НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

Звіт по лабораторній роботі №1
«Моделювання систем»
«Перевірка генератора випадкових чисел на відповідність закону розподілу»

Студент: Галько М.В.
Група: ІП-01

Київ, 2023

ЗМІСТ

Завдання до практичної роботи

- ✓ 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**

Виконання роботи

Архітектура програми

Для реалізації програми необхідно визначити її архітектуру. Оскільки маємо справу з генераціями чисел із різним розподілом, необхідно створити абстрактний клас Генератор і його реалізації в залежності від розподілу. Також, маємо виконати перевірку згенерованих чисел. Кожному розподілу відповідає формула для його перевірки. Цю функцію можемо також визначити у Генераторах.

Щодо самої перевірки, необхідним етапом є розділення чисел на інтервали для їх майбутньої перевірки. Для цієї задачі створимо клас Інтервал, у якому визначимо атрибути для збереження інтервалів та допоміжні функціїї для роботи із ними.

Також, для перевірки потребуємо вирахування Chi^2 параметру. Саме вирахування є однією функцією тому не потребуємо додаткового класу. Однак, готове число необхідно порівнювати із табличним значенням. Тому створимо допоміжних клас для утримання цих даних та декілька методів для порівняння chi обрахованого та табличного.

В кінці кінців, для проведення повного аналізу та реалізації кожного з етапів (генерація, отримання розподілу, розбиття на інтервали, обрахунок chi параметру, перевірка відповідності самого розподілу) розробимо загальний клас GeneratorAnalyser.

Отже отримуємо наступну структуру:

Клас GeneratorAnalyser

Клас відповідає за аналіз генераторів випадкових чисел та обробку їх результатів.

Важливі атрибути:

1) _numbers: список випадкових чисел, згенерованих генератором;
2) _intervals: список інтервалів, у які розбиваються згенеровані числа;
3) _unitedIntervals: список об'єднаних інтервалів після аналізу.

Важливі методи:

1) RunFullAnalysis: метод для запуску аналізу генератора (генерування чисел, їх розбиття на інтервали, об'єднання малих інтервалів, обчислення хі-квадрат, вивід результатів);
2) GetChiAndCheck: обчислення хі-квадрат та перевірка його відповідність критерію згоди;
3) PrintIntervals: виведення інтервалів та їх кількості.

Клас Interval

Клас відповідає за роботу з інтервалами, розділення чисел на інтервали та їх об'єднання.

Важливі атрибути:

1) StartPoint: початкова точка інтервалу;
2) EndPoint: кінцева точка інтервалу;
3) Count: кількість чисел, які потрапляють в цей інтервал.

Важливі методи:

1) IsInInterval: перевіряє, чи належить число інтервалу;
2) SplitNumbersIntoEqualIntervals: розділяє числа на рівні інтервали (число інтервалів задається);
3) UniteSmallIntervals: об'єднує менші інтервали в більші.

Клас ChiCriticalValuesHelper

Клас надає довідник критичних значень хі-квадрат для різних ступенів свободи.

Важливі атрибути:

DataDictionary: зберігає табличні значення хі-квадрат для 0.95 для 20 ступенів свободи.

Важливі методи:

1) CheckChiSquared: перевірка на відповідність значення хі-квадрат критичному значенню для заданого ступеня свободи;

2) GetChiCriticalValue(v): Повертає критичне значення хі-квадрат для заданого ступеня свободи.

Абстрактний клас Generator

Клас є базовим для всіх генераторів випадкових чисел. Він визначає загальну структуру генератора та абстрактний метод GenerateNumber(). Крім того, цей клас надає метод Generate, який генерує випадкові числа вказаного розміру.

Важливі методи та властивості:

1) abstract GenerateNumber: абстрактний метод для генерації випадкового числа;
2) Generate(int size): визначений метод для генерації списку випадкових чисел заданого розміру;
3) abstract GetIntegralFunc(): абстрактний метод для визначення функції інтегралу, необхідної для аналізу.

Клас ExponentialGenerator

Генератор випадкових чисел за експоненційним розподілом із параметром Lambda (значення за замовчуванням 1).

Клас NormalGenerator

Генератор випадкових чисел за нормальним розподілом із параметрами Alpha та Sigma (значення за замовчуванням 0 та 1 відповідно).

Клас UniformGenerator

Генератор випадкових чисел за рівномірним розподілом із параметрами A та C (значення за замовчуванням 5^13 та 2^31 відповідно).

Принцип роботи генераторів

ExponentialGenerator (Експоненційний генератор)

Параметри:

1) Lambda: цей параметр відповідає за швидкість спаду експоненційного розподілу. Вибір величини Lambda дозволяє регулювати, наскільки швидко значення спадають від максимуму до нуля.

Принцип розподілу: генератор використовує експоненційний розподіл, який в математичному вигляді визначається функцією щільності ймовірності: $f(x) = \lambda * \exp(-\lambda * x)$, де λ - параметр, що визначається користувачем. Експоненційний розподіл часто використовується для моделювання інтервалів між подіями, які відбуваються з певною сталою інтенсивністю.

NormalGenerator (Нормальний генератор)

Параметри:

1) Sigma: цей параметр відповідає за дисперсію (σ) нормального розподілу. Вибір величини Sigma впливає на розкид значень навколо середнього;
2) Alpha: Цей параметр відповідає за математичне сподівання (μ) нормального розподілу. Вибір значення Alpha зсуває розподіл вліво чи вправо.

Принцип розподілу: Генератор використовує нормальний (гауссівський) розподіл, який характеризується щільністю ймовірності вигляду: $f(x) = (1 / (\sigma * \sqrt{(2\pi)})) * \exp(-(x - \mu)^2 / (2\sigma^2))$, де σ - дисперсія, а μ - математичне сподівання. Нормальний розподіл використовується для моделювання великої кількості явищ у природі, де значення навколо середнього є більш імовірними, а зі збільшенням відхилення від середнього ймовірність спадає.

UniformGenerator (Рівномірний генератор)

Параметри:

1) A та C: визначають майбутній розподіл генерації випадкових чисел у формулі: Z = A * Z % C;
2) Z: значення Z генерується на початку та використовується для генерації випадкових чисел.

Принцип розподілу: Генератор використовує рівномірний розподіл, де кожне число має однакову ймовірність потрапити в будь-який інтервал. У рівномірному розподілі функція щільності ймовірності стала в межах визначеного інтервалу. Математично це виглядає як: f(x) = 1 / (b - a), де a та b - границі інтервалу. Рівномірний розподіл використовується для моделювання ситуацій, де всі можливі значення мають однакову ймовірність.

Реалізації методу генерування випадкового числа

ExponentialGenerator

```
protected override double GenerateNumber() =>
(-1 / Lambda) * Math.Log(_random.NextDouble());
```

NormalGenerator

```
protected override double GenerateNumber(){
    double u = Enumerable.Range(0, 12).Select(_
=> _random.NextDouble()).Sum() - 6;
    return Sigma * u + Alpha;
}
```

UniformGenerator

```
protected override double GenerateNumber(){
    Z = A * Z % C;
    return (double)Z / C;
}
```

Інтегральні функції для перевірки випадково згенерованих чисел

Кожен із генераторів (ExponentialGenerator, NormalGenerator, UniformGenerator) має функцію GetIntegralFunc(), яка повертає інтегральну функцію відповідного розподілу. Ця інтегральна функція визначає ймовірність того, що випадкова величина, згенерована відповідним генератором, потрапить в заданий інтервал.

ExponentialGenerator:

Функція GetIntegralFunc() повертає інтегральну функцію для експоненційного розподілу відповідно до параметра λ (Lambda), який визначається при створенні генератора:

```
public override Func<double, double, double> GetIntegralFunc() =>(start, end) =>
(1 - Math.Exp(-Lambda * end)) - (1 - Math.Exp(-Lambda * start));
```

Інтегральна функція обчислюється наступним чином:

NormalGenerator:

Функція GetIntegralFunc() повертає інтегральну функцію для нормального розподілу відповідно до параметрів Sigma (σ - дисперсія) і Alpha (μ - математичне сподівання), які визначаються при створенні генератора:

```
public override Func<double, double, double> GetIntegralFunc() => (start, end) => {double Func(double x) => 1 / (Sigma * Sqrt(2 * PI)) * Exp(-(Pow(x - Alpha, 2) / (2 * Pow(Sigma, 2))));
        return Integrate.OnClosedInterval(Func, start, end);
};
```

UniformGenerator:

Функція GetIntegralFunc() повертає інтегральну функцію для рівномірного розподілу:

```
public override Func<double, double, double> GetIntegralFunc() => (start, end) => end - start;
```

Інтервали

Для побудови гістограм та перевірки згенерованих чисел необхідно розділити ці числа на проміжки. Для їх формування знаходить мінімальне та максимальне значення зі згенерованого проміжку. Значення за замовчуванням для кількості інтервалів оберемо 20. На таку кількість розбивається проміжок від мінімуму до максимуму. Для кожного проміжку необхідно підрахувати кількість влучень випадкових величин. На основі цього можемо сформувати графіки, але для перевірки параметра хі-квадрат цього недостатньо.

Для перевірки хі-квадрат маємо об’єднати інтервали в яких кількість влучень менша за 5.

Отже отримаємо наступну реалізацію методів для розбиття на інтервали:

```
public static List<Interval>
SplitNumbersIntoEqualIntervals(List<double> numbers, int
intervalsCount = 20)
{
    List<Interval> intervals = new();
    double min = numbers.Min();
    double max = numbers.Max();
    double step = (max - min) / intervalsCount;

    for (int i = 0; i < intervalsCount; i++)
    {
        double intervalStart = i * step + min;
        double intervalEnd = intervalStart + step;

        intervals.Add(CreateAndFillInterval(intervalStart,
intervalEnd, numbers));
    }

    foreach (var number in numbers
    if (number == max)
        intervals[^1].Count++;

    return intervals;
}

public static Interval CreateAndFillInterval(double startPoint,
double endPoint, List<double> numbers)
{
    Interval interval = new(startPoint, endPoint);
    foreach (var number in numbers)
    {
```

```
            if (interval.IsInInterval(number))
            {
                interval.Count++;
            }
        }

        return interval;
    }
```

Метод IsInInterval повертає true якщо число входить у інтервал: зліва включно та справа – ні. Через не включеність справа, найбільші числа не увійдуть такою перевіркою в інтервал. Для цього у останньому інтервалі збільшується кількість чисел, що належить йому.

Далі маємо реалізацію методу для об’єднання малих інтервалів у більший:

```
public static List<Interval> UniteSmallIntervals(List<Interval>
intervals)
{
    List<Interval> unitedIntervals = new List<Interval>(intervals);
    for (int i = 0; i < unitedIntervals.Count; i++){
        if (unitedIntervals[i].Count < IntervalMinCapacity){
            int leftIndex = DefineLeftIndexForMerging(i,
unitedIntervals);
            int rightIndex = leftIndex + 1;

            var newInterval = MergeIntervalsLeftWithRight(
unitedIntervals[leftIndex], unitedIntervals[rightIndex]);
            unitedIntervals[leftIndex] = newInterval;
            unitedIntervals.RemoveAt(rightIndex);
            i--;
        }
    }
    return unitedIntervals;
}
```

Метод DefineLeftIndexForMerging допомагає визначити, хто із сусідніх від малого інтервалу менший. Із тим інтервалом він і об’єднується. Для цього і визначаємо лівого сусіда (для визначення початку інтервалу) та правого (для визначення кінця інтервалу):

Метод MergeIntervalsLeftWithRight створює новий інтервал із початком лівого та кінцем правого інтервалів, та сумарною кількістю чисел обох.

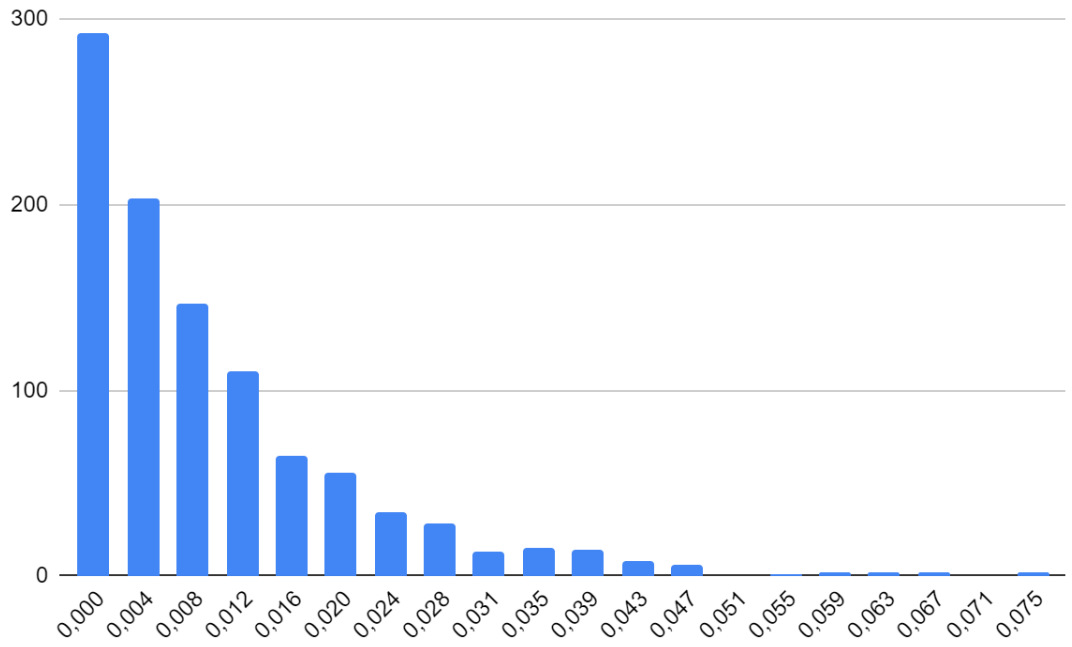
Гістограми та фундаментальні статистичні характеристики

В якості фундаментальних статистичних характеристик розглянемо середнє і дисперсію випадкових чисел. Отже сформуємо наступні методи:

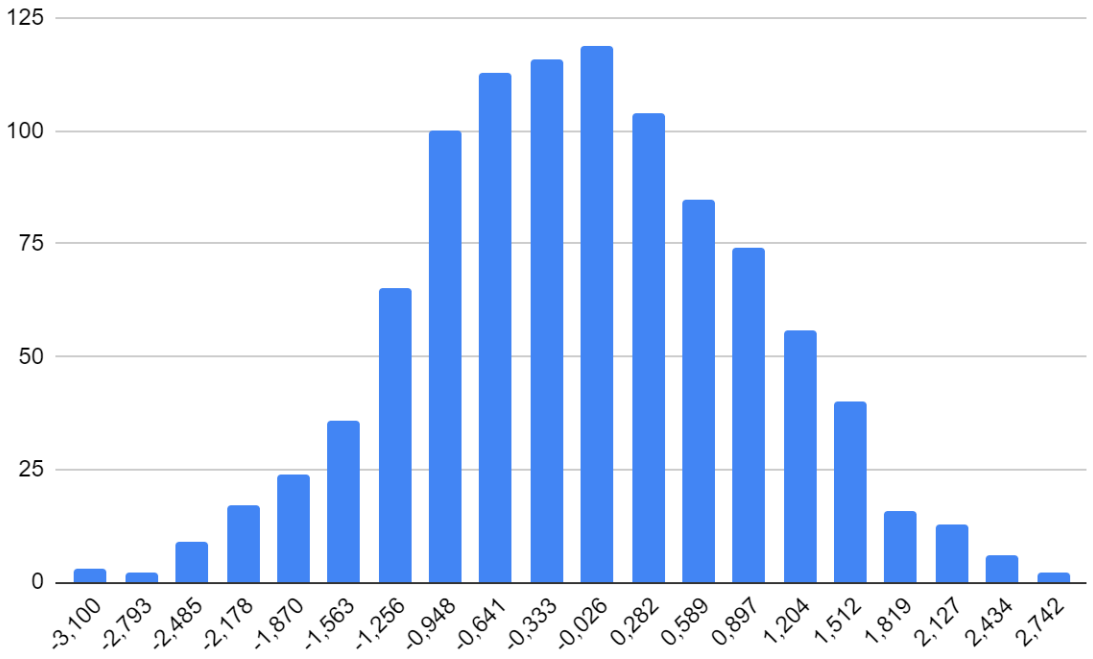
```
private double GetAverage() => _numbers.Average();

private double GetDispersion()
{
    double average = GetAverage();
    double dispersion = 0;
        foreach (var number in _numbers) dispersion += Math.Pow(number - average, 2);

    return dispersion / (_numbers.Count - 1);
}
```

Для гістограм сформуємо дані розподілів і в Excel відобразимо їх. На осі абсцис маємо значення початку інтервалу, вісь ординат в свою чергу демонструє кількість згенерованих чисел, що входять до інтервалу.

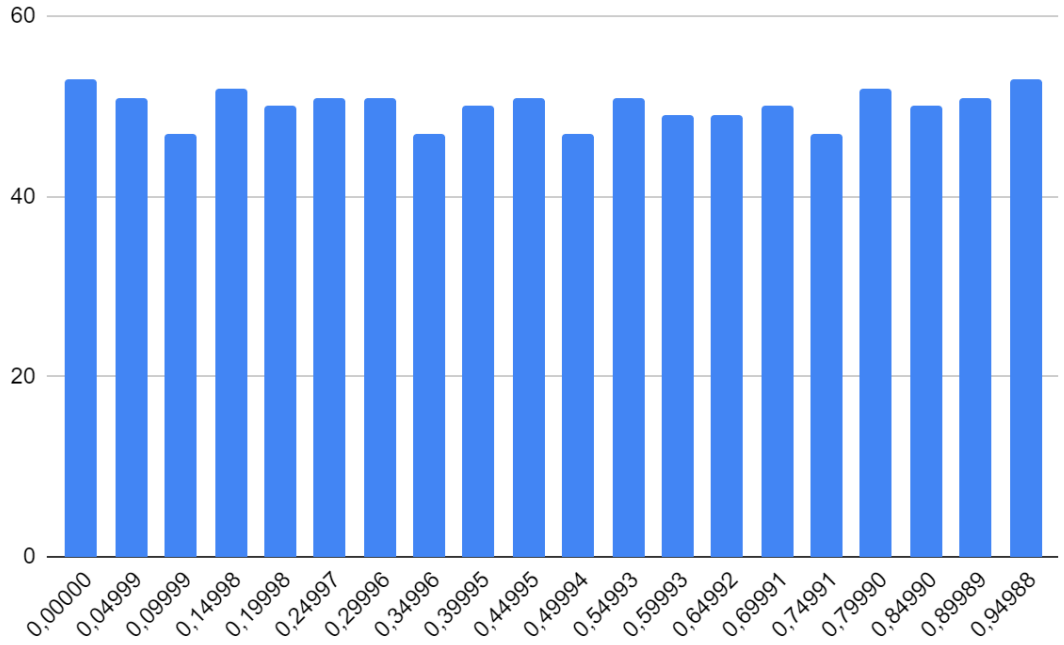
Експоненційний:



Нормальний:



Рівномірний:



На гістограмах видно, що згенерований розподіл дійсно відображає характер розподілу, що мав бути реалізованим. Щодо значень середнього та дисперсії, розглянемо їх значення вже після визначення найкращих параметрів для генерації.

Перевірка розподілу через χ^2(ксі-квадрат)

Для перевірки необхідно порахувати значення ксі-квадрату. Отже, сформуємо наступний метод:

```
    private double CalculateChiSquared(List<Interval> intervals,
Func<double, double, double> integralFunc)
    {
        double chi = 0;
        foreach (var interval in intervals)
        {
            double npi = GetNumbersCountInAllIntervals(intervals) *
                         integralFunc(interval.StartPoint,
interval.EndPoint);
            chi += Math.Pow((interval.Count - npi), 2) / npi;
        }

        return chi;
    }
```

Під час обрахунку маємо надати інтегральну функцію для відповідного генератора та зменшений варіант інтервалів для надання кількості чисел у ньому. Далі потрібно порівняти кінцеве значення ксі-квадрат із табличним значенням. Для зберігання табличних значень маємо клас ChiCriticalValuesHelper. Отже, далі сформуємо наступну перевірку, де ступінь свободи v обчислюється як різниця загальної кількості інтервалів у зменшеному варіанті та двох:

```
    tableChi =
ChiCriticalValuesHelper.GetChiCriticalValue(_unitedIntervals.Count
- 2);
    return chi < tableChi;
```

Тепер перейдемо до виведення характеристик кожного з генераторів:

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
EXPONENTIAL
Intervals count: 20 -> 16
Average: 0.011691494436157452
Dispersion: 0.00013162657534206634
Chi: 18.470 < 23.685 OK
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
NORMAL
Intervals count: 20 -> 17
Average: 0.10147001185380343
Dispersion: 0.009990586764310217
Chi: 8.472 < 24.996 OK
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
UNIFORM
Intervals count: 20 -> 20
Average: 0.4830398758368682
Dispersion: 0.08404655744152037
Chi: 19.287 < 28.869 OK
```

Як бачимо, кількість інтервалів зменшилася у перших двох (через малу кількість чисел у частині, що спадає та границях розподілу відповідно), у рівномірному – ні (через рівномірність розподілу). Можемо бачити значення середнього числа згенерованого розподілу та дисперсію. В кінці ж, демонструється кінцеве значення ксі-квадрат, що порівнюється з табличним відповідно до ступеня свободи. Слово “ОК” відображається в усіх випадках і каже, що перевірка розподілу прошла успішно. Тобто, із ймовірністю 0.95 можна стверджувати, що знайдений закон розподілу відповідає згенерованому.

Тестування параметрів генератора

Перейдемо до тестування параметрів усіх генераторів. Нехай, для кожного значення параметра відбудеться тестування у 1000 ітерацій. Якщо фінальне ксі-квадрат менше за табличний варіант то будемо збільшувати показник chiIsOk. В кінці будемо виводити процент того наскільки генерація була успішною:

```
    public void TestChiIsOkPercent(Generator generator, int
TESTS_COUNT = 1000)
    {
        int chiIsOk = Enumerable.Range(0, TESTS_COUNT).Select(_ =>
GetChiIsOkAfterAnalysis(generator) ? 1 : 0).Sum();
        double goodChiPercent = (double)chiIsOk / TESTS_COUNT *
100;
        Console.Out.WriteLine($"Percent of good chi:
{goodChiPercent:F2}%");
    }
```

Отже, перейдемо до результатів при різних варіантах параметрів генераторів:

Експоненційний:

λ	0.1	10.1	20.1	30.1	40.1	50.1	60.1	70.1	80.1	90.1
%(χ^2)	90.60	89.50	91.20	90.30	91.20	88.50	90.70	88.50	90.70	89.70

Нормальний:

σ	0.1	0.1	0.1	0.1	25.1	25.1	25.1	25.1
α	0.1	25.1	50.1	75.1	0.1	25.1	50.1	75.1
%(χ^2)	90.40	90.70	91.00	90.70	91.20	89.70	89.70	90.10

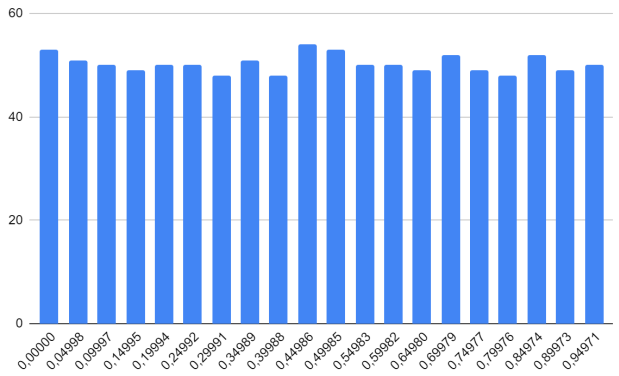
σ	50.1	50.1	50.1	50.1	75.1	75.1	75.1	75.1
α	0.1	25.1	50.1	75.1	0.1	25.1	50.1	75.1
%(χ^2)	90.00	90.20	89.40	91.80	89.20	89.70	89.20	89.60

Рівномірний:

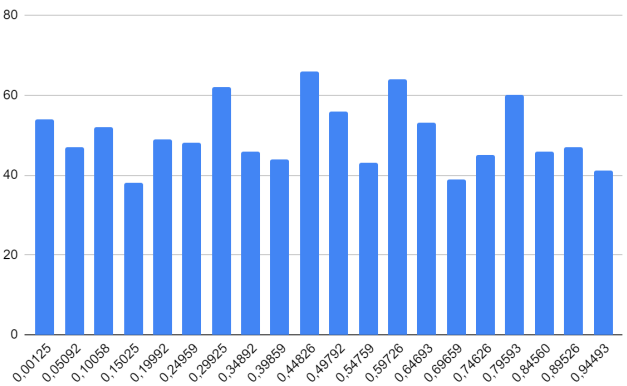
a	3^4	3^4	3^4	4^10	4^10	4^10	5^13	5^13	5^13
c	6^15	6^10	3^17	6^15	6^10	3^17	6^15	6^10	3^17
%(χ^2)	91.70	92.80	100.0	92.00	99.60	92.40	0.90	94.40	96.30

Судячи з результатів наведених у таблицях, видно, що зміна параметрів не сильно впливає на процент успішності генерації. У свою чергу, у Рівномірному розподілі важливо підібрати хороші параметри для досягнення найкращого проценту успіху генерації. Так при значеннях 3^4 та 3^17 досягаємо успіху в усіх тестуваннях, тоді як при 5^13 та 6^15 отримуємо лише 1% успіху від усіх генерацій.

Оскільки графики перших двох генераторів не мають особливим чином змінитися, перейдемо до фінального вигляду гістограми рівномірного розподілу при найкращих значеннях (a:3^4, c:3^17):



Та найгірших (a:5^13, c:6^15), але все ж успішного:



Висновок

Досліджуючи різні методи генерації випадкових чисел та перевіряючи їхню коректність і відповідність розподілам, я прийшла до наступних висновків:

1) Реалізація усіх методів генерації розподілу справді відповідає теоретичним розподілам з відповідними параметрами для кожного з них. Перевірка заснована на попередньому аналізі гістограм, використанні інтегральної функції, та обрахунок ксі-квадрату. Фінальні результати свідчать про те, що розподіли відповідають очікуваному.
2) Останнім кроком був розгляд рівномірного розподілу, де докладний підбір параметрів виявився важливим фактором для досягнення успішної генерації. Результати тестування показали, що вибір параметрів може впливати на якість генерації, але з правильними значеннями, можна досягти задовільних результатів.

Загалом, наша робота демонструє важливість ретельного аналізу і тестування генераторів випадкових чисел для забезпечення їхньої коректності та відповідності теоретичним розподілам. Також, зазначимо, що зміна параметрів може впливати на результати генерації, і їх правильний вибір важливий для досягнення успішних результатів.

У висновку, наша робота демонструє, що наші генератори випадкових чисел є ефективними та коректними, і можуть бути використані для подальших досліджень та моделювання в різних областях.

Код

GeneratorAnalyser.cs

```
using System.Globalization;
using System.Text;
using Lab1.Generators;
using static Lab1.ChiCriticalValuesHelper;

namespace Lab1;

public class GeneratorAnalyser
{
    private List<double> _numbers = new();
    private List<Interval> _intervals = new();
    private List<Interval> _unitedIntervals = new();
    private double chi;
    private double tableChi;

    public int NumbersCount { get; set; } = 1000;
    public int IntervalsCount { get; set; } = 20;

    public bool GetChiIsOkAfterAnalysis(Generator generator)
    {
        _numbers = generator.Generate(NumbersCount);
        _intervals = Interval.SplitNumbersIntoEqualIntervals(_numbers,
IntervalsCount);
        _unitedIntervals = Interval.UniteSmallIntervals(_intervals);
        chi = CalculateChiSquared(_unitedIntervals, generator.GetIntegralFunc());
        tableChi = GetChiCriticalValue(_unitedIntervals.Count - 2);
        return chi < tableChi;
    }

    public double GetChiAfterAnalysisWithPrint(Generator generator)
    {
        bool chiIsOk = GetChiIsOkAfterAnalysis(generator);
        PrintIntervalsAsColumn(_intervals);
        Console.Out.WriteLine($"Intervals count: {_intervals.Count} ->
{_unitedIntervals.Count}");
        Console.Out.WriteLine($"Average: {GetAverage()}");
        Console.Out.WriteLine($"Dispersion: {GetDispersion()}");
        Console.Out.WriteLine($"Chi: {chi:F3} {(chiIsOk ? "<" : ">= ")} {tableChi}
{(chiIsOk ? "OK" : "NOT OK")}");
        return chi;
    }

    public void TestChiIsOkPercent(Generator generator, int TESTS_COUNT = 1000)
    {
        int chiIsOk = Enumerable.Range(0, TESTS_COUNT).Select(_ =>
GetChiIsOkAfterAnalysis(generator) ? 1 : 0).Sum();
        double goodChiPercent = (double)chiIsOk / TESTS_COUNT * 100;
        Console.Out.WriteLine($"Percent of good chi: {goodChiPercent:F2}%");
    }

    private double GetAverage() => _numbers.Average();

    private double GetDispersion()
    {
        double average = GetAverage();
        double dispersion = 0;
```

```
        foreach (var number in _numbers) dispersion += Math.Pow(number - average,
2);

        return dispersion / (_numbers.Count - 1);
    }

    private double CalculateChiSquared(List<Interval> intervals, Func<double,
double, double> integralFunc)
    {
        double chi = 0;
        foreach (var interval in intervals)
        {
            double npi = GetNumbersCountInAllIntervals(intervals) *
                         integralFunc(interval.StartPoint, interval.EndPoint);
            chi += Math.Pow((interval.Count - npi), 2) / npi;
        }

        return chi;
    }

    private int GetNumbersCountInAllIntervals(List<Interval> intervals) =>
        Enumerable.Sum(intervals, interval => interval.Count);

    private void PrintIntervalsAsColumn(List<Interval> intervals)
    {
        CultureInfo customCulture = new CultureInfo("fr-FR") { NumberFormat = {
NumberDecimalSeparator = "," } };

        foreach (var interval in intervals)
        {

Console.Out.WriteLine($"{interval.StartPoint.ToString(customCulture)}");
        }

        foreach (var interval in intervals)
        {
            Console.Out.Write($"{interval.Count}\n");
        }

        Console.Out.WriteLine("");
    }

    public void PrintIntervals(List<Interval> intervals)
    {
        StringBuilder sb = new();
        foreach (var interval in intervals)
        {
            string format = "+0.00;-0.00;0.000";
            sb.Append(
                $"[{interval.StartPoint.ToString(format)};
{interval.EndPoint.ToString(format)}) {interval.Count} ");
            for (int i = 0; i < interval.Count; i++)
            {
                // sb.Append(".");
            }

            sb.AppendLine();
        }

        Console.WriteLine(sb.ToString());
    }
```

```
}
```

Generator.cs

```
namespace Lab1.Generators;

public abstract class Generator
{
   protected abstract double GenerateNumber();

   public List<double> Generate(int size)
   {
       List<double> numbers = new();
       for (int i = 0; i < size; i++)
       {
           numbers.Add(GenerateNumber());
       }

       return numbers;
   }

   public abstract Func<double, double, double> GetIntegralFunc();
}
```

ExponentialGenerator.cs

```
namespace Lab1.Generators;

public class ExponentialGenerator : Generator
{
   public double Lambda { get; set; }

   private readonly Random _random = new Random();

   public ExponentialGenerator(double lambda = 1)
   {
       Lambda = lambda;
   }

   protected override double GenerateNumber() => (-1 / Lambda) *
Math.Log(_random.NextDouble());

   public override Func<double, double, double> GetIntegralFunc() =>
       (start, end) => (1 - Math.Exp(-Lambda * end)) - (1 - Math.Exp(-Lambda *
start));
}
```

NormalGenerator.cs

```
using MathNet.Numerics;
using static System.Math;

namespace Lab1.Generators;

public class NormalGenerator : Generator
{
   private readonly Random _random = new Random();
```

```
    public double Sigma { get; set; } // dispersion
    public double Alpha { get; set; } // mathematical expectation

    public NormalGenerator(double sigma = 1, double alpha = 0)
    {
        Sigma = sigma;
        Alpha = alpha;
    }

    protected override double GenerateNumber()
    {
        double u = Enumerable.Range(0, 12).Select(_ => _random.NextDouble()).Sum()
- 6;
        return Sigma * u + Alpha;
    }

    public override Func<double, double, double> GetIntegralFunc() => (start, end)
=>
    {
        double Func(double x) => 1 / (Sigma * Sqrt(2 * PI)) * Exp(-(Pow(x - Alpha,
2) / (2 * Pow(Sigma, 2))));
        return Integrate.OnClosedInterval(Func, start, end);
    };
}
```

UniformGenerator.cs

```
namespace Lab1.Generators;

public class UniformGenerator : Generator
{
    public double A { get; set; } // 5^13 == 1220703125
    public double C { get; set; } // 2^31 == 2147483648
    public double Z { get; set; }

    public UniformGenerator(double a = 1220703125, double c = 2147483648)
    {
        A = a;
        C = c;
        Z = new Random().NextDouble();
    }

    protected override double GenerateNumber()
    {
        Z = (A * Z) % C;
        return (double)Z / C;
    }

    public override Func<double, double, double> GetIntegralFunc() => (start, end)
=> end - start;
}
```

Interval.cs

```
namespace Lab1;

public class Interval
{
    public double StartPoint { get; private set; }
```

```
    public double EndPoint { get; private set; }
    public int Count { get; private set; }
    private const int IntervalMinCapacity = 5;

    public Interval(double startPoint, double endPoint)
    {
        StartPoint = startPoint;
        EndPoint = endPoint;
    }

    public bool IsInInterval(double number) => number >= StartPoint && number <
EndPoint;

    public static List<Interval> SplitNumbersIntoEqualIntervals(List<double>
numbers, int intervalsCount = 20)
    {
        List<Interval> intervals = new();
        double min = numbers.Min();
        double max = numbers.Max();
        double step = (max - min) / intervalsCount;

        for (int i = 0; i < intervalsCount; i++)
        {
            double intervalStart = i * step + min;
            double intervalEnd = intervalStart + step;

            intervals.Add(CreateAndFillInterval(intervalStart, intervalEnd,
numbers));
        }

        foreach (var number in numbers)
        {
            if (number == max)
            {
                intervals[^1].Count++;
            }
        }
        return intervals;
    }

    public static Interval CreateAndFillInterval(double startPoint, double
endPoint, List<double> numbers)
    {
        Interval interval = new(startPoint, endPoint);
        foreach (var number in numbers)
        {
            if (interval.IsInInterval(number))
            {
                interval.Count++;
            }
        }

        return interval;
    }

    public static List<Interval> UniteSmallIntervals(List<Interval> intervals)
    {
        List<Interval> unitedIntervals = new List<Interval>(intervals);
        for (int i = 0; i < unitedIntervals.Count; i++)
        {
            if (unitedIntervals[i].Count < IntervalMinCapacity)
```

```
            {
                int leftIndex = DefineLeftIndexForMerging(i, unitedIntervals);
                int rightIndex = leftIndex + 1;

                if (leftIndex < 0 || rightIndex >= unitedIntervals.Count)
                {
                    Console.WriteLine("Something went wrong");
                }
                var newInterval =
MergeIntervalsLeftWithRight(unitedIntervals[leftIndex],
unitedIntervals[rightIndex]);
                unitedIntervals[leftIndex] = newInterval;
                unitedIntervals.RemoveAt(rightIndex);

                i--;
            }
        }

        return unitedIntervals;
    }

    private static int DefineLeftIndexForMerging(int i, List<Interval> intervals)
    {
        if (i == 0) return i;
        if (i == intervals.Count - 1) return i - 1;
        int leftIndex = i - 1;
        if (i + 1 < intervals.Count - 1)
        {
            if (intervals[i + 1].Count < intervals[i - 1].Count)
                leftIndex = i;
        }

        return leftIndex;
    }

    public static Interval MergeIntervalsLeftWithRight(Interval left, Interval
right)
    {
        Interval newInterval = new Interval(left.StartPoint, right.EndPoint)
        {
            Count = left.Count + right.Count
        };

        return newInterval;
    }
}
```

ChiCriticalValuesHelper.cs

```
namespace Lab1;

public static class ChiCriticalValuesHelper
{
    private static readonly Dictionary<int, double> DataDictionary = new
Dictionary<int, double>
    {
        { 1, 3.841 },
        { 2, 5.991 },
        { 3, 7.815 },
        { 4, 9.488 },
```

```
        { 5, 11.070 },
        { 6, 12.592 },
        { 7, 14.067 },
        { 8, 15.507 },
        { 9, 16.919 },
        { 10, 18.307 },
        { 11, 19.675 },
        { 12, 21.026 },
        { 13, 22.362 },
        { 14, 23.685 },
        { 15, 24.996 },
        { 16, 26.296 },
        { 17, 27.587 },
        { 18, 28.869 },
        { 19, 30.144 },
        { 20, 31.410 }
    };

    public static bool CheckChiSquared(double chi, int v) => chi <
GetChiCriticalValue(v);

    public static double GetChiCriticalValue(int v) => v < 1 ? DataDictionary[1] :
DataDictionary[v];
}
```