МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Кафедра многопроцессорных систем и сетей

Аладко Анастасия Дмитриевна

Отчет по лабораторным работам по курсу

“Математическое моделирование”

студентки 3 курса 12 группы

|  |
| --- |
| **Преподаватель** |
| *Лобач Виктор Иванович*  доцент кафедры ММАД,  канд. физ.-мат. наук |
|  |

*Минск 2024*

Оглавление

Лабораторная работа № 1. Моделирование БСВ………………………………3

Лабораторная работа № 2. Моделирование дискретных СВ ..…..…………….8

Лабораторная работа № 3. Моделирование непрерывных СВ………………12

Лабораторная работа № 4. Вычисление определенных интегралов методом Монте-Карло……………………………………………………………….……20

Лабораторная работа № 5. Решение СЛАУ методом Монте-Карло………...24

**Лабораторная работа № 1.**

**Моделирование БСВ**

**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 .
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.

**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.

**main.py**

import math

import scipy.stats as stats

M = 2 \*\* 31

N = 1000

def MCM(a0, bet):

a\_star = bet

alpha1 = []

alpha1.append(a0/M)

for i in range(1, N):

a\_star = (bet \* a\_star) % M

alpha1.append(a\_star / M)

return alpha1

def mclaren(b, c, k):

v = b[:k]

alpha1 = []

for i in range(N):

s = ((int)(c[i] \* k)) % k

alpha1.append(v[s])

v[s] = b[(i + k) % N]

return alpha1

#эмпирическая функция распределения

def elem\_less\_map(alpha1):

alpha1.sort()

temp = 1/N

d = {}

summ = 0

for i in alpha1:

d[i] = summ

summ += temp

return d

def pirson(seq):

seq.sort()

for i in range(N):

seq[i] = seq[i] / seq[-1]

l = 10

p = []

seg = [0 for i in range(l)]

x = 0

for index in range(l):

p.append(1 / l)

for i in range(N):

if x <= seq[i] < x + 1 / l:

seg[index] += 1

x += 1 / l

s = 0

for i in range(l):

s += ((seg[i] - N \* p[i]) \*\* 2) / (N \* p[i])

return s

def kolmagorov(alpha1):

alpha1.sort()

min = alpha1[0]

max = alpha1[-1]

d = elem\_less\_map(alpha1)

answer = 0

for i in alpha1:

if (abs(d[i] - ((i - min)/(max - min))) > answer):

answer = abs(d[i] - ((i - min)/(max - min)))

return answer

def main():

bet = 79507

a0 = bet

K = 64

seq1 = MCM(a0, bet)

print('Критерий Колмагорова 1: %f' % (kolmagorov(seq1)))

print('Критерий Пирсона 1: %f' % (pirson(seq1)))

seq2 = mclaren(seq1, MCM(2 \* a0, 2 \* bet), K)

print('Критерий Колмагорова 2: %f' % (kolmagorov(seq2)))

print('Критерий Пирсона 2: %f' % (pirson(seq2)))

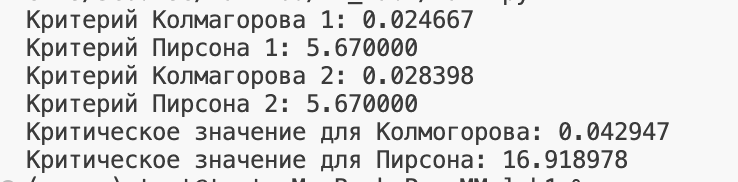
print('Критическое значение для Колмогорова: %f' % (math.sqrt(-0.5 \* math.log(0.05 / 2) / N)))

print('Критическое значение для Пирсона: %f' % stats.chi2.ppf(0.95, 9))

if \_\_name\_\_ == "\_\_main\_\_":

main()

Результат выполнения программы:



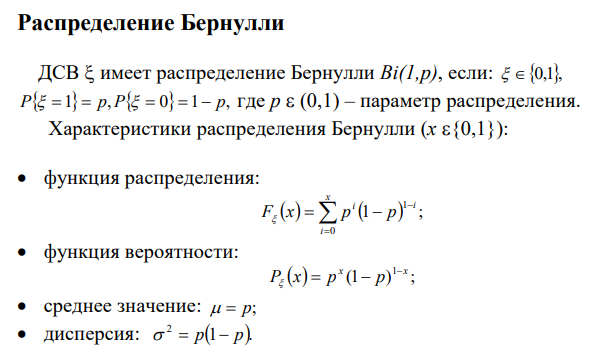
**Лабораторная работа № 2.**

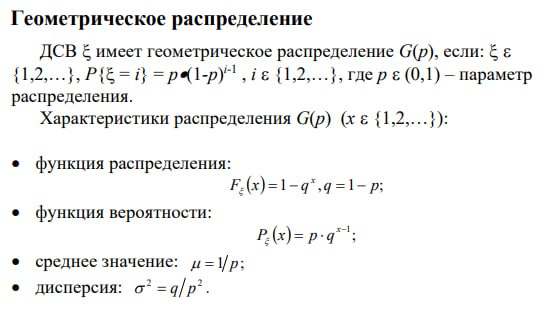
**Моделирование дискретных СВ**

**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 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.

**Теория:**

****

**main.py**

import math

import matplotlib.pyplot as plt

import scipy.stats as sps

# Функция для расчета критерия Пирсона

def pirson\_stat(values, intervals, distribution, param):

stat = 0

step = (max(values) - min(values)) / intervals

x = min(values)

y = x + step

for i in range(1, intervals):

observed = 0

for val in values:

if x < val <= y:

observed += 1

expected = len(values) \* (distribution(y, param) - distribution(x, param))

stat += ((observed - expected) \*\* 2 + 1e-9) / (expected + 1e-9)

x = y

y += step

return stat

# Функции распределения

def geom\_cdf(x, p):

return sps.geom.cdf(x + 1, p)

def bern\_cdf(x, p):

return sps.bernoulli.cdf(x, p)

INTERVALS = 11

PIRSON\_QUANTIL = 19.7

SAMPLES = 1000

# Распределение Бернулли

P\_bernoulli = 0.9

data\_bern = sps.bernoulli(P\_bernoulli).rvs(SAMPLES)

E\_bern = sum(data\_bern) / len(data\_bern)

D\_bern = sum((x - E\_bern) \*\* 2 for x in data\_bern) / len(data\_bern)

print('Распределение Бернулли:')

print(f'Истинное математическое ожидание: {P\_bernoulli}. Полученное: {E\_bern}')

print(f'Истинная дисперсия: {P\_bernoulli \* (1 - P\_bernoulli)}. Полученная: {D\_bern}')

pirson\_bern = pirson\_stat(data\_bern, INTERVALS, bern\_cdf, P\_bernoulli)

print(f"Критерий Пирсона: {pirson\_bern}, квантиль: {PIRSON\_QUANTIL}")

if pirson\_bern < PIRSON\_QUANTIL:

print("Гипотеза о соответствии принята")

else:

print("Гипотеза о соответствии отвергнута")

# Геометрическое распределение

P\_geometric = 0.7

data\_geom = sps.geom(P\_geometric).rvs(SAMPLES) - 1

E\_geom = sum(data\_geom) / len(data\_geom)

D\_geom = sum((x - E\_geom) \*\* 2 for x in data\_geom) / len(data\_geom)

print('\nГеометрическое распределение:')

print(f'Истинное математическое ожидание: {(1 - P\_geometric) / P\_geometric}. Полученное: {E\_geom}')

print(f'Истинная дисперсия: {(1 - P\_geometric) / P\_geometric\*\*2}. Полученная: {D\_geom}')

pirson\_geom = pirson\_stat(data\_geom, INTERVALS, geom\_cdf, P\_geometric)

print(f"Критерий Пирсона: {pirson\_geom}, квантиль: {PIRSON\_QUANTIL}")

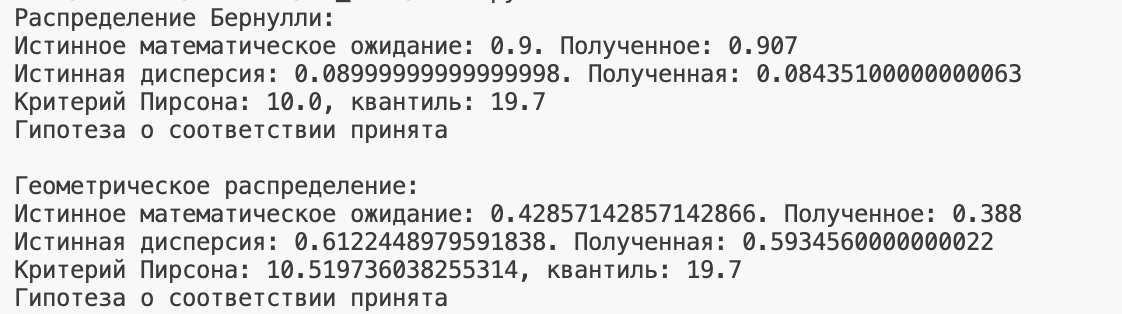
if pirson\_geom < PIRSON\_QUANTIL:

print("Гипотеза о соответствии принята")

else:

print("Гипотеза о соответствии отвергнута")

Результат выполнения программы:

****

**Лабораторная работа № 3.**

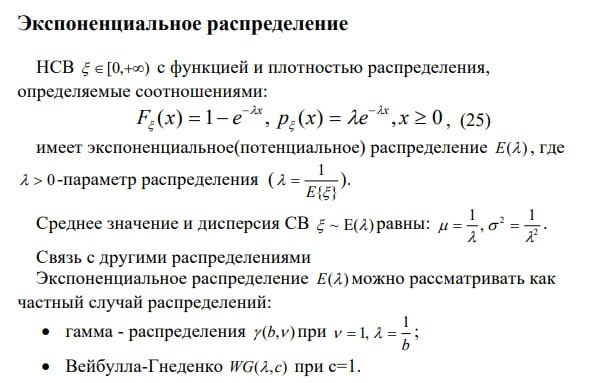
**Моделирование непрерывных СВ**

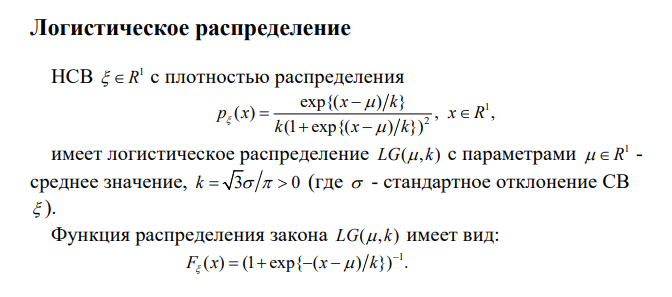
**Условие:**

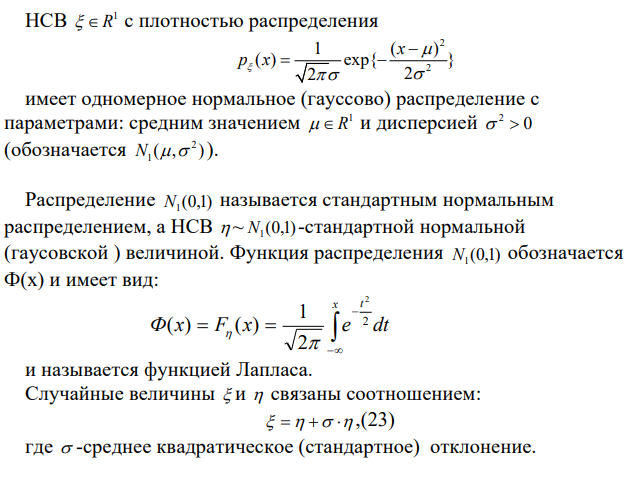
Смоделировать непрерывную  случайную величину (задания на стр. 25-47). Исследовать точность моделирования.

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 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.

**Теория:**

****

****

****

**distributions.py**

from cmath import log

import math

import random

class NormalDistribution:

def \_\_init\_\_(self, m, s2):

self.m\_ = m

self.s2\_ = s2

self.n\_ = 24

def rand(self):

result = 0

for \_ in range(self.n\_):

result +=random.random()

result -= self.n\_ / 2

result \*= math.sqrt(12 / self.n\_)

return result \* math.sqrt(self.s2\_) + self.m\_

class ExponentialDistribution:

def \_\_init\_\_(self, lyambda):

self.lyambda = lyambda

def rand(self):

a = random.random()

return -1 / self.lyambda \* math.log(a)

class LogisticDistribution:

def \_\_init\_\_(self, c, m):

self.c = c

self.m = m

self.normal\_dist = NormalDistribution(0, 1)

def rand(self):

p = random.random()

return self.c + self.m \* math.log(p / (1 - p))

**main.py**

import math

import scipy

from scipy import stats

import distributions

import matplotlib.pyplot as plt

eps = 1e-9

#гипотеза согласия

def pierson\_criteria(random\_values, partitions: int, prob\_func, \*args):

random\_values.sort()

step = (random\_values[-1] - random\_values[0]) / partitions

current\_level = random\_values[0] + step

n = 1000

frequency = [0] \* partitions

counter = 0

j = 0

for value in random\_values:

if value - current\_level > eps:

current\_level += step

frequency[j] = counter

j += 1

counter = 0

counter += 1

frequency[partitions - 1] = counter

result = 0

current\_level = random\_values[0]

for i in range(partitions):

low = prob\_func(\*args, current\_level)

up = prob\_func(\*args, current\_level + step)

p = up - low

result += ((frequency[i] - n \* p) \*\* 2) / (n \* p)

current\_level += step

kvantil = stats.distributions.chi2.ppf(1 - 0.05, partitions)

if result <= kvantil:

print("Pirson criteria is passed!")

else:

print("Pirson criteria was not passed(")

print(f"{result} < {kvantil}")

#максимальное отклонение

def kolmogorov\_criteria(random\_values, partitions: int, prob\_func, \*args):

critical\_value = 1.36

random\_values.sort()

step = (random\_values[-1] - random\_values[0]) / partitions

current\_level = random\_values[0] + step

n = 10000

frequency = [0] \* partitions

counter = 0

j = 0

for value in random\_values:

if value - current\_level > eps:

current\_level += step

frequency[j] = counter

j += 1

counter = 0

counter += 1

frequency[partitions - 1] = counter

result = 0

current\_level = random\_values[0]

max\_d = -1e10

current\_d = 0

current\_level = random\_values[0]

for i in range(partitions):

low = prob\_func(\*args, current\_level)

up = prob\_func(\*args, current\_level + step)

p = up - low

current\_d = (frequency[i] / 1000) - p

current\_level += step

if current\_d > max\_d:

max\_d = current\_d

if math.sqrt(n) \* max\_d <= critical\_value:

print("Kolmogorov criteria is passed!")

else:

print("Kolmogorov criteria was not passed(")

print(f"{math.sqrt(n) \* max\_d} <= {critical\_value}")

def get\_dist\_stats(result: list):

e\_value = 0

for number in result:

e\_value += number

e\_value /= len(result)

dispersion = 0

for number in result:

dispersion += (number - e\_value) \*\* 2

dispersion /= (len(result) - 1)

return e\_value, dispersion

def normal(m, d, x):

return scipy.stats.norm.cdf(x, m, d)

def e\_pow(x):

return math.e \*\* (- 0.5 \* ((x - 5) / math.sqrt(9)) \*\* 2)

def exponential(lyambda, x):

return 1 - (math.e \*\* (-lyambda \* x))

def logistic(c, m, x):

return 1 / (1 + math.e \*\* (-(x - c) / m))

def main():

n = 1000

m = -4

d = 4

normal\_dist = distributions.NormalDistribution(m, d)

result = []

for \_ in range(n):

result.append(normal\_dist.rand())

print(

"===================== [ Normal Distribution Test ] ======================")

print("Theoretical expected value is", m,

"\nTheoretical dispersion is ", d)

practical\_m, practical\_d = get\_dist\_stats(result)

print(

f"Practical expected value is {practical\_m}\nPractical dispersion is {practical\_d}")

pierson\_criteria(result, 10, normal, m, d)

kolmogorov\_criteria(result, 3000, normal, m, d)

lyambda = 0.5

exponential\_dist = distributions.ExponentialDistribution(lyambda)

result = []

for \_ in range(n):

result.append(exponential\_dist.rand())

print(

"===================== [ Exponential Distribution Test ] =====================")

print("Theoretical expected value is ", 1/lyambda,

"\nTheoretical dispersion is ", 1 / (lyambda \*\* 2))

practical\_m, practical\_d = get\_dist\_stats(result)

print(

f"Practical expected value is {practical\_m}\nPractical dispersion is {practical\_d}")

pierson\_criteria(result, 1000, exponential, lyambda)

kolmogorov\_criteria(result, 3000, exponential, lyambda)

result = []

for \_ in range(n):

result.append(exponential\_dist.rand())

c = 0

m = 1.5

logistic\_dist = distributions.LogisticDistribution(c, m)

result = []

for \_ in range(n):

result.append(logistic\_dist.rand())

print(

"===================== [ Logistic Distribution Test ] ======================")

print("Theoretical expected value is", c,

"\nTheoretical dispersion is", (math.pi \*\* 2 / 3) \* (m \*\* 2))

practical\_m, practical\_d = get\_dist\_stats(result)

print(

f"Practical expected value is {practical\_m}\nPractical dispersion is {practical\_d}")

pierson\_criteria(result, 10, logistic, c, m)

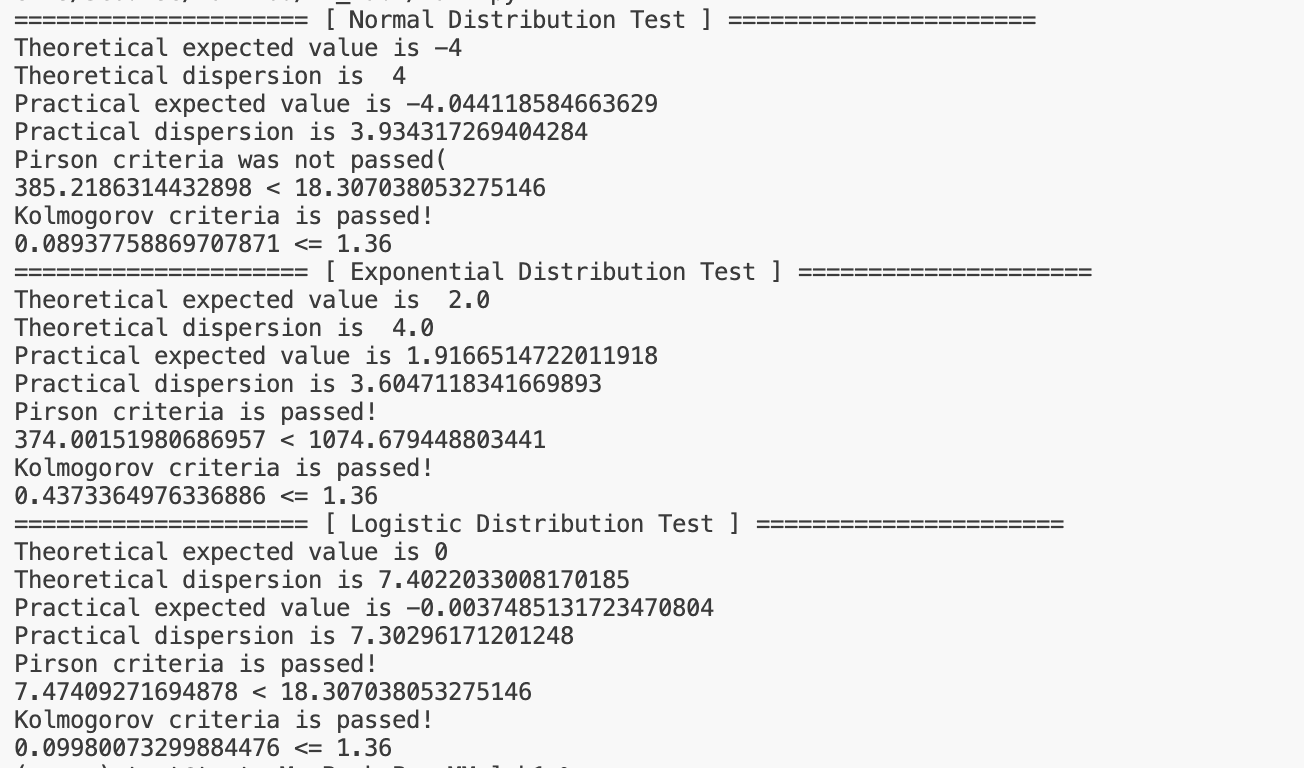
kolmogorov\_criteria(result, 3000, logistic, c, m)

return 0

if \_\_name\_\_ == "\_\_main\_\_":

main()

Результат выполнения программы:

****

**Лабораторная работа № 4.**

**Вычисление определенных интегралов методом Монте-Карло**

**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. Для этого построить график зависимости точности вычисленного методом Монте-Карло интеграла от числа итераций n.



**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].

**main.py**

import random

import math

import matplotlib.pyplot as plt

import numpy as np

def MonteCarlo1(func, density):

ITERS = int(1000)

results = [0]

for i in range(ITERS):

x = random.uniform(1, 3)

results.append(results[len(results) - 1] + func(x) / density())

for i in range(1, len(results)):

results[i] /= i

return results

def MonteCarlo2(func, density):

ITERS = int(100000)

results = [0]

for i in range(ITERS):

x = random.uniform(-3, 3)

y = random.uniform(-3, 3)

if abs(x) + abs(y) < 3:

results.append(results[len(results) - 1] + func(x, y) / density())

for i in range(1, len(results)):

results[i] /= i

return results

def func1(x):

return 1 / (np.power(x, 4) + 3 \* np.power(x, 2) + 17)

def func2(x, y):

return (x \* np.power(y, 2) + 1) \* np.sin(x)

def density2():

return 6

def density1():

return 1

#1

real\_value1 = 0.022721

integral = MonteCarlo1(func1, density1)

print("First integral: " + str(integral[len(integral) - 1]))

print("Real value: " + str(real\_value1))

for i in range(len(integral)):

integral[i] = abs(integral[i] - real\_value1)

plt.subplot(1, 2, 1)

plt.title("Integral error")

plt.plot(integral)

#2

real\_value2 = 0.023

double\_integral = MonteCarlo2(func2, density2)

print("Second integral: " + str(double\_integral[len(double\_integral) - 1]))

print("Real value: " + str(real\_value2))

for i in range(len(double\_integral)):

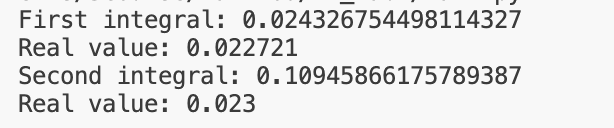
double\_integral[i] = abs(double\_integral[i] - real\_value2)

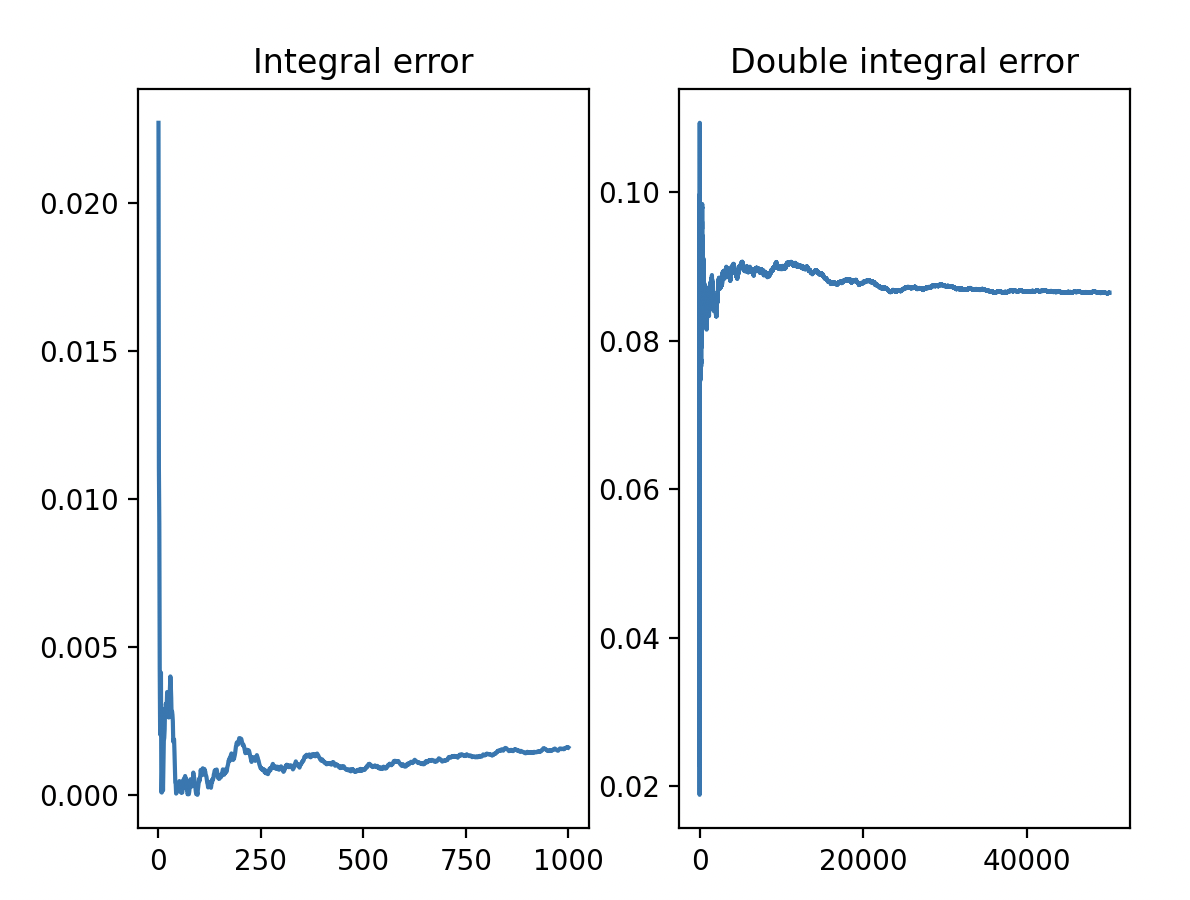
plt.subplot(1, 2, 2)

plt.title("Double integral error")

plt.plot(double\_integral)

plt.show()

Результат выполнения программы:

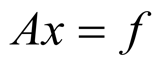
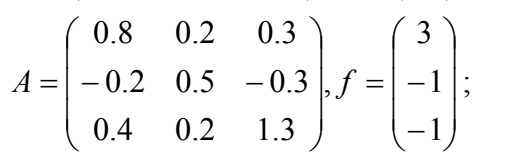
****

**Лабораторная работа № 5.**

**Решение СЛАУ методом Монте-Карло**

**Условие:**

Решить систему линейных уравнений, используя метод Монте-Карло.

1. Решить систему линейных алгебраических уравнений  методом Монте-Карло.
2. Сравнить с решением данного уравнения, полученным в произвольном математическом пакете.
3. Построить график зависимости точности решения от длины цепи маркова и числа смоделированных цепей маркова. 

**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.

**main.py**

import random

from scipy.linalg import solve

from matplotlib import pyplot

x = [[]]

def step\_markov(state, P, size):

r = random.random()

for i in range(size):

r -= P[state][i]

if r <= 0:

return i

return size - 1

def model\_markov(A, f, P, start, depth):

Q = 1

sum = 0

old = start

for i in range(depth):

new = step\_markov(old, P, len(P[old]))

if P[old][new] > 0:

Q = Q \* A[old][new] / P[old][new]

else:

Q = 0

sum += Q \* f[new]

old = new

return sum

def slae\_solve(A, f, depth, iterations):

P = []

size = len(A)

B = []

for i in range(size):

B.append([])

P.append([])

for j in range(size):

if i == j:

B[i].append(1 - A[i][j])

else:

B[i].append(-A[i][j])

P[i].append(1 / size)

answer = []

for coord in range(size):

sum = 0

x.append([])

for i in range(iterations):

e = model\_markov(B, f, P, coord, depth)

sum += e

x[coord].append(f[coord] + sum / iterations)

answer.append(f[coord] + sum / iterations)

return answer

A = [

[0.8, 0.2, 0.3],

[-0.2, 0.5, -0.3],

[0.4, 0.2, 1.3]

]

f = [-3, -1, -1]

result = slae\_solve(A, f, 100, 10000)

print("Result calculated by me:")

print(result)

pythonX = solve(A, f)

print("Result calculated with python library:")

print(pythonX)

pyplot.subplot(1,3,1)

pyplot.title("x1")

pyplot.plot(x[0])

pyplot.subplot(1,3,2)

pyplot.title("x2")

pyplot.plot(x[1])

pyplot.subplot(1,3,3)

pyplot.title("x3")

pyplot.plot(x[2])

pyplot.show()

Результат выполнения программы: