**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ**

**Кафедра теории вероятностей и математической статистики**

**ИТОГОВЫЙ ОТЧЕТ**

Имитационное и статистическое моделирование

|  |
| --- |
| Зуйкевич Лидии Анатольевны, студентки 4 курса, 7 группы  Преподаватель:  Е. А. Шибеко |

Минск, 2023

ОГЛАВЛЕНИЕ

[Глава 1. ЛАБОРАТОРНАЯ РАБОТА 1 3](#_Toc155994351)

[1.1 Условие 3](#_Toc155994352)

[1.2 Теоретические сведения 3](#_Toc155994353)

[1.3 Листинг программы 6](#_Toc155994354)

[1.4 Результаты 9](#_Toc155994355)

[Глава 2. ЛАБОРАТОРНАЯ РАБОТА 2 12](#_Toc155994356)

[2.1 Условие 12](#_Toc155994357)

[2.2 Теоретические сведения 12](#_Toc155994358)

[2.3 Листинг программы 13](#_Toc155994359)

[2.4 Результаты 17](#_Toc155994360)

[Глава 3. ЛАБОРАТОРНАЯ РАБОТА 3 20](#_Toc155994361)

[3.1 Условие 20](#_Toc155994362)

[3.2 Теоретические сведения 21](#_Toc155994363)

[3.3 Листинг программы 22](#_Toc155994364)

[3.4 Результаты 27](#_Toc155994365)

[Глава 4. ЛАБОРАТОРНАЯ РАБОТА 4 32](#_Toc155994366)

[4.1 Условие 32](#_Toc155994367)

[4.2 Теоретические сведения 32](#_Toc155994368)

[4.3 Листинг программы 33](#_Toc155994369)

[4.4 Результаты 35](#_Toc155994370)

[Глава 5. ЛАБОРАТОРНАЯ РАБОТА 5 37](#_Toc155994371)

[5.1 Условие 37](#_Toc155994372)

[5.2 Теоретические сведения 37](#_Toc155994373)

[5.3 Листинг программы 39](#_Toc155994374)

[5.4 Результаты 41](#_Toc155994375)

# ЛАБОРАТОРНАЯ РАБОТА 1

## 1.1 Условие

Используя метод Макларена-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.
4. Построить диаграмму рассеяния и гистограмму для 1) и 2) на доп балл.

Параметры варианта:

1. *a*0 = β = 78 125, K = 256

## 1.2 Теоретические сведения

Базовой случайной величиной (БСВ) непрерывная СВ, равномерно распределённая на полуинтервале [0,1).

Для моделирования на ЭВМ реализаций БСВ используются специальные программы, называемые программными датчиками БСВ. В основе программных датчиков БСВ лежат рекуррентные формулы.

Мультипликативный конгруэнтный метод

Согласно этому методу псевдослучайная последовательность реализаций БСВ определяется по рекуррентным формулам:

Где - параметры датчика (натуральные числа): - множитель (<M), M – модуль, - стартовое значение (нечётное число).

Операция означает вычет числа z по модулю М:

где [x] – целая часть числа x. Период датчика , коэффициент использования БСВ k = 1. Значения параметров определяются из условия максимума периода Т. значение М зависит от способа представления целых чисел в ЭВМ.

Метод Макларена-Марсальи

Метод основан на комбинировании двух простейших программных датчиков БСВ (например, мультипликативных конгруэнтных).

Пусть  – псевдослучайные последовательности, порождаемые независимо работающими датчиками;  – результирующая псевдослучайная последовательность реализаций БСВ;  – вспомогательная таблица K чисел.

Процесс вычисления  включает следующие этапы:

• первоначальное заполнение таблицы V:



• случайный выбор из таблицы:



• обновление табличных значений:



Данный метод позволяет ослабить зависимость между членами псевдослучайной последовательности  и получить сколь угодно большие значения её периода Т при условии, что периоды Т1, Т2 исходных датчиков являются взаимно простыми числами.

Коэффициент использования БСВ для данного датчика k = 1/2 (за исключением первой реализации, для моделирования которой используется

K + 1 реализация).

Критерий согласия χ2

Пусть некоторый алгоритм осуществляет моделирование случайной величины . В результате *n* – кратного обращения к этому алгоритму образуется случайная выборка объема *n*, которую обозначим через *X*, *X* =, где

*xi* – реализация смоделированной случайной величины . Необходимо при помощи выборки *X* проверить гипотезу *H*0 о том, что Конкурирующая гипотеза 

Критерий согласия χ2**.** В выборке *X* находим минимальный и максимальный элементы: *x*- = min{*x*i}, *x*+ = max{*x*i} и осуществим разбиение числовой прямой на *k* ячеек: (-, [*x*- + *h*, *x*-+ 2*h*), …, [*x*+ -2*h*, *x*+ - *h*), [*x+* - *h*, + ), где *h* = .

Теоретическая вероятность попадания  в *i* – ю ячейку [*x*- + (*i* - 1)*h*, *x*- + *ih*), если верна гипотеза *H*0:



Выбираем *k* так, чтобы в каждую ячейку попадало не менее 5 выборочных значений. Подсчитываем число *n*i выборочных значений, попавших в *i*-ю ячейку. Вычисляем статистику χ2:



Решающее правило:

Принимается гипотеза {

где порог выбирается так, чтобы вероятность ошибки первого рода равнялась заданному уровню значимости :



В качестве обычно выбирается одно из следующих значений: 0,001; 0,01; 0,05.

При верной гипотезе *H*0 и случайная величина χ2 имеет распределение χ2с *k* – 1 степенью свободы. Функция распределения этой случайной величины имеет вид:



Эта функция табулирована. Порог можно определить по таблицам распределения с *k*- 1 степенью свободы:



Критерий согласия Колмогорова

Для случайной абсолютно-непрерывной величины  функция распределения является непрерывной функцией. Обозначим через порядковые статистики, т.е. выборочные статистики, упорядоченные в порядке их возрастания. Эмпирическая функция распределения СВ :



Расстояние Колмогорова между эмпирической функцией распределения  и :

.

Решающее правило:

Принимается гипотеза 

где порог выбирается так, чтобы вероятность ошибки первого рода равнялась заданному уровню значимости :



Если верна гипотеза *H*0 и *n* значительно больше единицы (практически для *n* 20), то независимо от  случайная величина  имеет распределение Колмогорова:



это распределение табулировано. Это позволяет, аналогично предыдущему пункту, определить порог .

## 1.3 Листинг программы

import math  
import numpy as np  
from scipy.stats import chi2, kstest, uniform  
import matplotlib.pyplot as plt  
  
  
class MultiplicativeCongruentMethod:  
 def \_\_init\_\_(self, a, b, m):  
 self.prev\_el = a  
 self.beta = b  
 self.M = m  
  
 def next\_element(self):  
 z = self.beta \* self.prev\_el  
 self.prev\_el = z - self.M \* int(z / self.M)  
 return self.prev\_el / self.M  
  
 def generate\_n(self, num\_el):  
 return [self.next\_element() for \_ in range(num\_el)]  
  
  
class MultiplicativeCongruentialGenerator:  
 def \_\_init\_\_(self, size, beta, alpha0, m):  
 self.size = size  
 self.beta = beta  
 self.alpha0 = alpha0  
 self.m = m  
 self.seq = np.zeros(size)  
 self.prev = alpha0  
  
 def generate\_sequence(self):  
 for i in range(self.size):  
 z = self.beta \* self.prev  
 self.prev = z - self.m \* int(z / self.m)  
 self.seq[i] = self.prev / self.m  
  
 def print\_seq(self):  
 print(f"Реализация БСВ мультипликативным конгруэнтным методом при beta = {self.beta}, alpha0 = {self.alpha0}, m = {self.m}: ")  
 print(self.seq)  
  
  
class MacLarenMarsagliaGenerator:  
 def \_\_init\_\_(self, n, k):  
 self.n = n  
 self.k = k  
 self.alpha = np.zeros(n)  
  
 def maclaren\_marsaglia\_method(self, c, b):  
 *# len(c) = n, len(b) = n + k* v = b[:self.k]  
 for i in range(self.n):  
 s = int(c[i] \* self.k)  
 self.alpha[i] = v[s]  
 v[s] = b[i + self.k]  
  
 def print\_seq(self):  
 print(f"Реализация БСВ методом МакЛарена Марсальи, k = {self.k}: ", '\n', self.alpha)  
  
  
class StatTests:  
 def \_\_init\_\_(self, seq, e, name):  
 self.seq = seq  
 self.e = e  
 self.name = name  
 self.intervals = self.find\_intervals()  
  
 def find\_intervals(self):  
 n = 1 + int(math.log2(len(self.seq)))  
 max\_el = np.max(self.seq)  
 min\_el = np.min(self.seq)  
 h = (max\_el - min\_el) / n  
 a = np.zeros(n)  
 for i in range(n - 1):  
 a[i] = min\_el + (i + 1) \* h  
 a[-1] = 1  
 return a  
  
 def observed\_frequencies(self, seq):  
 n = len(self.intervals)  
 freq = np.zeros(n)  
 sort\_seq = np.sort(seq)  
 i = 0  
 k = 0  
 while i < n:  
 while k < len(seq) and sort\_seq[k] < self.intervals[i]:  
 freq[i] += 1  
 k += 1  
 i += 1  
 return freq  
  
 def expected\_frequencies(self):  
 n = len(self.intervals)  
 exp\_freq = np.zeros(n)  
 exp\_freq[0] = uniform.cdf(self.intervals[0])  
 for i in range(1, n - 1):  
 exp\_freq[i] = uniform.cdf(self.intervals[i]) - uniform.cdf(self.intervals[i - 1])  
 exp\_freq[-1] = 1 - uniform.cdf(self.intervals[-2])  
 return exp\_freq  
  
 def chisquare\_test(self):  
 print("Критерий хи-квадрат: ")  
 obs\_freq = self.observed\_frequencies(self.seq)  
 exp\_freq = self.expected\_frequencies()  
 k = len(self.intervals)  
 n = len(self.seq)  
 stats = sum(((obs\_freq[i] - n \* exp\_freq[i]) \*\* 2) / n \* exp\_freq[i] for i in range(k))  
 critical\_value = chi2.ppf(self.e, k)  
 print(f"Значение статистики хи квадрат: {stats}", f", критическое значение: {critical\_value}")  
 if stats < critical\_value:  
 print(f"Гипотеза о равномерном распределении принимается с уровнем значимости eps = {self.e}")  
 else:  
 print(f"Гипотеза о равномерном распределении отклоняется с уровнем значимости eps = {self.e}")  
  
 def ks\_test(self):  
 print("Критерий согласия Колмогорова: ")  
 stats = kstest(self.seq, 'uniform')  
 print(stats)  
 if stats[1] >= self.e:  
 print(f"Гипотеза о равномерном распределении принимается с уровнем значимости eps = {self.e}")  
 else:  
 print(f"Гипотеза о равномерном распределении отклоняется с уровнем значимости eps = {self.e}")  
  
 def histogram(self):  
 plt.hist(self.seq, bins=len(self.intervals))  
 plt.xticks(self.intervals, labels=np.round(self.intervals, 2))  
 plt.title(f"Гистограмма: {self.name}")  
 plt.show()  
  
 def scatter\_plot(self):  
 plt.plot(np.arange(len(self.seq)), self.seq, 'ro')  
 plt.yticks(self.intervals, labels=np.round(self.intervals, 2))  
 plt.title(f"Диаграмма рассеяния: {self.name}")  
 plt.show()  
  
 def standart\_tests(self):  
 self.ks\_test()  
 self.chisquare\_test()  
 self.histogram()  
 self.scatter\_plot()  
  
a1 = 78125  
*#a2 = 12167*a2 = 32771  
*#a2 = 29537*n = 1000  
k = 256  
m = 2 \*\* 31  
mcg1 = MultiplicativeCongruentialGenerator(n, a1, a1, m)  
mcg1.generate\_sequence()  
mcg1.print\_seq()  
mcg2 = MultiplicativeCongruentialGenerator(n + k, a2, a2, m)  
mcg2.generate\_sequence()  
mmg = MacLarenMarsagliaGenerator(1000, 256)  
mmg.maclaren\_marsaglia\_method(mcg1.seq, mcg2.seq)  
mmg.print\_seq()  
print()  
eps = 0.05  
print("Тесты для реализации мультипликативным конгруэнтным методом:")  
st\_test1 = StatTests(mcg1.seq, eps, "Мультипикативный конгруэнтный метод")  
st\_test1.ks\_test()  
st\_test1.chisquare\_test()  
st\_test1.histogram()  
st\_test1.scatter\_plot()  
print()  
print("Тесты для реализации методом МакЛарена Марсальи:")  
st\_test2 = StatTests(mmg.alpha, eps, "Метод МакЛарена Марсальи")  
st\_test2.ks\_test()  
st\_test2.chisquare\_test()  
st\_test2.histogram()  
st\_test2.scatter\_plot()

## 1.4 Результаты

Реализация БСВ мультипликативным конгруэнтным методом при beta = 78125, alpha0 = 78125, m = 2147483648:

[8.42170943e-01 6.04925031e-01 7.68070944e-01 5.42509316e-01 … 9.03877759e-01 8.83262509e-01 5.77189208e-01 9.50108883e-01]

Тесты для реализации мультипликативным конгруэнтным методом:

Критерий согласия Колмогорова:

KstestResult(statistic=0.021834087517112494, pvalue=0.7184791877010861, statistic\_location=0.1971659124828875, statistic\_sign=1)

Гипотеза о равномерном распределении принимается с уровнем значимости eps = 0.05

Критерий хи-квадрат:

Значение статистики хи квадрат: 0.08569543206646724 , критическое значение: 3.9402991361190605

Гипотеза о равномерном распределении принимается с уровнем значимости eps = 0.05

Тесты для реализации методом МакЛарена Марсальи:

Критерий согласия Колмогорова:

KstestResult(statistic=0.03416052658483387, pvalue=0.18929534496033917, statistic\_location=0.41683947341516614, statistic\_sign=1)

Гипотеза о равномерном распределении принимается с уровнем значимости eps = 0.05

Критерий хи-квадрат:

Значение статистики хи квадрат: 0.14069317559868239 , критическое значение: 3.9402991361190605

Гипотеза о равномерном распределении принимается с уровнем значимости eps = 0.05

|  |
| --- |
| Рисунок 1.1 – Гистограмма выборки, полученной с помощью мультипликативного конгруэнтного метода |

|  |
| --- |
| Рисунок 1.2 – Диаграмма рассеяния выборки, полученной с помощью мультипликативного конгруэнтного метода |

|  |
| --- |
| Рисунок 1.3 – Гистограмма выборки, полученной с помощью метода МакЛарена Марсальи |

|  |
| --- |
| Рисунок 1.4 – Диаграмма рассеяния выборки, полученной с помощью метода МакЛарена Марсальи |

# ЛАБОРАТОРНАЯ РАБОТА 2

## 2.1 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 использовать χ2-критерий Пирсона с уровнем значимость ε=0.05. Проверить, что вероятность ошибки I рода стремится к 0.05.

Параметры варианта:

Бернулли – Bi(1, *p*), *p* = 0.2; Геометрическое – G(*p*), p = 0.6;

## 2.2 Теоретические сведения

Для моделирования ДСВ из заданных распределений будем использовать специальные алгоритмы, а не общие.

Моделирование ДСВ из распределения Бернулли

ДСВ имеет распределение Бернулли Bi(1, *p*), если

, где - параметр распределения.

Алгоритм моделирования состоит из двух шагов:

1. Моделирование реализации БСВ α.
2. Принятие решения о том, что реализацией является значение

x, определяемое по правилу:

Коэффициент использования БСВ k = 1.

Моделирование ДСВ из геометрического распределения

ДСВ имеет геометрическое распределение G(p), если:

, где – параметр распределения.

Алгоритм моделирования состоит из двух шагов:

1. Моделирование реализации α БСВ.
2. Принятие решения о том, что реализацией является значение x, определяемое соотношением:

здесь [z] означает округление числа z в большую сторону до ближайшего целого значения.

Коэффициент использования БСВ k = 1.

Изменения реализации критерия Пирсона

Несколько изменим реализацию критерия для дискретного случая. В качестве ожидаемых вероятностей будем использовать вероятности получить то или иное значение, этими вероятностями задается само дискретное распределение (таблица). Для получения наблюдаемых вероятностей не буде делить область значений на интервалы, а будем считать количество появлений каждого из возможных значений. Для проверки того, что ошибка 1 рода стремится к 0,05, запустим каждый алгоритм 100 раз и применим к каждой выборке критерий Пирсона. Вероятность ошибки 1 рода вычислим как количество раз, когда гипотеза была отклонена, разделенное на 100.

Для реализации БСВ будем использовать мультипликативный конгруэнтный метод.

## 2.3 Листинг программы

MultiplicativeCongruentalGenerator.py

class MultiplicativeCongruentialMethod:  
 def \_\_init\_\_(self, a, b, m):  
 self.prev\_el = a  
 self.beta = b  
 self.m = m  
  
 def next\_elem(self):  
 z = self.beta \* self.prev\_el  
 self.prev\_el = z - self.m \* int(z / self.m)  
 return self.prev\_el / self.m

ChiSquareTest.py

import math  
import numpy as np  
from scipy.stats import chi2  
  
  
class DiscreteChiSquareTest:  
 def \_\_init\_\_(self, seq, e, dist, p):  
 self.seq = seq  
 self.e = e  
 self.distribution = dist  
 self.p = p  
 self.values = self.find\_values()  
  
 def find\_values(self):  
 max\_el = np.max(self.seq)  
 min\_el = np.min(self.seq)  
 return np.arange(min\_el, max\_el + 1)  
  
 def observed\_frequencies(self):  
 n = len(self.values)  
 freq = np.zeros(n)  
 sort\_seq = np.sort(self.seq)  
 i = 0  
 k = 0  
 while i < n:  
 while k < len(self.seq) and sort\_seq[k] == self.values[i]:  
 freq[i] += 1  
 k += 1  
 i += 1  
 return freq  
  
 def expected\_frequencies(self):  
 if self.distribution == "Bi":  
 return [(1 - self.p), self.p]  
 if self.distribution == "G":  
 return [self.p \* ((1 - self.p) \*\* (elem - 1)) for elem in self.values]  
  
 def chisquare\_test(self, print\_res):  
 obs\_freq = self.observed\_frequencies()  
 exp\_freq = self.expected\_frequencies()  
 k = len(self.values)  
 n = len(self.seq)  
 stats = sum(((obs\_freq[i] - n \* exp\_freq[i]) \*\* 2) / n \* exp\_freq[i] for i in range(k))  
 if self.distribution == "Bi":  
 k += 2  
 if self.distribution == "G":  
 obs\_freq[-3] += obs\_freq[-2] + obs\_freq[-1]  
 exp\_freq[-3] += exp\_freq[-2] + exp\_freq[-1]  
 k -= 3  
 critical\_value = chi2.ppf(self.e, k)  
 res = 1 if stats < critical\_value else 0  
 if print\_res:  
 print("Критерий хи-квадрат: ")  
 print(f"exp freq {exp\_freq}")  
 print(f"obs freq {obs\_freq}")  
 print(f"Значение статистики хи квадрат: {stats}", f", критическое значение: {critical\_value}")  
 if res == 1:  
 print(f"Гипотеза о распределении {self.distribution}({self.p}) принимается с уровнем значимости eps = {self.e}")  
 else:  
 print(f"Гипотеза о распределении {self.distribution}({self.p}) отклоняется с уровнем значимости eps = {self.e}")  
 return res

DiscreteRandomVariable.py

import math  
import numpy as np  
import MultiplicativeCongruentalGenerator as mcg  
import ChiSquareTest as Chi2Test  
import matplotlib.pyplot as plt  
  
  
class BernoulliDRV:  
 def \_\_init\_\_(self, p, mcs):  
 self.p = p  
 self.mcs = mcs  
  
 def generate\_drv(self):  
 a = self.mcs.next\_elem()  
 return 1 if a <= self.p else 0  
  
  
class GeometricDRV:  
 def \_\_init\_\_(self, p, mcs):  
 self.q = 1 - p  
 self.mcs = mcs  
  
 def generate\_drv(self):  
 a = self.mcs.next\_elem()  
 return math.ceil(math.log(a) / math.log(self.q))  
  
  
class DRVSequence:  
 def \_\_init\_\_(self, n, dist, p, mcm):  
 self.n = n  
 self.seq = np.zeros(n)  
 self.distribution = dist  
 self.p = p  
 self.mcs = mcm  
  
 def generate\_sequence(self):  
 if self.distribution == "Bi":  
 generator = BernoulliDRV(self.p, self.mcs)  
 self.seq = [generator.generate\_drv() for elem in self.seq]  
 if self.distribution == "G":  
 generator = GeometricDRV(self.p, self.mcs)  
 self.seq = [generator.generate\_drv() for elem in self.seq]  
  
 def exp\_mean(self):  
 if self.distribution == "G":  
 return 1 / self.p  
 if self.distribution == "Bi":  
 return self.p  
  
 def exp\_variance(self):  
 if self.distribution == "G":  
 return (1 - self.p) / (self.p \*\* 2)  
 if self.distribution == "Bi":  
 return self.p \* (1 - self.p)  
  
 def est\_mean(self):  
 return sum(self.seq[i] for i in range(self.n)) / self.n  
  
 def est\_variance(self):  
 mean = self.est\_mean()  
 return sum((self.seq[i] - mean) \*\* 2 for i in range(self.n)) / (self.n + 1)  
  
 def print\_seq(self):  
 print(f"Сгенерированная последовательность из {self.distribution}({self.p})")  
 print(\*self.seq)  
 print("Среднее: ")  
 print(f"Несмещенная оценка: {self.est\_mean()}, истинное значение: {self.exp\_mean()}, "  
 f"разность: {abs(self.exp\_mean() - self.est\_mean())}")  
 print("Дисперсия: ")  
 print(f"Несмещенная оценка: {self.est\_variance()}, истинное значение: {self.exp\_variance()},"  
 f" разность: {abs(self.exp\_variance() - self.est\_variance())}")  
  
 def histogram(self):  
 plt.hist(self.seq)  
 max\_el = np.max(self.seq)  
 min\_el = np.min(self.seq)  
 values = np.arange(min\_el, max\_el + 1)  
 plt.xticks(values, labels=values)  
 plt.title(f"Гистограмма для ДСВ из {self.distribution}")  
 plt.show()  
  
 def scatter\_plot(self):  
 plt.plot(np.arange(len(self.seq)), self.seq, 'ro')  
 plt.title(f"Диаграмма рассеяния для ДСВ из {self.distribution}")  
 plt.show()  
  
  
def type1error(bi\_p, g\_p, num, e, n2):  
 bi\_res = 0  
 g\_res = 0  
 mgen = mcg.MultiplicativeCongruentialMethod(75465, 75465, 2 \*\* 31)  
  
 for \_ in range(n2):  
 g = DRVSequence(num, "G", g\_p, mgen)  
 g.generate\_sequence()  
 g\_test = Chi2Test.DiscreteChiSquareTest(g.seq, e, "G", g\_p)  
 g\_res += g\_test.chisquare\_test(False)  
  
 bi = DRVSequence(num, "Bi", bi\_p, mgen)  
 bi.generate\_sequence()  
 bi\_test = Chi2Test.DiscreteChiSquareTest(bi.seq, e, "Bi", bi\_p)  
 bi\_res += bi\_test.chisquare\_test(False)  
  
 bi\_error = 1 - (bi\_res / n2)  
 g\_error = 1 - (g\_res / n2)  
 print(f"Вероятность ошибки первого рода для ДСВ из Bi({bi\_p}): {bi\_error}")  
 print(f"Вероятность ошибки первого рода для ДСВ из G({g\_p}): {g\_error}")  
  
  
n = 1000  
beta = 78121  
m = 2 \*\* 31  
mcm = mcg.MultiplicativeCongruentialMethod(beta, beta, m)  
eps = 0.05  
bi\_seq = DRVSequence(n, "Bi", 0.2, mcm)  
bi\_seq.generate\_sequence()  
bi\_seq.print\_seq()  
bi\_seq.histogram()  
bi\_seq.scatter\_plot()  
chi2bi = Chi2Test.DiscreteChiSquareTest(bi\_seq.seq, eps, "Bi", 0.2)  
chi2bi.chisquare\_test(True)  
print()  
g\_seq = DRVSequence(n, "G", 0.6, mcm)  
g\_seq.generate\_sequence()  
g\_seq.print\_seq()  
g\_seq.histogram()  
g\_seq.scatter\_plot()  
chi2g = Chi2Test.DiscreteChiSquareTest(g\_seq.seq, eps, "G", 0.6)  
chi2g.chisquare\_test(True)  
print()  
type1error(0.2, 0.6, n, eps, 100)

## 2.4 Результаты

Сгенерированная последовательность из Bi(0.2)

0 0 0 1 0 0 1 0 0 1 0 0 0 0 … 1 0 1 0 0 0 0 0 0 0 0 0 0

Среднее:

Несмещенная оценка: 0.204, истинное значение: 0.2, разность: 0.003999999999999976

Дисперсия:

Несмещенная оценка: 0.16222177822178022, истинное значение: 0.16000000000000003, разность: 0.002221778221780185

Критерий хи-квадрат:

Значение статистики хи квадрат: 0.016, критическое значение: 0.7107230213973239

Гипотеза о распределении Bi(0.2) принимается с уровнем значимости eps = 0.05

Сгенерированная последовательность из G(0.6)

1 4 2 1 1 1 1 2 2 1 1 1 4 1 3 … 3 2 1 2 1 1 1 1 1

Среднее:

Несмещенная оценка: 1.649, истинное значение: 1.6666666666666667, разность: 0.01766666666666672

Дисперсия:

Несмещенная оценка: 1.0347642357642333, истинное значение: 1.1111111111111112, разность: 0.0763468753468779

Критерий хи-квадрат:

Значение статистики хи квадрат: 0.13015391892543898 , критическое значение: 0.7107230213973239

Гипотеза о распределении G(0.6) принимается с уровнем значимости eps = 0.05

Вероятность ошибки первого рода для ДСВ из Bi(0.2): 0.030000000000000027

Вероятность ошибки первого рода для ДСВ из G(0.6): 0.010000000000000009

|  |
| --- |
| Рисунок 2.1 – Гистограмма для выборки из распределения Бернулли |

|  |
| --- |
| Рисунок 2.2 – Диаграмма рассеяния для выборки из распределения Бернулли |

|  |
| --- |
| Рисунок 2.3 – Гистограмма для выборки из геометрического распределения |

|  |
| --- |
| Рисунок 2.4 – Диаграмма рассеяния для выборки из геометрического распределения |

# ЛАБОРАТОРНАЯ РАБОТА 3

## 3.1 Условие

Смоделировать непрерывную случайную величину (задания на стр. 257). 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.

Параметры варианта:

*m* = 0, *s*2 = 1; Логистическое *LG*(*a*,*b*), *a* = 2, *b* = 3; Лапласа *L*(*a*), *a* = 2.

## 3.2 Теоретические сведения

Одним из самых распространенных методов для моделирования НСВ является метод обратной функции. Этот же метод используется для моделирования заданных распределений.

Моделирование СВ из стандартного нормального распределения

НСВ с плотностью распределения

имеет стандартное нормальное распределение .

Функция распределения закона называется функцией Лапласа и имеет вид:

Алгоритм моделирования основан на ЦПТ:

1. Моделируется N = 12 реализаци1 БСВ.
2. Вычисляется значение x реализации по формуле:

Коэффициент использования БСВ k = 1/12.

Моделирование СВ из логистического распределения

НСВ с плотностью распределения

имеет логистическое распределение LG(a, b).

Функция распределения закона LG(a, b) имеет вид:

Алгоритм моделирования:

1. Моделируется реализация y БСВ.
2. Вычисляется значение x реализации по формуле:

Коэффициент использования БСВ k = 1.

Моделирование СВ из распределения Лапласа

НСВ с плотностью распределения

имеет логистическое распределение L(a).

Функция распределения закона L(a) имеет вид:

Алгоритм моделирования:

1. Моделируется реализация y БСВ.
2. Вычисляется значение x реализации по формуле:

Коэффициент использования БСВ k = 1.

Используем реализации критерия и критерия Колмогорова из лабораторной работы 1. Для проверки того, что ошибка 1 рода стремится к 0,05, запустим каждый алгоритм 1000 раз и применим к каждой выборке критерий Пирсона и критерий Колмогорова. Вероятность ошибки 1 рода вычислим как количество раз, когда гипотеза была отклонена, разделенное на 1000.

Для реализации БСВ будем использовать мультипликативный конгруэнтный метод.

## 3.3 Листинг программы

MultiplicativeCongruentalGenerator.py

class MultiplicativeCongruentialMethod:  
 def \_\_init\_\_(self, a, b, m):  
 self.prev\_el = a  
 self.beta = b  
 self.m = m  
  
 def next\_elem(self):  
 z = self.beta \* self.prev\_el  
 self.prev\_el = z - self.m \* int(z / self.m)  
 return self.prev\_el / self.m

Tests.py

import math  
import numpy as np  
from scipy.stats import chi2, chisquare, kstest  
  
  
class Tests:  
 def \_\_init\_\_(self, e, cdf, distribution):  
 self.e = e  
 self.cdf = cdf  
 self.distribution = distribution  
  
 @staticmethod  
 def find\_intervals(self, seq):  
 n = 1 + int(math.log2(len(seq)))  
 max\_el = np.max(seq)  
 min\_el = np.min(seq)  
 h = (max\_el - min\_el) / n  
 a = np.zeros(n)  
 for i in range(n - 1):  
 a[i] = min\_el + (i + 1) \* h  
 a[-1] = np.Inf  
 return a  
  
 @staticmethod  
 def observed\_frequencies(self, seq, intervals):  
 n = len(intervals)  
 freq = np.zeros(n)  
 sort\_seq = np.sort(seq)  
 i = 0  
 k = 0  
 while i < n:  
 while k < len(seq) and sort\_seq[k] < intervals[i]:  
 freq[i] += 1  
 k += 1  
 i += 1  
 return freq  
  
 def expected\_frequencies(self, intervals, l):  
 n = len(intervals)  
 exp\_freq = np.zeros(n)  
 exp\_freq[0] = l \* self.cdf(intervals[0])  
 for i in range(1, n):  
 exp\_freq[i] = l \* (self.cdf(intervals[i]) - self.cdf(intervals[i - 1]))  
 return exp\_freq  
  
 def chisquare\_test(self, seq, print\_res):  
 intervals = self.find\_intervals(self, seq)  
 obs\_freq = self.observed\_frequencies(self, seq, intervals)  
 exp\_freq = self.expected\_frequencies(intervals, len(seq))  
 k = len(intervals)  
 n = len(seq)  
 stats = chisquare(obs\_freq, exp\_freq)  
 critical\_value = chi2.ppf(1 - self.e, k + 2)  
 res = 1 if stats.statistic < critical\_value else 0  
 if print\_res:  
 print("Критерий хи-квадрат: ")  
 print(f"Значение статистики хи квадрат: {stats.statistic}", f", критическое значение:", chi2.ppf(1 - self.e, k - 1))  
 if res == 1:  
 print(f"Гипотеза о распределении {self.distribution} принимается с уровнем значимости eps = {self.e}")  
 else:  
 print(f"Гипотеза о распределении {self.distribution} отклоняется с уровнем значимости eps = {self.e}")  
 return res  
  
 def ks\_test(self, seq, print\_res):  
 stats = kstest(seq, self.cdf)  
 res = 1 if stats[1] >= self.e else 0  
 if print\_res:  
 print("Критерий согласия Колмогорова: ")  
 print(stats)  
 if res == 1:  
 print(f"Гипотеза о распределении {self.distribution} принимается с уровнем значимости eps = {self.e}")  
 else:  
 print(f"Гипотеза о распределении {self.distribution} отклоняется с уровнем значимости eps = {self.e}")  
 return res

Continuous Random Variables Generator.py

import MultiplicativeCongruentalGenerator as mcg  
import Tests  
import math  
import matplotlib.pyplot as plt  
import numpy as np  
from scipy.stats import norm, logistic, laplace  
  
  
class NormalVariableGenerator:  
 def \_\_init\_\_(self, mean, sigma, num, gen):  
 self.mean = mean  
 self.sigma = sigma  
 self.num = num  
 self.gen = gen  
  
 def next\_element(self):  
 norm\_var = sum(self.gen.next\_elem() for \_ in range(self.num))  
 return (norm\_var - 6) \* self.sigma + self.mean  
  
 def exp\_mean(self):  
 return self.mean  
  
 def exp\_variance(self):  
 return self.sigma \*\* 2  
  
 def cdf(self, x):  
 return norm.cdf(x, loc=self.mean, scale=self.sigma)  
  
  
class LogisticVariableGenerator:  
 def \_\_init\_\_(self, mu, k, gen):  
 self.mu = mu  
 self.k = k  
 self.gen = gen  
  
 def next\_element(self):  
 y = self.gen.next\_elem()  
 return self.mu + self.k \* math.log(y / (1 - y))  
  
 def exp\_mean(self):  
 return self.mu  
  
 def exp\_variance(self):  
 return ((self.k \* math.pi) \*\* 2) / 3  
  
 def cdf(self, x):  
 return logistic.cdf(x, loc=self.mu, scale=self.k)  
  
  
class LaplaceVariableGenerator:  
 def \_\_init\_\_(self, p, gen):  
 self.p = p  
 self.gen = gen  
  
 def next\_element(self):  
 y = self.gen.next\_elem()  
 return math.log(2 \* y) / self.p if y < 0.5 else - math.log(2 - 2 \* y) / self.p  
  
 @staticmethod  
 def exp\_mean():  
 return 0  
  
 def exp\_variance(self):  
 return 2 / (self.p \*\* 2)  
  
 def cdf(self, x):  
 return laplace.cdf(x, loc=0, scale=1/self.p)  
  
  
class CRVSequence:  
 def \_\_init\_\_(self, generator, seq\_len, test\_times, distribution, e):  
 self.generator = generator  
 self.seq\_len = seq\_len  
 self.test\_times = test\_times  
 self.e = e  
 self.cdf = generator.cdf  
 self.distribution = distribution  
 self.seq = self.generate\_sequence()  
  
 def generate\_sequence(self):  
 return [self.generator.next\_element() for \_ in range(self.seq\_len)]  
  
 def est\_mean(self):  
 return sum(self.seq[i] for i in range(self.seq\_len)) / self.seq\_len  
  
 def est\_variance(self):  
 mean = self.est\_mean()  
 return sum((self.seq[i] - mean) \*\* 2 for i in range(self.seq\_len)) / (self.seq\_len + 1)  
  
 def tests(self):  
 tests = Tests.Tests(self.e, self.cdf, self.distribution)  
 tests.chisquare\_test(self.seq, True)  
 tests.ks\_test(self.seq, True)  
 self.type1error(tests)  
  
 def type1error(self, tests):  
 xi\_res = 0  
 k\_res = 0  
 for \_ in range(self.test\_times):  
 temp = self.generate\_sequence()  
 xi\_res += tests.chisquare\_test(temp, False)  
 k\_res += tests.ks\_test(temp, False)  
  
 xi\_error = 1 - (xi\_res / self.test\_times)  
 k\_error = 1 - (k\_res / self.test\_times)  
 print(f"Вероятность ошибки первого рода для СВ из {self.distribution}, критерий хи-квадрат: {xi\_error}")  
 print(f"Вероятность ошибки первого рода для СВ из {self.distribution}, критерий Колмогорова: {k\_error}")  
  
 def histogram(self):  
 plt.hist(self.seq)  
 max\_el = np.max(self.seq)  
 min\_el = np.min(self.seq)  
 values = np.arange(min\_el, max\_el + 1)  
 plt.xticks(values, labels=labels=np.round(values))  
 plt.title(f"Гистограмма для СВ из {self.distribution}")  
 plt.show()  
  
 def scatter\_plot(self):  
 plt.plot(np.arange(len(self.seq)), self.seq, 'ro')  
 plt.title(f"Диаграмма рассеяния для СВ из {self.distribution}")  
 plt.show()  
  
 def print\_results(self):  
 print(f"Сгенерированная последовательность из {self.distribution}:")  
 print(\*self.seq)  
 print("Среднее: ")  
 print(f"Несмещенная оценка: {self.est\_mean()}, истинное значение: {self.generator.exp\_mean()}, "  
 f"разность: {abs(self.generator.exp\_mean() - self.est\_mean())}")  
 print("Дисперсия: ")  
 print(f"Несмещенная оценка: {self.est\_variance()}, истинное значение: {self.generator.exp\_variance()},"  
 f" разность: {abs(self.generator.exp\_variance() - self.est\_variance())}")  
  
 self.histogram()  
 self.scatter\_plot()  
 self.tests()  
  
  
n = 1000  
times = 1000  
beta = 78121  
beta3 = 78117  
m = 2 \*\* 31  
eps = 0.05  
gen = mcg.MultiplicativeCongruentialMethod(beta, beta, m)  
gen3 = mcg.MultiplicativeCongruentialMethod(beta3, beta3, m)  
mean = 0  
sigma = 1  
norm\_gen = NormalVariableGenerator(mean, sigma, 12, gen)  
norm\_seq = CRVSequence(norm\_gen, n, times, f"N({mean}, {sigma\*\*2})", eps)  
norm\_seq.print\_results()  
print()  
a = 2  
laplace\_gen = LaplaceVariableGenerator(a, gen)  
laplace\_seq = CRVSequence(laplace\_gen, n, times, f"L({a})", eps)  
laplace\_seq.print\_results()  
print()  
mu = 2  
k = 3  
lg\_gen = LogisticVariableGenerator(mu, k, gen3)  
lg\_seq = CRVSequence(lg\_gen, n, times, f"LG({mu}, {k})", eps)  
lg\_seq.print\_results()

## 3.4 Результаты

Сгенерированная последовательность из N(0, 1):

-0.06461082585155964 -0.6113977897912264 -0.28605531342327595 … 0.34055538289248943 -0.9830965865403414 0.536145044490695

Среднее:

Несмещенная оценка: -0.0058087344914674755, истинное значение: 0, разность: 0.0058087344914674755

Дисперсия:

Несмещенная оценка: 0.9868882045735924, истинное значение: 1, разность: 0.013111795426407613

Критерий хи-квадрат:

Значение статистики хи квадрат: 3.04929942218059 , критическое значение: 16.918977604620448

Гипотеза о распределении N(0, 1) принимается с уровнем значимости eps = 0.05

Критерий согласия Колмогорова:

KstestResult(statistic=0.030100227153228742, pvalue=0.31885245843308174, statistic\_location=-0.04036824591457844, statistic\_sign=1)

Гипотеза о распределении N(0, 1) принимается с уровнем значимости eps = 0.05

Вероятность ошибки первого рода для СВ из N(0, 1), критерий хи-квадрат: 0.016000000000000014

Вероятность ошибки первого рода для СВ из N(0, 1), критерий Колмогорова: 0.049000000000000044

Сгенерированная последовательность из L(2):

-1.0144969822242897 -0.47153697694455315 -0.276128128621111 -0.5744549885940434 … -0.021010941572200736 -0.7415647995844128 0.6299212607336467

Среднее:

Несмещенная оценка: -0.013180897094282376, истинное значение: 0, разность: 0.013180897094282376

Дисперсия:

Несмещенная оценка: 0.47488812887599713, истинное значение: 0.5, разность: 0.02511187112400287

Критерий хи-квадрат:

Значение статистики хи квадрат: 7.411225885631511 , критическое значение: 16.918977604620448

Гипотеза о распределении L(2) принимается с уровнем значимости eps = 0.05

Критерий согласия Колмогорова:

KstestResult(statistic=0.027570735435932914, pvalue=0.4251309968644753, statistic\_location=0.016705247749752017, statistic\_sign=1)

Гипотеза о распределении L(2) принимается с уровнем значимости eps = 0.05

Вероятность ошибки первого рода для СВ из L(2), критерий хи-квадрат: 0.051000000000000045

Вероятность ошибки первого рода для СВ из L(2), критерий Колмогорова: 0.05600000000000005

Сгенерированная последовательность из LG(2, 3):

7.010294215396152 0.7811929720528834 4.977906539742194 …

-4.4722746972882685 -17.828794650519704 -4.23734540411138

Среднее:

Несмещенная оценка: 2.0595419107403066, истинное значение: 2, разность: 0.05954191074030657

Дисперсия:

Несмещенная оценка: 29.268206090590613, истинное значение: 29.608813203268074, разность: 0.3406071126774606

Критерий хи-квадрат:

Значение статистики хи квадрат: 2.9282037723124033 , критическое значение: 16.918977604620448

Гипотеза о распределении LG(2, 3) принимается с уровнем значимости eps = 0.05

Критерий согласия Колмогорова:

KstestResult(statistic=0.01953584844246506, pvalue=0.8325180402308459, statistic\_location=-2.457741151409933, statistic\_sign=-1)

Гипотеза о распределении LG(2, 3) принимается с уровнем значимости eps = 0.05

Вероятность ошибки первого рода для СВ из LG(2, 3), критерий хи-квадрат: 0.05500000000000005

Вероятность ошибки первого рода для СВ из LG(2, 3), критерий Колмогорова: 0.04300000000000004

|  |
| --- |
| Рисунок 3.1 – Гистограмма для выборки из стандартного нормального распределения |

|  |
| --- |
| Рисунок 3.2 – Диаграмма рассеяния для выборки из стандартного нормального распределения |

|  |
| --- |
| Рисунок 3.3 – Гистограмма для выборки из распределения Лапласа |

|  |
| --- |
| Рисунок 3.4 – Диаграмма рассеяния для выборки из распределения Лапласа |

|  |
| --- |
| Рисунок 3.5 – Гистограмма для выборки из логистического распределения |

|  |
| --- |
| Рисунок 3.6 – Диаграмма рассеяния для выборки из логистического распределения |

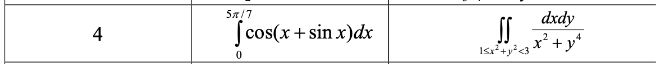
# ЛАБОРАТОРНАЯ РАБОТА 4

## 4.1 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*.

Параметры варианта



## 4.2 Теоретические сведения

Метод Монте-Карло для вычисления интегралов

Рассмотрим задачу приближенного вычисления интеграла

Где - подмножество из . При n = 1 имеем определенный интеграл вида

Схема вычислений как многомерных, так и одномерных интегралов абсолютно аналогична.

Пусть η – произвольная случайная величина с плотностью распределения вероятностей

Предполагается только, что существуют моменты случайных величин, встречающиеся ниже. Рассмотрим случайную величину, являющуюся функциональным преобразованием случайной величины η:

Можно показать, что . Поэтому в качестве приближенного значения интеграла можно использовать статистическую оценку , построенную в выборке из n независимых случайных величин :

В качестве случайной величины возьмем СВ, равномерно распределенную на интервале . Плотность распределения такой СВ .

Для вычисления двумерного интеграла будем действовать несколько иначе. Т.к. область интегрирования – кольцо, ограничим эту область квадратом со сторонами, равными внешнему диаметру кольца b = 3. В качестве будем использовать две СВ из равномерного распределения на интервале (по факту, это двумерный случайный вектор, равномерно распределенный в квадрате). Если точка с координатами попадает в кольцо, то к сумме добавляем значение . Приближенное вычисление интеграла будет равно данной сумме, умноженной на площадь квадрата и деленной на n.

Для генерирования СВ из равномерного распределения будем использовать встроенную функцию Python. Для построения графика зависимости отклонения полученного значения от «точного» возьмем значения n от 1000 до 10000 с шагом 10. Для двумерного интеграла также построим график попадания смоделированных в область интегрирования.

В качестве точных значений обоих интегралов взяты приближенные значения, найденные с помощью Wolfram Mathematica.

|  |
| --- |
| Рисунок 4.1 – Вычисление приближенных значений интегралов, взятых в качестве точных |

## 4.3 Листинг программы

from random import uniform  
import matplotlib.pyplot as plt  
import numpy as np  
import math  
  
  
class MonteCarloIntegration:  
 def \_\_init\_\_(self, function, a, b, integral\_value, dim):  
 self.function = function  
 self.a = a  
 self.b = b  
 self.integral\_value = integral\_value  
 self.dim = dim  
 self.in\_area\_x = []  
 self.out\_of\_area\_x = []  
 self.in\_area\_y = []  
 self.out\_of\_area\_y = []  
  
 def uniform\_pdf(self, x):  
 return 1 / (self.b - self.a)  
  
 def integrate\_x(self, n):  
 ksi\_sum = 0  
  
 for \_ in range(n):  
 eta = uniform(self.a, self.b)  
 ksi\_val = self.function(eta) / self.uniform\_pdf(eta)  
 ksi\_sum += ksi\_val  
 approx\_integral\_value = ksi\_sum / n  
 return approx\_integral\_value  
  
 def integrate\_xy(self, n):  
 sum = 0  
 for \_ in range(n):  
 x = uniform(-math.sqrt(self.b), math.sqrt(self.b))  
 y = uniform(-math.sqrt(self.b), math.sqrt(self.b))  
 if self.a <= x \*\* 2 + y \*\* 2 < self.b:  
 sum += self.function(x, y)  
 if n == 10000:  
 self.in\_area\_x.append(x)  
 self.in\_area\_y.append(y)  
 elif n == 10000:  
 self.out\_of\_area\_x.append(x)  
 self.out\_of\_area\_y.append(y)  
 area = (2 \* math.sqrt(self.b)) \*\* 2  
 return area \* sum / n  
  
 def plot\_integral\_value(self):  
 n = np.arange(1000, 10001, step=10)  
 dif = 0  
 if self.dim == 1:  
 dif = [self.integrate\_x(i) - self.integral\_value for i in n]  
 plt.title("График отклонения от точного значения для интеграла 1")  
 if self.dim == 2:  
 dif = [self.integrate\_xy(i) - self.integral\_value for i in n]  
 plt.title("График отклонения от точного значения для интеграла 2")  
 print(f"Точное значение: {self.integral\_value}")  
 print(f"Найденное значение при n = 1000: {dif[0] + self.integral\_value}") print(f"Найденное значение при n = 10000: {dif[-10] + self.integral\_value}")plt.plot(n, dif)  
 plt.show()  
 if self.dim == 2:  
 plt.plot(self.in\_area\_x, self.in\_area\_y, 'o', color='olive')  
 plt.plot(self.out\_of\_area\_x, self.out\_of\_area\_y, 'o', color='gold')  
 plt.title("График попадания сгенерированных СВ в область")  
 plt.show()  
  
  
def func1(x):  
 return math.cos(x + math.sin(x))  
  
  
def func2(x, y):  
 return 1 / (x \*\* 2 + y \*\* 4)  
  
  
obj1 = MonteCarloIntegration(func1, 0, (5 \* math.pi) / 7, -0.485736, 1)  
obj1.plot\_integral\_value()  
obj2 = MonteCarloIntegration(func2, 1, 3, 3.21825, 2)  
obj2.plot\_integral\_value()

## 4.4 Результаты

Одномерный интеграл:

Точное значение: -0.485736

Найденное значение при n = 1000: -0.47534780687675343

Найденное значение при n = 10000: -0.488075912618409

Двумерный интеграл:

Точное значение: 3.21825

Найденное значение при n = 1000: 3.2452610877111807

Найденное значение при n = 10000: 3.2140494151158046

|  |
| --- |
| Рисунок 4.2 – График зависимости отклонения найденного значения от точного для одномерного интеграла |

|  |
| --- |
| Рисунок 4.3 – График зависимости отклонения найденного значения от точного для двумерного интеграла |

|  |
| --- |
| Рисунок 4.4 – График попадания сгенерированных в область интегрирования |

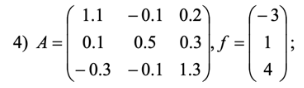
# ЛАБОРАТОРНАЯ РАБОТА 5

## 5.1 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.
3. Построить график зависимости точности решения от длины цепи Маркова и числа смоделированных цепей Маркова.

Параметры варианта



## 5.2 Теоретические сведения

Пусть система алгебраических уравнений задана в виде

Где – вектор-столбец неизвестных, – вектор правых частей и – матрица системы.

Для применения метода Монте-Карло все собственные значения матрицы по модулю должны быть меньше 1.

Определим некоторый вектор h следующим образом:

, где 1 находится на позиции k для вычисления компоненты .

Моделируем вспомогательную цепь Маркова: , где

. n-вектор вероятностей начальных состояний цепи Маркова: . Матрица переходных вероятностей размерности n x n имеет вид:

Каждому состоянию цепи Маркова приписываем веса, которые вычисляются по формулам:

Строим СВ по формуле:

Где - номер реализации цепи Маркова. Тогда приближенное

решение вычисляется в зависимости от вектора h по формуле:

Система была приведена к требуемому виду. Точное решение найдено с помощью Wolfram Mathematica.

|  |
| --- |
| Рисунок 5.1 – Требуемый вид системы и нахождение точного решения |

Для построения графика зависимости отклонения решения от точного от длины цепи и количества цепей зададим количество цепей и длину цепи от 1000 до 10000 с шагом 1000 (чтобы вычисления были быстрее).

## 5.3 Листинг программы

from random import uniform  
import matplotlib.pyplot as plt  
import numpy as np  
import math  
  
  
class MarkovChainSimulation:  
 def \_\_init\_\_(self, n, pi, p):  
 self.n = n  
 self.pi = pi  
 self.p = p  
 self.c = np.arange(n)  
 *# вспомогательный вектор q* self.q\_v = [sum(pi[:j]) for j in range(n + 1)]  
 *# вспомогательная матрица Q* self.q\_m = [[sum(p[i][:j]) for j in range(n + 1)] for i in range(n)]  
  
 def simulation(self, mc\_len):  
 mc = np.zeros(mc\_len)  
 v0 = 0  
 alpha = uniform(0, 1)  
 for i in range(self.n):  
 if self.q\_v[i] <= alpha < self.q\_v[i + 1]:  
 mc[0] = int(self.c[i])  
 v0 = i  
 break  
 for k in range(1, mc\_len):  
 alpha = uniform(0, 1)  
 for i in range(self.n):  
 if self.q\_m[v0][i] <= alpha < self.q\_m[v0][i + 1]:  
 mc[k] = int(self.c[i])  
 v0 = i  
 break  
 return mc  
  
  
class MonteCarloSolving:  
 def \_\_init\_\_(self, dim, matrix, f\_vector, solution, mc\_gen):  
 self.n = dim  
 self.a = matrix  
 self.f = f\_vector  
 self.exp = solution  
 self.mc\_gen = mc\_gen  
  
 def calculate\_weights(self, mc, h, mc\_len):  
 q = np.zeros(mc\_len)  
 if self.mc\_gen.pi[mc[0]] > 0:  
 q[0] = h[mc[0]] / self.mc\_gen.pi[mc[0]]  
 else:  
 q[0] = 0  
  
 for i in range(1, mc\_len):  
 if self.mc\_gen.p[mc[i - 1]][mc[i]] > 0:  
 q[i] = q[i - 1] \* self.a[mc[i - 1]][mc[i]] / self.mc\_gen.p[mc[i - 1]][mc[i]]  
 else:  
 q[i] = 0  
 return q  
  
 def generate\_ksi(self, h, mc\_len):  
 mc = [int(elem) for elem in self.mc\_gen.simulation(mc\_len)]  
 q = self.calculate\_weights(mc, h, mc\_len)  
 ksi = sum(q[i] \* self.f[mc[i]] for i in range(mc\_len))  
 return ksi  
  
 def solve(self, l, mc\_len):  
 h = [1, 0, 0]  
 x = 0  
 for \_ in range(l):  
 x += self.generate\_ksi(h, mc\_len)  
 x /= l  
 h = [0, 1, 0]  
 y = 0  
 for \_ in range(l):  
 y += self.generate\_ksi(h, mc\_len)  
 y /= l  
 h = [0, 0, 1]  
 z = 0  
 for \_ in range(l):  
 z += self.generate\_ksi(h, mc\_len)  
 z /= l  
 f\_sol = [x, y, z]  
 rez = [self.exp[i] - f\_sol[i] for i in range(self.n)]  
 if l == 10000:  
 print(f'Приближенное решение: {f\_sol}')  
 print(f'Точное решение: {self.exp}')  
 print(f'Вектор невязки: {rez}')  
 return rez  
  
  
dim = 3  
pi = [1/3, 1/3, 1/3]  
p = [[1/3, 1/3, 1/3], [1/3, 1/3, 1/3], [1/3, 1/3, 1/3]]  
mc\_gen = MarkovChainSimulation(dim, pi, p)  
mc\_len = 1000  
matrix = [[-0.1, 0.1, -0.2], [-0.1, 0.5, -0.3], [0.3, 0.1, -0.3]]  
f = [-3, 1, 4]  
acc\_sol = [-3.07018, 1.14035, 2.45614]  
sol = MonteCarloSolving(dim, matrix, f, acc\_sol, mc\_gen)  
rez\_x = []  
rez\_y = []  
rez\_z = []  
l\_lims = np.arange(1000, 10001, step=1000)  
for l in l\_lims:  
 f\_rez = sol.solve(l, mc\_len)  
 rez\_x.append(f\_rez[0])  
 rez\_y.append(f\_rez[1])  
 rez\_z.append(f\_rez[2])  
  
plt.plot(l\_lims, rez\_x)  
plt.title('График зависимости невязки от L для x')  
plt.show()  
plt.plot(l\_lims, rez\_y)  
plt.title('График зависимости невязки от L для y')  
plt.show()  
plt.plot(l\_lims, rez\_z)  
plt.title('График зависимости невязки от L для z')  
plt.show()  
  
rez\_len\_x = []  
rez\_len\_y = []  
rez\_len\_z = []  
len\_lims = np.arange(1000, 10001, step=1000)  
for m\_len in len\_lims:  
 f\_rez = sol.solve(1000, m\_len)  
 rez\_len\_x.append(f\_rez[0])  
 rez\_len\_y.append(f\_rez[1])  
 rez\_len\_z.append(f\_rez[2])  
  
plt.plot(len\_lims, rez\_len\_x)  
plt.title('График зависимости невязки от длины цепи для x')  
plt.show()  
plt.plot(len\_lims, rez\_len\_y)  
plt.title('График зависимости невязки от длины цепи для y')  
plt.show()  
plt.plot(len\_lims, rez\_len\_z)  
plt.title('График зависимости невязки от длины цепи для z')  
plt.show()

## 5.4 Результаты

Приближенное решение: [-3.042418986235477, 1.0933566682442784, 2.387896016551205]

Точное решение: [-3.07018, 1.14035, 2.45614]

Вектор невязки: [-0.027761013764523224, 0.04699333175572162, 0.06824398344879512]

|  |
| --- |
| Рисунок 5.2 – График зависимости невязки от количества ЦМ для компоненты x |

|  |
| --- |
| Рисунок 5.3 – График зависимости невязки от количества ЦМ для компоненты y |

|  |
| --- |
| Рисунок 5.4 – График зависимости невязки от количества ЦМ для компоненты z |

|  |
| --- |
| Рисунок 5.5 – График зависимости невязки от длины ЦМ для компоненты x |

|  |
| --- |
| Рисунок 5.6 – График зависимости невязки от длины ЦМ для компоненты y |

|  |
| --- |
| Рисунок 5.7 – График зависимости невязки от длины ЦМ для компоненты z |