МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

ОТЧЕТ О ВЫПОЛНЕНИИ ЛАБОРАТОРНЫХ РАБОТ

по предмету ИиСМ

Выполнил:
Черепенников Роман
Михайлович
Сутдент 4 курса 8 группы

Минск, 2022

СОДЕРЖАНИЕ

Лабораторная 1. Моделирование БСВ

1.1 Условие

Необходимо:

1. Осуществить моделирование n = 1000 реализаций БСВ с помощью мультипликативного конгруэнтного метода (МКМ) с параметрами $a_0 = a_{01}$, $\beta = max(c_1, M - c_1)$, $M = 2^{31}$ и вывести 100-ый, 900-ый и 1000-ый элементы сгенерированной последовательности.

2. Осуществить моделирование $n = 1000$ реализаций БСВ с помощью метода Макларена-Марсальи, используя в качестве простейших датчиков БСВ датчики D1 – датчик из первого задания, D2 – датчик по методу МКМ с параметрами $a_0 = a_{02}$, $\beta = max(c_2, M - c_2)$, $M = 2^{31}$

Условия варианта:

$$a_{01} = 445423$$

$$c_1 = 90474281$$

$$a_{02} = 275803775$$

$$c_2 = 42062397$$

1.2 Выполнение работы

Исходный код:

```
class MultiplicativeCongruetteGenerator(AbstractRandomGenerator):
    def __init__(
            self,
            a0: int,
            b: int,
            M: int,
    ):
        self.a0 = a0
        self.M = M
        self.b = b
        self.prev = self.a0

    def next_element(self):
        z = self.b * self.prev
        self.prev = z - self.M * int(z / self.M)
```

```
        return self.prev / self.M

    def reset(self):
        self.prev = self.a0


class MacLarenMarsagliaGenerator(AbstractRandomGenerator):
    def __init__(
            self,
            gen1: AbstractRandomGenerator,
            gen2: AbstractRandomGenerator,
            K: int,
    ):
        self.gen1 = gen1
        self.gen2 = gen2
        self.K = K
        self.V = [self.gen1.next_element() for _ in range(K)]

    def next_element(self):
        s = int(self.gen2.next_element() * self.K)
        val = self.V[s]
        self.V[s] = self.gen1.next_element()
        return val

    def reset(self):
        self.gen1.reset()
        self.gen2.reset()

sns.set()
M = 2 ** 31
a01 = 445423
c1 = 90474281
gen1 = MultiplicativeCongruetteGenerator(
    a01, max([c1, M - c1]), M,
)
n = 30
generated = [gen1.next_element() for _ in range(1, n + 1)]
print(generated)
a02 = 275803775
c2 = 42062397
gen2 = MultiplicativeCongruetteGenerator(
    a02, max([c2, M - c2]), M,
```

```
)
K = 160
gen = MacLarenMarsagliaGenerator(gen1, gen2, K)
generated = [gen.next_element() for _ in range(100000)]
_, bins, _ = plt.hist(generated, bins=10, density=True)
plt.plot(bins, np.ones_like(bins), lw=2, color='red')
plt.show()
```

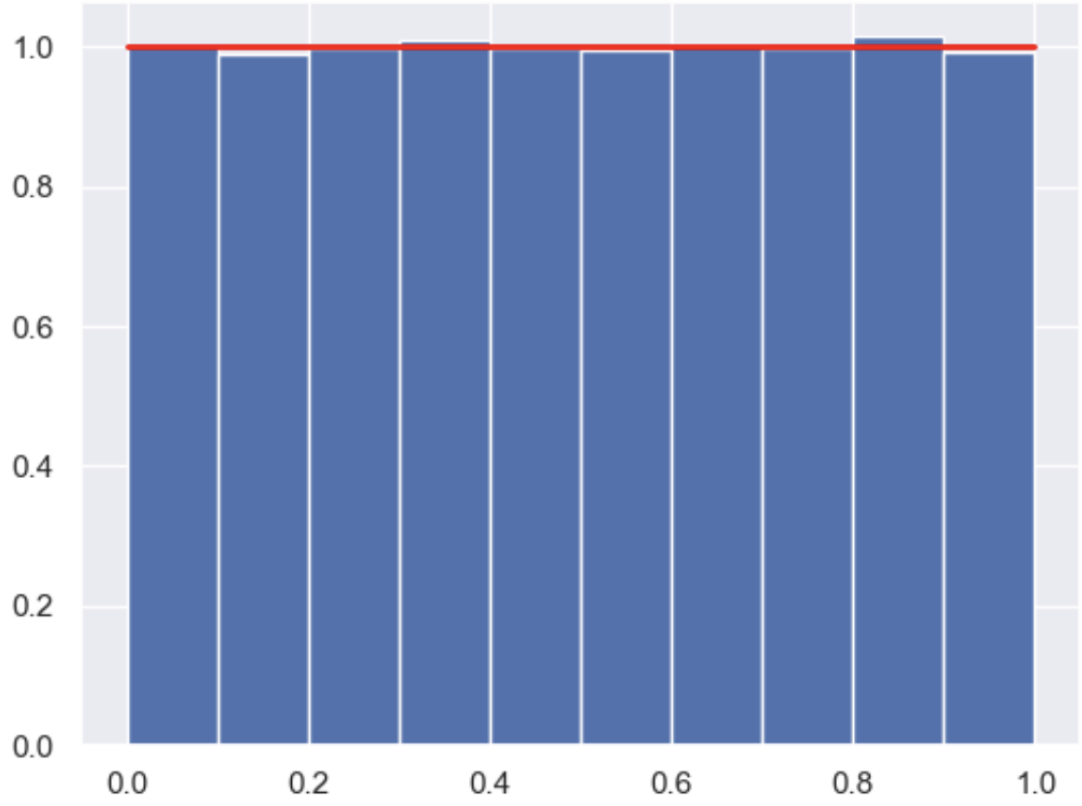

Рисунок 1.1 — Гистограмма значений, полученных с помощью метода Макларена-Марсальи

Лабораторная 2. Моделирование ДСВ

2.1 Условие

Необходимо осуществить моделирование n = 1000 реализаций СВ из заданных дискретных распределений для этого можно использовать любой генератор БСВ (как реализованный в 1-ой лабораторной работе, так и встроенный в язык программирования). Вывести на экран несмещенные оценки математического ожидания и дисперсии, сравнить их с истинными значениями.

Условия варианта:

1. Бернулли $p = 0.16$
2. Геометрическое $p = 0.3$
3. Биномиальное $m = 15, p = 0.35$
4. Пуассона $\lambda = 1$

2.2 Выполнение работы

Исходный код:

```
class BernoulliGenerator(AbstractRandomGenerator):
    def __init__(self, p: float):
        self.p = p

    def reset(self):
        super().reset()

    def next_element(self):
        a = random.random()
        return 1 if a <= self.p else 0


class GeometricGenerator(AbstractRandomGenerator):
    def __init__(self, p):
        self.q = 1.0 - p

    def next_element(self):
        return math.ceil(
```

```
            math.log(random.random()) / math.log(self.q)
        )

    def reset(self):
        super().reset()


class BinomialGenerator(AbstractRandomGenerator):
    def __init__(self, m: int, p: float):
        self.m = m
        self.bern = BernoulliGenerator(p)

    def next_element(self):
        x = 0
        for _ in range(self.m):
            x += self.bern.next_element()
        return x

    def reset(self):
        super().reset()


class PoissonGenerator(AbstractRandomGenerator):
    def __init__(self, l: float):
        self.l = l

    def next_element(self):
        p = math.exp(-self.l)
        x = 0
        r = random.random()
        r = r - p
        while r >= 0:
            x = x + 1
            p = p * self.l / x
            r = r - p
        return x

    def reset(self):
        super().reset()

bern = BernoulliGenerator(0.16)
hist(bern, 'Bernoulli', bernoulli, p=0.16)
```

```
geom_gen = GeometricGenerator(0.3)
hist(geom_gen, 'Geometric', geom, p=0.3)

bin = BinomialGenerator(15, 0.35)
hist(bin, 'Binomial', binom, n=15, p=0.35)

pois = PoissonGenerator(1.0)
hist(pois, 'Poisson', poisson, mu=1.0)
print(f'Коэффициент эксцесса для Пуассона : {excess(pois)}')
print(f'Коэффициент ассиметрии для Пуассона : {skewness(pois)}')
x, y = emperical_dist_function(pois)
y1 = poisson.cdf(x, mu=1.0)
plt.scatter(x, y, label='Emperical')
plt.scatter(x, y1, label='Theoretical', marker='x')
plt.legend()
plt.show()
```

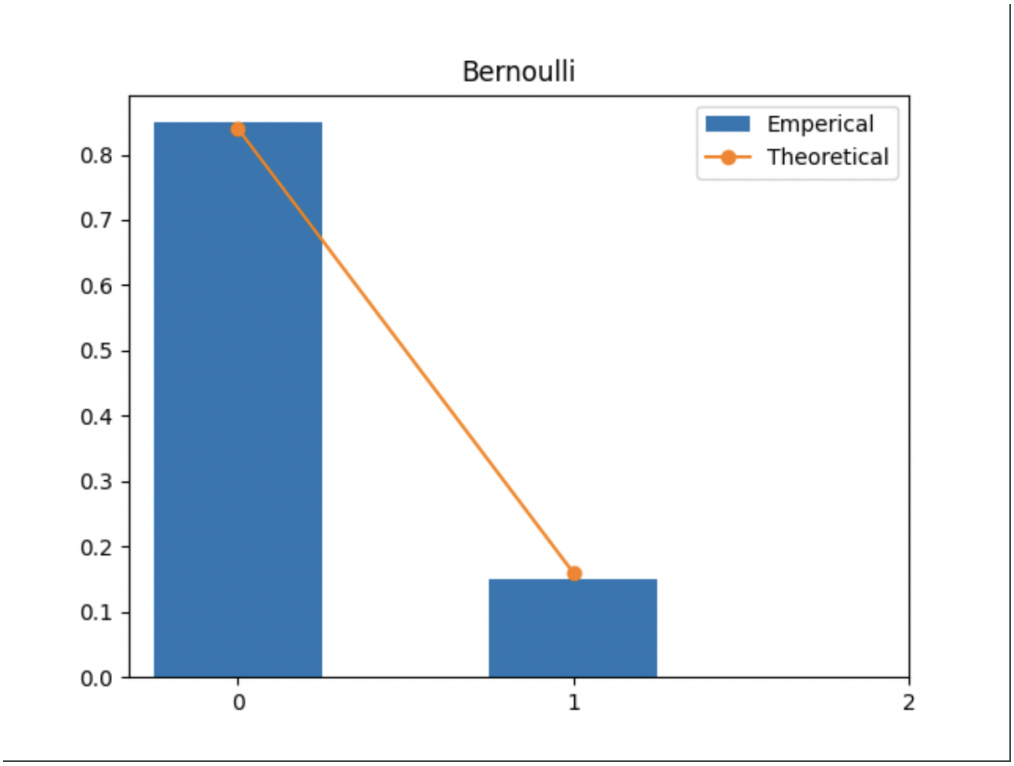

Рисунок 2.1 — Гистограмма значений для распределения Бернулли

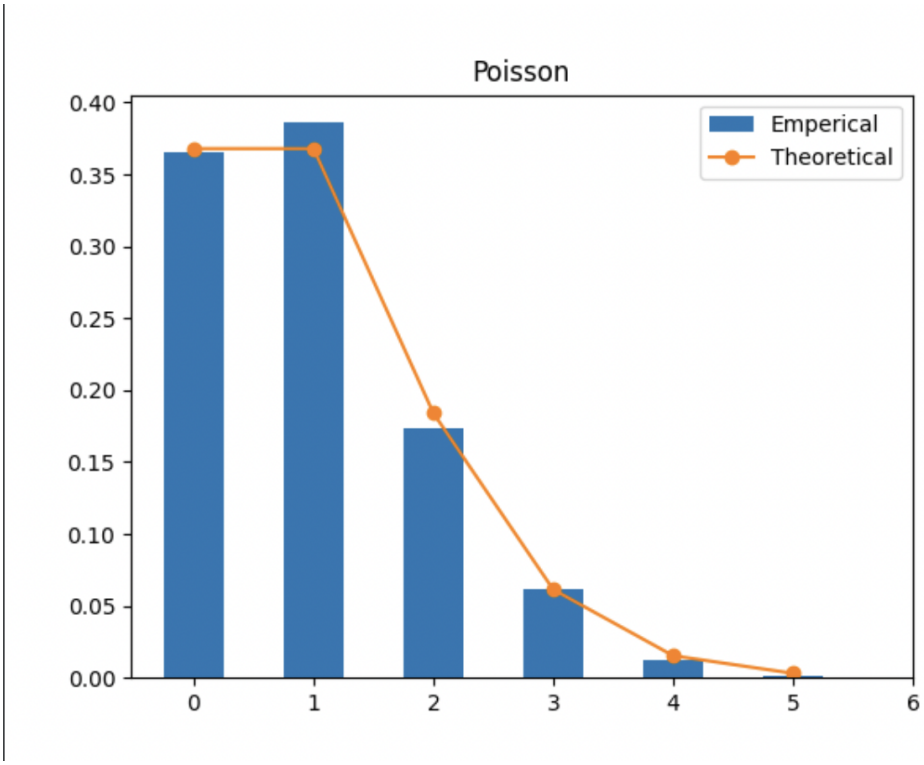

Рисунок 2.2 — Гистограмма значений для распределения Пуассона

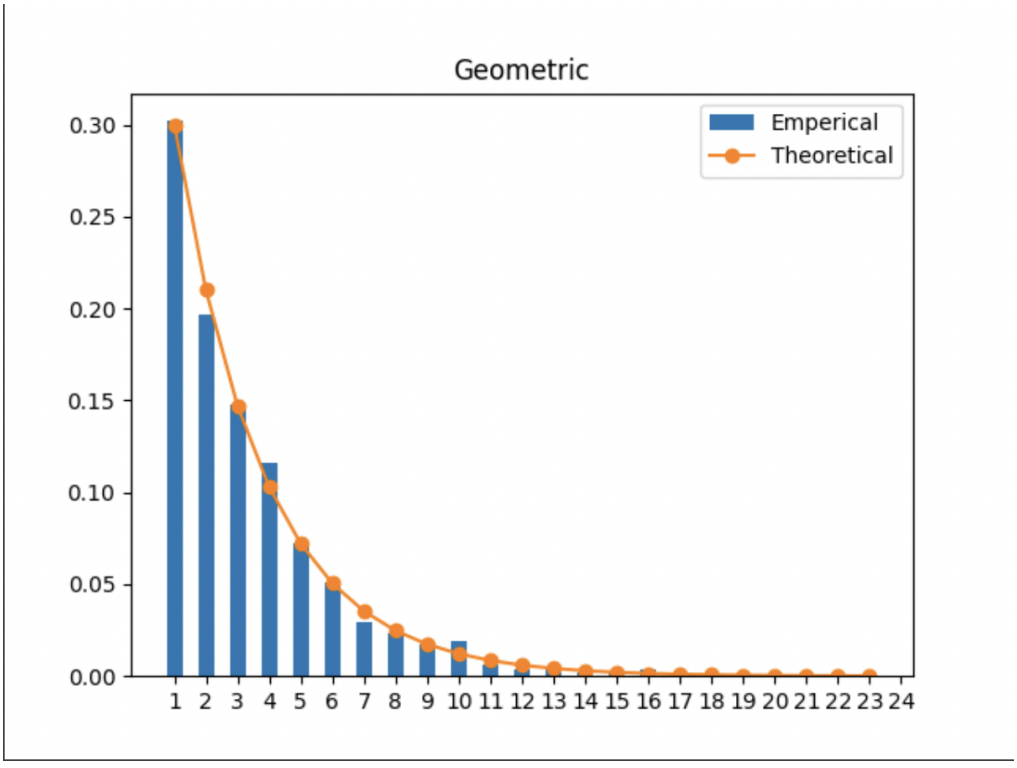

Рисунок 2.3 — Гистограмма значений для геометрического распределения

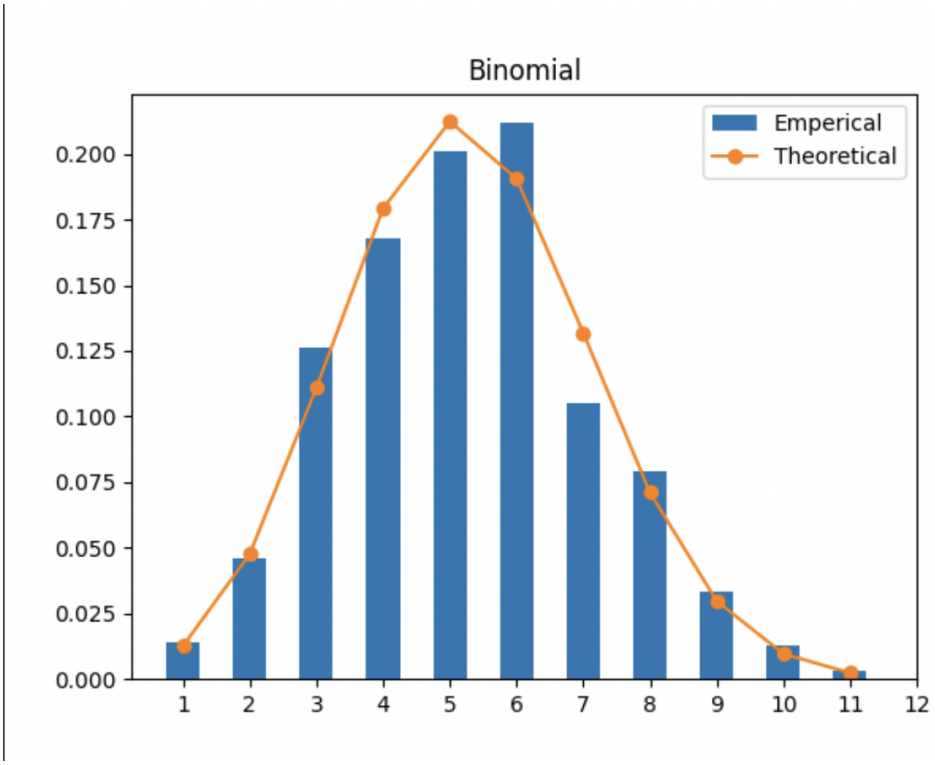

Рисунок 2.4 — Гистограмма значений для биномиального распределения

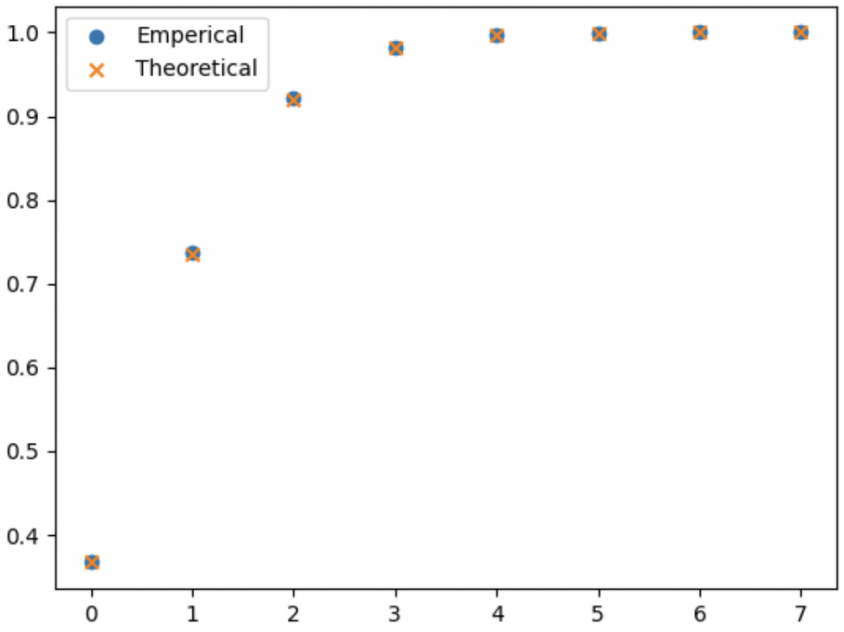

Рисунок 2.5 — Сравнение эмперической и теоретической функций распределения для распределния Пуассона

Лабораторная 3. Моделирование НСВ

3.1 Условие

Необходимо:

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).

Условия варианта:

1. Нормальное $m = 2, s^2 = 16$

2. Экспоненциальное $a = 0.25$

3. Лапласа $a = 1.5$

4. Коши $a = 1, b = 2$

5. Лог-нормальное $m = -1, s^2 = 2$

3.2 Выполнение работы

Исходный код:

```
class NormalGenerator(AbstractRandomGenerator):
    def __init__(self, m: float, s2: float):
        self.m = m
        self.s2 = s2
        self._next = None

    def next_element(self):
        if self._next is not None:
            res = self._next
            self._next = None
        else:
            a1 = random.random()
```

```
            a2 = random.random()
            self._next = (
            math.sqrt(-2 * math.log(a1))
            * math.cos(math.pi * 2 * a2)
            )
            res = (
            math.sqrt(-2 * math.log(a1))
            * math.sin(math.pi * 2 * a2)
            )

        res = self.m + res * math.sqrt(self.s2)
        return res


    def reset(self):
        self.a1 = None
        self.a2 = None


class StandardNormalGenerator(AbstractRandomGenerator):
    def __init__(self):
        self.gen = NormalGenerator(0, 1)

    def next_element(self):
        return self.gen()

    def reset(self):
        return self.gen.reset()


class LogNormalGenerator(AbstractRandomGenerator):
    def __init__(self, m: float, s2: float):
        self.gen = NormalGenerator(m, s2)

    def next_element(self):
        return math.exp(self.gen())

    def reset(self):
        self.gen.reset()


class LaplaceGenerator(AbstractRandomGenerator):
    def __init__(self, a: float):
```

```
        self.a = a

    def next_element(self):
        x = random.random()
        if x < 0.5:
            res = 1 / self.a * math.log(2 * x)
        else:
            res = - 1 / self.a * math.log(2 * (1 - x))
        return res

    def reset(self):
        pass


class ExponentialGenerator(AbstractRandomGenerator):
    def __init__(self, a: float):
        self.a = a

    def next_element(self):
        x = random.random()
        return - 1/ self.a * math.log(x)

    def reset(self):
        pass


class CauchyGenerator(AbstractRandomGenerator):
    def __init__(self, a: float, b: float):
        self.a = a
        self.b = b

    def next_element(self):
        x = random.random()
        res = self.a + self.b * math.tan(math.pi * (x - 0.5))
        return res

    def reset(self):
        pass


E = []
D = []
Et = []
```

```
Dt = []
titles = []

loc = 2
scale = 4
n = NormalGenerator(loc, scale**2)
title = f'Normal({loc}, {scale**2})'
hist(
    n, title
)
generated = [n() for _ in range(10000)]
E.append(mean(generated))
D.append(dispersion(generated))
Et.append(loc)
Dt.append(scale**2)
titles.append(title)

a = 0.25
title = f'Exponential({a})'
exp = ExponentialGenerator(a)
hist(
    exp, title,
)
generated = [exp() for _ in range(10000)]
E.append(mean(generated))
D.append(dispersion(generated))
Et.append(1/a)
Dt.append(1/a**2)
titles.append(title)

a = 1.5
title = f'Laplace({a})'
lapl = LaplaceGenerator(a)
hist(
    lapl, title
)
generated = [lapl() for _ in range(10000)]
E.append(mean(generated))
D.append(dispersion(generated))
Et.append(0)
Dt.append(2 / a**2)
titles.append(title)
```

```
a = 1
b = 2
title = f'Cauchy({a}, {b})'
cauchy = CauchyGenerator(a, b)
generated = [cauchy() for _ in range(10000)]
E.append(mean(generated))
D.append(dispersion(generated))
Et.append(None)
Dt.append(None)
titles.append(title)
generated = [g for g in generated if -10 < g < 10]
plt.hist(generated, label='Emperical')
plt.title(title)
plt.show()

m = -1
s2 = 2
title = f'LogNormal({m}, {s2})'
ln = LogNormalGenerator(m, s2)
hist(
    ln, title
)
generated = [ln() for _ in range(100000)]
e = math.exp(m + s2 / 2)
d = math.exp(2 * m + s2) * (math.exp(s2) - 1)
E.append(mean(generated))
D.append(dispersion(generated))
Et.append(e)
Dt.append(d)
titles.append(title)

data = {
    "Распределение": titles,
    "Мат. ожидание (теоретически)": Et,
    "Мат. ожидание (оценка)": E,
    "Диспресия (теоретически)": Dt,
    "Диспресия (оценка)": D,
}

print(tabulate(
    data, headers="keys",
    missingval="Не определено",
    tablefmt="simple_grid",
```

```
    floatfmt=".4f",
    numalign='center'
))
```

Распределение	Мат. ожидание (теоретически)	Мат. ожидание (оценка)	Диспресия (теоретически)	Диспресия (оценка)
Normal(2, 16)	2.0000	1.9972	16.0000	16.4093
Exponential(0.25)	4.0000	3.9654	16.0000	16.2141
Laplace(1.5)	0.0000	-0.0029	0.8889	0.8931
Cauchy(1, 2)	Не определено	-0.9220	Не определено	6021.2044
LogNormal(-1, 2)	1.0000	1.0021	6.3891	5.7798

Рисунок 3.1 — Сравнение теоретических значений мат. ожиданий и дисперсии с полученными оценками

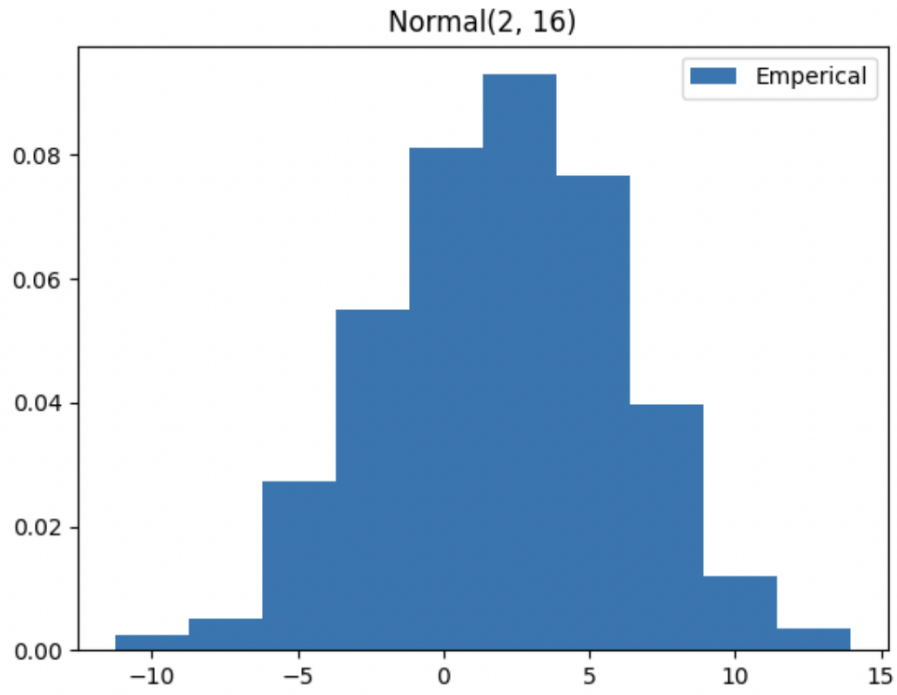

Рисунок 3.2 — Гистограмма значений для нормального распределения

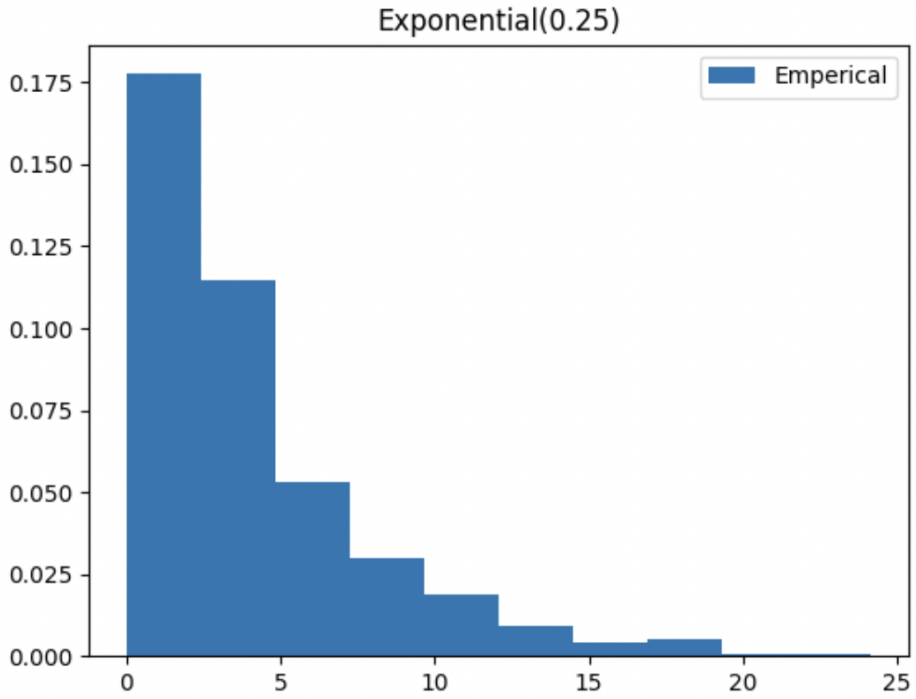

Рисунок 3.3 — Гистограмма значений для экспоненциального распределения

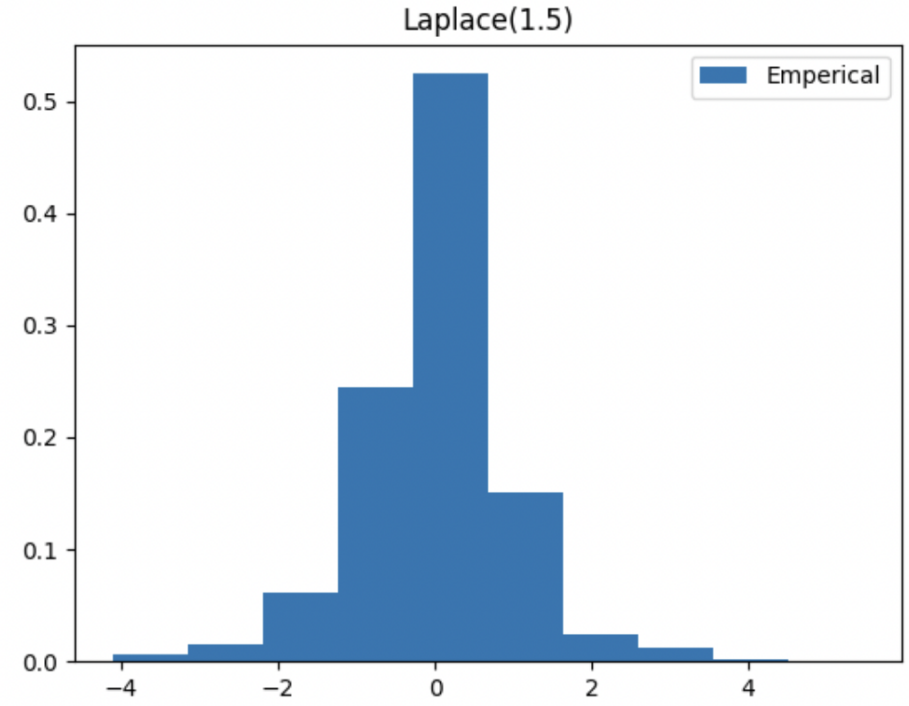

Рисунок 3.4 — Гистограмма значений для распределения Лапласа

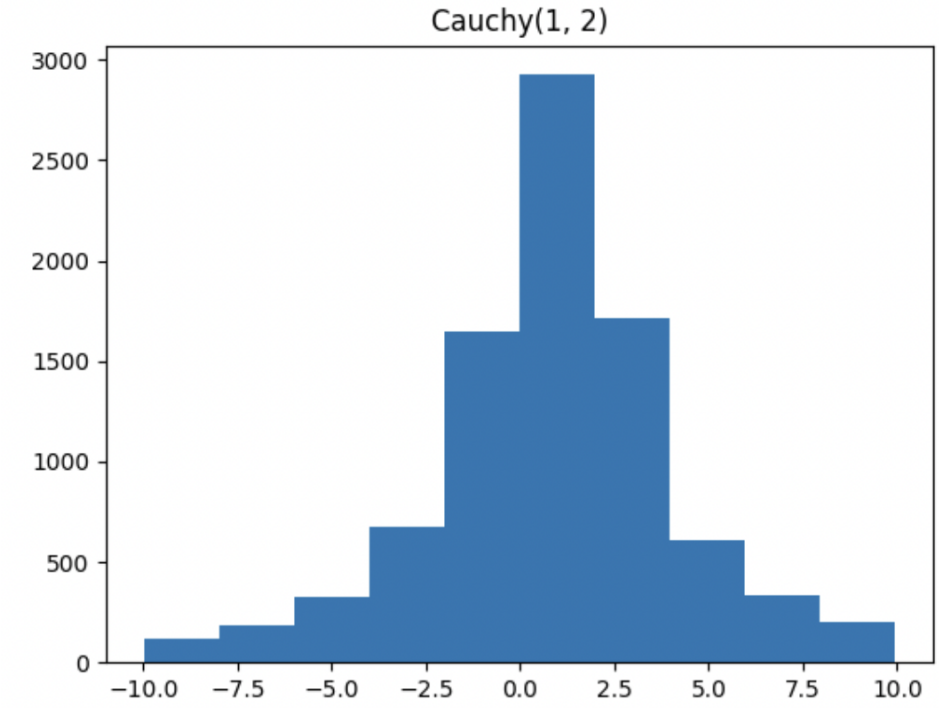

Рисунок 3.5 — Гистограмма значений для распределения Коши

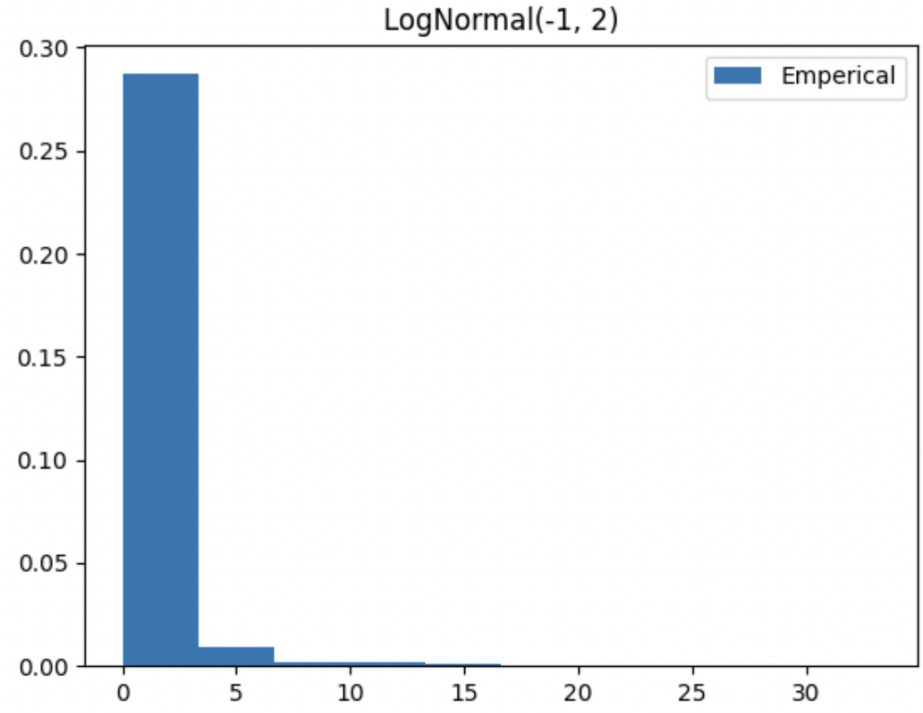

Рисунок 3.6 — Гистограмма значений для лог-нормального распределения

Лабораторная 4. Метод Монте-Карло вычисления интегралов

4.1 Условие

Необходимо:

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.

Условие варианта:

$$\int\limits_{-\pi\sqrt{e}}^{e\sqrt{\pi}} |sin(x)|arctg(|x|^3)dx$$

$$\iint\limits_{|x|<2, x^2 \le y \le 4} \frac{\sqrt{y + sin^2(x)}}{e^{-xy}} dxdy$$

4.2 Выполнение работы

Исходный код

```
import math
import random
import matplotlib.pyplot as plt
import numpy as np
from tqdm.auto import tqdm

def f1(x):
    return (
        abs(math.sin(x)) * math.atan(abs(x) ** 3)
    )
```

```
def f2(x, y):
    return math.sqrt(y + math.sin(x) ** 2) / math.exp(-x*y)


def task1(n_its):
    lower = -math.sqrt(math.e) * math.pi
    upper = math.sqrt(math.pi) * math.e
    fn = [
        f1(x) for x in [random.uniform(lower, upper) for _ in range(n_
    ]
    return (upper - lower) * sum(fn) / len(fn)


def task2(n_its):
    x_lower = -2
    x_upper = 2
    y_upper = 4
    Xs = []
    Ys = []
    while(len(Xs) != n_its):
        x = random.uniform(x_lower, x_upper)
        y = random.uniform(0, y_upper)
        if y >= x**2:
            Ys.append(y)
            Xs.append(x)
    fn = [f2(x, y) for x, y in zip(Xs, Ys)]
    mean_fn = np.array(fn).mean()
    S = 32 / 3
    return S * mean_fn


#Task 1
iters = np.linspace(100, 10**5, 100).astype(np.int32)
vals = np.array([task1(nit) for nit in tqdm(iters)])
theor_val = 8.39794
errors = abs(theor_val - vals)
plt.plot(iters, errors)
plt.xlabel('Number of iterations')
plt.ylabel('Absolute error')
plt.title('Task 1')
plt.show()
print(f'Calculated value: {task1(10**6):.5f}')
print(f'Theoretical value: {theor_val:.5f}')
```

```
# Task 2
iters = np.linspace(100, 10**5, 100).astype(np.int32)
vals = np.array([task2(nit) for nit in tqdm(iters)])
theor_val = 618.542
errors = abs(theor_val - vals)
plt.plot(iters, errors)
plt.xlabel('Number of iterations')
plt.ylabel('Absolute error')
plt.title('Task 2')
plt.show()
print(f'Calculated value: {task2(10**6):.5f}')
print(f'Theoretical value: {theor_val:.5f}')
```

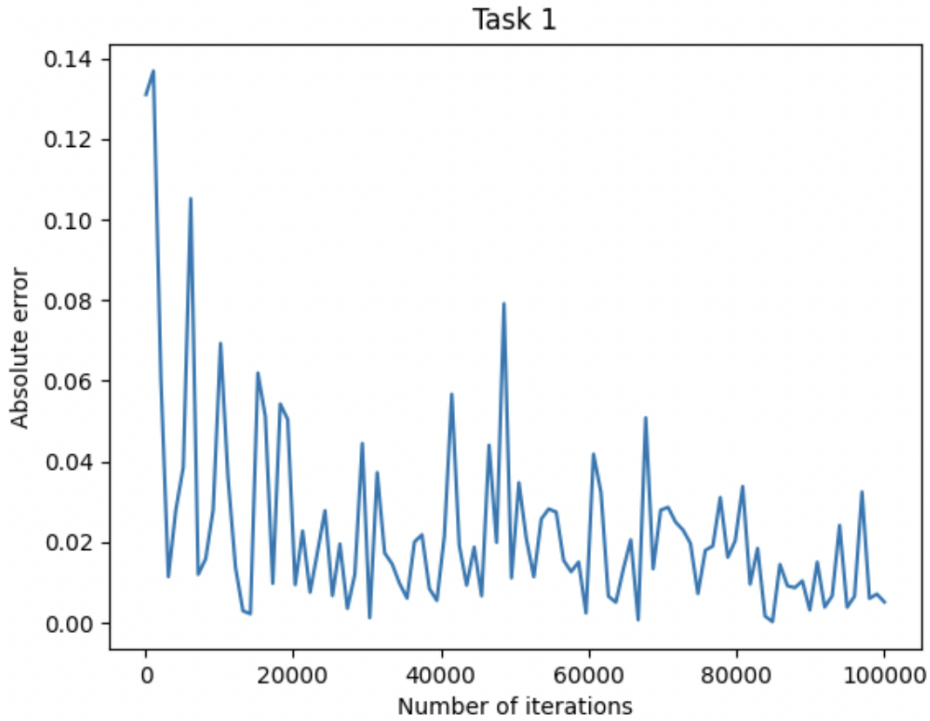

Рисунок 4.1 — Зависимость точности вычисления интеграла от количества итераций (одномерный интеграл)

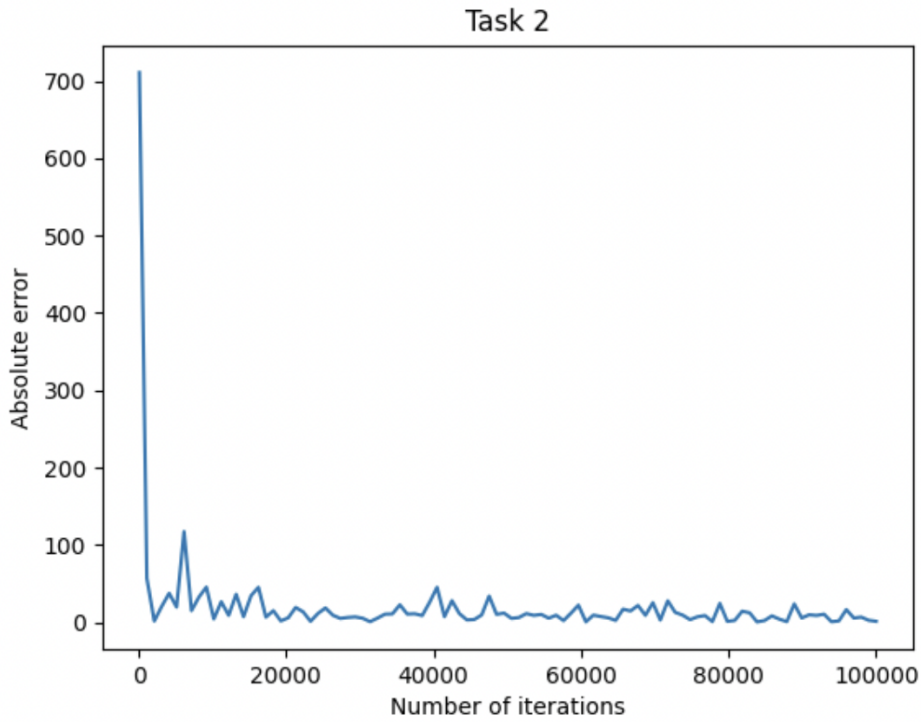

Рисунок 4.2 — Зависимость точности вычисления интеграла от количества итераций (двумерный интеграл)

Лабораторная 5. Метод Монте-Карло решения СЛАУ

5.1 Условие

Необходимо методом Монте-Карло решить систему

$$\begin{bmatrix} 0.7 & -0.2 & 0.3 \\ 0.4 & 1.3 & 0.1 \\ 0.2 & 0.1 & 1.1 \end{bmatrix} x = \begin{bmatrix} 3 \\ 1 \\ 1 \end{bmatrix}$$

5.2 Выполнение работы

Исходный код:

```
import math

import numpy as np
import random
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm
sns.set()
from matplotlib import patches


def get_markov_chain(n: int, len_: int):
    pi = np.ones((n,), dtype=np.float32) * 1 / n
    P = np.ones((n, n), dtype=np.float32) * 1 / n
    return markov_chain(pi, P, len_), pi, P


def get_observation(pi):
    x = random.random()
    st = 0.0
    i = 0
    for v in pi:
        if st <= x <= st + v:
            return i
        i += 1
        st += v
```

```
def markov_chain(pi, P, len_):
    observations = [get_observation(pi)]
    for _ in range(len_ - 1):
        current_observ = observations[-1]
        observations.append(get_observation(P[current_observ]))
    return observations


def one_chain_realization(A, f, h, markov_len):
    mc, pi, P = get_markov_chain(A.shape[0], markov_len)

    qs = []
    if pi[mc[0]] > 0:
        qs.append(h[mc[0]] / pi[mc[0]])
    else:
        qs.append(0)

    for k in range(1, markov_len):
        i = mc[k]
        i_prev = mc[k-1]
        if P[i_prev, i] > 0:
            g = A[i_prev, i] / P [i_prev, i]
        else:
            g = 0
        qs.append(qs[-1] * g)

    sum_ = 0.0
    for k in range(markov_len):
        sum_ += qs[k] * f[mc[k]]

    return sum_



def solve_one_variable(A,f, h , markov_len, num_realizations=10000):
    xis = [
    one_chain_realization(A, f, h, markov_len) for _ in tqdm(
    range(num_realizations),
    total=num_realizations,
    desc='Markov chain realizations',
    leave=False
```

```
    )]
    return np.array(xis).mean()

def solve_monte_carlo(A: np.ndarray, f: np.ndarray, markov_len):
    mc_sol = np.zeros((A.shape[0],))
    h = np.identity(A.shape[0])
    for i in range(A.shape[0]):
        mc_sol[i] = solve_one_variable(A, f, h[i], markov_len)
    return mc_sol

A = np.array([
     [0.7, -0.2, 0.3],
     [0.4, 1.3, 0.1],
     [0.2, 0.1, 1.1]
 ])
 b = np.array([3, 1, 1])
 correct_solution = np.linalg.solve(A, b)
 markov_lens = np.linspace(10, 100, 10, dtype=np.int32)
 A = np.identity(A.shape[0]) - A
 f = b
 print(A)
 print(np.linalg.norm(A))
 solutions = np.array([
 solve_monte_carlo(A, f, ml) for ml in tqdm(
 markov_lens, desc='Length of markov chain', leave=False)
 ])
 r = [np.linalg.norm(sol - correct_solution) for sol in solutions]
 print(f'Точное решение: {correct_solution}')
 print(f'Решение методом Монте Карло: {solutions[np.argmin(r)]}')
 plt.plot(markov_lens, r)
 plt.xlabel('Длина цепи Маркова')
 plt.ylabel('Невязка решения')
 plt.show()
```

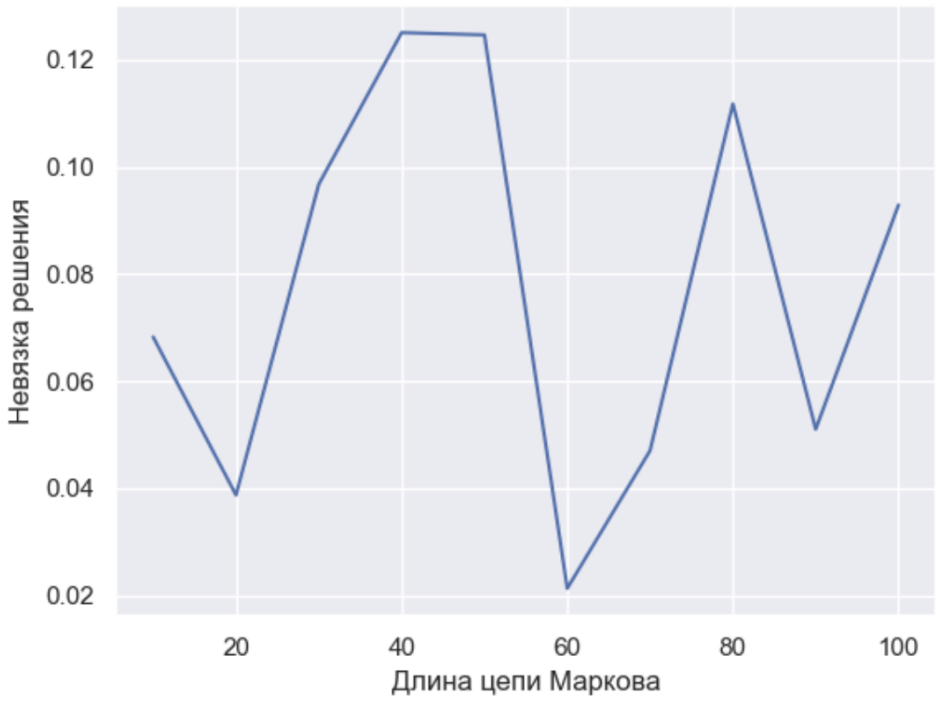

Рисунок 5.1 — Зависимость точности решения СЛАУ от длины генерируемой цепи Маркова