МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет прикладной математики и информатики

Кафедра математического моделирования и анализа данных

Кражевский Алексей Игоревич

Отчет по лабораторным работам по курсу “Математическое моделирование ”

студента 2 курса 15 группы

Работа сдана 2022г.

зачтена __________ 2022 г.

____
(подпись преподавателя)

Преподаватель
Лобач Сергей Викторович
Ассистент кафедры
математического моделирования и
анализ данных ФПМИ

Минск 2022

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, $M = 2^{31}$.
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 χ^2-критерия Пирсона с уровнем значимости ε = 0.05.

Вариант задания:

8) a0 = β = 262 147, K = 256

Теория:

Мультипликативный конгруэнтный метод:

Псевдослучайная последовательность $\alpha_1,\ \alpha_2,\ ...,\alpha_n$ строится по следующим рекуррентным формулам:

$$\alpha_t = \alpha_t^* / M,\ \ \alpha_t^* = \{\beta\alpha_{t-1}^*\} \bmod M \quad (t = 1, 2, ...),$$

где β, M, α_0^* - параметры датчика: β - множитель (β <M), M – модуль, $\alpha_0^* \in \{1, ..., M-1\}$ - стартовое значение (нечетное число).
В данной работе брались значения: M=2147483648, $\alpha_0^* = \beta$ =262 147.

Метод Маклорена-Марсальи:

Пусть $\{\beta_t\}$, $\{c_t\}$ - псевдослучайные последовательности, порожденные независимо работающими датчиками; $\{\alpha_t\}$ - результирующая псевдослучайная последовательность реализация БСВ;
V={V(0), V(1), …,V(K-1)} – вспомогательная таблица K чисел.

Процесс вычисления $\{\alpha_t\}$ включает следующие этапы:

- первоначальное заполнение таблицы

$$V\colon\ V(i) = \beta_i,\ \text{i}=\overline{\text{0,K-1}};$$

- случайный выбор из таблицы:

$$\alpha_t = V(s),\ \text{s}=[\text{c}_t \cdot \text{K}];$$

-обновление табличных значений:

$$V(s) = b_{t+K},\ \text{t}=0,\ 1,\ 2,\ \ldots.$$

В данной работе в качестве $\{\beta_t\}$ бралась последовательность (из 1000 элементов), полученная мультипликативным конгруэнтным методом, описанным выше. В качестве $\{c_t\}$, бралась последовательности (из 1000) элементов, полученная аналогичным способом с тем же M и $\beta' = 3\beta + 1$. K=256.

χ^2 - критерий согласия Пирсона:

Область возможных значений случайной величины разбивается на интервалы $[x_{k-1}, x_k)$, $k = \overline{1, K}$.

Рассматривается следующая статистика,

$$\chi^2 = \sum_{k=1}^{K} \frac{(\nu_k - n \cdot p_k)}{n \cdot p_k},$$

n – объем выборки,

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

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

Проверяется условие $\chi^2 < \Delta$, где $\Delta = G^{-1}(1-\varepsilon)$, G функция распределения распределения χ^2, ε - уровень значимости (ε=0.05).

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

Критерий согласия Колмогорова:

Рассматривается статистика:

$$D_n = \sup_{x \in \mathbb{R}} | F_\xi(x) - F_0(x) | \in [0;1],$$

где

$$F_\xi(x) = \frac{1}{n} \sum_{i=1}^{n} I_{[-\infty;x]}(x_i), \ x \in \mathbb{R},$$

Проверяется условие $\sqrt{n} K_n < \Delta$, где $\Delta = K^{-1}(1-\varepsilon)$, K - функция распределения распределения Колмогорова, ε - уровень значимости.

Код программы:

```
import math
from random import random
import numpy as np


#константы
k = 256
a0 = 262147
b = a0
M = math.pow(2, 31)


now = a0
```

```python
def mult_cong_met():
    global now
    now = (now * b) % M
    return now / M


v = []
def first():
    for _ in range(k):
        v.append(mult_cong_met())


def maclaren():
    indx = int(random() % k)
    num = v[indx]
    v[indx] = mult_cong_met()
    return num


'''
v - объем выборки
n - количество разбиений (0,1)
c - критическое значение лямбда = 1.36 для 0,05
func - закон распределения СВ

sqrt(v) * Dn <= LAMBDAa, где
Dn = max|Fn(x) - F(x)|
'''
def kolmogorov(v, n, c, func):
    print ('Working', end = '')
    cof = 0
    for i in range (n):
        count = 0
        for _ in range (v):
            if func() < float(i / n):
                count += 1
        #print (i, ': ', count)
        print('.', end = '')

        f = float(count / v)
        if cof < abs(f - i / n):
            cof = abs(f - i / n)
        '''
        if cof < max(i / n - f, f - (i - 1) / n):
            cof = max(i / n - f, f - (i - 1) / n)
        '''
    #print (math.sqrt(v) * cof)
    if math.sqrt(v) * cof < c:
        return 'pass'
    else:
        return 'not pass'
```

```python
'''
ans = (Oi - Ei)^2 / Ei
'''
def pirson(v, n, c, func):
    arr = np.zeros(n+1, dtype = int)

    #подсчет числа степеней свободы
    for _ in range (v):
        a = round(func() * n)
        arr[a] += 1

    #подсчет ответа по формуле
    ans = 0
    for i in range(n):
        ans += math.pow(arr[i] - v/n, 2) / (v/n)
    #print (ans)
    if ans < c:
        return 'pass'
    else:
        return 'not pass'


if __name__ == '__main__':
    fout1 = open ('out1.txt', 'w')
    fout2 = open ('out2.txt', 'w')
    fout3 = open ('test_res.txt', 'w')
    first()

    for _ in range (1000):
        fout1.write(str(mult_cong_met()))
        fout1.write('\n')

    for _ in range (1000):
        fout2.write(str(maclaren()))
        fout2.write('\n')

    fout1.close()
    fout2.close()

    fout3.write('Test Results:\n')
    fout3.write('\tKolmogorov for maclaren:\n')
    fout3.write(kolmogorov (20000, 250, 1.36, maclaren))
    fout3.write('\n\tPirson for maclaren:\n')
    fout3.write(pirson (1000, 25, 37.6525, maclaren))

    fout3.close()
```

Результаты:

В файле out1.txt находятся 1000 реализаций БСВ мультипликативно-конгуэртного метода, а в файле out2.txt – 1000 реализаций БСВ методом макларена-марсальи.

Результат критерия Колмогорова и Пирсона:

```
Test Results:
    Kolmogorov for maclaren:
not pass
    Pirson for maclaren:
pass
```

Лабораторная работа 2.

Условие:

Смоделировать дискретную случайную величину (задания на стр. 18-22). 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 χ^2-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.

Вариант задания:

8) Отрицательное биномиальное – (r,p), r = 6, p = 0.25; Пуассона – П(λ), λ = 3

Теория:

Распределение Пуассона (с параметром λ):

Случайная величина ξ принимает только целые неотрицательные значения, причем $P(\xi = k) = \frac{\lambda^k}{k!}$, $k \in \mathbb{N}, k \geq 0$.

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 $\frac{\lambda^k}{k!}$ проверялось, в какой интервал попадает элемент последовательности БСВ.

Биномиальное распределение:

ДСВ $\xi \in \{0, 1, 2, \dots, \} \sim Bi(n, p)$, если

$$P\{\xi = i\} = C_n^i p^i (1-p)^{n-1}$$

$$C_n^i = \frac{n!}{i!\,(n-i)!}$$

Отрицательное биномиальное распределение:

Пусть $\{X_i\}_{i=1}^{\infty}$ — последовательность независимых случайных величин с распределением Бернулли, то есть

$$X_i = \begin{cases} 1, & p \\ 0, & q \equiv 1-p \end{cases}, \; i \in \mathbb{N}.$$

Построим случайную величину Y следующим образом. Пусть $k+r$ — номер r-го успеха в этой последовательности. Тогда $Y = k$. Более строго, положим $S_n = \sum_{i=1}^{n} X_i$. Тогда

$$Y = \inf\{n \mid S_n = r\} - r.$$

Распределение случайной величины Y, определённой таким образом, называется отрицательным биномиальным. Пишут: $Y \sim \mathrm{NB}(r, p)$.

Код программы:

main.py

```
from puasson import puasson_gen, puasson_e, puasson_d
from binomial import bernoulli_gen, binomial_gen
from neg_binomial import neg_binomial_e, neg_binomial_gen, neg_binomial_d
from mcg import mcg
import matplotlib.pyplot as plt


if __name__ == '__main__':
    g = puasson_gen(3, mcg(2**31, 16807, 16807))
    seq = [n for n in g]
    #print(seq)
    plt.plot(seq)
    plt.show()

    Ep = puasson_e(3)
    Dp = puasson_d(3)
    print(Ep, Dp)

    '''
    g = binomial_gen(4, 0.25, mcg(2**31, 16807, 16807))
    seq = [n for n in g]
    plt.plot(seq)
    plt.show()
    #print(seq)
    '''
```

```
    g = neg_binomial_gen(6, bernoulli_gen(0.25, mcg(2**31, 16087, 16087,
count=100000)))
    seq = [n for n in g]
    #print(seq)
    plt.plot(seq)
    plt.show()

    e = neg_binomial_e(6, 0.25)
    d = neg_binomial_d(6, 0.25)
```

mcg.py

```
def lcg(modulus, a, c, seed, count=1000):
    for _ in range(count):
        seed = (a * seed + c) % modulus
        yield seed / modulus


def mcg(modulus, a, seed, count=1000):
    yield from lcg(modulus, a, 0, seed, count)
```

neg_binomial.py

```
def neg_binomial_gen(r, random_generator, count=1000):
    ''' Y = inf{n | Sn = r} - r '''
    for _ in range(count):
        sn = 0
        indx = 0
        n = -1

        for rand in random_generator:
            #print (rand)
            sn += rand
            if sn == r:
                n = indx
                break
            indx += 1

        yield n - r


def neg_binomial_d(r, p):
    return r * (1-p) / (p ** 2)


def neg_binomial_e(r, p):
    return r * (1-p) / p
```

puasson.py

```
import math


def puasson_gen(lambd, random_generator):
```

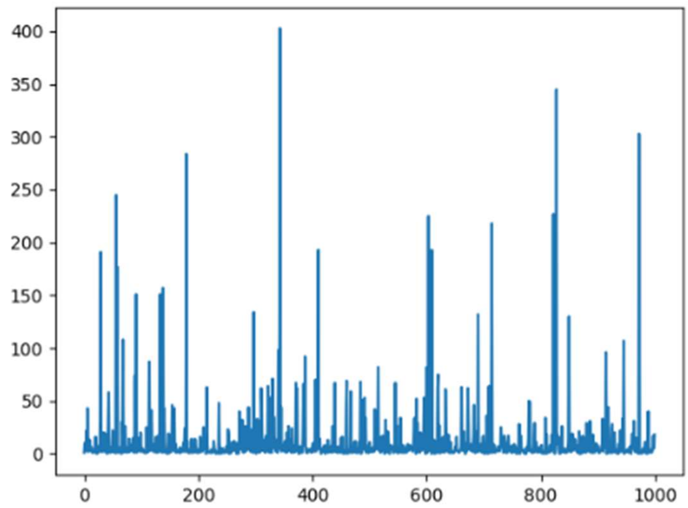
```
    for r in random_generator:
        p = math.exp(-lambd)
        a = 1.0
        x = 0

        while True:
            x += 1
            a *= r
            if a <= p:
                break

        yield x - 1

def puasson_e(lam):
    return lam


def puasson_d(lam):
    return lam
```

binomial.py

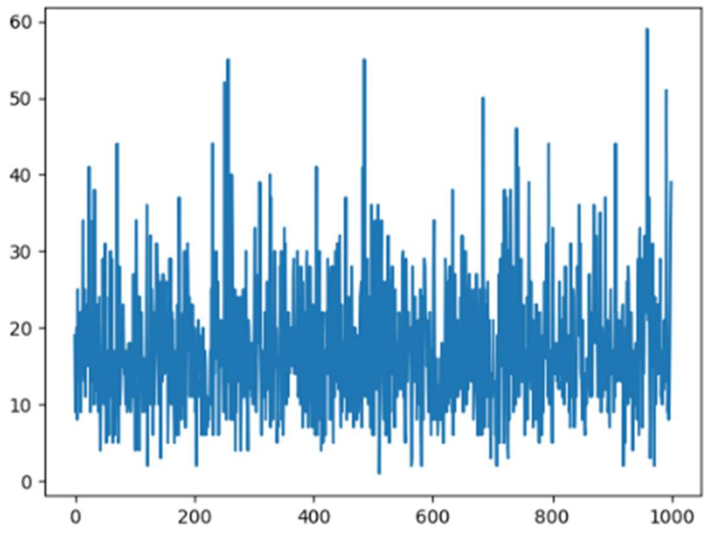
```
def binomial_gen(n, p, random_generator):
    for r in random_generator:
        x = 0
        for _ in range(n):
            if r < p:
                x += 1
        yield x


def bernoulli_gen(p, random_generator):
    yield from binomial_gen(1, p, random_generator)
```

Результат:

Пуассон:



Отрицательное биномиальное:



Мат. ожидание и дисперсия:

```
3 3
```

Лабораторная работа 3.

Условие:

Смоделировать непрерывную случайную величину (задания на стр. 25-47). 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 χ^2-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.

Условие задания:

8) m = -5, s2 = 25; Лапласа L(a), a = 1; Экспоненциальное E(a), a = 4.

Теория:

Нормальный закон распределения:

***Пример* 1.** $f_0(x)$ — плотность стандартного нормального распределения $\mathcal{N}(0,\ 1)$. Функция распределения (функция Лапласа) описывается интегралом вероятностей

$$\mathbf{F}_0(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt,$$

который аналитически вычислить не удается. Поэтому $\mathbf{F}_0(x)$ (и $\mathbf{F}_0^{-1}(y)$ тем более) не выражаются через элементарные функции. Метод обратной функции непосредственно применить нельзя.

Распределение Лапласа:

По определению, функция распределения — это интеграл от плотности распределения:

$$F(x) = \int_{-\infty}^{x} f(t)\,dt = \frac{\alpha}{2} \int_{-\infty}^{x} e^{-\alpha|t-\beta|}\,dt.$$

Для интегрирования необходимо рассмотреть два случая:

$$F(x) = \begin{cases} \frac{1}{2} e^{\alpha(x-\beta)}, & x \leqslant \beta, \\ 1 - \frac{1}{2} e^{-\alpha(x-\beta)}, & x > \beta. \end{cases}$$

Проверка свойств полученной функции:

1. $F(x)$ не убывает, так как $f(x)$ положительна.
2. $F(\beta - 0) = F(\beta + 0) = \dfrac{1}{2}$, следовательно, $F(x)$ непрерывна в точке β
3. $F(x)$ ограничена.
4. Пределы на бесконечностях:

$$\lim_{x\to-\infty} F(x) = \frac{1}{2} \lim_{x\to-\infty} e^{\alpha(x-\beta)} = 0,$$

$$\lim_{x\to+\infty} F(x) = 1 - \frac{1}{2} \lim_{x\to+\infty} e^{-\alpha(x-\beta)} = 1.$$

Экспоненциальное распределение:

Пример **2.** $f_0(x)$ — плотность экспоненциального (показательного) распределения ($E(\lambda)$)

$$f_0(x) = \lambda e^{-\lambda x}, \quad x \geq 0.$$

По формулам (3.29), (3.30) получаем

$$\mathbf{F}_0(x) = 1 - e^{-\lambda x}, \quad x \geq 0,$$

$$x = \mathbf{F}_0^{-1}(y) = -(1/\lambda)\ln(1 - y).$$

Для моделирования ξ^* с плотностью распределения $f_0(x)$ используем формулу

$$\xi = -\lambda^{-1}\ln(1 - \alpha),$$

где α — БСВ. Очевидно, что α и $1 - \alpha$ одинаково распределены, поэтому эквивалентным моделирующим алгоритмом является следующий:

$$\xi = -\mu\ln\alpha, \ \mu = \lambda^{-1} = \mathbf{E}\{\xi^*\}.$$

Код программы:

main.py

```
import math
from normal import normal_generator
from mult_met import mcg
from laplace import laplace_generator
from exponential import exponential_generator
import matplotlib.pyplot as plt
from criteria import pearson_criterion, kolmogorov_criterion
from disp import expectation, dispersion


def test_criterion(criterion, params):
    try:
        return "PASS" if criterion(*params) else "FAIL"
    except (ArithmeticError, ValueError):
        return "NOT PASS"


def normal_distribution(x, mu, sigma2):
    sigma = math.sqrt(sigma2)
    return math.exp(-0.5 * ((x - mu) / sigma) ** 2) / (sigma * math.sqrt(2 *
math.pi))


def lognormal_distribution(x, mu, sigma2):
    return math.exp(normal_distribution(x, mu, sigma2))


def exponential_distribution(x, a):
    return a*math.exp(-a*x)


def laplace_distribution(x, a):
    return math.copysign(1, -x) * math.exp(-a * x) / 2


def weibull_distribution(x, a, b):
    return 1 - math.exp(-a * x ** b)


if __name__ == '__main__':
    #print('Normal distribution:')
```

```
    normally_distributed = normal_generator(mu=-5, sigma2=25,
                                            modulus=2 ** 31, a=16807,
seed=16807,
                                            count=10000)
    #print(list(normally_distributed))
    seq = [g for g in normally_distributed]
    plt.plot(seq)
    plt.show()

    #print('Laplace distribution:')
    gen = mcg(modulus=2 ** 31, a=16807, seed=16807, count=1000)
    laplace_distributed = list(laplace_generator(1, gen))
    #print(laplace_distributed)
    seq = [g for g in laplace_distributed]
    plt.plot(seq)
    plt.show()

    #print('Exponential distribution:')
    gen = mcg(modulus=2 ** 31, a=16807, seed=16807, count=1000)
    exponentially_distributed = list(exponential_generator(4, gen))
    #print(exponentially_distributed)
    seq = [g for g in exponentially_distributed]
    plt.plot(seq)
    plt.show()

    mcg_params = dict(modulus=2 ** 31, a=16807, seed=16807, count=1000)
    distr = {"Normal distribution":
                 (normal_generator(mu=5, sigma2=25,
                                   **mcg_params), lambda x:
normal_distribution(x, mu=5, sigma2=25)),
             "Exponential distribution":
                 (exponential_generator(4,
                                        mcg(**mcg_params)), lambda x:
exponential_distribution(x, 4)),
             "Laplace distribution":
                 (laplace_generator(1,
                                    mcg(**mcg_params)), lambda x:
laplace_distribution(x, 1))}

    for name, (generator, func) in distr.items():
        print(name)
        seq = list(generator)
        if func:
            print("Pearson criterion:", test_criterion(pearson_criterion,
(seq, 25, func)))
            print("Kolmogorov criterion:",
test_criterion(kolmogorov_criterion, (seq, 25, func)))
        m = expectation(seq)
        print("Mean", m)
        print("Disp", dispersion(seq, m))
        print()
```

criteria.py

```
import math


def empirical(x, seq):
    res = 0
    for item in seq:
        if item < x:
            res += 1
    return res / len(seq)


def kolmogorov_criterion(seq, parts, distr_func):
```

```
    lo = min(seq)
    step = (max(seq) - min(seq)) / parts
    dn = 0
    for i in range(1, parts + 1):
        point = lo + i * step
        temp = math.fabs(empirical(point, seq) - distr_func(point))
        if dn < temp:
            dn = temp
    return math.sqrt(len(seq)) * dn < 1.36


def pearson_criterion(seq, parts, distr_func):
    lo = min(seq)
    if lo >= 0:
        lo = 0
    step = (max(seq) - lo) / parts
    frequency = [0] * parts
    for item in seq:
        index = int((item - lo) / step)
        if index == parts:
            index -= 1
        frequency[index] += 1
    res = 0
    for i in range(1, parts + 1):
        delta = len(seq) * (distr_func(lo + i * step) - distr_func(lo + (i -
1) * step))
        res += pow((frequency[i - 1] - delta), 2) / delta
    return res < 37.7
```

disp.py

```
def expectation(vec):
    res = 0
    for item in vec:
        res += item
    return res / len(vec)


def dispersion(vec, avg):
    res = 0
    for item in vec:
        res += pow(item - avg, 2)
    return res / (len(vec) - 1)
```

exponential.py

```
import math


def exponential_generator(a, generator):
    for n in generator:
        yield -math.log(n) / a
```

laplace.py

```
from mult_met import mcg
import math


def laplace_generator(a, generator):
    for n in generator:
        if n < 0.5:
            yield math.log(2 * n) / a
        else:
            yield -math.log(2 * (1 - n)) / a
```

mult_meg.py

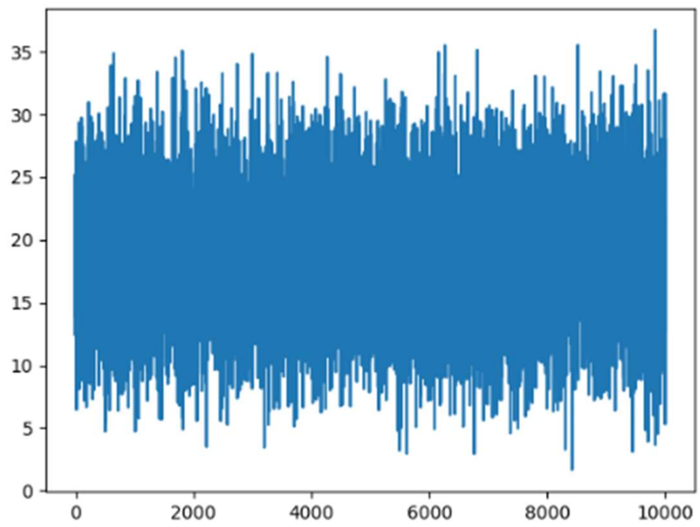
```python
def lcg(modulus, a, c, seed, count=100):
    """
    Linear congruential generator.
    """
    for i in range(count):
        seed = (a * seed + c) % modulus
        yield seed / modulus


def mcg(modulus, a, seed, count=100):
    """
    Multiplicative congruential generator.
    """
    yield from lcg(modulus, a, 0, seed, count)
```

normal.py

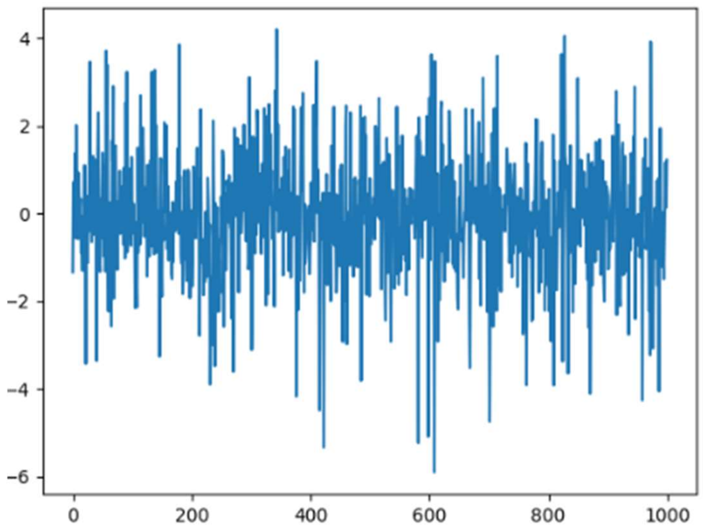
```python
from mult_met import mcg
import math


def normal_generator(mu, sigma2, modulus, a, seed, count):
    sequence = list(mcg(modulus, a, seed, count * 12))
    for i in range(count):
        yield mu + math.sqrt(sigma2) * sum(sequence[12 * i + j] for j in
range(12)) - 6
```

Результат:

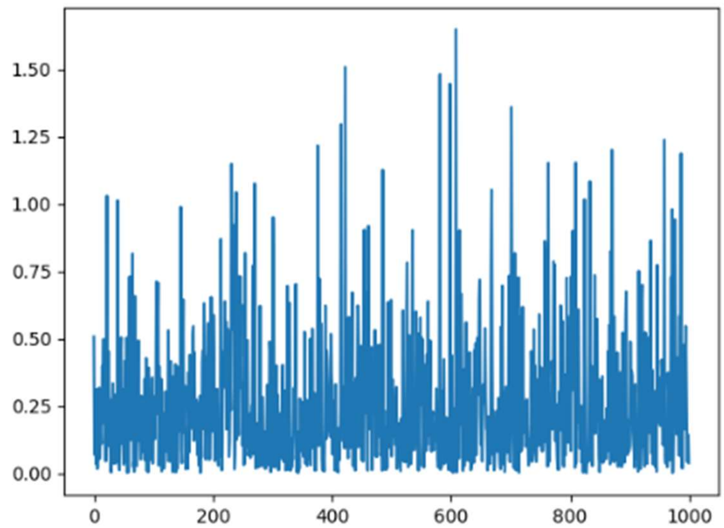
Нормальное распределение:



Распределение Лапласа:



Экспоненциальное распределение:



Критерии для распределений:

```
Normal distribution
Pearson criterion: PASS
Kolmogorov criterion: FAIL
Mean 28.705860751569272
Disp 25.623989646277376
```

```
Exponential distribution
Pearson criterion: PASS
Kolmogorov criterion: FAIL
Mean 0.25394371695586326
Disp 0.06183343649340454
```

```
Laplace distribution
Pearson criterion: PASS
Kolmogorov criterion: FAIL
Mean -0.045485452376975134
Disp 1.8056919874070252
```

Лабораторная работа 4.

Условие:

Вычислить значение интеграла, используя метод Монте-Карло. Оценить точность.

1. По методу Монте-Карло вычислить приближенное значения интегралов.
2. Сравнить полученное значение либо с точным значением (если его получится вычислить), либо с приближенным, полученным в каком-либо математическом пакете (например, в mathematica). Для этого построить график зависимости точности вычисленного методом Монте-Карло интеграла от числа итераций n.

Вариант задания:

8	$\int_0^2 e^x\sqrt{1+x}dx$	$\int_0^1\int_0^2 (x^2+y^2)dxdy$

Теория:

Метод Монте-Карло приближенного вычисления интеграла:

Необходимо вычислить $\int_{x_0}^{x_1} g(x)dx$.

Пусть η - произвольная случайная величина с плотностью распределения $P_\eta(x),\ x \in [x_0, x_1]$, имеющая конечный момент второго порядка.

Пусть $\xi = \dfrac{g(\eta)}{P_\eta(\eta)}$. Тогда $M\{\xi\} = a,\ D\{\xi\} < \infty$.

В качестве приближенного значения a можно взять

$$\frac{1}{n}\sum_{i=1}^{n}\xi_i = \frac{1}{n}\sum_{i=1}^{n}\frac{g(\eta_i)}{P_\eta(\eta_i)}.$$

В данной работе в качестве η бралась случайная величина, равномерно распределенная на отрезках, указанных в задании.

Код программы:

main.py

```
from MonteCarlo import monte_carlo, monte_carlo_double, compare_to_solution
from math import e, tan
import matplotlib.pyplot as plt


if __name__ == '__main__':
    integral1 = monte_carlo(0, 2, lambda x: ((e ** (-x)) * (1 + x) ** 0.5),
1000)
    print(f"1. {integral1}")

    integral2 = monte_carlo_double(0, 1, 0, 2, lambda x, y: x ** 2 + y ** 2,
1000)
    print(f"2. {integral2}")

    # compare monte-carlo1 result to solution = 1.11007
    seq1 = []
    for i in range(1000, 10000, 50):
        seq1.append(compare_to_solution(monte_carlo(0, 2, lambda x: ((e ** (-
x)) * (1 + x) ** 0.5), i), 1.11007))

    plt.plot(seq1)
    plt.title("Monte-Carlo method 1")
    plt.xlabel("Iterations (1000 + x*50)")
    plt.ylabel("Precision")
    plt.show()

    # compare monte-carlo2 result to solution = 3.3333333333
    seq2 = []
    for i in range(1000, 10000, 50):
        seq2.append(compare_to_solution(monte_carlo_double(0, 1, 0, 2, lambda
x, y: x ** 2 + y ** 2, i), 3.3333333333))

    plt.plot(seq2)
    plt.title("Monte-Carlo method 2")
    plt.xlabel("Iterations (1000 + x*50)")
    plt.ylabel("Precision")
    plt.show()
```

MonteCarlo.py

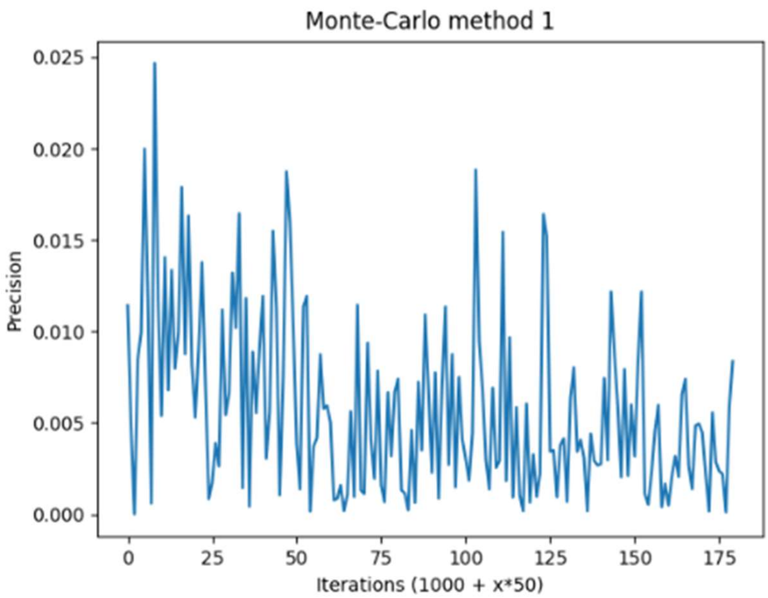
```
import numpy as np


def monte_carlo(left_bound, right_bound, function, n) -> float:
    """
    Monte-Carlo method implementation
    returns the value of the integral
    """
    sum_ = 0
    for i in range(n):
        sum_ += function(np.random.uniform(left_bound, right_bound)) *
(right_bound - left_bound)
```

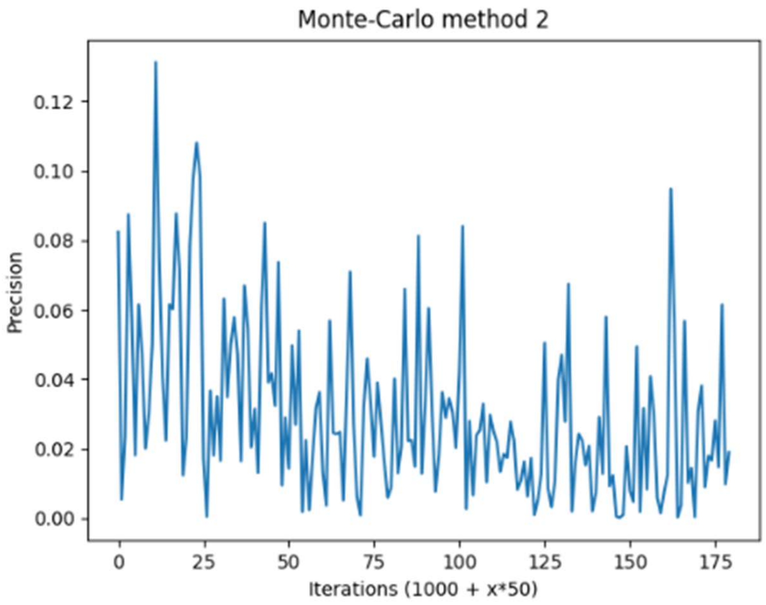
```
    return sum_ / n


def monte_carlo_double(left_bound1, right_bound1, left_bound2, right_bound2,
function, n) -> float:
    """
    Monte-Carlo method implementation for double integration
    returns the value of the integral
    """
    sum_ = 0

    for i in range(n):
        u1 = np.random.uniform(left_bound1, right_bound1)
        u2 = np.random.uniform(left_bound2, right_bound2)

        sum_ += function(u1, u2) * (right_bound1 - left_bound1) *
(right_bound2 - left_bound2)

    return sum_ / n


def compare_to_solution(generated_value, solution_value) -> float:
    """
    Compare generated value with solution value
    returns precision
    """
    return abs(generated_value - solution_value)
```

Результат:
Результаты вычисление интегралов:

```
1. 1.115405752599163
2. 3.3661024924076988
```

Графики:





Лабораторная работа 5.

Условие:

Решить систему линейных уравнений, используя метод Монте-Карло.

1. Решить систему линейных алгебраических уравнений $Ax = f$ методом Монте-Карло.
2. Сравнить с решением данного уравнения, полученным в произвольном математическом пакете.

Построить график зависимости точности решения от длины цепи маркова и числа смоделированных цепей маркова.

Вариант задания:

8) $A = \begin{pmatrix} 1.2 & -0.3 & 0.4 \\ 0.4 & 0.7 & -0.2 \\ 0.2 & -0.3 & 0.9 \end{pmatrix}, f = \begin{pmatrix} -4 \\ 2 \\ 0 \end{pmatrix}$

Теория:

Метод Монте-Карло приближенного решения системы линейных алгебраических уравнений:

Необходимо решить систему, представленную в виде $x = Ax + f$, где $x = (x_1, ..., x_n)^T$, $f = (f_1, ..., f_n)^T$, $A = (a_{ij}),\ i, j = \overline{1, n}$, собственные значения A по модулю меньше 1.

Наша цель – вычислить скалярное произведение вектора решения $x = (x_1, ..., x_n)^T$ с некоторым вектором $h = (h_1, ..., h_n)^T$.

Рассмотрим цепь Маркова с параметрами $\pi = (\pi_1, ..., \pi_n)^T$, $P = (p_{ij})$, такими что

$\pi_i \geq 0,\ \sum_{i=1}^{n} \pi_i = 1;$

$p_{ij} \geq 0,\ \sum_{j=1}^{n} p_{ij} = 1;$

$\pi_i > 0,$ если $h_i \neq 0;$

$p_{ij} > 0,$ если $a_{ij} \neq 0.$

Положим

$g_i^{(0)} = \begin{cases} h_i / \pi_i, & \pi_i > 0 \\ 0, & \pi_i = 0 \end{cases}, \ g_i^{(k)} = \begin{cases} a_{ij} / p_{ij}, & p_{ij} > 0 \\ 0, & p_{ij} = 0 \end{cases}.$

Выберем некоторое натуральное N и рассмотрим случайную величину

$$\xi_N = \sum_{m=0}^{N} Q_m f_{i_m},$$

Где $i_0 \to i_1 \to \ldots \to i_m$ - траекторая цепи Маркова.

Q_m опрядеяется как:

$$Q_0 = g_{i_0}^{(0)},\ Q_m = Q_{m-1} g_{i_{m-1} i_m}^{(m)}.$$

Тогда скалярное произведение вектором *h* и *x* приблизительно равно $E\{\xi_N\}$.

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

Код программы:

main.py

```
import matplotlib.pyplot as plt
import numpy as np
from MonteCarlo import monte_carlo_solve, norm
import sympy as sp


if __name__ == '__main__':
    # data according to the task
    a = [[1.2, -0.3, 0.4],
         [0.4, 0.7, -0.2],
         [0.2, -0.3, 0.9]]

    f = [-4, 2, 0]

    # accurate solution = [-2.829, 5.143, 2.343]
    n = 3
    markov_len = 50
    markov_cnt = 1000

    # x = Bx + f
    b = [[-0.2, 0.3, -0.4],
         [-0.4, 0.3, 0.2],
         [-0.2, 0.3, 0.1]]
    # probability matrix
    p = [[1 / 3, 1 / 3, 1 / 3],
         [1 / 3, 1 / 3, 1 / 3],
         [1 / 3, 1 / 3, 1 / 3]]
```

```
    x = monte_carlo_solve(b, n, f, p, markov_len, markov_cnt)
    print(f"[1] Monte-Carlo method: x = {x}")

    # get an accurate solution using sympy
    a1 = sp.Matrix(a)
    b1 = sp.Matrix(f)
    x1 = a1.inv() * b1
    print(f"[2] Accurate solution: x1 = {x1}")

    # difference between solutions
    print(f"[3] Norm of x-x1: {norm(x, x1, n)}")

    # task 3 - plot the solution
    X = np.loadtxt('Lab5Cnt.txt')
    Y = np.loadtxt('Lab5Len.txt')
    Z = np.loadtxt('Lab5Norms.txt')

    fig = plt.figure()
    ax = fig.gca(projection='3d')

    ax.plot_trisurf(X, Y, Z, linewidth=0, antialiased=True)
    plt.show()
```

MonteCarlo.py

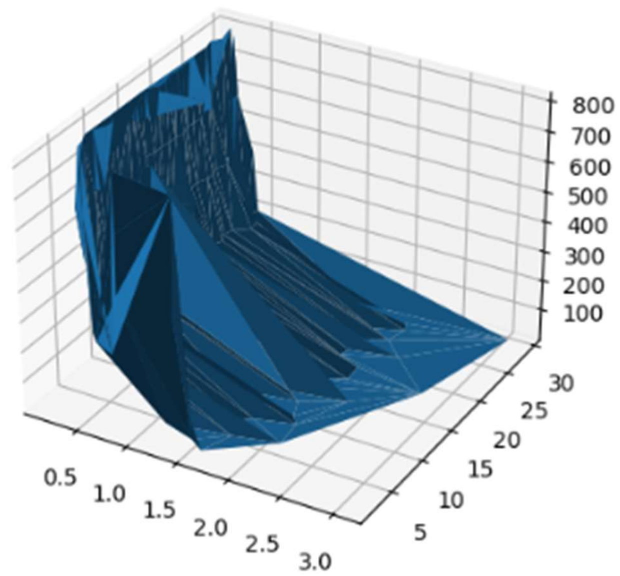
```
from numpy import random
from math import sqrt, pow


def norm(X, X1, n):
    return sqrt(sum([pow(X[i] - X1[i], 2) for i in range(n)]))


def markov_rand(P, state, size):
    rand = random.random()
    for i in range(size):
        rand -= P[state][i]
        if rand <= 0:
            return i
    return size - 1


def monte_carlo_solve(b, n, f, P, markov_len=50, markov_cnt=1000) -> list:
    """
    Solves the linear system x = Bx + f using the Monte-Carlo method
    :return list: x
    """
    solution = []
    for coordinate in range(n):
        x = 0
        for i in range(markov_cnt):
            prev_m = coordinate
            prev_q = 1
            for j in range(1, markov_len):
                next_m = markov_rand(P, prev_m, n)
                next_q = 0
                if P[prev_m][next_m] > 0:
                    next_q = prev_q * b[prev_m][next_m] / P[prev_m][next_m]
                x += next_q * f[next_m]
                prev_q = next_q
                prev_m = next_m
        solution.append(f[coordinate] + x / markov_cnt)
    return solution
```

Результат:

Вывод результата методом Монте-Карло, точного решения и нормы их разницы:

```
[1] Monte-Carlo method: x = [-2.9660046685971357, 5.087725668520265, 2.3770702497251075]
[2] Accurate solution: x1 = Matrix([[-2.82857142857143], [5.14285714285714], [2.34285714285714]])
[3] Norm of x-x1: 0.15197997107540312
```

График:



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.