Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Комп’ютерний практикум №1

Моделювання систем

Тема: Перевірка генератора випадкових чисел на відповідність закону розподілу

Виконав
студент групи ІП-11:
Панченко С. В.

Перевірив:
Стеценко І.В.

Київ 2024

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Дослідити та перевірити різні методи генерації випадкових чисел на відповідність заданим законам розподілу шляхом статистичного аналізу та застосування критерію згоди χ^2.

2 ЗАВДАННЯ

- ✓ Згенерувати 10000 випадкових чисел трьома вказаними нижче способами. **45 балів**.
  - Згенерувати випадкове число за формулою $x_i = -\frac{1}{\lambda}\ln\xi_i$, де ξ_i - випадкове число, рівномірно розподілене в інтервалі (0;1). Числа ξ_i можна створювати за допомогою вбудованого в мову програмування генератора випадкових чисел. Перевірити на відповідність експоненційному закону розподілу $F(x) = 1 - e^{-\lambda x}$. Перевірку зробити при різних значеннях λ.
  - Згенерувати випадкове число за формулами:

$$x_i = \sigma\mu_i + a$$

$$\mu_i = \sum_{i=1}^{12}\xi_i - 6,$$

    де ξ_i - випадкове число, рівномірно розподілене в інтервалі (0;1). Числа ξ_i можна створювати за допомогою убудованого в мову програмування генератора випадкових чисел. Перевірити на відповідність нормальному закону розподілу:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{(x-a)^2}{2\sigma^2}\right).$$

    Перевірку зробити при різних значеннях a і σ.
  - Згенерувати випадкове число за формулою $z_{i+1} = az_i \pmod c$, $x_{+1i} = z_{i+1}/c$, де $a=5^{13}$, $c=2^{31}$. Перевірити на відповідність рівномірному закону розподілу в інтервалі (0;1). Перевірку зробити при різних значеннях параметрів a і c.

- ✓ Для кожного побудованого генератора випадкових чисел побудувати гістограму частот, знайти середнє і дисперсію цих випадкових чисел. По виду гістограми частот визначити вид закону розподілу. **20 балів.**
- ✓ Відповідність заданому закону розподілу перевірити за допомогою критерію згоди χ^2. **30 балів**
- ✓ Зробити висновки щодо запропонованих способів генерування випадкових величин. **5 балів**

3 ВИКОНАННЯ

Наведемо нижче результати виконання програми для різних параметрів:

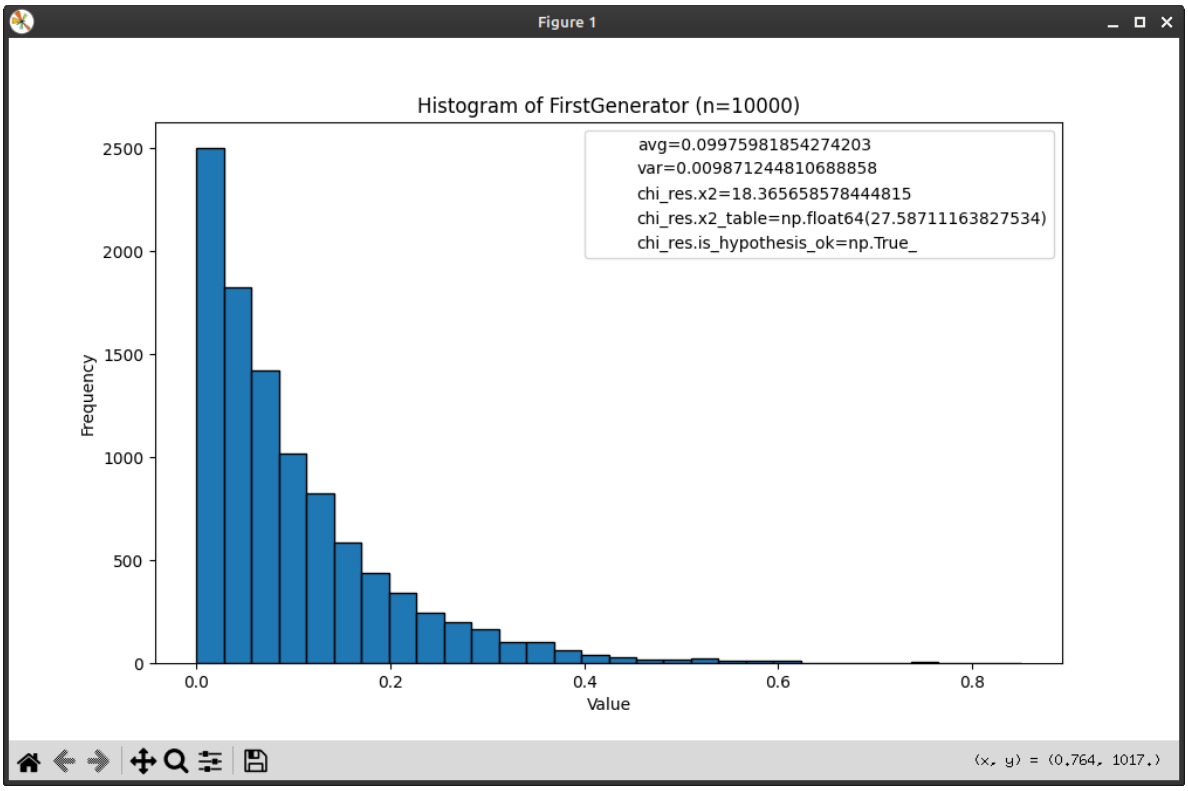

Рисунок 3.1 — Перший генератор, lamda = 10

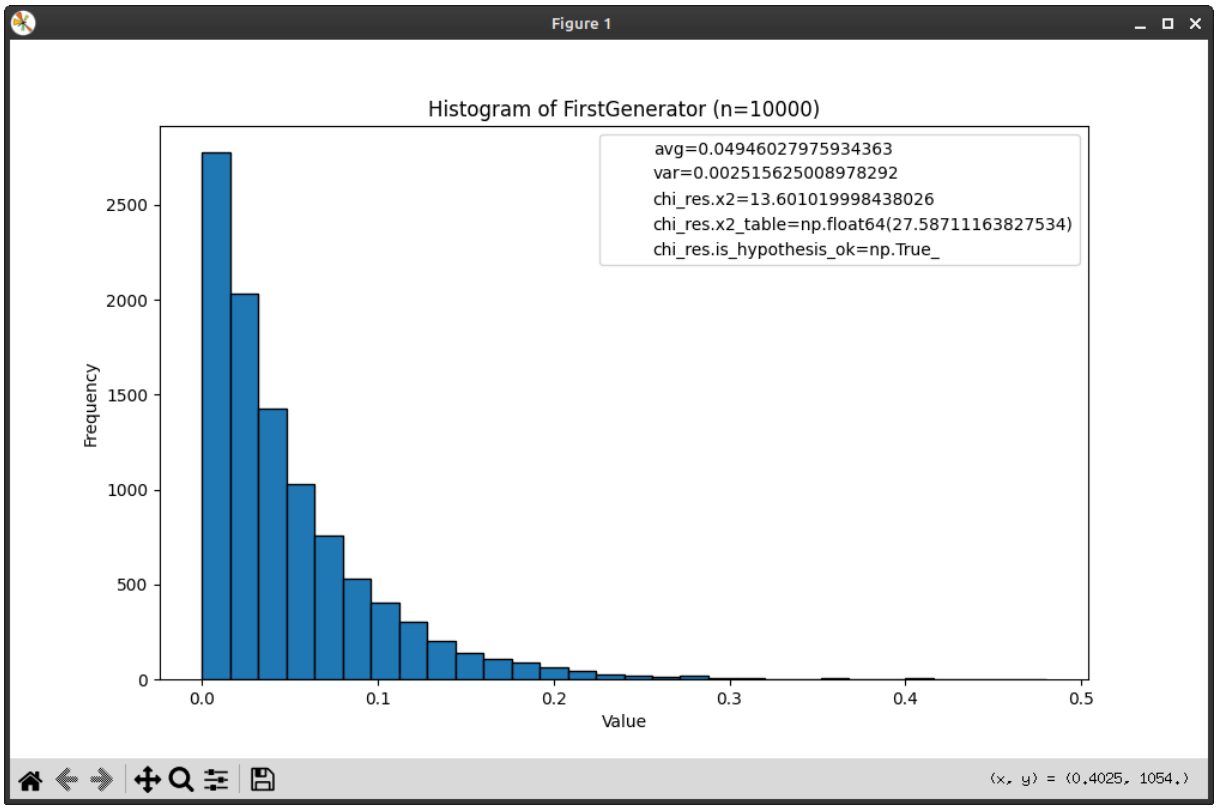

Рисунок 3.2 — Перший генератор, lamda = 20

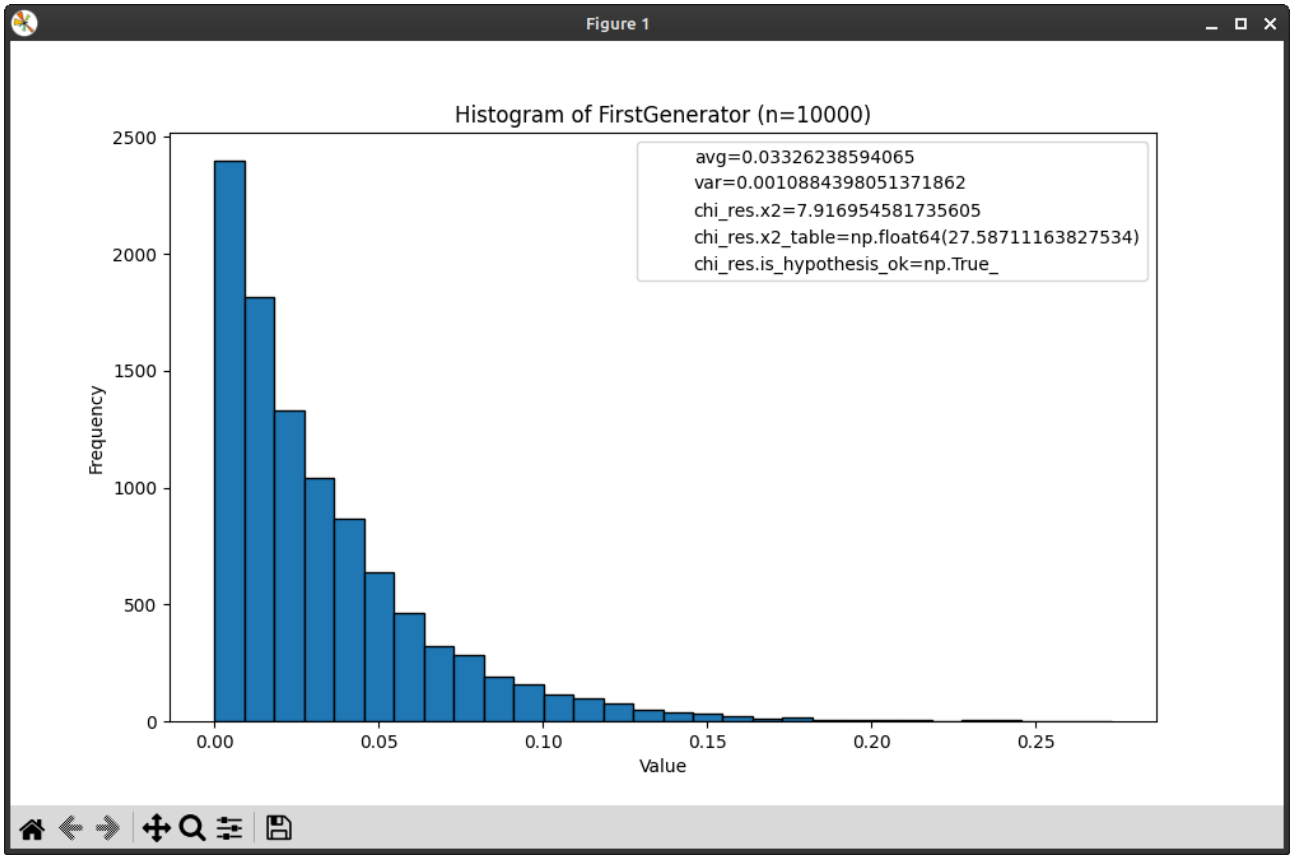

Рисунок 3.3 — Перший генератор, lamda = 30

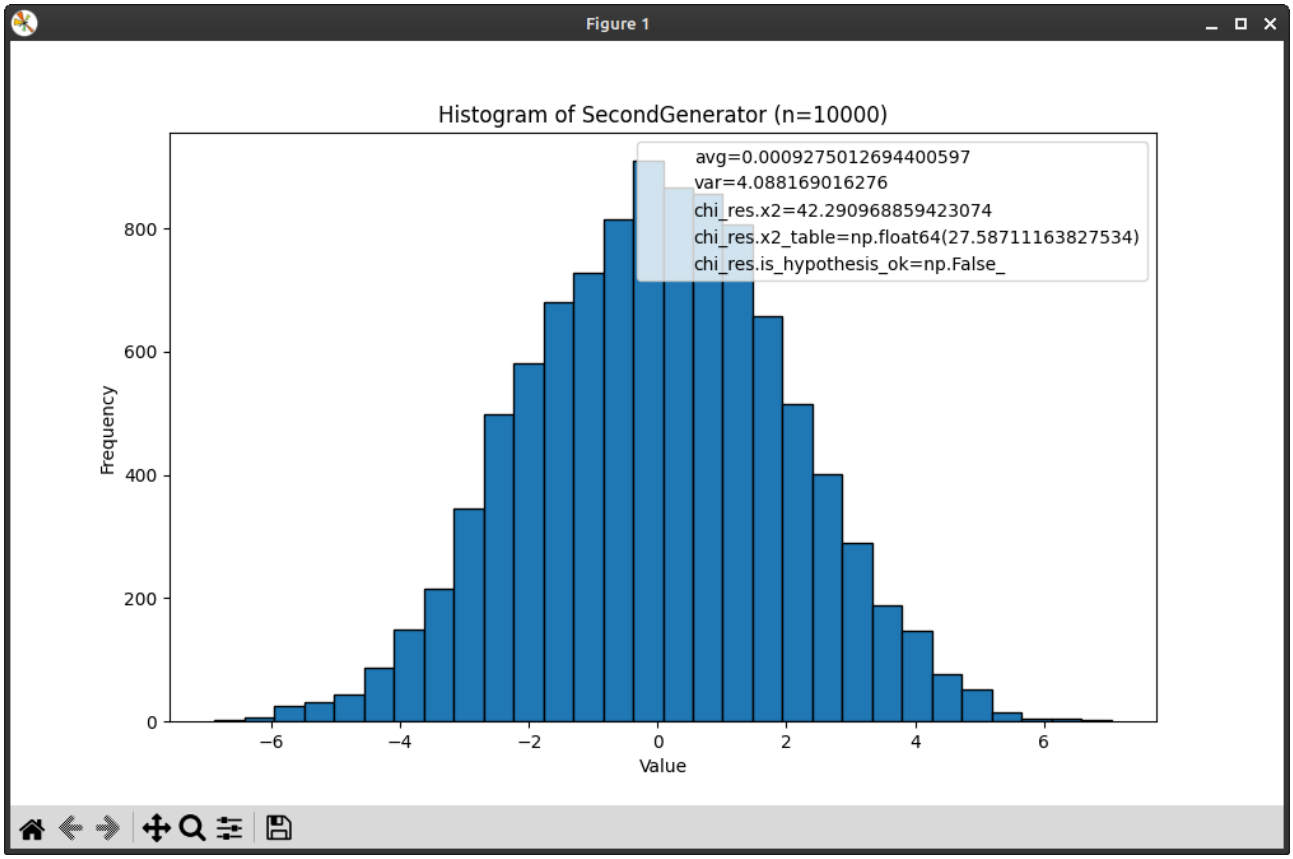

Рисунок 3.4 — Другий генератор, alpha = 0, sigma = 2

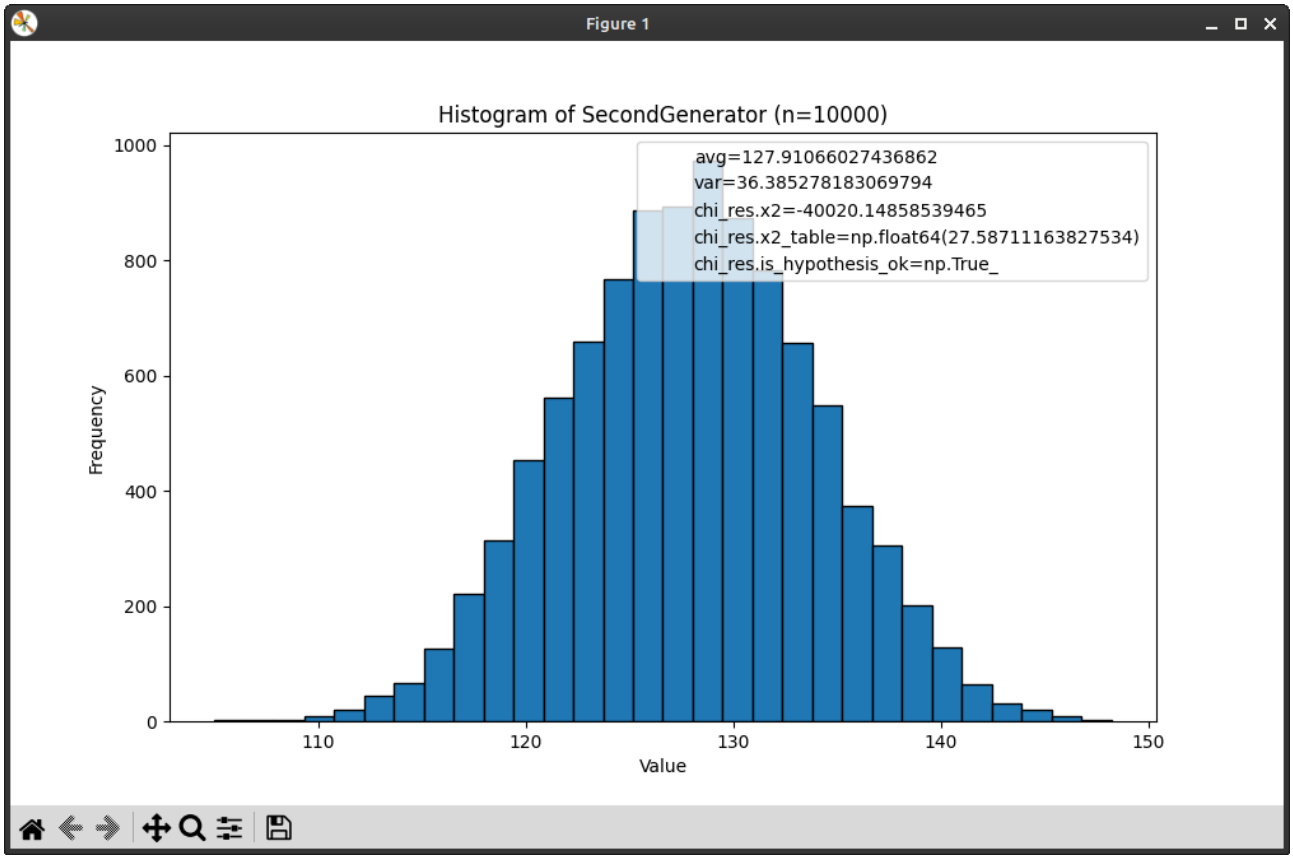

Рисунок 3.5 — Другий генератор, alpha = 128, sigma = -6

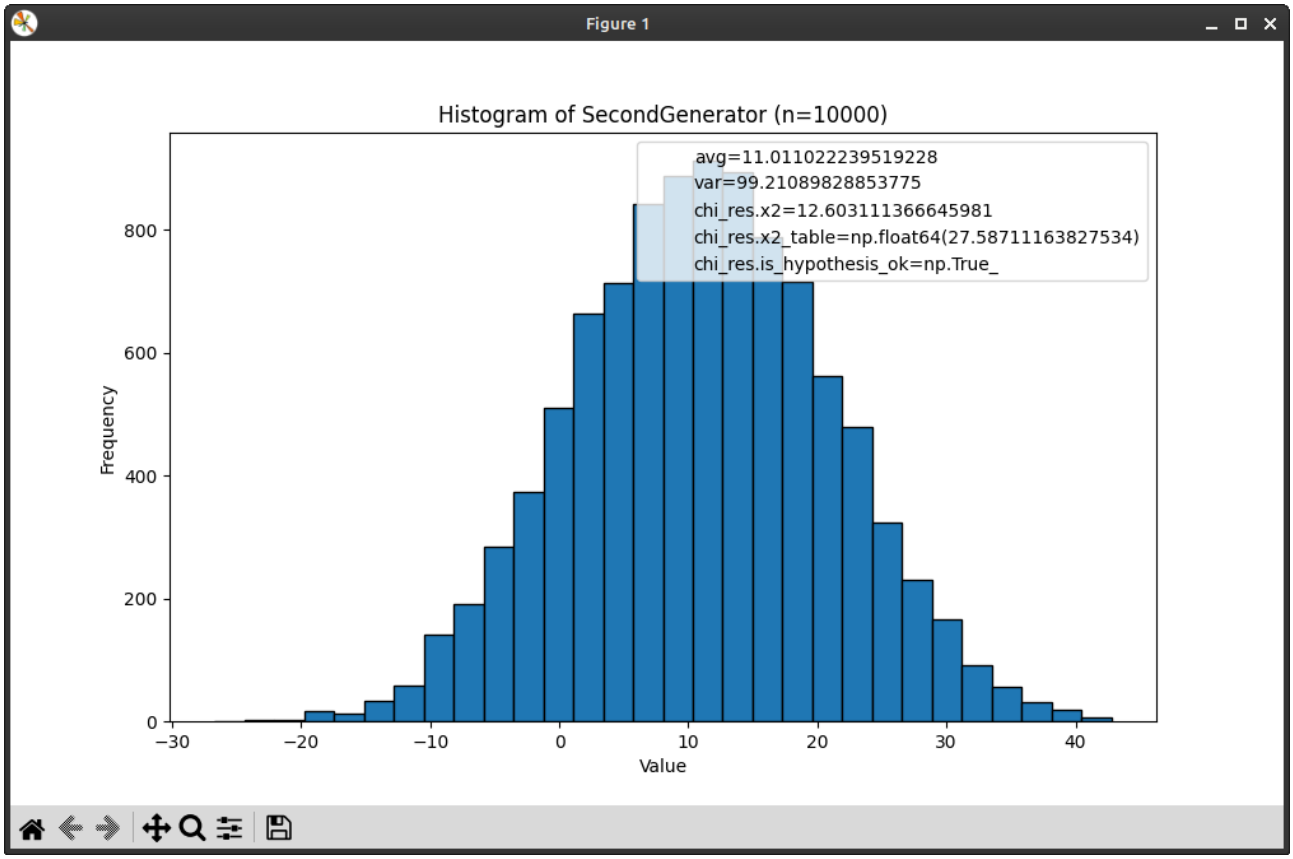

Рисунок 3.6 — Другий генератор, alpha = 11, sigma = 10

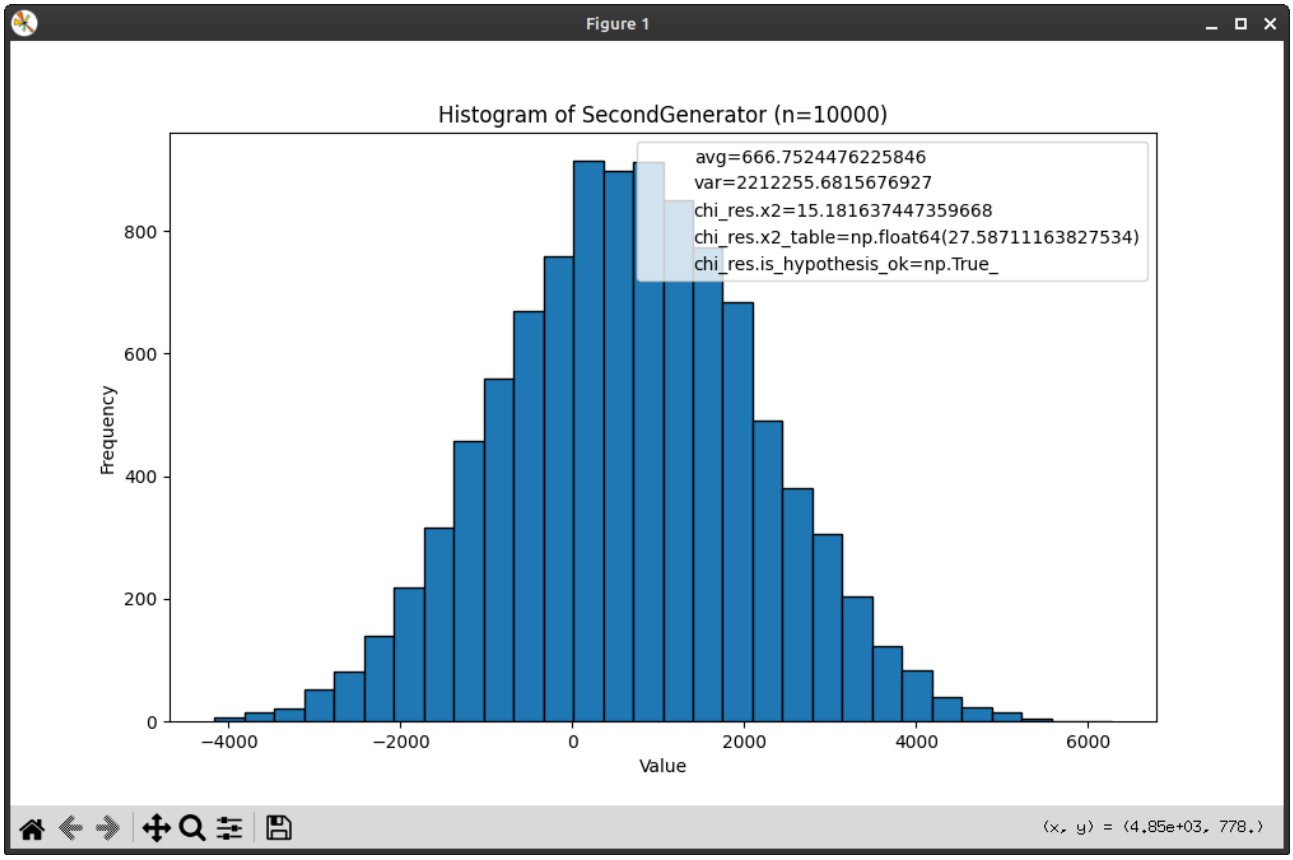

Рисунок 3.7 — Другий генератор, alpha = 674, sigma = 1478

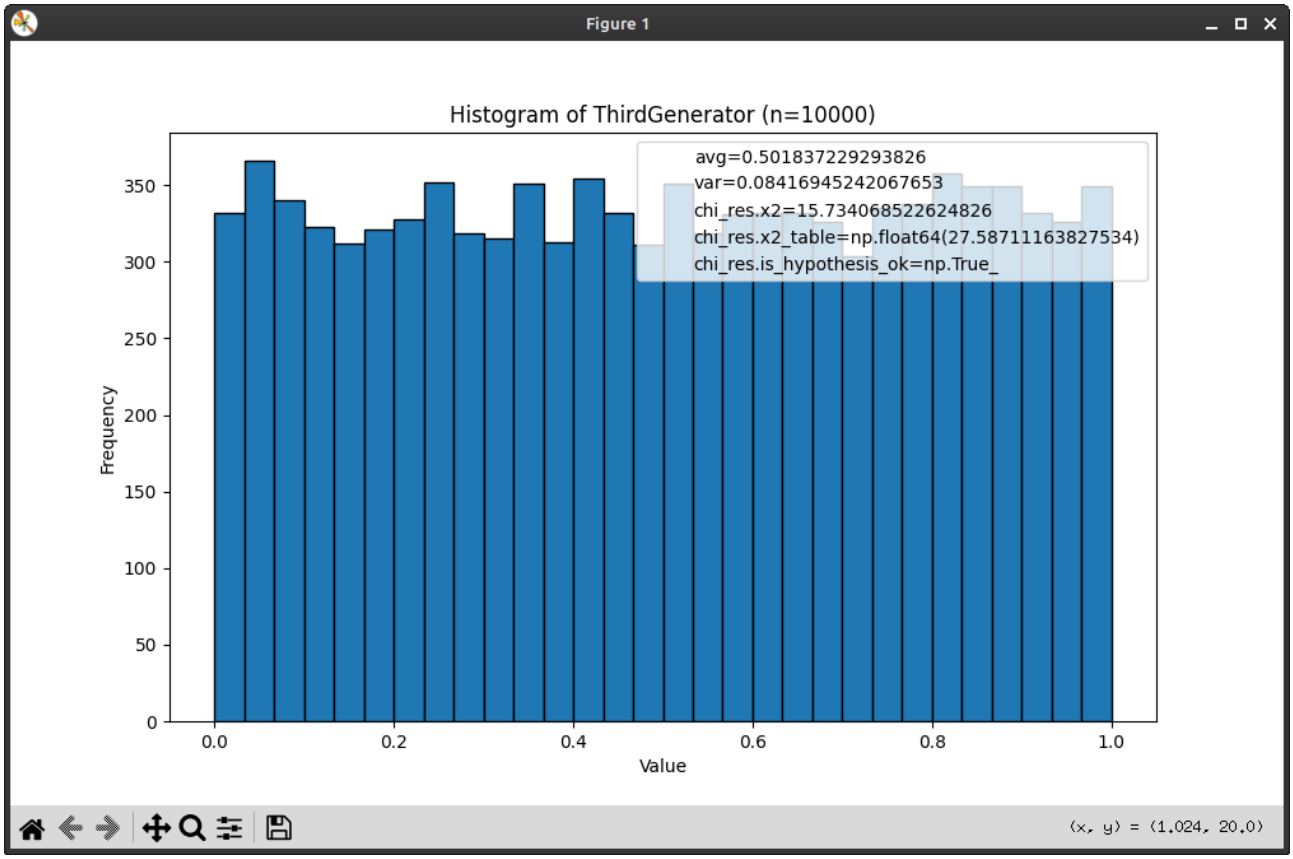

Рисунок 3.8 — Другий генератор, a = 5^13, c = 2^31

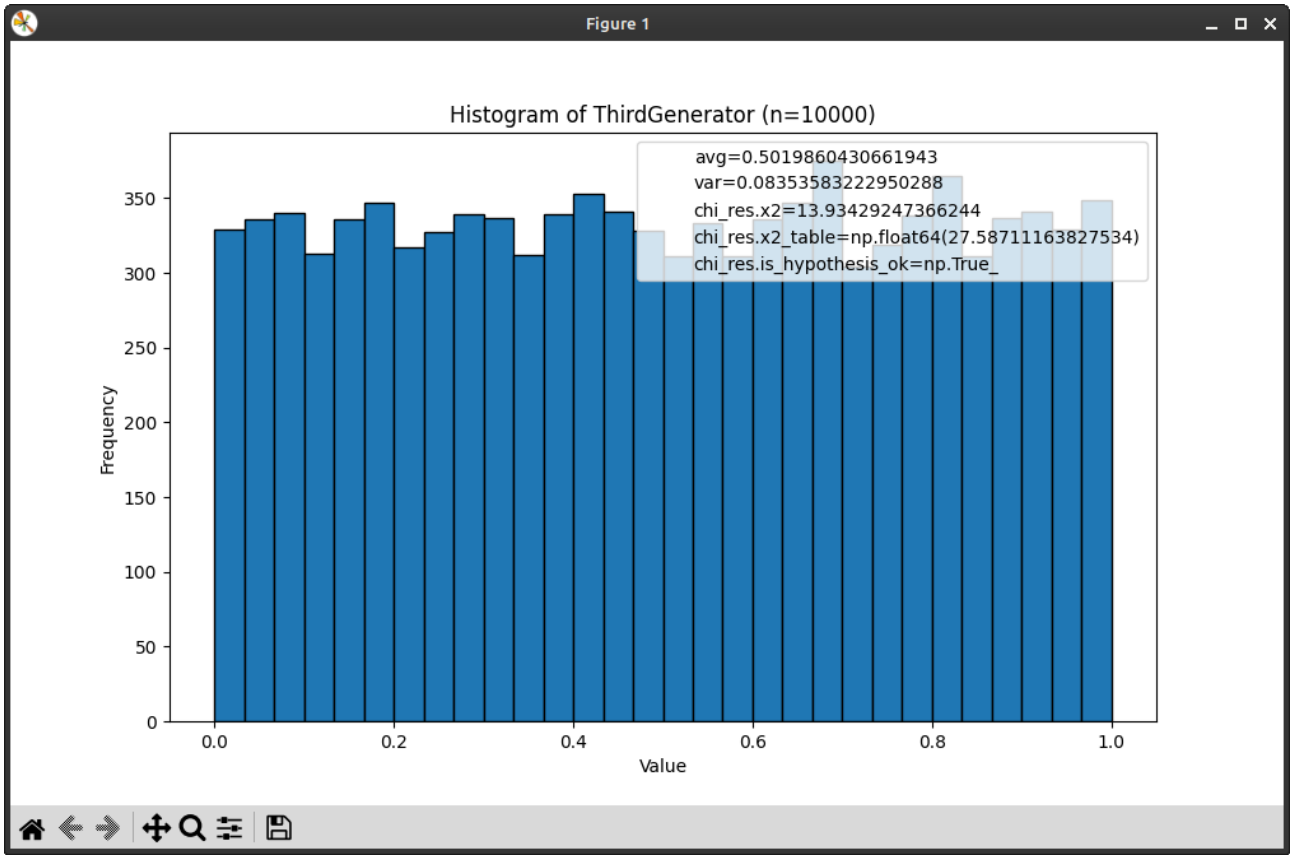

Рисунок 3.9 — Другий генератор, a = 2^31, c = 5^13

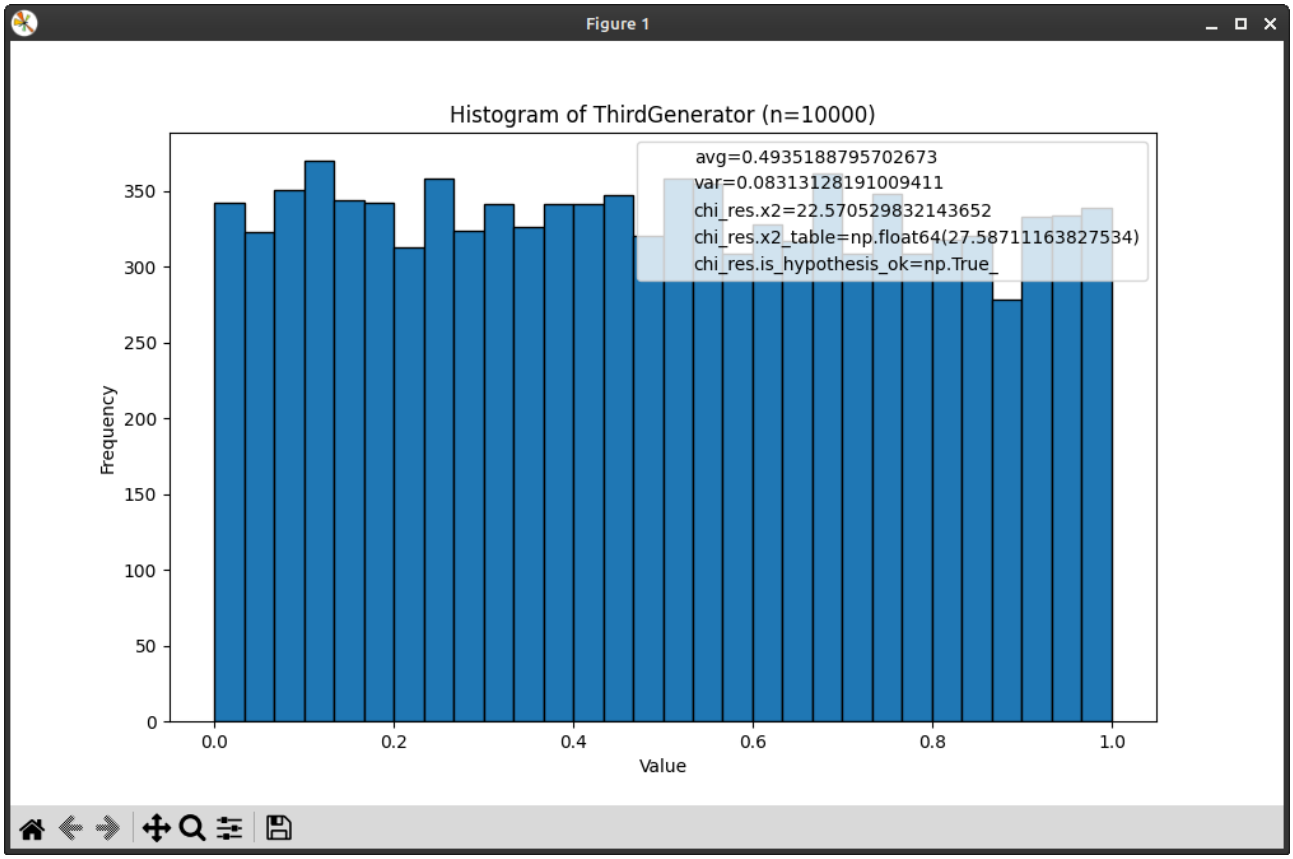

Рисунок 3.10 — Другий генератор, a = 2^4, c = 6^13

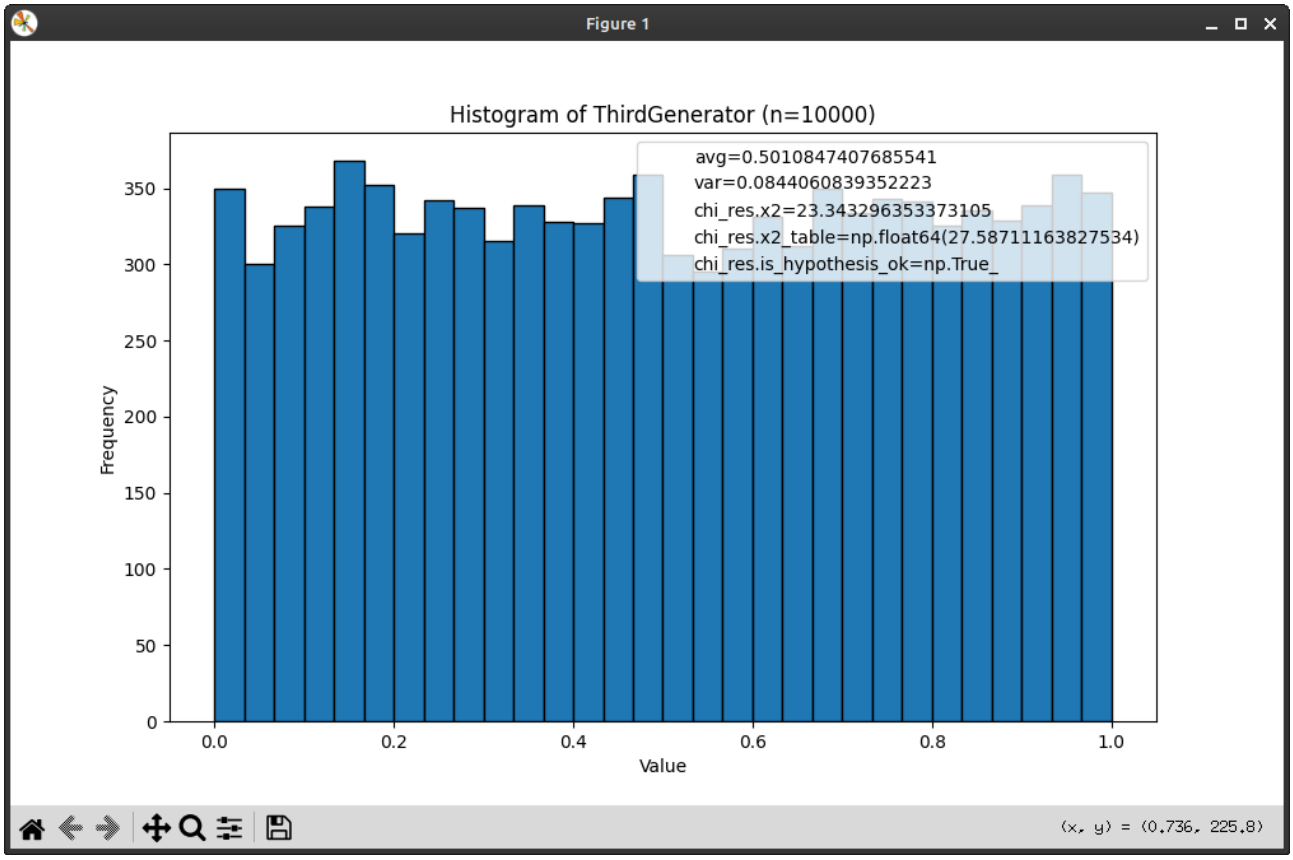

Рисунок 3.11 — Другий генератор, a = 4^7, c = 7^9

ДОДАТОК А ТЕКСТИ ПРОГРАМНОГО КОДУ

Тексти програмного коду

(Найменування програми (документа))

Жорсткий диск

(Вид носія даних)

(Обсяг програми (документа), арк.)

Студента групи ІП-11 4 курсу

Панченка С. В

```
import math
import attr
import random
import numpy as np
import matplotlib.pyplot as plt
from typing import TypeAlias, Callable, Protocol
from scipy import stats

IntegralFunc: TypeAlias = Callable[[float, float], float]

@attr.frozen
class FirstGenerator:
    lam: float

    def generate_number(self) -> float:
        return (-1 / self.lam) * math.log(random.random())

    def calculate_distribution(self, x: float) -> float:
        return 1 - math.exp(-self.lam * x)

@attr.frozen
class SecondGenerator:
    alpha: float
    sigma: float

    def generate_number(self) -> float:
        u = sum([random.random() for _ in range(1, 13)]) - 6
        return self.sigma * u + self.alpha

    def calculate_distribution(self, x: float) -> float:
        return ( 1 + math.erf((x - self.alpha)/(math.sqrt(2) * self.sigma))) / 2
```

```
@attr.mutable
class ThirdGenerator:
    a: float
    c: float
    z: float = attr.field(factory=random.random)

    def generate_number(self) -> float:
        self.z = math.fmod(self.a * self.z, self.c)
        return self.z / self.c

    def calculate_distribution(self, x: float) -> float:
        if x < 0:
            return 0
        elif x > 1:
            return 1
        else:
            return x

class GeneratorProtocol(Protocol):
    def generate_number(self) -> float:
        ...

    def calculate_distribution(self, x: float) -> float:
        ...

@attr.frozen
class CalcCountsInIntervalResult:
    data: list[int]
    interval_size: float
```

```
def calc_counts_in_interval(
    numbers: list[float],
    interval_count: np.uint32
) -> CalcCountsInIntervalResult:
    min_val = min(numbers)
    max_val = max(numbers)
    interval_size = float((max_val - min_val) / interval_count)
    counts_in_interval = [0 for _ in numbers]
    for number in numbers:
        index = int((number - min_val) / interval_size)
        if number == max_val:
            index -= 1
        counts_in_interval[index] += 1
    return CalcCountsInIntervalResult(counts_in_interval, interval_size)

class CalculateDistributionProtocol(Protocol):
    def calculate_distribution(self, x: float) -> float:
        ...

def calc_chi_value(
    calculate_distribution: Callable[[float], float],
    numbers: list[float],
    interval_count: np.uint32
):
    counts_in_interval = calc_counts_in_interval(numbers, interval_count)
    min_val = min(numbers)

    x2 = 0
    count_in_interval = 0
    left_index = 0
```

```
    for i in range(interval_count):
        count_in_interval += counts_in_interval.data[i]
        if count_in_interval < 5 and i != interval_count - 1:
            continue

        left = min_val + counts_in_interval.interval_size * left_index
        right = min_val + counts_in_interval.interval_size * (i + 1)
                    expected_count = len(numbers) * (calculate_distribution(right) -
calculate_distribution(left))

        x2 += (count_in_interval - expected_count) ** 2 / expected_count

        left_index = i + 1
        count_in_interval = 0
    return x2

@attr.frozen
class CheckChiValueRes:
    x2: float
    x2_table: float
    is_hypothesis_ok: bool

def check_chi_value(
    calculate_distribution: Callable[[float], float],
    numbers: list[float],
    interval_count: np.uint32,
    significance_level: float
) -> CheckChiValueRes:
    x2 = calc_chi_value(calculate_distribution, numbers, interval_count)
    degrees_of_freedom = interval_count - 1 - 2
     x2_table: float = stats.chi2.ppf(1 - significance_level, degrees_of_freedom) #type:
```

```
ignore
    return CheckChiValueRes(x2, x2_table, x2 < x2_table)

def calc_average(numbers: list[float]) -> float:
    return sum(numbers) / len(numbers)

def calc_variance(numbers: list[float]) -> float:
    avg = calc_average(numbers)
    return sum([math.pow(x - avg, 2) for x in numbers]) / len(numbers)

def write_to_legend(message: str) -> None:
    plt.plot([], [], ' ', label=message) # type: ignore

def analyze_generator(
    generator: GeneratorProtocol,
    number_count: np.uint32,
    interval_count: np.uint32,
    significance_level: float
) -> None:
    numbers = [generator.generate_number() for _ in range(number_count)]

    avg = calc_average(numbers)
    var = calc_variance(numbers)
            chi_res = check_chi_value(generator.calculate_distribution, numbers,
interval_count, significance_level)

    plt.figure(figsize=(10, 6)) # type: ignore
    plt.hist(numbers, bins=30, edgecolor='black') # type: ignore
     plt.title(f'Histogram of {generator.__class__.__name__} (n={number_count})') #
type: ignore
    plt.xlabel('Value') #type: ignore
```

```
    plt.ylabel('Frequency') #type: ignore

    write_to_legend(f'{avg=}')
    write_to_legend(f'{var=}')
    write_to_legend(f'{chi_res.x2=}')
    write_to_legend(f'{chi_res.x2_table=}')
    write_to_legend(f'{chi_res.is_hypothesis_ok=}')
    plt.legend(loc='upper right') # type: ignore

    plt.show() #type: ignore

def main() -> None:
    number_count = np.uint32(10000)
    interval_count = np.uint32(20)
    significance_level = 0.05

    generators: list[GeneratorProtocol] = [
        FirstGenerator(10),
        FirstGenerator(20),
        FirstGenerator(30),

        SecondGenerator(0, 2),
        SecondGenerator(128, -6),
        SecondGenerator(11, 10),
        SecondGenerator(674, 1478),

        ThirdGenerator(5 ** 13, 2 ** 31),
        ThirdGenerator(2 ** 31, 5 ** 13),
        ThirdGenerator(3 ** 4, 6 ** 13),
        ThirdGenerator(4 ** 7, 7 ** 9),
    ]
```

```
    for gen in generators:
        analyze_generator(gen, number_count, interval_count, significance_level)

if __name__ == '__main__':
    main()
```