Министерство образования Республики Беларусь

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

Факультет прикладной математики и информатики

**Кафедра**

Сергиенко Лев Эдуардович

Отчет по лабораторным работам по курсу

“Математическое моделирование”

студента 3 курса 12 группы

|  |  |
| --- | --- |
| Работа сдана 2024г. | **Преподаватель** |
| зачтена \_\_\_\_\_\_\_\_\_\_ 2024 г.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись преподавателя) | *Лобач Виктор Иванович*  доцент кафедры ММАД,  канд. физ.-мат. наук |
|  |  |

*Минск 2024*

**Лабораторная работа 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 a0, 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: a0 = β = 32771, K = 128

**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*=2^31 , ==32771.

**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*=128.

** - критерий согласия Пирсона:**

Область возможных значений случайной величины разбивается на интервалы .

Рассматривается следующая статистика,

,

*n* – объем выборки,

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

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

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

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

**Критерий согласия Колмогорова:**

Рассматривается статистика:

где

,

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

**Код программы:**

1. import math

2. import numpy as np

3. from scipy.stats import chi2

4.

5. # Multiplicative Congruential Method

6. def mcm(a, beta, M, n):

7.     x = a

8.     results = []

9.     for \_ in range(n):

10.         x = (beta \* x) % M

11.         results.append(x / M)

12.     return results

13.

14. # MacLaren-Marsaglia Method

15. def mmm(gen1, gen2, K, n):

16.     table = gen1[:K]

17.     results = []

18.     for i in range(n):

19.         j = int(gen2[i] \* K)

20.         results.append(table[j])

21.         table[j] = gen1[K + i]

22.     return results

23.

24. # Empirical Distribution Function

25. def empirical\_distribution(data):

26.     sorted\_data = sorted(data)

27.     n = len(data)

28.     edf = []

29.     for i in range(n):

30.         edf.append((i + 1) / n)

31.     return sorted\_data, edf

32.

33. # Kolmogorov criterion

34. def kolm\_criterion(data, e=0.05):

35.     sorted\_data, edf = empirical\_distribution(data)

36.     n = len(data)

37.     d\_max = 0

38.     for i in range(n):

39.         theoretical\_cdf = sorted\_data[i]  # for uniform distribution F(x) = x

40.         d = abs(edf[i] - theoretical\_cdf)

41.         if d > d\_max:

42.             d\_max = d

43.

44.     critical = math.sqrt(-0.5 \* math.log(e / 2) / n)

45.     return d\_max, critical, d\_max < critical

46.

47. # χ²-Pearson criterion

48. def pearson(data, e=0.05, num\_bins=15):

49.     n = len(data)

50.     if n == 0:

51.         raise ValueError("Data cannot be empty")

52.

53.     bin\_edges = np.linspace(0, 1, num\_bins + 1)  # Define the bin edges

54.

55.     # Count the number of elements in each interval

56.     observed, \_ = np.histogram(data, bins=bin\_edges)

57.

58.     expInBin = n / num\_bins

59.

60.     stat = np.sum((observed - expInBin) \*\* 2 / expInBin)  # χ² = Σ((Oi - Ei)² / Ei)

61.

62.     # Critical value for χ² with (num\_bins - 1) degrees of freedom and significance level e

63.     critical = chi2.ppf(1 - e, num\_bins - 1)

64.

65.     return stat, critical, stat < critical

66.

67. def test\_data(data, e=0.05):

68.     d\_max, crit, kolm\_passed = kolm\_criterion(data, e)

69.     print(

70.         f"Kolmogorov criterion: D\_max = {d\_max:.4f}, critical: {crit:.4f}, passed: {kolm\_passed}"

71.     )

72.     chi2\_stat, crit, chi2\_passed = pearson(data, e)

73.     print(

74.         f"χ²-Pearson criterion: χ² = {chi2\_stat:.4f}, critical: {crit:.4f}, passed: {chi2\_passed}"

75.     )

76.

77. def main():

78.     a = 32771  # initial value

79.     beta = 32771  # multiplier

80.     M = 2\*\*31

81.     n = 1000

82.     K = 128

83.     e = 0.05

84.

85.     print("Checking MCM sequence:")

86.     mcm\_seq = mcm(a, beta, M, n)

87.     test\_data(mcm\_seq, e)

88.     print("\nChecking MacLaren-Marsaglia sequence:")

89.     mmm\_seq = mmm(mcm(3 \* beta, 3 \* beta, M, n + K), mcm\_seq, K, n)

90.     test\_data(mmm\_seq, e)

91.   main()

**Результаты:**

1. Checking MCM sequence:

2. Kolmogorov criterion: D\_max = 0.0385, critical: 0.0429, passed: True

3. χ²-Pearson criterion: χ² = 12.8000, critical: 23.6848, passed: True

4.

5. Checking MacLaren-Marsaglia sequence:

6. Kolmogorov criterion: D\_max = 0.0166, critical: 0.0429, passed: True

7. χ²-Pearson criterion: χ² = 4.8200, critical: 23.6848, passed: True

**Лабораторная работа 2.**

**Условие:**

Смоделировать дискретную случайную величину.

Исследовать точность моделирования.

1) Осуществить моделирование n = 1000 реализаций СВ из заданных дискретных распределений.

2) Вывести на экран несмещенные оценки математического ожидания и дисперсии, сравнить их с истинными значениями.

3) Для каждой из случайных величин построить свой χ2 -критерием Пирсона с уровнем значимость ε=0.05. Проверить, что вероятность ошибки I рода стремится к 0.05.

4) Осуществить проверку каждой из сгенерированных выборок каждым из построенных критериев.

Вариант: Бернулли – Bi(1,p), p = 0.75; Пуассона – П(λ), λ = 1;

**Теория:**

**Распределение Бернулли (с параметром pp)**  
Случайная величина X в случае распределения Бернулли принимает значение 1 с вероятностью p и значение 0 с вероятностью 1 - p. Формально это можно записать так:

P(X=1)=p, P(X=0)=1−p

Чтобы сгенерировать выборку из распределения Бернулли, в данной работе использовался следующий метод:

1. Генерируется равномерная случайная величина U из интервала [0;1].
2. Если U≤p, то принимается X = 1.
3. Если U>p, то принимается X = 0.

Такой подход эквивалентен тому, что мы «подбрасываем монетку», у которой сторона «орёл» (аналог 1) выпадает с вероятностью p, а «решка» (аналог 0) — с вероятностью 1−p.

В коде это реализовано:

* Создаёт пустой список для хранения сгенерированных значений.
* В цикле n раз генерирует равномерное число U.
* Сравнивает U с p. Если U≤p, записывает 1, иначе 0.
* Возвращает итоговую выборку в виде массива.

Таким образом, получается выборка из n Бернуллиевских случайных величин с параметром p.

**Распределение Пуассона (с параметром λ=1)**  
Случайная величина X принимает только целые неотрицательные значения:

X∈{0,1,2,…}

Закон распределения задаётся формулой:

При λ = 1 это выражение упрощается к:

В данной работе для получения выборки из распределения Пуассона с параметром λ=1 сначала моделировалась последовательность равномерных случайных величин (БСВ), каждая из которых попадала в интервал [0;1]. Затем для каждого значения этой последовательности генерировалась сумма экспоненциальных интервалов длины

где U\_i — очередная равномерная БСВ. Сумма этих экспоненциальных интервалов вычислялась до тех пор, пока она не превысит единицу. Количество таких «шагов», совершённых прежде, чем сумма стала больше 1, и есть значение случайной величины X. Иначе говоря, мы моделируем поток Пуассона на промежутке времени длины 1 и считаем, сколько событий произошло за это время. Полученная таким образом выборка соответствует распределению Пуассона с параметром λ=1.

**Код программы:**

1. import numpy as np

2. import math

3. from scipy import stats

4.

5. n = 1000

6. p = 0.75

7. lambda\_param = 1

8. alpha = 0.05

9.

10. def bernoulli\_simulation(p, n):

11.     bernoulli\_rv = []

12.     for \_ in range(n):

13.         u = np.random.uniform(0, 1)

14.         if u <= p:

15.             bernoulli\_rv.append(1)

16.         else:

17.             bernoulli\_rv.append(0)

18.     return np.array(bernoulli\_rv)

19.

20. def poisson\_simulation(lambd, n):

21.     samples = []

22.     for \_ in range(n):

23.         N = -1

24.         S = 0

25.

26.         while S < 1:

27.             U = np.random.uniform(0, 1)

28.             E = -np.log(U) / lambd

29.             S += E

30.             N += 1

31.

32.         samples.append(N)

33.     return np.array(samples)

34.

35. def estimates(rv):

36.     mean\_estimate = np.mean(rv)         # вычисляем оценку мат. ожидания

37.     var\_estimate = np.var(rv, ddof=1)   # вычисляем оценку дисперсии

38.     return mean\_estimate, var\_estimate

39.

40. def chi\_square\_test\_bernoulli(observed, p, n):

41.     expected\_counts = np.array([n \* (1 - p), n \* p])

42.     observed\_counts = np.array([np.sum(observed == 0), np.sum(observed == 1)])

43.

44.     chi2\_stat = np.sum((observed\_counts - expected\_counts) \*\* 2 / expected\_counts)

45.     chi2\_crit = stats.chi2.ppf(1 - alpha, df=1)

46.     return chi2\_stat, chi2\_crit

47.

48. def chi\_square\_test\_poisson(observed, lambda\_param, n, threshold=5):

49.     unique\_values, counts = np.unique(observed, return\_counts=True)

50.     # Вычисляем вероятности для каждого уникального значения, используя распределение Пуассона

51.     expected\_probs = stats.poisson.pmf(unique\_values, lambda\_param)

52.     # Вычисляем ожидаемое количество для каждого уникального значения

53.     expected\_counts = n \* expected\_probs

54.

55.     # Агрегируем малые значения, если их ожидания меньше порогового значения threshold

56.     mask = expected\_counts >= threshold

57.     if not np.all(mask):

58.         observed\_aggregated = np.append(counts[mask], np.sum(counts[~mask]))

59.         expected\_aggregated = np.append(expected\_counts[mask], np.sum(expected\_counts[~mask]))

60.     else:

61.         observed\_aggregated = counts

62.         expected\_aggregated = expected\_counts

63.

64.     chi2\_stat = np.sum((observed\_aggregated - expected\_aggregated) \*\* 2 / expected\_aggregated)

65.     chi2\_crit = stats.chi2.ppf(1 - alpha, df=len(observed\_aggregated) - 1)

66.     return chi2\_stat, chi2\_crit

67.

68. bernoulli\_rv = bernoulli\_simulation(p, n)

69. poisson\_rv = poisson\_simulation(lambda\_param, n)

70.

71. true\_mean\_bernoulli = p

72. true\_var\_bernoulli = p \* (1 - p)

73.

74. true\_mean\_poisson = lambda\_param

75. true\_var\_poisson = lambda\_param

76.

77. mean\_bernoulli, var\_bernoulli = estimates(bernoulli\_rv)

78. mean\_poisson, var\_poisson = estimates(poisson\_rv)

79.

80. def print\_statistics(distribution\_name, param\_name, param, mean, true\_mean, var, true\_var):

81.     print(f"{distribution\_name} ({param\_name}={param}):")

82.     print(f"Оценка мат. ожидания: {mean}, Истинное значение: {true\_mean}")

83.     print(f"Оценка дисперсии: {var}, Истинное значение: {true\_var}\n")

84.

85. def perform\_chi\_square\_test(distribution\_name, chi\_square\_test\_func, rv, param, n):

86.     chi2, crit = chi\_square\_test\_func(rv, param, n)

87.     print(f"χ²-критерий для {distribution\_name}: χ² = {chi2}, критическое значение = {crit}")

88.

89.     if chi2 > crit:

90.         print(f"Критерий χ² НЕ пройден для {distribution\_name}.")

91.     else:

92.         print(f"Критерий χ² пройден для {distribution\_name}.")

93.

94. print\_statistics("Бернулли", "p", p, mean\_bernoulli, true\_mean\_bernoulli, var\_bernoulli, true\_var\_bernoulli)

95. print\_statistics("Пуассон", "λ", lambda\_param, mean\_poisson, true\_mean\_poisson, var\_poisson, true\_var\_poisson)

96.

97. perform\_chi\_square\_test("Бернулли", chi\_square\_test\_bernoulli, bernoulli\_rv, p, n)

98. perform\_chi\_square\_test("Пуассон", chi\_square\_test\_poisson, poisson\_rv, lambda\_param, n)

99.

100. def type\_1\_error\_rate(test\_func, distribution\_simulation\_func, param, n, num\_simulations=1000):

101.     errors = 0

102.     for \_ in range(num\_simulations):

103.         rv = distribution\_simulation\_func(param, n)

104.         chi2, crit = test\_func(rv, param, n)

105.         if chi2 > crit:

106.             errors += 1

107.     return errors / num\_simulations

108.

109. bernoulli\_error\_rate = type\_1\_error\_rate(chi\_square\_test\_bernoulli, bernoulli\_simulation, p, n)

110. print(f"Оценка вероятности ошибки I рода для распределения Бернулли: {bernoulli\_error\_rate}")

111.

112. poisson\_error\_rate = type\_1\_error\_rate(chi\_square\_test\_poisson, poisson\_simulation, lambda\_param, n)

113. print(f"Оценка вероятности ошибки I рода для распределения Пуассона: {poisson\_error\_rate}")

Результыт:

1. Бернулли (p=0.75):

2. Оценка мат. ожидания: 0.738, Истинное значение: 0.75

3. Оценка дисперсии: 0.1935495495495495, Истинное значение: 0.1875

4.

5. Пуассон (λ=1):

6. Оценка мат. ожидания: 0.995, Истинное значение: 1

7. Оценка дисперсии: 1.0139889889889888, Истинное значение: 1

8.

9. χ²-критерий для Бернулли: χ² = 0.768, критическое значение = 3.841458820694124

10. Критерий χ² пройден для Бернулли.

11. χ²-критерий для Пуассон: χ² = 1.4079896616777243, критическое значение = 11.070497693516351

12. Критерий χ² пройден для Пуассон.

13. Оценка вероятности ошибки I рода для распределения Бернулли: 0.054

14. Оценка вероятности ошибки I рода для распределения Пуассона: 0.055

**Лабораторная работа 3.**

**Условие:**

Смоделировать непрерывную случайную величину. Исследовать точность моделирования.

1) Осуществить моделирование n = 1000 реализаций СВ из нормального закона распределения N(m, s^2) с заданными параметрами. Вычислить несмещенные оценки математического ожидания и дисперсии, сравнить их с истинными.

2) Смоделировать n = 1000 СВ из заданных абсолютно непрерывных распределений. Вычислить несмещенные оценки математического ожидания и дисперсии, сравнить их с истинными значениями (если это возможно).

3) Для каждой из случайных величин построить свой критерий Колмогорова с уровнем значимость ε=0.05. Проверить, что вероятность ошибки I рода стремится к 0.05.

4) Для каждой из случайных величин построить свой χ2 -критерий Пирсона с уровнем значимость ε=0.05. Проверить, что вероятность ошибки I рода стремится к 0.05.

5) Осуществить проверку каждой из сгенерированных выборок каждым из построенных критериев.

Вариант: m = 0, s^2 = 1; Лапласа L(a), a = 0.5; Вейбулла W(a,b), a = 1, b = 0.5.

**Теория:**

**Моделирование непрерывной случайной величины из нормального распределения N(m,σ^2)**

**Нормальное (Гауссово) распределение** с параметрами m (математическое ожидание) и σ^2 (дисперсия) имеет плотность вероятности:

Основные характеристики:

* **Математическое ожидание** E[X]=m
* **Дисперсия**
* **Стандартное нормальное распределение** — частный случай при m=0 и

**Метод Бокса–Мюллера для моделирования**

Для получения выборки из N(m,σ2) часто используется метод Бокса–Мюллера, позволяющий сгенерировать две независимые случайные величины, каждая из которых имеет стандартное нормальное распределение N(0,1). Шаги метода:

1. Сгенерировать две равномерно распределённые величины U\_1, U\_2 на [0;1].
2. Вычислить
3. Получить две независимые стандартные нормальные случайные величины:
4. При необходимости перейти к путём линейного преобразования

**Моделирование из других абсолютно непрерывных распределений**

В данной лабораторной работе дополнительно моделируются случайные величины из **распределения Лапласа L(a)** и **распределения Вейбулла W(a,b).**

**Распределение Лапласа L(a)**

**Распределение Лапласа** (двустороннее экспоненциальное) с параметром a > 0 имеет плотность вероятности:

Основные характеристики:

* Математическое ожидание: E[X]=0
* Дисперсия:

**Метод моделирования** (обратное преобразование):

1. Сгенерировать U∼Uniform(0,1).
2. Если U≤0, то

ЕслиU > 0.5, то

**Распределение Вейбулла W(a,b)**

**Распределение Вейбулла** с параметрами a>0 и b>0 имеет плотность:

Основные характеристики:

где Γ(⋅) — гамма-функция.

**Метод моделирования** (обратное преобразование):

1. Сгенерировать U∼Uniform(0,1).
2. Положить

Тогда X имеет распределение Вейбулла с параметрами a и b.

**Проверка вероятности ошибки I рода**

**Ошибка I рода** — это вероятность отвергнуть верную гипотезу (в нашем случае — гипотезу о том, что выборка действительно взята из F).  
Чтобы убедиться, что критерии работают корректно на уровне α=0.05, можно провести имитационный эксперимент:

1. Многократно (скажем, 1000 раз) сгенерировать выборку размера n из **того самого** распределения F.
2. Каждый раз применять критерий Колмогорова или **χ2**.
3. Подсчитать долю экспериментов, где гипотеза была отвергнута.  
   Эта доля должна быть близка к 0.05.

**Код программы:**

1. import math

2. import random

3. from scipy.stats import chi2

4. import numpy as np

5.

6. n = 1000

7. alpha = 0.05

8. num\_simulations = 1000

9.

10. # Моделирование нормального распределения N(0,1) с помощью преобразования Бокса-Мюллера

11. def generate\_normal(n):

12.     normals = []

13.     for \_ in range((n + 1) // 2):

14.         U1 = random.uniform(0, 1)

15.         U2 = random.uniform(0, 1)

16.

17.         # Преобразование Бокса-Мюллера

18.         R = math.sqrt(-2 \* math.log(U1))

19.         theta = 2 \* math.pi \* U2

20.         Z1 = R \* math.cos(theta)

21.         Z2 = R \* math.sin(theta)

22.

23.         normals.extend([Z1, Z2])

24.

25.     return normals[:n]

26.

27. # Моделирование распределения Лапласа L(a) с помощью обратного преобразования

28. def generate\_laplace(n, a):

29.     laplace\_vars = []

30.     for \_ in range(n):

31.         U = random.uniform(0, 1)

32.

33.         if U <= 0.5:

34.             X = (1 / a) \* math.log(2 \* U)

35.         else:

36.             X = -(1 / a) \* math.log(2 \* (1 - U))

37.

38.         laplace\_vars.append(X)

39.

40.     return laplace\_vars

41.

42. # Моделирование распределения Вейбулла с помощью обратного преобразования

43. def generate\_weibull(n, a, b):

44.     weibull\_vars = []

45.     for \_ in range(n):

46.         U = random.uniform(0, 1)

47.

48.         X = b \* (-math.log(U)) \*\* (1 / a)

49.

50.         weibull\_vars.append(X)

51.

52.     return weibull\_vars

53.

54. # Функции распределения для каждой случайной величины

55.

56. def standard\_normal\_cdf(x):

57.     return 0.5 \* (1 + math.erf(x / math.sqrt(2)))  #  erf - функция ошибок

58.

59. def laplace\_cdf(x, a):

60.     if x < 0:

61.         return 0.5 \* math.exp(a \* x)

62.     else:

63.         return 1 - 0.5 \* math.exp(-a \* x)

64.

65. def weibull\_cdf(x, a, b):

66.     if x < 0:

67.         return 0

68.     else:

69.         return 1 - math.exp(-((x / b) \*\* a))

70.

71. # Функция для проведения теста Колмогорова-Смирнова

72. def ks\_test(sample, cdf\_func, args=(), alpha=0.05):

73.     n = len(sample)

74.     sorted\_sample = sorted(sample)

75.     D\_max = 0

76.

77.     for i, x in enumerate(sorted\_sample, start=1):

78.         F\_x = cdf\_func(x, \*args)

79.         F\_n = i / n

80.

81.         D\_current = abs(F\_n - F\_x)

82.         if D\_current > D\_max:

83.             D\_max = D\_current

84.

85.     D\_alpha = math.sqrt(-0.5 \* math.log(alpha / 2) / n)

86.

87.     result = D\_max > D\_alpha

88.

89.     return D\_max, D\_alpha, result

90.

91. # Функция для проведения теста хи-квадрат

92. def chi\_squared\_test(sample, cdf\_func, args=(), alpha=0.05, k=10):

93.     n = len(sample)

94.     # Определение границ интервалов (бинов)

95.     bin\_edges = [np.percentile(sample, 100 \* i / k) for i in range(k + 1)]

96.     # Наблюдаемые частоты в каждом бине

97.     observed, \_ = np.histogram(sample, bins=bin\_edges)

98.

99.     expected = []

100.     for i in range(k):

101.         lower = bin\_edges[i]

102.         upper = bin\_edges[i + 1]

103.

104.         F\_upper = cdf\_func(upper, \*args)

105.         F\_lower = cdf\_func(lower, \*args)

106.

107.         # Ожидаемая частота в бине

108.         expected\_freq = n \* (F\_upper - F\_lower)

109.         # Предотвращение деления на ноль

110.         expected.append(expected\_freq if expected\_freq > 0 else 1e-6)

111.

112.     chi2\_stat = sum((o - e) \*\* 2 / e for o, e in zip(observed, expected))

113.     chi2\_critical = chi2.ppf(1 - alpha, df=k - 1)

114.

115.     result = chi2\_stat > chi2\_critical

116.

117.     return chi2\_stat, chi2\_critical, result

118.

119. def check\_I\_error(generate\_func, cdf\_func, args=(), alpha=0.05):

120.     ks\_passed = 0

121.     chi2\_passed = 0

122.     for \_ in range(num\_simulations):

123.         sample = generate\_func(n)

124.         D, D\_alpha, ks\_result = ks\_test(sample, cdf\_func, args, alpha)

125.         chi2\_stat, chi2\_critical, chi2\_result = chi\_squared\_test(

126.             sample, cdf\_func, args, alpha, k=10

127.         )

128.         if not ks\_result:

129.             ks\_passed += 1

130.         if not chi2\_result:

131.             chi2\_passed += 1

132.

133.     ks\_success\_rate = 100 \* (ks\_passed / num\_simulations)

134.     chi2\_success\_rate = 100 \* (chi2\_passed / num\_simulations)

135.

136.     return ks\_success\_rate, chi2\_success\_rate

137.

138. def print\_results(

139.     name,

140.     sample\_mean,

141.     theoretical\_mean,

142.     sample\_var,

143.     theoretical\_var,

144.     D,

145.     D\_alpha,

146.     ks\_passed,

147.     chi2\_stat,

148.     chi2\_critical,

149.     chi2\_passed,

150.     success\_rate\_ks,

151.     success\_rate\_chi2,

152. ):

153.     get\_checkmark = lambda passed: "✅" if passed else "❌"

154.

155.     print(f"> {name}:")

156.     print(f"{'':<12}{'Фактическое':>12}{'Теоретическое':>15}")

157.     print(f"{'Среднее':<12}{sample\_mean:>12.4f}{theoretical\_mean:>15.4f}")

158.     print(f"{'Дисперсия':<12}{sample\_var:>12.4f}{theoretical\_var:>15.4f}")

159.     print("-" \* 55)

160.     checkmark\_ks = get\_checkmark(not ks\_passed)

161.     checkmark\_chi2 = get\_checkmark(not chi2\_passed)

162.     print(

163.         f"Колмогорова-Смирнова: {D:.3f} < {D\_alpha:.3f} | Тест пройден: {checkmark\_ks}"

164.     )

165.     print(

166.         f"Хи-квадрат Пирсона: {chi2\_stat:.4f} < {chi2\_critical:.4f} | Тест пройден: {checkmark\_chi2}"

167.     )

168.     print(f"Успех KS: {success\_rate\_ks:.1f}%")

169.     print(f"Успех хи-квадрат: {success\_rate\_chi2:.1f}%")

170.     print("-" \* 55 + "\n")

171.

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

173.     # Нормальное распределение

174.     normals = generate\_normal(n)

175.     mean\_normal = sum(normals) / n

176.     var\_normal = sum((x - mean\_normal) \*\* 2 for x in normals) / (n - 1)

177.     theoretical\_mean\_normal = 0

178.     theoretical\_var\_normal = 1

179.     D\_normal, D\_alpha\_normal, result\_normal = ks\_test(

180.         normals, standard\_normal\_cdf, alpha=alpha

181.     )

182.     chi2\_stat\_normal, chi2\_critical\_normal, chi2\_result\_normal = chi\_squared\_test(

183.         normals, standard\_normal\_cdf, args=(), alpha=alpha, k=10

184.     )

185.     success\_normal\_ks, success\_normal\_chi2 = check\_I\_error(

186.         generate\_normal, standard\_normal\_cdf, args=(), alpha=alpha

187.     )

188.     print\_results(

189.         "Нормальное распределение N(0,1)",

190.         mean\_normal,

191.         theoretical\_mean\_normal,

192.         var\_normal,

193.         theoretical\_var\_normal,

194.         D\_normal,

195.         D\_alpha\_normal,

196.         result\_normal,

197.         chi2\_stat\_normal,

198.         chi2\_critical\_normal,

199.         chi2\_result\_normal,

200.         success\_normal\_ks,

201.         success\_normal\_chi2,

202.     )

203.

204.     # Распределение Лапласа

205.     a\_laplace = 0.5

206.     laplace\_vars = generate\_laplace(n, a\_laplace)

207.     mean\_laplace = sum(laplace\_vars) / n

208.     var\_laplace = sum((x - mean\_laplace) \*\* 2 for x in laplace\_vars) / (n - 1)

209.     theoretical\_mean\_laplace = 0

210.     theoretical\_var\_laplace = 2 / (a\_laplace\*\*2)

211.     D\_laplace, D\_alpha\_laplace, result\_laplace = ks\_test(

212.         laplace\_vars, laplace\_cdf, args=(a\_laplace,), alpha=alpha

213.     )

214.     chi2\_stat\_laplace, chi2\_critical\_laplace, chi2\_result\_laplace = chi\_squared\_test(

215.         laplace\_vars, laplace\_cdf, args=(a\_laplace,), alpha=alpha, k=10

216.     )

217.     success\_laplace\_ks, success\_laplace\_chi2 = check\_I\_error(

218.         lambda n: generate\_laplace(n, a\_laplace),

219.         laplace\_cdf,

220.         args=(a\_laplace,),

221.         alpha=alpha,

222.     )

223.     print\_results(

224.         f"Распределение Лапласа L(a={a\_laplace})",

225.         mean\_laplace,

226.         theoretical\_mean\_laplace,

227.         var\_laplace,

228.         theoretical\_var\_laplace,

229.         D\_laplace,

230.         D\_alpha\_laplace,

231.         result\_laplace,

232.         chi2\_stat\_laplace,

233.         chi2\_critical\_laplace,

234.         chi2\_result\_laplace,

235.         success\_laplace\_ks,

236.         success\_laplace\_chi2,

237.     )

238.

239.     # Распределение Вейбулла

240.     a\_weibull = 1

241.     b\_weibull = 0.5

242.     weibull\_vars = generate\_weibull(n, a\_weibull, b\_weibull)

243.     mean\_weibull = sum(weibull\_vars) / n

244.     var\_weibull = sum((x - mean\_weibull) \*\* 2 for x in weibull\_vars) / (n - 1)

245.     theoretical\_mean\_weibull = b\_weibull \* math.gamma(1 + 1 / a\_weibull)

246.     theoretical\_var\_weibull = b\_weibull\*\*2 \* (

247.         math.gamma(1 + 2 / a\_weibull) - (math.gamma(1 + 1 / a\_weibull)) \*\* 2

248.     )

249.     D\_weibull, D\_alpha\_weibull, result\_weibull = ks\_test(

250.         weibull\_vars, weibull\_cdf, args=(a\_weibull, b\_weibull), alpha=alpha

251.     )

252.     chi2\_stat\_weibull, chi2\_critical\_weibull, chi2\_result\_weibull = chi\_squared\_test(

253.         weibull\_vars, weibull\_cdf, args=(a\_weibull, b\_weibull), alpha=alpha, k=10

254.     )

255.     success\_weibull\_ks, success\_weibull\_chi2 = check\_I\_error(

256.         lambda n: generate\_weibull(n, a\_weibull, b\_weibull),

257.         weibull\_cdf,

258.         args=(a\_weibull, b\_weibull),

259.         alpha=alpha,

260.     )

261.     print\_results(

262.         f"Распределение Вейбулла W(a={a\_weibull}, b={b\_weibull})",

263.         mean\_weibull,

264.         theoretical\_mean\_weibull,

265.         var\_weibull,

266.         theoretical\_var\_weibull,

267.         D\_weibull,

268.         D\_alpha\_weibull,

269.         result\_weibull,

270.         chi2\_stat\_weibull,

271.         chi2\_critical\_weibull,

272.         chi2\_result\_weibull,

273.         success\_weibull\_ks,

274.         success\_weibull\_chi2,

275.     )

276.

**Результат:**

1. > Нормальное распределение N(0,1):

2. Фактическое Теоретическое

3. Среднее 0.0103 0.0000

4. Дисперсия 1.0429 1.0000

5. -------------------------------------------------------

6. Колмогорова-Смирнова: 0.027 < 0.043 | Тест пройден: ✅

7. Хи-квадрат Пирсона: 7.5795 < 16.9190 | Тест пройден: ✅

8. Успех KS: 95.2%

9. Успех хи-квадрат: 93.8%

10. -------------------------------------------------------

11.

12. > Распределение Лапласа L(a=0.5):

13. Фактическое Теоретическое

14. Среднее -0.1291 0.0000

15. Дисперсия 8.1571 8.0000

16. -------------------------------------------------------

17. Колмогорова-Смирнова: 0.035 < 0.043 | Тест пройден: ✅

18. Хи-квадрат Пирсона: 6.9892 < 16.9190 | Тест пройден: ✅

19. Успех KS: 95.2%

20. Успех хи-квадрат: 95.1%

21. -------------------------------------------------------

22.

23. > Распределение Вейбулла W(a=1, b=0.5):

24. Фактическое Теоретическое

25. Среднее 0.5158 0.5000

26. Дисперсия 0.2907 0.2500

27. -------------------------------------------------------

28. Колмогорова-Смирнова: 0.017 < 0.043 | Тест пройден: ✅

29. Хи-квадрат Пирсона: 5.9083 < 16.9190 | Тест пройден: ✅

30. Успех KS: 96.5%

31. Успех хи-квадрат: 95.1%

32. -------------------------------------------------------

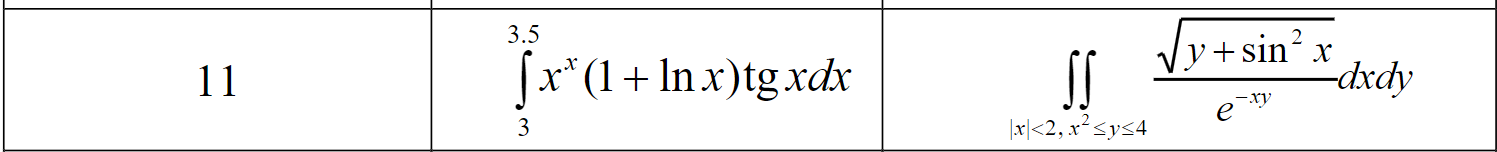
**Лабораторная работа 4**

**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.



**Теория:**

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

Необходимо вычислить .

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

Пусть  Тогда 

В качестве приближенного значения *a* можно взять



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

**Код программы:**

1. import numpy as np

2. import matplotlib.pyplot as plt

3. from scipy.integrate import quad, dblquad

4. from tabulate import tabulate

5.

6. def f1(x):

7.     return x\*\*x \* (1 + np.log(x)) \* np.tan(x)

8.

9. a\_L1 = 3

10. b\_L1 = 3.5

11.

12. exact\_value\_L1, \_ = quad(f1, a\_L1, b\_L1)

13. print("Точное значение L1 (приблизительно):", exact\_value\_L1)

14.

15. N\_values\_L1 = np.logspace(2, 7, num=15, dtype=int)

16. errors\_L1 = []

17. results\_L1 = []

18. for N in N\_values\_L1:

19.     x\_random = np.random.uniform(a\_L1, b\_L1, N)

20.     f\_values = f1(x\_random)

21.     # 1/N \* sum(f(x\_i)) / (1 / (b - a))

22.     integral\_estimate = (b\_L1 - a\_L1) \* np.mean(f\_values)

23.     error = np.abs(integral\_estimate - exact\_value\_L1)

24.     errors\_L1.append(error)

25.     results\_L1.append([N, integral\_estimate, error])

26.

27. print(

28.     tabulate(

29.         results\_L1,

30.         headers=["N", "Оценка интеграла", "Ошибка"],

31.         tablefmt="grid",

32.         floatfmt=".6f",

33.     )

34. )

35.

36. def f2(y, x):

37.     return np.sqrt(y + np.sin(x) \*\* 2) \* np.exp(x \* y)

38.

39. # Параметры интегрирования

40. x\_min = -2

41. x\_max = 2

42. y\_max = 4

43.

44. # Точное значение интеграла

45. exact\_value\_L2, \_ = dblquad(f2, x\_min, x\_max, lambda x: x\*\*2, y\_max)

46. print("Точное значение L2 (приблизительно):", exact\_value\_L2)

47.

48. # Количество выборок

49. N\_values\_L2 = np.logspace(3, 7, num=15, dtype=int)

50. errors\_L2 = []

51. results\_L2 = []

52.

53. # Функция для усеченного экспоненциального распределения

54. def sample\_truncated\_exponential(min, max, h):

55.     u = np.random.uniform(0, 1)

56.     cdf\_min = 1 - np.exp(-h \* min)

57.     cdf\_max = 1 - np.exp(-h \* max)

58.     y = -np.log(1 - u \* (cdf\_max - cdf\_min) - cdf\_min) / h

59.     return y

60.

61. # Основная функция для вычисления интеграла

62. def monte\_carlo\_integration(N):

63.     weights = []

64.

65.     for \_ in range(N):

66.         x = np.random.uniform(x\_min, x\_max)

67.         y\_min = x\*\*2

68.         y = np.random.uniform(y\_min, y\_max)

69.

70.         f\_value = f2(y, x) \* (y\_max - y\_min) \* (x\_max - x\_min)

71.

72.         weights.append(f\_value)

73.

74.     # Оценка интеграла

75.     integral\_estimate = np.mean(weights)

76.     return integral\_estimate

77.

78. for N in N\_values\_L2:

79.     integral\_estimate = monte\_carlo\_integration(N)

80.     error = np.abs(integral\_estimate - exact\_value\_L2)

81.     errors\_L2.append(error)

82.     results\_L2.append([N, integral\_estimate, error])

83.

84. # Вывод результатов

85. print(

86.     tabulate(

87.         results\_L2,

88.         headers=["N", "Оценка интеграла", "Ошибка"],

89.         tablefmt="grid",

90.         floatfmt=".6f",

91.     )

92. )

Результат:

1. Точное значение L1 (приблизительно): 8.506555930307984

2. +-----------------+--------------------+----------+

3. | N | Оценка интеграла | Ошибка |

4. +=================+====================+==========+

5. | 100.000000 | 8.349363 | 0.157193 |

6. +-----------------+--------------------+----------+

7. | 227.000000 | 9.082931 | 0.576375 |

8. +-----------------+--------------------+----------+

9. | 517.000000 | 8.701543 | 0.194987 |

10. +-----------------+--------------------+----------+

11. | 1178.000000 | 8.304496 | 0.202059 |

12. +-----------------+--------------------+----------+

13. | 2682.000000 | 8.458351 | 0.048205 |

14. +-----------------+--------------------+----------+

15. | 6105.000000 | 8.435827 | 0.070729 |

16. +-----------------+--------------------+----------+

17. | 13894.000000 | 8.444518 | 0.062038 |

18. +-----------------+--------------------+----------+

19. | 31622.000000 | 8.587386 | 0.080830 |

20. +-----------------+--------------------+----------+

21. | 71968.000000 | 8.521572 | 0.015016 |

22. +-----------------+--------------------+----------+

23. | 163789.000000 | 8.476108 | 0.030448 |

24. +-----------------+--------------------+----------+

25. | 372759.000000 | 8.526480 | 0.019925 |

26. +-----------------+--------------------+----------+

27. | 848342.000000 | 8.501954 | 0.004602 |

28. +-----------------+--------------------+----------+

29. | 1930697.000000 | 8.503411 | 0.003145 |

30. +-----------------+--------------------+----------+

31. | 4393970.000000 | 8.513563 | 0.007007 |

32. +-----------------+--------------------+----------+

33. | 10000000.000000 | 8.504734 | 0.001822 |

34. +-----------------+--------------------+----------+

35. Точное значение L2 (приблизительно): 618.5419826403217

36. +-----------------+--------------------+-----------+

37. | N | Оценка интеграла | Ошибка |

38. +=================+====================+===========+

39. | 1000.000000 | 655.743445 | 37.201462 |

40. +-----------------+--------------------+-----------+

41. | 1930.000000 | 650.028589 | 31.486607 |

42. +-----------------+--------------------+-----------+

43. | 3727.000000 | 615.584282 | 2.957701 |

44. +-----------------+--------------------+-----------+

45. | 7196.000000 | 622.388864 | 3.846882 |

46. +-----------------+--------------------+-----------+

47. | 13894.000000 | 597.408535 | 21.133447 |

48. +-----------------+--------------------+-----------+

49. | 26826.000000 | 621.773007 | 3.231025 |

50. +-----------------+--------------------+-----------+

51. | 51794.000000 | 612.009245 | 6.532738 |

52. +-----------------+--------------------+-----------+

53. | 100000.000000 | 610.849423 | 7.692559 |

54. +-----------------+--------------------+-----------+

55. | 193069.000000 | 620.710215 | 2.168232 |

56. +-----------------+--------------------+-----------+

57. | 372759.000000 | 620.207972 | 1.665989 |

58. +-----------------+--------------------+-----------+

59. | 719685.000000 | 616.726374 | 1.815609 |

60. +-----------------+--------------------+-----------+

61. | 1389495.000000 | 619.357499 | 0.815516 |

62. +-----------------+--------------------+-----------+

63. | 2682695.000000 | 618.762731 | 0.220748 |

64. +-----------------+--------------------+-----------+

65. | 5179474.000000 | 619.218675 | 0.676692 |

66. +-----------------+--------------------+-----------+

67. | 10000000.000000 | 618.329053 | 0.212930 |

**A graph with blue lines and dots

Description automatically generatedA graph with a line

Description automatically generated**68. +-----------------+--------------------+-----------+

**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.

**A number of numbers on a white background

Description automatically generated**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 остальных – 0.

В данной работе выбиралось

**Код программы:**

1. import numpy as np

2. import matplotlib.pyplot as plt

3.

4. n = 3

5.

6. A = np.array(

7.     [

8.         [0.7, -0.3, -0.2],

9.         [-0.2, 0.2, -0.7],

10.         [-0.5, 0.1, 2],

11.     ]

12. )

13. f = np.array([-1, -3, 8])

14.

15. B = np.array(

16.     [

17.         [0, 0.3 / 0.7, 0.2 / 0.7],

18.         [0.2, 0, -0.2],

19.         [0.5 / 2, -0.1 / 2, 0],

20.     ]

21. )

22. g = np.array([-1 / 0.7, -3.45454545, 8 / 2])

23.

24. # Начальные вероятности для цепи Маркова (должны суммироваться в 1)

25. pi = np.array([1 / 3, 1 / 3, 1 / 3])

26.

27. # Матрица переходов для цепи Маркова

28. p = np.array([[0, 0.5, 0.5], [0.5, 0, 0.5], [0.5, 0.5, 0]])

29.

30. x\_direct = np.linalg.solve(A, f)

31.

32. def solve\_monte\_carlo\_all\_coords(N, m):

33.     # Результирующий вектор

34.     x\_sum = np.zeros(n)

35.

36.     # Чтобы удобнее было инициализировать Q, определим "h" как единичную матрицу:

37.     # h[l, i] = 1, если l == i, иначе 0.

38.     # Тогда Q[l,k] будет соответствовать случаю, когда "h" берётся для компоненты l.

39.     h = np.eye(n)

40.

41.     for \_ in range(m):

42.         # 1. Инициализация цепи Маркова

43.         alpha = np.random.rand()

44.         if alpha < pi[0]:

45.             i\_chain = [0]

46.         elif alpha < pi[0] + pi[1]:

47.             i\_chain = [1]

48.         else:

49.             i\_chain = [2]

50.

51.         # 2. Генерация цепи Маркова длины N

52.         for k in range(1, N + 1):

53.             alpha = np.random.rand()

54.             cumulative\_prob = 0.0

55.             for next\_state in range(n):

56.                 cumulative\_prob += p[i\_chain[-1]][next\_state]

57.                 if alpha < cumulative\_prob:

58.                     i\_chain.append(next\_state)

59.                     break

60.

61.         # 3. Вычисление весов Q для всех l = 0..n-1

62.         #    Q будет размером (n, N+1), т.к. для каждого l своя строка

63.         Q = np.zeros((n, N + 1))

64.

65.         # Инициализация Q[:, 0]

66.         # Для каждого l: Q[l, 0] = h[l, i\_chain[0]] / pi[i\_chain[0]] (если pi[i\_chain[0]]>0)

67.         # h[l, i\_chain[0]] = 1, если l == i\_chain[0], иначе 0

68.         if pi[i\_chain[0]] > 0:

69.             for l in range(n):

70.                 Q[l, 0] = h[l, i\_chain[0]] / pi[i\_chain[0]]

71.

72.         for k in range(1, N + 1):

73.             pr = p[i\_chain[k - 1]][i\_chain[k]]

74.             if pr > 0:

75.                 for l in range(n):

76.                     Q[l, k] = Q[l, k - 1] \* B[i\_chain[k - 1]][i\_chain[k]] / pr

77.             else:

78.                 # Если pr == 0, то переход невозможен; обнулим соответствующие веса

79.                 Q[:, k] = 0

80.

81.         # 4. Накопим вклад в x (для каждой компоненты l)

82.         #    ksi[l] = sum\_{k=0..N} Q[l, k] \* g[i\_chain[k]]

83.         ksi = np.zeros(n)

84.         for l in range(n):

85.             for k in range(N + 1):

86.                 ksi[l] += Q[l, k] \* g[i\_chain[k]]

87.

88.         x\_sum += ksi

89.

90.     x\_est = x\_sum / m

91.     return x\_est

92.

93. print("Решение методом прямого вычисления:")

94. print(x\_direct)

95.

96. print("Решение методом Монте-Карло:")

97. print(solve\_monte\_carlo\_all\_coords(N=1000, m=1000))

98.

**Результат:**

1. Решение методом прямого вычисления:

2. [-2.38636364 -4.65909091 3.63636364]

3. Решение методом Монте-Карло:

A graph of a graph with numbers and lines

Description automatically generated with medium confidence4. [-2.20121772 -5.07585652 3.44908949]

**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.