  
 **double** e = p - urd(gen);  
 e >= 0 ? x += 1 : y += 1;  
 }  
 nbi[i] = y;  
 }  
 cout << **"\nNegative binomial"** << endl;  
 cout << **"Expected value: "** << expectedValue(nbi) << endl;  
 cout << **"True expected value: "**<< r \* (1 - p) / p << endl;  
 cout << **"Variance: "** << variance(nbi) << endl;  
 cout << **"True variance: "**<< r \* (1 - p) / p / p << endl;  
 cout << **"Pearson: "** << pearson\_negative\_binomial(nbi, p, r) << endl;  
 cout << **"Pearson critical: "**<< 123.2252214533618 << endl;  
  
 *// Poisson* λ = 2;  
 array<**double**, n> pu;  
 **for**(**int** i = 0; i < n; i += 1) {  
 **int** x = -1;  
 **double** e = exp(-λ), y = 1.0;  
 **while**(y > e) {  
 y \*= urd(gen);  
 x += 1;  
 }  
 pu[i] = x;  
 }  
 cout << **"\nPoisson"** << endl;  
 cout << **"Expected value: "** << expectedValue(pu) << endl;  
 cout << **"True expected value: "**<< λ << endl;  
 cout << **"Variance: "** << variance(pu) << endl;  
 cout << **"True variance: "**<< λ << endl;  
 cout << **"Pearson: "** << pearson\_poisson(pu, λ) << endl;  
 cout << **"Pearson critical: "**<< 23.684791304840576 << endl;  
 **return** 0;  
}

Результат:



**Лабораторная работа 3.**

**Условие:**

Смоделировать непрерывную случайную величину. Исследовать точность моделирования.

1. Осуществить моделирование *n* = 1000 реализаций СВ из нормального закона распределения *N*(*m*, *s*2) с заданными параметрами. Вычислить несмещенные оценки математического ожидания и дисперсии, сравнить их с истинными.
2. Смоделировать *n* = 1000 СВ из заданных абсолютно непрерывных распределений. Вычислить несмещенные оценки математического ожидания и дисперсии, сравнить их с истинными значениями (если это возможно).
3. Для каждой из случайных величин построить свой критерий Колмогорова с уровнем значимость ε=0.05. Проверить, что вероятность ошибки I рода стремится к 0.05.
4. Для каждой из случайных величин построить свой χ2-критерий Пирсона с уровнем значимость ε=0.05. Проверить, что вероятность ошибки I рода стремится к 0.05.
5. Осуществить проверку каждой из сгенерированных выборок каждым из построенных критериев.

*m* = 5, *s*2 = 9; Коши *C*(*a*,*b*), *a* = -1, *b* = 3; Лапласа *L*(*a*), *a* = 2.

**Теория:**

#### Равномерное распределение

НСВ  имеет равномерно е распределение на интервале [a, b), обозначаемое R(a, b), если функция и плотность распределения  определяются соотношениями:

 (21)

Для произвольных значений параметров распределения a, b распределение R(a, b) обобщает распределение R(0, 1) БСВ α.

Среднее значение и дисперсия  равны:

.

#### Одномерное нормальное распределение

НСВ  с плотностью распределения



имеет одномерное нормальное (гауссово) распределение с параметрами: средним значением  и дисперсией  (обозначается ).

Распределение  называется стандартным нормальным распределением, а НСВ ~-стандартной нормальной (гаусовской ) величиной. Функция распределения  обозначается Ф(x) и имеет вид:

 (22)

и называется функцией Лапласа.

Случайные величины и  связаны соотношением:

,(23)

где -среднее квадратическое (стандартное) отклонение. Таким образом, задача моделирования ~ сводится к моделированию стандартной гаусовской СВ  и применению формулы (23). Опишем некоторые алгоритмы моделирования ~.

Лог - нормальное распределение

НСВ с плотностью распределения

.

имеет логарифмически-нормальное распределение (лог-нормальное распределение) с параметрами формы (стандартное отклонение СВ ).

Среднее значение и дисперсия ~определяются формулами:



Очевидно, СВ имеет распределение , если СВ распределена по нормальному закону , причем . И наоборот: если ~, то СВ . Эта связь распределений лежит в основе моделирующего алгоритма ().

#### Экспоненциальное распределение

НСВ  с функцией и плотностью распределения, определяемые соотношениями:

, (25)

имеет экспоненциальное(потенциальное) распределение , где -параметр распределения ().

Среднее значение и дисперсия СВ равны: .

Связь с другими распределениями

Экспоненциальное распределение можно рассматривать как частный случай распределений:

* гамма - распределения при ;
* Вейбулла-Гнеденко  при с=1.

**Код программы:**

#include **<iostream>**#include **<fstream>**#include **<iomanip>**#include **<math.h>**#include **<algorithm>**#include **<array>**#include **<random>**#include **<ctime>  
  
using namespace** std;  
  
**const int** n = 1000;  
  
**template** <**class** T = **unsigned long**>  
T C(**unsigned long** n, **unsigned long** k) {  
 **unsigned long** i;  
 T b;  
 **if** (0 == k || n == k) {  
 **return** 1;  
 }  
 **if** (k > n) {  
 **return** 0;  
 }  
 **if** (k > (n - k)) {  
 k = n - k;  
 }  
 **if** (1 == k) {  
 **return** n;  
 }  
 b = 1;  
 **for** (i = 1; i <= k; ++i) {  
 b \*= (n - (k - i));  
 **if** (b < 0) **return** -1;  
 b /= i;  
 }  
 **return** b;  
}  
  
**long long** factorial(**long long** n) {  
 **return** (n == 1 || n == 0) ? 1 : factorial(n - 1) \* n;  
}  
  
**int** sgn(**int** x) {  
 **return** x == 0 ? 0 : x > 0 ? 1 : -1;  
}  
  
**template** <size\_t ***N***>  
**double** expectedValue(array<**double**, ***N***> a) {  
 **return** accumulate(a.begin(), a.end(), 0.0, plus<>()) / N ;  
}  
  
**template** <size\_t N>  
**double** variance(array<**double**, N> a) {  
 **double** middle = expectedValue(a);  
 **return** accumulate(a.begin(), a.end(), 0.0, [&middle](**double** accumulator, **double** e) {  
 **return** accumulator + (e - middle) \* (e - middle);  
 }) / N;  
}  
  
**template** <size\_t N>  
**double** pearson\_normal(array<**double**, N> a, **double** m, **int** s\_2) {  
 **double** s = 0;  
 **for**(**int** i = -13; i <= 23; i += 1) {  
 **int** j = i + 1;  
 **double** v = (  
 erf((j - m) / (sqrt(2 \* s\_2))) -  
 erf((i - m) / (sqrt(2 \* s\_2)))  
 ) / 2;  
 **double** n = **double**(count\_if(a.begin(), a.end(), [&i](**double** j) {**return** i < j && j <= (i + 1);})) / N;  
 s += pow(n - v, 2) / v;  
 }  
 **return** s \* N;  
}  
  
**template** <size\_t N>  
**double** ks\_normal(array<**double**, N> a, **double** m, **int** s\_2) {  
 **double** max = 0;  
 **for**(**int** i = -13; i <= 23; i += 1) {  
 **int** j = i + 1;  
 **double** v = (  
 erf((j - m) / (sqrt(2 \* s\_2))) -  
 erf((i - m) / (sqrt(2 \* s\_2)))  
 ) / 2;  
 **double** n = **double**(count\_if(a.begin(), a.end(), [&i](**double** j) {**return** i < j && j <= (i + 1);})) / N;  
 **double** b = abs(n - v);  
 **if**(max < b) max = b;  
 }  
 **return** max;  
}  
  
**template** <size\_t N>  
**double** pearson\_uniform(array<**double**, N> a) {  
 **double** s = 0;  
 **for**(**int** i = 0; i < 10; i += 1) {  
 **double** v = 1 / 10.0;  
 **double** n = **double**(count\_if(a.begin(), a.end(), [&i](**double** j) { **return** i < j && j <= (i + 1); })) / N;  
 s += pow(n - v, 2) / v;  
 }  
 **return** s \* N;  
}  
  
**template** <size\_t N>  
**double** ks\_uniform(array<**double**, N> a) {  
 **double** max = 0;  
 **for**(**int** i = 0; i < 10; i += 1) {  
 **double** v = 1 / 10.0;  
 **double** n = **double**(count\_if(a.begin(), a.end(), [&i](**double** j) { **return** i < j && j <= (i + 1); })) / N;  
 **double** b = abs(n - v);  
 **if**(max < b) max = b;  
 }  
 **return** max;  
}  
  
**template** <size\_t N>  
**double** pearson\_exponential(array<**double**, N> a, **double** λ) {  
 **double** s = 0;  
 **double** sr = 1 - exp(-37 \* λ);  
 **for**(**int** i = 0; i < 37; i += 1) {  
 **int** j = i + 1;  
 **double** v = (  
 (1 - exp(-j \* λ)) -  
 (1 - exp(-i \* λ))  
 ) / (sr);  
 **double** n = **double**(count\_if(a.begin(), a.end(), [&i](**double** j) { **return** i < j && j <= (i + 1); })) / N;  
 s += pow(n - v, 2) / v;  
 }  
 **return** s \* N;  
}  
  
**template** <size\_t N>  
**double** ks\_exponential(array<**double**, N> a, **double** λ) {  
 **double** max = 0;  
 **double** sr = 1 - exp(-37 \* λ);  
 **for**(**int** i = 0; i < 37; i += 1) {  
 **int** j = i + 1;  
 **double** v = (  
 (1 - exp(-j \* λ)) -  
 (1 - exp(-i \* λ))  
 ) / (sr);  
 **double** n = **double**(count\_if(a.begin(), a.end(), [&i](**double** j) { **return** i < j && j <= (i + 1); })) / N;  
 **double** b = abs(n - v);  
 **if**(max < b) max = b;  
 }  
 **return** max;  
}  
  
**template** <size\_t N>  
**double** pearson\_laplace(array<**double**, N> a, **double** λ) {  
 **double** s = 0;  
 **double** sr = 1 - exp(-9 \* λ);  
 **for**(**int** i = -9; i < 9; i += 1) {  
 **int** j = i + 1;  
 **double** v = (λ) \* (  
 (exp(-λ \* j) \* (pow(exp(λ \* j) - 1, 2) \* (-sgn(j)) + 2 \* exp(λ \* j) + exp(2 \* λ \* j) - 1)) / (2 \* λ) -  
 (exp(-λ \* i) \* (pow(exp(λ \* i) - 1, 2) \* (-sgn(i)) + 2 \* exp(λ \* i) + exp(2 \* λ \* i) - 1)) / (2 \* λ)  
 ) / (sr \* 2);  
 **double** n = **double**(count\_if(a.begin(), a.end(), [&i](**double** j) { **return** i < j && j <= (i + 1); })) / N;  
 s += pow(n - v, 2) / v;  
 }  
 **return** s \* N;  
}  
  
**template** <size\_t N>  
**double** ks\_laplace(array<**double**, N> a, **double** λ) {  
 **double** max = 0;  
 **double** sr = 1 - exp(-9 \* λ);  
 **for**(**int** i = -9; i < 9; i += 1) {  
 **int** j = i + 1;  
 **double** v = (λ) \* (  
 (exp(-λ \* j) \* (pow(exp(λ \* j) - 1, 2) \* (-sgn(j)) + 2 \* exp(λ \* j) + exp(2 \* λ \* j) - 1)) / (2 \* λ) -  
 (exp(-λ \* i) \* (pow(exp(λ \* i) - 1, 2) \* (-sgn(i)) + 2 \* exp(λ \* i) + exp(2 \* λ \* i) - 1)) / (2 \* λ)  
 ) / (sr \* 2);  
 **double** n = **double**(count\_if(a.begin(), a.end(), [&i](**double** j) { **return** i < j && j <= (i + 1); })) / N;  
 **double** b = abs(n - v);  
 **if**(max < b) max = b;  
 }  
 **return** max;  
}  
  
**template** <size\_t N>  
**double** pearson\_cauchy(array<**double**, N> a, **double** x0, **double** γ) {  
 **double** h = (\*max\_element(a.begin(), a.end()) - \*min\_element(a.begin(), a.end())) / 10;  
 **double** i = \*min\_element(a.begin(), a.end());  
 **double** s = 0;  
 **for**(**int** t = 0; t < 10; t += 1) {  
 **double** j = i + h;  
 **double** v = (  
 atan((j - x0) / γ) -  
 atan((i - x0) / γ)  
 ) / M\_PI;  
 **double** n = **double**(count\_if(a.begin(), a.end(), [&i, &h](**double** j) { **return** i <= j && j < (i + h); })) / N;  
 s += pow(n - v, 2) / v;  
 i += h;  
 }  
 **return** s \* N;  
}  
  
**template** <size\_t N>  
**double** ks\_cauchy(array<**double**, N> a, **double** x0, **double** γ) {  
 **double** h = (\*max\_element(a.begin(), a.end()) - \*min\_element(a.begin(), a.end())) / 10;  
 **double** i = \*min\_element(a.begin(), a.end());  
 **double** max = 0;  
 **for**(**int** t = 0; t < 10; t += 1) {  
 **double** j = i + h;  
 **double** v = (  
 atan((j - x0) / γ) -  
 atan((i - x0) / γ)  
 ) / M\_PI;  
 **double** n = **double**(count\_if(a.begin(), a.end(), [&i, &h](**double** j) { **return** i <= j && j < (i + h); })) / N;  
 **double** b = abs(n - v);  
 **if**(max < b) max = b;  
 i += h;  
 }  
 **return** max;  
}  
  
**template** <size\_t N>  
**double** pearson\_logistic(array<**double**, N> a, **double** μ, **double** k) {  
 **double** s = 0;  
 **for**(**int** i = -53; i < 57; i += 1) {  
 **int** j = i + 1;  
 **double** v = (  
 pow(1 + exp(-(j - μ) / k), -1) -  
 pow(1 + exp(-(i - μ) / k), -1)  
 );  
 **double** n = **double**(count\_if(a.begin(), a.end(), [&i](**double** j) { **return** i < j && j <= (i + 1); })) / N;  
 s += pow(n - v, 2) / v;  
 }  
 **return** s \* N;  
}  
  
**template** <size\_t N>  
**double** ks\_logistic(array<**double**, N> a, **double** μ, **double** k) {  
 **double** max = 0;  
 **for**(**int** i = -53; i < 57; i += 1) {  
 **int** j = i + 1;  
 **double** v = (  
 pow(1 + exp(-(j - μ) / k), -1) -  
 pow(1 + exp(-(i - μ) / k), -1)  
 );  
 **double** n = **double**(count\_if(a.begin(), a.end(), [&i](**double** j) { **return** i < j && j <= (i + 1); })) / N;  
 **double** b = abs(n - v);  
 **if**(max < b) max = b;  
 }  
 **return** max;  
}  
  
**int** main() {  
 **double** λ;  
 mt19937 gen;  
 uniform\_real\_distribution<> urd(0, 1);  
 gen.seed(time(0));  
  
 *// Normal* **int** m = 5, s\_2 = 9;  
 array<**double**, n> no;  
 **for**(**int** i = 0; i < n; i += 1) {  
 **double** x = 0;  
 **for**(**int** j = 0; j < 12; j += 1){  
 x += urd(gen);  
 }  
 no[i] = m + sqrt(s\_2) \* (x - 6);  
 }  
 cout << **"Normal"** << endl;  
 cout << **"Expected value: "** << expectedValue(no) << endl;  
 cout << **"True expected value: "**<< m << endl;  
 cout << **"Variance: "** << variance(no) << endl;  
 cout << **"True variance: "**<< s\_2 << endl;  
 cout << **"Pearson: "** << pearson\_normal(no, m, s\_2) << endl;  
 cout << **"Pearson critical: "**<< 50.99846016571065 << endl;  
 cout << **"KS test: "** << ks\_normal(no, m, s\_2) << endl;  
 cout << **"KS critical: "**<< 0.04294689290274677 << endl;  
  
 *// Uniform 0 10* **double** d1 = 0.0, d2 = 10.0;  
 array<**double**, n> un;  
 **for**(**int** i = 0; i < n; i += 1) {  
 un[i] = urd(gen) \* (d2 - d1) + d1;  
 }  
 cout << **"\nUniform 0 10"** << endl;  
 cout << **"Expected value: "** << expectedValue(un) << endl;  
 cout << **"True expected value: "**<< (d1 + d2) / 2 << endl;  
 cout << **"Variance: "** << variance(un) << endl;  
 cout << **"True variance: "**<< (d1 - d2) \* (d1 - d2) / 12 << endl;  
 cout << **"Pearson: "** << pearson\_uniform(un) << endl;  
 cout << **"Pearson critical: "**<< 16.918977604620448 << endl;  
 cout << **"KS test: "** << ks\_uniform(un) << endl;  
 cout << **"KS critical: "**<< 0.04294689290274677 << endl;  
  
 *// Exponential* λ = 0.5;  
 array<**double**, n> ex;  
 **for**(**int** i = 0; i < n; i += 1) {  
 ex[i] = -log(urd(gen)) / λ;  
 }  
 cout << **"\nExponential"** << endl;  
 cout << **"Expected value: "** << expectedValue(ex) << endl;  
 cout << **"True expected value: "**<< 1 / λ << endl;  
 cout << **"Variance: "** << variance(ex) << endl;  
 cout << **"True variance: "**<< 1 / λ / λ << endl;  
 cout << **"Pearson: "** << pearson\_exponential(ex, λ) << endl;  
 cout << **"Pearson critical: "**<< 50.99846016571065 << endl;  
 cout << **"KS test: "** << ks\_exponential(ex, λ) << endl;  
 cout << **"KS critical: "**<< 0.04294689290274677 << endl;  
  
 *// Laplace* λ = 2;  
 array<**double**, n> la;  
 **for**(**int** i = 0; i < n; i += 1) {  
 **double** y = urd(gen);  
 la[i] = y < 0.5 ? log(2 \* y) / λ : - log(2 \* (1 - y)) / λ;  
 }  
 cout << **"\nLaplace"** << endl;  
 cout << **"Expected value: "** << expectedValue(la) << endl;  
 cout << **"True expected value: "**<< 0 << endl;  
 cout << **"Variance: "** << variance(la) << endl;  
 cout << **"True variance: "**<< 2 / λ / λ << endl;  
 cout << **"Pearson: "** << pearson\_laplace(la, λ) << endl;  
 cout << **"Pearson critical: "**<< 28.869299430392623 << endl;  
 cout << **"KS test: "** << ks\_laplace(la, λ) << endl;  
 cout << **"KS critical: "**<< 0.04294689290274677 << endl;  
  
 *// Cauchy* **double** x0 = 2.0, γ = 3.0;  
 array<**double**, n> ca;  
 **for**(**int** i = 0; i < n; i += 1) {  
 ca[i] = x0 + γ \* tan(M\_PI \* (urd(gen) - 0.5));  
 }  
 cout << **"\nCauchy"** << endl;  
 cout << **"Expected value: don't exist"** << endl;  
 cout << **"Variance: "** << variance(ca) << endl;  
 cout << **"True variance: infinity "**<< endl;  
 cout << **"Pearson: "** << pearson\_cauchy(ca, x0, γ) << endl;  
 cout << **"Pearson critical: "**<< 16.918977604620448 << endl;  
 cout << **"KS test: "** << ks\_cauchy(ca, x0, γ) << endl;  
 cout << **"KS critical: "**<< 0.04294689290274677 << endl;  
  
 *// Logistic* **double** μ = 2.0, k = 3.0;  
 array<**double**, n> lo;  
 **for**(**int** i = 0; i < n; i += 1) {  
 **double** y = urd(gen);  
 lo[i] = μ + k \* log(y / (1 - y));  
 }  
 cout << **"\nLogistic"** << endl;  
 cout << **"Expected value: "** << expectedValue(lo) << endl;  
 cout << **"True expected value: "**<< μ << endl;  
 cout << **"Variance: "** << variance(lo) << endl;  
 cout << **"True variance: "**<< pow(M\_PI \* k, 2) / 3 << endl;  
 cout << **"Pearson: "** << pearson\_logistic(lo, μ, k) << endl;  
 cout << **"Pearson critical: "**<< 135.48017792835952 << endl;  
 cout << **"KS test: "** << ks\_logistic(lo, μ, k) << endl;  
 cout << **"KS critical: "**<< 0.04294689290274677 << endl;  
 **return** 0;  
}

**Результат:**

****

****

**Лабораторная работа 4.1.**

**Условие:**

Вычислить интегралы методом Монте-Карло:

|  |  |
| --- | --- |
|  |  |

**Теория:**

**Метод Монте-Карло приближенного вычисления интеграла:**

Необходимо вычислить .

Пусть  - произвольная случайная величина с плотностью распределения  имеющая конечный момент второго порядка.

Пусть  Тогда 

В качестве приближенного значения *a* можно взять



В данной работе в качестве  бралась случайная величина, равномерно распределенная на [0;1].

**Код программы:**

**from** math **import** \*  
**import** numpy **as** np  
  
data = [(21, 1), (20001, 1), (21, 10), (20001, 10), (20001, 1000), (2000001, 1000), (2000001, 10000)]  
print(**"Right answer is something about 2.07959"**)  
**for** (n, ranges) **in** data:  
 h = (ranges \* 2) / (n - 1)  
 nodes = [i **for** i **in** np.arange(-ranges, ranges + h, h)]  
 answer = sum([exp(-(x \*\* 4)) \* ((1 + x \*\* 2) \*\* (1 / 2)) **for** x **in** nodes]) \* (ranges \* 2 / n)  
 print(**"Answer: {0}, ranges = (-{1}, {1}), number of nodes = {2}"**.format(answer, ranges, n))

**import** math  
**import** random  
  
circle\_r = 7 \*\* (1 / 2)  
  
print(**"Right answer is something about 7.82449"**)  
**for** n **in** [10 \*\* i **for** i **in** range(3, 9)]:  
 nodes = []  
 **for** counter **in** range(n):  
 x = y = 0  
 **while** (x \*\* 2 + y \*\* 2) < 1:  
 alpha = 2 \* math.pi \* random.random()  
 r = circle\_r \* math.sqrt(random.random())  
 x = r \* math.cos(alpha)  
 y = r \* math.sin(alpha)  
 **if** (x \*\* 2 + y \*\* 2) > 7:  
 print(x \*\* 2 + y \*\* 2)  
 nodes.append((x, y))  
  
 answer = sum([(x \*\* 3 + math.exp(y)) / (x \*\* 2 + 2 \* y \*\* 2) **for** (x, y) **in** nodes]) \* (6 \* math.pi / n)  
 print(**"Answer: {0}, number of nodes = {1}"**.format(answer, n))

Результат:





**Лабораторная работа 4.2.**

**Условие:**

Решить систему линейных алгебраических уравнений методом Монте-Карло:

**Теория:**

**Метод Монте-Карло приближенного решения системы линейных алгебраических уравнений:**

Необходимо решить систему, представленную в виде , где , собственные значения *A* по модулю меньше 1.

Наша цель – вычислить скалярное произведение вектора решения  с некоторым вектором .

Рассмотрим цепь Маркова с параметрами  такими что





 если 

 если 

Положим



Выберем некоторое натуральное *N* и рассмотрим случайную величину



Где 🡪🡪…🡪 - траекторая цепи Маркова.

*Qm* опряделяется как:



Тогда скалярное произведение вектором *h* и *x* приблизительно равно .

Можем найти *x*, скалярно умножая его на векторы *h* у которых в одной позиции стоит 1, а в остьльных – 0.

В данной работе выбиралось 

**Код программы:**

**import** numpy **as** np  
  
x = y = 0  
A = [  
 [0.6, -0.3],  
 [-0.3, 0.2]  
]  
f = [2, 3]  
h = [1, 0]  
pi = [0.5, 0.5]  
  
p = [  
 [0.5, 0.5],  
 [0.5, 0.5]  
]  
  
N = 10  
m = 10000  
  
i = [0] \* N  
Q = [0] \* N  
  
ξ = [0] \* m  
  
**for** j **in** range(m):  
 mca = np.random.uniform(0.0, 1.0, N + 1)  
 i[0] = 0 **if** mca[0] < pi[0] **else** 1  
  
 **for** k **in** range(1, N):  
 i[k] = 0 **if** mca[k] < 0.5 **else** 1  
  
 Q[0] = h[i[0]] / pi[i[0]] **if** pi[i[0]] > 0 **else** 0  
  
 **for** k **in** range(1, N):  
 **if** p[i[k - 1]][i[k]] > 0:  
 Q[k] = (Q[k - 1] \* A[i[k - 1]][i[k]]) / p[i[k - 1]][i[k]]  
 **else**:  
 Q[k] = 0  
  
 **for** k **in** range(N):  
 ξ[j] += Q[k] \* f[i[k]]  
  
print(sum(ξ) / m)  
print(**"Right answer is (3.04, 2.609)"**)

**Результат:**

**Задание 5**

**1. Постановка задачи**

***47. Моделирование работы продовольственного магазина.***

Небольшой продовольственный магазин состоит из 4-х прилавков и одной кассы на выходе из магазина.   
Покупатели приходят в магазин, образуя пуассоновский поток, составляющий λ c.   
Войдя в магазин, каждый покупатель берет корзинку и может обойти один или несколько прилавков, отбирая продукты.  
Вероятность обхода конкретного прилавка указана в следующей таблице:  
  
*|* Прилавок *|* Вероятность покупки *|* Время обхода прилавка *|  
|:--------:|:-------------------:|:---------------------:|  
|* 1 *|* 0.65 *|* 180±60с *|  
|* 2 *|* 0.85 *|* 160±30с *|  
|* 3 *|* 0.70 *|* 100±25с *|  
|* 4 *|* 0.90 *|* 50±15с *|*После того как товар отобран, покупатель становится в конец очереди к кассе.   
Время обслуживания покупателя в кассе прямо пропорционально числу сделанных покупок, на одну покупку уходит время t.

Разработать GPSS-модель функционирования системы в течение восьми часов.

Первоначальный перечень экспериментов:

λ=95 t=3.

**2. Имитационная модель на GPSS/РС (текст программы)**

*input\_flow\_mean VARIABLE 95; lambda  
time\_shop EQU 3.0; t  
  
POISS FUNCTION RN1,C24 (Poisson Process)  
0.0,0.0/0.1,0.104/0.2,0.222/0.3,0.355/0.4,0.509/0.5,0.69/  
0.6,0.915/0.7,1.2/0.75,1.38/0.8,1.6/0.84,1.83/0.88,7.12/  
0.9,2.3/0.92,2.52/0.94,2.81/0.95,2.99/0.96,3.2/0.97,3.5/  
0.98,3.9/0.99,4.6/0.995,5.3/0.998,6.2/0.999,7/0.9997,8  
  
; ---------------------------------------------------------------  
  
 GENERATE V$input\_flow\_mean,FN$POISS ; Input flow: lambda\*PoissonDistr()  
   
 ASSIGN NZap,0  
 TRANSFER .65,METKA1,PURCHASE1  
PURCHASE1 ASSIGN NZap+,time\_shop  
 ADVANCE 180,60  
METKA1 TRANSFER .85,METKA2,PURCHASE2  
PURCHASE2 ASSIGN NZap+,time\_shop  
 ADVANCE 160,30  
METKA2 TRANSFER .70,METKA3,PURCHASE3  
PURCHASE3 ASSIGN NZap+,time\_shop  
 ADVANCE 100,25  
METKA3 TRANSFER .90,METKA4,PURCHASE4  
PURCHASE4 ASSIGN NZap+,time\_shop  
 ADVANCE 50,15  
METKA4 QUEUE TILL  
 SEIZE TILL  
 DEPART TILL  
  
 ADVANCE P$NZap  
 RELEASE TILL  
   
 TERMINATE   
; ---------------------------------------------------------------  
  
 GENERATE 28800 ; 8 hr = 8\*60\*60 = 28800 s  
 TERMINATE 1  
 START 1*

……………………………………………………………………………………

**3. Имитационная модель на Python (текст программы)**

**import** numpy **as** np  
**import** random  
count = 0  
all\_time = 0  
total\_time = 28800  
m = []  
t = 3  
**while True**:  
 s = np.random.poisson(95, 1)[0]  
 all\_time += s  
 **if** total\_time < all\_time:  
 **break** m.append({  
 **"time\_arrival"**: all\_time  
 })  
 count += 1  
  
**for** c, e **in** enumerate(m):  
 time\_go\_to\_till = e[**"time\_arrival"**]  
 n = 0  
 **if** random.random() > 0.35:  
 time\_go\_to\_till += (180 + ((random.random() \* 2) - 1) \* 60)  
 n += 1  
 **if** random.random() > 0.15:  
 time\_go\_to\_till += (160 + ((random.random() \* 2) - 1) \* 30)  
 n += 1  
 **if** random.random() > 0.30:  
 time\_go\_to\_till += (100 + ((random.random() \* 2) - 1) \* 25)  
 n += 1  
 **if** random.random() > 0.10:  
 time\_go\_to\_till += (50 + ((random.random() \* 2) - 1) \* 15)  
 n += 1  
 m[c][**"time\_go\_to\_till"**] = time\_go\_to\_till  
 m[c][**"n"**] = n  
m = [x **for** x **in** m **if** x[**"time\_go\_to\_till"**] < total\_time]  
m = sorted(m, key=**lambda** e: e[**"time\_go\_to\_till"**])  
  
current\_time = m[0][**"time\_go\_to\_till"**]  
**for** c, e **in** enumerate(m):  
 **if** m[c][**"time\_go\_to\_till"**] > current\_time:  
 current\_time = m[c][**"time\_go\_to\_till"**]  
 m[c][**"time\_start\_till"**] = current\_time  
 m[c][**"time\_end\_till"**] = current\_time + m[c][**"n"**] \* t  
 current\_time += m[c][**"n"**] \* t  
m = [x **for** x **in** m **if** x[**"time\_go\_to\_till"**] < total\_time]  
  
max\_line = 0  
**for** c, e **in** enumerate(m):  
 local\_max\_line = 0  
 local\_e = m[c]  
 **for** i **in** range(c + 1, len(m)):  
 **if** m[c][**"time\_end\_till"**] > m[i][**"time\_go\_to\_till"**]:  
 local\_max\_line += 1  
 **else**:  
 **break  
 if** local\_max\_line > max\_line:  
 max\_line = local\_max\_line  
  
print(**"Общее количество всех обслуженных покупателей: {0}"**.format(len(m)))  
print(**"Среднее время в кассе: {0}"**.format(sum([x[**"time\_end\_till"**] - x[**"time\_start\_till"**] **for** x **in** m]) / len(m)))  
print(**"Среднее время в очереди: {0}"**.format(sum([x[**"time\_start\_till"**] - x[**"time\_go\_to\_till"**] **for** x **in** m]) / len(m)))  
print(**"Максимальное количество людей в очереди: {0}"**.format(max\_line))

**Задание 6**

**1. Постановка задачи**

***18. Моделирование вычислительной системы с удаленными терминалами.***

Моделирование вычислительной системы с удаленными терминалами   
Вычислительная система представляет собой двухпроцессорный комплекс, который   
обслуживает местных пользователей и три однотипных удаленных терминала.   
На каждом терминале задача формируется в среднем через t сек, а время выполнения   
задачи в процессорном блоке имеет математическое ожидание µ сек (экспоненциальное   
распределение). После выполнения задача возвращается на соответствующий терминал,   
инициируя тем самым формирование новой задачи. Время передачи данных по каналу связи  
распределяется равномерно в пределах от a до b сек.

Разработать GPSS-модель функционирования системы в течение одного часа.

Первоначальный перечень экспериментов:

*t = 25, μ = 20, a = 10, b = 30*

**2. Имитационная модель на GPSS/РС (текст программы)**

*proc STORAGE 2   
  
 GENERATE 25  
 TRANSFER ,datt  
  
 GENERATE 25  
 TRANSFER ,datt  
  
 GENERATE 25  
 TRANSFER ,datt  
  
datt ADVANCE 20,10   
 TRANSFER ,work  
  
work QUEUE waittime   
 ENTER proc  
 SEIZE waittime  
 DEPART waittime  
 ADVANCE (EXPONENTIAL(1,0,20))  
 RELEASE waittime  
 LEAVE proc  
 TERMINATE  
  
 GENERATE 3600  
 TERMINATE 1  
 START 1*

……………………………………………………………………………………

**Литература**

1. Харин Ю.С., Малюгин В.И., Кирлица В.П., Лобач В.И., Хацкевич Г.А. Основы имитационного и статистического моделирования. Учебное пособие. Минск: ДизайнПРО, 1997 – 228 с.
2. Лобач В.И., Кирлица В.П., Малюгин В.И., Сталевская С.Н. Имитационное и статистическое моделирование. Практикум для студентов математических и экономических специальностей. Минск, БГУ, 2004 –189 с.