МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет прикладной математики и информатики

Петров Андрей Александрович

Отчет по лабораторным работам по курсу
“Математическое моделирование”
студента 2 курса 14 группы

Работа сдана ________ 2021г.

зачтена ___________2021 г.

____
(подпись преподавателя)

Преподаватель
Лобач Сергей Викторович
Ассистент кафедры математического моделирования и анализ данных ФПМИ

Минск 2021

ОГЛАВЛЕНИЕ

ЛАБОРАТОРНАЯ РАБОТА 1. 3
ЛАБОРАТОРНАЯ РАБОТА 2. 7
ЛАБОРАТОРНАЯ РАБОТА 3. 12
ЛАБОРАТОРНАЯ РАБОТА 4.1 17
ЛАБОРАТОРНАЯ РАБОТА 4.2 19
ЛАБОРАТОРНАЯ РАБОТА 5 20
СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ 23

ЛАБОРАТОРНАЯ РАБОТА 1.

Условие:

Используя метод Маклерена-Марсальи построить датчик БСВ (1 датчик должен быть мультипликативно конгруентный, второй – на выбор). Исследовать точность построенной БСВ.

1) Осуществить моделирование $n = 1000$ реализаций БСВ с помощью мультипликативного конгруэнтного метода (МКМ) с параметрами a_0, β, $M = 2^{31}$.

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.

Теория:

Мультипликативный конгруэнтный метод:

Псевдослучайная последовательность $\alpha_1, \alpha_2, ..., \alpha_n$ строится по следующим рекуррентным формулам:

$$\alpha_t = \alpha_t^* / M, \ \ \alpha_t^* = \{\beta\alpha_{t-1}^*\} \operatorname{mod} M \quad (t = 1, 2, ...),$$

где β, M, α_0^* - параметры датчика: β - множитель ($\beta < M$), M – модуль, $\alpha_0^* \in \{1, ..., M-1\}$ - стартовое значение (нечетное число).

В данной работе брались значения: M=2147483648, $\alpha_0^* = \beta$ =65539.

Метод Маклорена-Марсальи:

Пусть $\{\beta_t\}$, $\{c_t\}$ - псевдослучайные последовательности, порожденные независимо работающими датчиками; $\{\alpha_t\}$ - результирующая псевдослучайная последовательность реализация БСВ;

V={V(0), V(1), …,V(K-1)} – вспомогательная таблица K чисел.

Процесс вычисления $\{\alpha_t\}$ включает следующие этапы:

- первоначальное заполнение таблицы

$$V: V(i) = \beta_i, \ \text{i}=\overline{0,K-1};$$

- случайный выбор из таблицы:

$$\alpha_t = V(s), \ \text{s}=[\text{c}_t \cdot \text{K}];$$

-обновление табличных значений:

$$V(s) = b_{t+K}, \ \text{t}=0, 1, 2, ...$$

В данной работе в качестве $\{\beta_t\}$ бралась последовательность (из 100 элементов), полученная мультипликативным конгруэнтным методом, описанным выше. В качестве $\{c_t\}$, бралась последовательности (из 10000) элементов, полученная аналогичным способом с тем же М и $\beta' = 3\beta + 1$. K=100.

χ^2 - *критерий согласия Пирсона:*

Область возможных значений случайной величины разбивается на интервалы $[x_{k-1}, x_k)$, $k = \overline{1, K}$.

Рассматривается следующая статистика,

$$\chi^2 = \sum_{k=1}^{K} \frac{(\nu_k - n \cdot p_k)}{n \cdot p_k},$$

n – объем выборки,

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

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

Проверяется условие $\chi^2 < \Delta$, где $\Delta = G^{-1}(1-\varepsilon)$, G функция распределения распределения χ^2, ε - уровень значимости (обычно ε=0.05).

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

Критерий согласия Колмогорова:

Рассматривается статистика:

$$D_n = \sup_{x \in \square} | F_\xi(x) - F_0(x) | \in [0;1],$$

где $F_\xi(x) = \dfrac{1}{n} \sum_{i=1}^{n} I_{[-\infty;x]}(x_i),\ x \in \square$,

Проверяется условие $\sqrt{n} K_n < \Delta$, где $\Delta = K^{-1}(1-\varepsilon)$, K - функция распределения распределения Колмогорова, ε - уровень значимости.

Код программы:

```
using System;
using System.IO;

namespace lab1
{
    internal static class Program
    {
        private const long A0 = 78125;
        private const long Beta = A0;
        private const long M = 2147483648L;
        private const int NumImplements = 1000;
        private const int K = 256;
        private const double CriticalNum = 16.91898;
        private const double CriticalNumD = 1.63;

        private static double[] MultiMethod(double[] a , long beta, int n){
            var aWithStar = new double[n];
            aWithStar[0] = beta*beta % M;
```

```
            a[0] = aWithStar[0] / M;

            for (int i = 1; i < n; i++){
                aWithStar[i] = aWithStar[i - 1]*Beta % M;
                a[i] = aWithStar[i] / M;
            }
            return a;
        }

        private static double[] MethodMacLarenMarsaglia(double[] a, double[] b,
double[] c){
            var temp = new double[K];
            Array.Copy(b,0, temp, 0, K);

            for (var i = 0; i < NumImplements; i++){
                var s =((int)(c[i]*K))% K;
                a[i] = temp[s];
                temp[s] = c[(i + K) % K];
            }

            return a;
        }

        private static double TestPirson(double[] a, int L){
            Array.Sort(a);
            double xi = 0;
            var i = 0;

            for (int j = 1; j <= L; j++){
                var count = 0;
                while ((i < NumImplements) && (a[i] < (double) j / L)){
                    i++;
                    count++;
                }
                xi += Math.Pow(count - (double)(NumImplements / L), 2) /
(double)(NumImplements / L);
            }

            Console.WriteLine("xi^2={0} | critical number: {1}", xi, CriticalNum
);

            return xi;
        }

        private static double TestKolmogorov(double[] a){
            Array.Sort(a);
            double D = 0;

            for (int i = 0; i < NumImplements; i++){
                D = Math.Max(D, Math.Abs(((double)i + 1) / NumImplements) -
a[i]);
            }

            Console.WriteLine("D={0} | critical number: {1}", D*Math.Sqrt(1000),
CriticalNumD );
            return D;
        }


        public static void Main(String[] args) {
            var firstSeq = MultiMethod(new double[NumImplements], A0,
NumImplements);
            var secondSeq = MultiMethod(new double[K], 2*Beta + 1, K);
            var thirdSeq = new double[NumImplements];
```

```
            thirdSeq = MethodMacLarenMarsaglia(thirdSeq, secondSeq, firstSeq);

            TestPirson(firstSeq, 10);
            TestPirson(thirdSeq, 10);

            TestKolmogorov(firstSeq);
            TestKolmogorov(thirdSeq);

            StreamWriter sw = new StreamWriter("data.txt");
            sw.WriteLine("MultiMethod 1#:");
            foreach (var f in firstSeq)
            {
                sw.Write("{0} ", f);
            }
            sw.WriteLine("\nMultiMethod 2#:");
            foreach (var s in secondSeq)
            {
                sw.Write("{0} ", s);
            }
            sw.WriteLine("\nMethodMacLarenMarsaglia:");
            foreach (var th in thirdSeq)
            {
                sw.Write("{0} ", th);
            }
            sw.Close();

        }
    }
}
```

Результат выполнения программы:

```
xi^2=8,620000000000001 | critical number: 16,91898
xi^2=27,14 | critical number: 16,91898
D=0,6904544718552611 | critical number: 1,63
D=1,1282483180911198 | critical number: 1,63
```

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 χ^2-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.

Теория:

Распределение Пуассона (с параметром λ):

Случайная величина ξ принимает только целые неотрицательные значения, причем

$$P(\xi = k) = \frac{\lambda^k}{k!},\ \mathrm{k} \in \square, k \geq 0.$$

В данной работе сначала моделировалась последовательность БСВ, а потом по каждой БСВ строился соответствующий элемент выборки распределения Пуассона: отрезок [0;1] разбивался на интервалы длин $\frac{\lambda^k}{k!}$ проверялось, в какой интервал попадает элемент последовательности БСВ.

Код программы:

```
public class DRV
    {
        private static int NumberImplementations { get; set; }
        internal DRV(int n)
        {
            NumberImplementations = n;
        }
        public static List<int> ModelingGeometric(params double[] param)
        {
            var p = param[0];
            var sequence = new List<int>();
            var rnd = new Random();

            for (var i = 0; i < NumberImplementations; i++)
            {
                sequence.Add( (int) (Math.Log(rnd.NextDouble(), 1 - p) + 1));
            }
            return sequence;
        }
        public static List<int> ModelingBernoulli(params double[] param)
        {
            var p = param[0];
            var sequence = new List<int>();
            var rnd = new Random();

            for (var i = 0; i < NumberImplementations; i++)
```

```
            {
                sequence.Add(rnd.NextDouble() < p ? 1 : 0);
            }
            return sequence;
        }
        public static List<int> ModelingBinomial(params double[] param)
        {
            var m = param[0];
            var p = param[1];
            double  x = 0;
            var sequence = new int[NumberImplementations];
            var rnd = new Random();

            for(var j = 0; j < NumberImplementations; j++)
            {
                for(var i = 0; i < m; i++)
                {
                    if (rnd.NextDouble() < p)
                    {
                        x++;
                    }
                }
                sequence[j] = (int) x;
                x = 0;
            }
            return new List<int>(sequence);
        }
        public static List<int> ModelingNegBinomial(params double[] param)
        {
            var r = param[0];
            var p = param[1];
            double rem = r, x = 0;
            var sequence = new int[NumberImplementations];
            var rnd = new Random();

            for(var j = 0; j < NumberImplementations; j++)
            {
                while (rem != 0)
                {
                    if (rnd.NextDouble() > p)
                    {
                        x++;
                    }
                    else
                    {
                        rem--;
                    }
                }
                sequence[j] = (int) x;
                rem = r;
                x = 0;
            }
            return new List<int>(sequence);
        }
    }
```

```
public static class ListDrvExtension
    {
        public static double GetDispersion(this List<int> sequence,
                                           double mathExpectation)
        {
            var sum = sequence.Sum(seq => (seq - mathExpectation) * (seq -
```

```
                                                                    mathExpectation));
            return  sum / (sequence.Count - 1);
        }

        public static double GetMathExpectation(this List<int> sequence)
        {
            var sum = sequence.Aggregate<int, double>(0, (current, seq) =>
                                                        current + seq);
            return sum/sequence.Count;
        }

        public static double CriterionPearsonBernoulli(this List<int> sequence,
                                                                double p)
        {
            var n = sequence.Count;
            double xi = 0;

            var valuesNum = new[] {0, 0};
            var probabilities = new[] {1-p, p};

            foreach (var seq in sequence)
            {
                valuesNum[seq]++;
            }
            for (int i = 0; i < 2; i++)
            {
                xi += ((double) valuesNum[i] / n - probabilities[i]) *
                      ((double) valuesNum[i] / n - probabilities[i]) /
                      probabilities[i];
            }
            return xi;
        }

        public static double CriterionPearsonGeometric(this List<int> sequence,
                                                                double p)
        {
            var n = sequence.Count;
            double xi = 0;
            var maxValue = sequence.Max();

            var valuesNum = new int[maxValue+1];
            var probabilities = new double[maxValue+1];

            for (var i = 0; i <= maxValue; i++)
            {
                probabilities[i] = Math.Pow(1 - p, i) * p;
            }
            foreach (var seq in sequence)
            {
                valuesNum[seq]++;
            }
            for (int i = 0; i <= maxValue; i++)
            {
                xi += ((double) valuesNum[i] / n - probabilities[i]) *
                      ((double) valuesNum[i] / n - probabilities[i]) /
                      probabilities[i];
            }
            return xi;
        }
        public static double CriterionPearsonBinomial(this List<int> sequence,
                                                        params double[] param)
        {
            var m = (int) param[0] + 1;
            var p = param[1];
```

```
        var n = sequence.Count;
        double xi = 0;

        var valuesNum = new int[m];
        var probabilities = new double[m];

        for (var i = 0; i < m; i++)
        {
            probabilities[i] = Cominations((m-1),i)*Math.Pow(p,i)*
                                                  Math.Pow(1-p, m-1-i);
        }
        foreach (var seq in sequence)
        {
            valuesNum[seq]++;
        }
        for (var i = 0; i < m; i++)
        {
            xi += ((double) valuesNum[i] / n - probabilities[i]) *
                  ((double) valuesNum[i] / n - probabilities[i]) /
                  probabilities[i];
        }
        return xi;
    }
    public static double CriterionPearsonNegBinomial(this List<int>
                                                   sequence, params double[] param)
    {
        var r = (int) param[0];
        var p = param[1];
        var n = sequence.Count;
        double xi = 0;
        var maxValue = sequence.Max();

        var valuesNum = new int[maxValue+1];
        var probabilities = new double[maxValue+1];

        foreach (var seq in sequence)
        {
            valuesNum[seq]++;
        }

        for (var i = 0; i <= maxValue; i++)
        {
            probabilities[i] = Cominations(i + r - 1, i) * Math.Pow(p, r) *
                                             Math.Pow(1 - p, i);
        }

        for (var i = 0; i <= maxValue; i++)
        {
            xi += ((double) valuesNum[i] / n - probabilities[i]) *
                  ((double) valuesNum[i] / n - probabilities[i]) /
                  probabilities[i];
        }
        return xi;
    }

    private static long Cominations(int allNumbers, int perGroup){
        if (perGroup < 0 || perGroup > allNumbers){
            return 0;
        }
        if (perGroup == 0 || perGroup == allNumbers){
            return 1;
        }
        perGroup = Math.Min(perGroup, allNumbers - perGroup);
        long c = 1;
```

```
            for (int i = 0; i < perGroup; i++) {
                c = c * (allNumbers - i) / (i + 1);
            }
            return c;
        }
    }
```

Результат выполнения программы:

```
log>    Criterion Pearson's for GEOMETRIC distribution: 1,5522360263321762
        Calculated math expectation: 1,701
        Theoretical math expectation: 1,6666666666666667
        Calculated dispersion: 1,1427417417417352
        Theoretical dispersion: 1,1111111111111112

log>    Criterion Pearson's for BERNOULLI distribution: 0,0014062499999999982
        Calculated math expectation: 0,185
        Theoretical math expectation: 0,2
        Calculated dispersion: 0,15092592592592705
        Theoretical dispersion: 0,16000000000000003

log>    Criterion Pearson's for BINOMIAL distribution: 0,009886764035291984
        Calculated math expectation: 2,08
        Theoretical math expectation: 1,9999997999999999
        Calculated dispersion: 1,429029029029023
        Theoretical dispersion: 1,3333332666666597

log>    Criterion Pearson's for NEGATIVE BINOMIAL distribution: 0,05246856591965019
        Calculated math expectation: 16,171
        Theoretical math expectation: 16
        Calculated dispersion: 79,38914814814821
        Theoretical dispersion: 79,99999999999999
```

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 χ^2-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.

Код программы:

```
public class Crv
{
    private int NumberImplementations { get; set; }
    public static int NumberSection{ get; set; }
    internal Crv(int n, int s)
    {
        NumberImplementations = n;
        NumberSection = s;
    }

    public List<double> ModelingNormal(params double[] param)
    {
        var p1 = param[0];
        var p2 = param[1];
        var sequence = new List<double>();
        var rnd = new Random();

        for (var i = 0; i < NumberImplementations; i++)
        {
            double sum = -6;
            for (int j = 0; j < 12; j++)
            {
                sum += rnd.NextDouble();
            }
            sequence.Add(p1 + sum * Math.Sqrt(p2));
        }
        return sequence;
    }

    public List<double> ModelingLogNormal(params double[] param)
    {
        var p1 = param[0];
        var p2 = param[1];
```

```
        var sequence = new List<double>();
        var rnd = new Random();

        for (var i = 0; i < NumberImplementations; i++)
        {
            double sum = -6;
            for (int j = 0; j < 12; j++)
            {
                sum += rnd.NextDouble();
            }
            sequence.Add(Math.Exp(p1 + sum * Math.Sqrt(p2)));
        }
        return sequence;
    }

    public List<double> ModelingLogistics(params double[] param)
    {
        var p1 = param[0];
        var p2 = param[1];
        var sequence = new List<double>();
        var rnd = new Random();

        for (var i = 0; i < NumberImplementations; i++)
        {
            var y = rnd.NextDouble();
            sequence.Add(p1 + p2 * Math.Log(y / (1 - y)));
        }
        return sequence;
    }

    public List<double> ModelingLaplace(params double[] param)
    {
        var p = param[0];
        var sequence = new List<double>();
        var rnd = new Random();

        for (var i = 0; i < NumberImplementations; i++)
        {
            var y = rnd.NextDouble();
            if (y < 0.5)
                sequence.Add(1 / p * Math.Log(2 * y));
            else
                sequence.Add(-1 / p * Math.Log(2 * (1 - y)));
        }

        return sequence;
    }

    public List<double> ModelingExponential(params double[] param)
    {
        var p = param[0];
        var sequence = new List<double>();
        var rnd = new Random();

        for (var i = 0; i < NumberImplementations; i++)
        {
            var y = rnd.NextDouble();
            sequence.Add(-Math.Log(y) / p);
        }
        return sequence;
    }

    public double NormalFunc(double value, params double[] param)
    {
```

```
        var m = param[0];
        var s2 = param[1];

        AdaptiveIntegrator integrator = new AdaptiveIntegrator();
        Func<double, double> f1 = (x) => Math.Exp(-Math.Pow(x - m, 2) / 2 / s2);
        integrator.Integrate(f1, -1000, value);
        return integrator.Result / Math.Sqrt(2 * Math.PI * s2);
    }

    public double LogNormalFunc(double value, params double[] param)
    {
        var m = param[0];
        var s2 = param[0];
        AdaptiveIntegrator integrator = new AdaptiveIntegrator();
        Func<double, double> f1 = (x) =>Math.Exp(-Math.Pow(Math.Log(x) - m, 2) /
                                                              2 / s2);
        integrator.Integrate(f1, -1000, value);
        return integrator.Result / Math.Sqrt(2 * Math.PI * s2);
    }

    public double LogisticsFunc(double value, params double[] param)
    {
        var p1 = param[0];
        var p2 = param[1];
        return 1 / (1 + Math.Exp(-(value - p1) / p2));
    }

    public double ExpFunc(double value, params double[] param)
    {
        var p = param[0];
        return 1 - Math.Exp(-p * value);
    }
    public double LaplaceFunc(double value, params double[] param)
    {
        var p = param[0];
        if (value < 0)
            return 0.5 * Math.Exp(p * value);
        return 1 - 0.5 * Math.Exp(-p * value);
    }
}
```

```
public static class ListCrvExtension
{
    public delegate double DistributionDelegate(double x,
                                                   params double[] param);

    public static double GetDispersion(this List<double> sequence,
                                        double mathExpectation)
    {
        var sum = sequence.Sum(seq => (seq - mathExpectation) * (seq -
                                                   mathExpectation));
        return  sum / (sequence.Count - 1);
    }

    public static double GetMathExpectation(this List<double> sequence)
    {
        return sequence.Sum()/sequence.Count;
    }

    public static double CriterionPearson(this List<double> sequence,
                DistributionDelegate distribution, params double[] param)
    {
        int j = 0;
```

```
        double xi = 0;
        var n = sequence.Count;
        double minValue = sequence.Min();
        double cellSize = (sequence.Max()- minValue) / Crv.NumberSection;
        sequence.Sort();

        for (var i = 1; i <= Crv.NumberSection; i++)
        {
            var frequency = 0;
            var border = minValue + (cellSize * i);
            while ((j < n) && (sequence[j] < border))
            {
                j++;
                frequency++;
            }
            var probability = distribution( minValue + cellSize * i, param) -
                          distribution( minValue + cellSize * (i-1), param);
            xi += (frequency - n * probability) * (frequency - n * probability)
                                   / n * probability;
        }
        return xi;
    }

    public static double CriterionKolmogorov(this List<double> sequence,
                        DistributionDelegate distribution, params double[] param)
    {
        double supr = 0;
        int n = sequence.Count;
        sequence.Sort();

        for (int i = 0; i < n; i++)
        {
            if (supr < Math.Abs((double) (i + 1) / n - distribution(sequence[i],
                                                                      param)))
                supr = Math.Abs((double) (i + 1) / n - distribution(sequence[i],
                                                                      param));
        }
        return Math.Sqrt(n) * supr;
    }

}
```

Результат выполнения программы:

```
log>    NORMAL DISTRIBUTION
        Criterion Pearson's: 0,08621081331769144
        Criterion Kolmogorov's: 0,5618571644432927
        Calculated math expectation: -0,016948905596951527
        Theoretical math expectation: 0
        Calculated dispersion: 14,876571775705962
        Theoretical dispersion: 16

log>    LOGNORMAL DISTRIBUTION
        Criterion Pearson's: не число
        Criterion Kolmogorov's: 0
        Calculated math expectation: 6150,188322616885
        Theoretical math expectation: 2,872649550817832E+56
        Calculated dispersion: 4383668439,338079
        Theoretical dispersion: 1,2472475573565074E+224

log>    LOGISTICS DISTRIBUTION
        Criterion Pearson's: 0,029712194509738277
        Criterion Kolmogorov's: 0,8074369870541868
        Calculated math expectation: 0,9754094047326202
        Theoretical math expectation: 1
        Calculated dispersion: 3,2468510502230594
        Theoretical dispersion: 3,289868133696453


##########################################################

log>    NORMAL DISTRIBUTION #2
        Criterion Pearson's: 0,07065270662933232
        Criterion Kolmogorov's: 0,5465065130797381
        Calculated math expectation: -4,929543494633187
        Theoretical math expectation: -5
        Calculated dispersion: 26,610430304619516
        Theoretical dispersion: 25

log>    LAPLACE DISTRIBUTION
        Criterion Pearson's: 0,2060337537563545
        Criterion Kolmogorov's: 0,6260158338597166
        Calculated math expectation: 0,012490880127655l5
        Theoretical math expectation: 0
        Calculated dispersion: 2,335901267074439
        Theoretical dispersion: 2

log>    EXPONENTIAL DISTRIBUTION
        Criterion Pearson's: 0,010636467946613904
        Criterion Kolmogorov's: 0,5783724632446494
        Calculated math expectation: 0,2440931613853039
        Theoretical math expectation: 0,25
        Calculated dispersion: 0,05864287511013637
        Theoretical dispersion: 0,0625
```

ЛАБОРАТОРНАЯ РАБОТА 4.1

Условие:
Вычислить интеграл методом Монте-Карло:

Теория:
Метод Монте-Карло приближенного вычисления интеграла:

Необходимо вычислить $\int\limits_{x_0}^{x_1} g(x)dx$.

Пусть η - произвольная случайная величина с плотностью распределения $P_\eta(x),\ x \in [x_0, x_1]$, имеющая конечный момент второго порядка.

Пусть $\xi = \dfrac{g(\eta)}{P_\eta(\eta)}$. Тогда

$$M\{\xi\} = a,\ D\{\xi\} < \infty.$$

В качестве приближенного значения a можно взять

$$\frac{1}{n}\sum_{i=1}^{n}\xi_i = \frac{1}{n}\sum_{i=1}^{n}\frac{g(\eta_i)}{P_\eta(\eta_i)}.$$

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

Код программы:

```
internal static class Program{
    private static double GetRandomNumber(double minimum, double maximum)
    {
        return new Random().NextDouble() * (maximum - minimum) + minimum;
    }

    private static void Main(string[] args){
        const int n = 1000000;
        var x = new double[n];
        var a = new double[n];
        const double maxLim = (5 * Math.PI) / 7;
        double sum = 0;
        for (var i = 0; i < n; i++)
        {
            x[i] = GetRandomNumber(0, maxLim);
            a[i] = Math.Cos(x[i] + Math.Sin(x[i]));

            sum += a[i];
        }
        Console.WriteLine((sum / n) * maxLim);
    }
}
```

Результат выполнения программы:

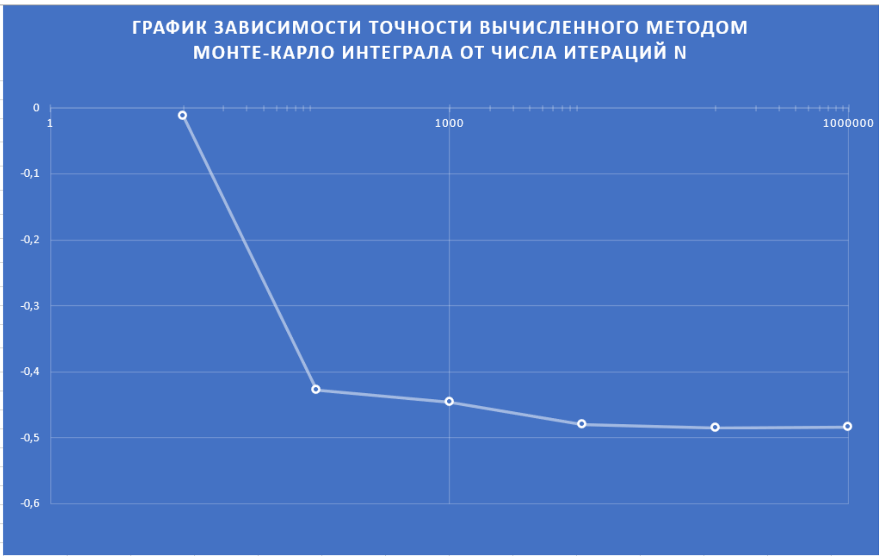
```
-0,4880660097517646


Process finished with exit code 0.
```

График:

$$\int_0^{5\pi/7} \cos(x + \sin x)\,dx$$

N	Result
10	-0,012259738
100	-0,428277977
1000	-0,446558814
10000	-0,480001101
100000	-0,484552914
1000000	-0,484143373

Exact value
-0,485736144



ЛАБОРАТОРНАЯ РАБОТА 4.2

Условие:

Вычислить двойной интеграл методом Монте-Карло:

Код программы:

```
internal static class Program {
    private static void Main() {
        double exact_value = 3.21825;
        var rnd = new Random();
        var res = 0.0;
        var n = 1000000;
        for (var i = 0; i < n; i++) {
            var x = 0.0;
            var y = 0.0;
            while (true) {
                x = rnd.NextDouble() * 2 * Math.Sqrt(3) - Math.Sqrt(3);
                y = rnd.NextDouble() * 2 * Math.Sqrt(3) - Math.Sqrt(3);
                var value = Math.Pow(x, 2) + Math.Pow(y, 2);
                if (value >= 1 && value < 3)
                    break;
            }

            res += 2 * Math.PI / (Math.Pow(x, 2) + Math.Pow(y, 4));
        }
        Console.WriteLine("Monte-Carlo: {0}\nExact value: {1}", res / n,
                                      exact_value);
    }
}
```

Результат выполнения программы:

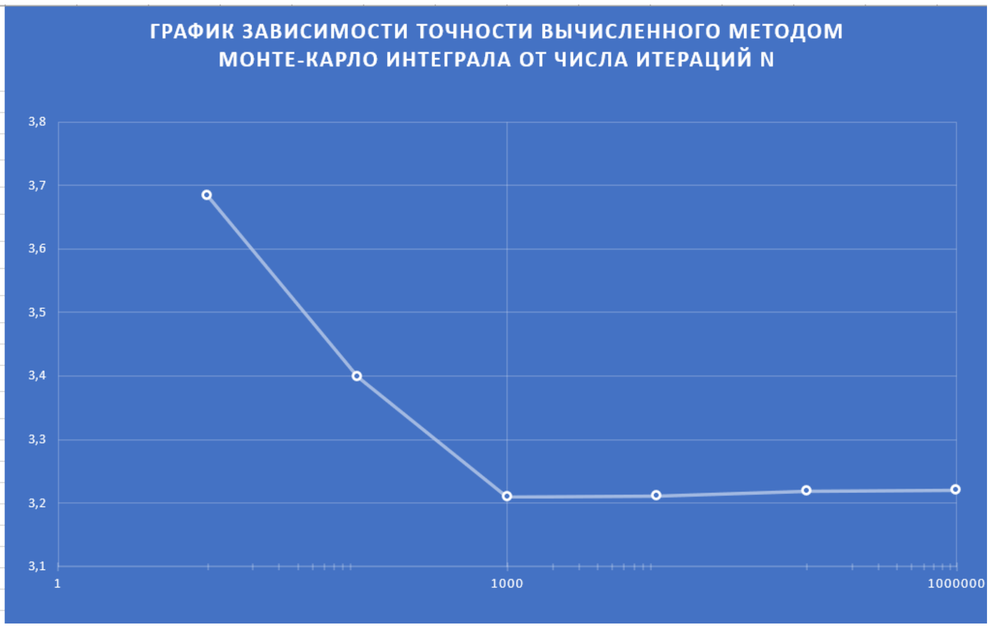
```
Monte-Carlo: 3,217961655035037
Exact value: 3,21825
```

График:

$$\iint\limits_{1\le x^2+y^2<3} \frac{dxdy}{x^2+y^4}$$

N	Result
10	3,684303521
100	3,398725504
1000	3,208914405
10000	3,211762243
100000	3,219166746
1000000	3,219574338

Exact value
3.21825



ЛАБОРАТОРНАЯ РАБОТА 5

Условие:

Решить систему линейных алгебраических уравнений методом Монте-Карло.

Теория:

Метод Монте-Карло приближенного решения системы линейных алгебраических уравнений:

Необходимо решить систему, представленную в виде $x = Ax + f$, где $x = (x_1, ..., x_n)^T$, $f = (f_1, ..., f_n)^T$, $A = (a_{ij}),\ i, j = \overline{1, n}$, собственные значения A по модулю меньше 1.

Наша цель – вычислить скалярное произведение вектора решения $x = (x_1, ..., x_n)^T$ с некоторым вектором $h = (h_1, ..., h_n)^T$.

Рассмотрим цепь Маркова с параметрами $\pi = (\pi_1, ..., \pi_n)^T,\ P = (p_{ij}),$ такими что

$\pi_i \ge 0,\ \sum_{i=1}^{n} \pi_i = 1;$

$p_{ij} \ge 0,\ \sum_{j=1}^{n} p_{ij} = 1;$

$\pi_i > 0,$ если $h_i \ne 0;$

$p_{ij} > 0,$ если $a_{ij} \ne 0.$

Положим

$$g_i^{(0)} = \begin{cases} h_i / \pi_i,\ \pi_i > 0 \\ 0,\ \pi_i = 0 \end{cases},\ g_i^{(k)} = \begin{cases} a_{ij} / p_{ij},\ p_{ij} > 0 \\ 0,\ p_{ij} = 0 \end{cases}.$$

Выберем некоторое натуральное N и рассмотрим случайную величину

$$\xi_N = \sum_{m=0}^{N} Q_m f_{i_m},$$

где $i_0 \rightarrow i_1 \rightarrow \ldots \rightarrow i_m$ - траекторая цепи Маркова.

Q_m опряделяется как:

$$Q_0 = g_{i_0}^{(0)},\ Q_m = Q_{m-1} g_{i_{m-1} i_m}^{(m)}.$$

Тогда скалярное произведение вектором h и x приблизительно равно $E\{\xi_N\}$.

Можем найти x, скалярно умножая его на векторы h у которых в одной позиции стоит 1, а в остьльных – 0.

В данной работе выбиралось $\pi_i = \frac{1}{2},\ \mathrm{p}_{ij} = \frac{1}{2},\ N$=1000, n=1000.

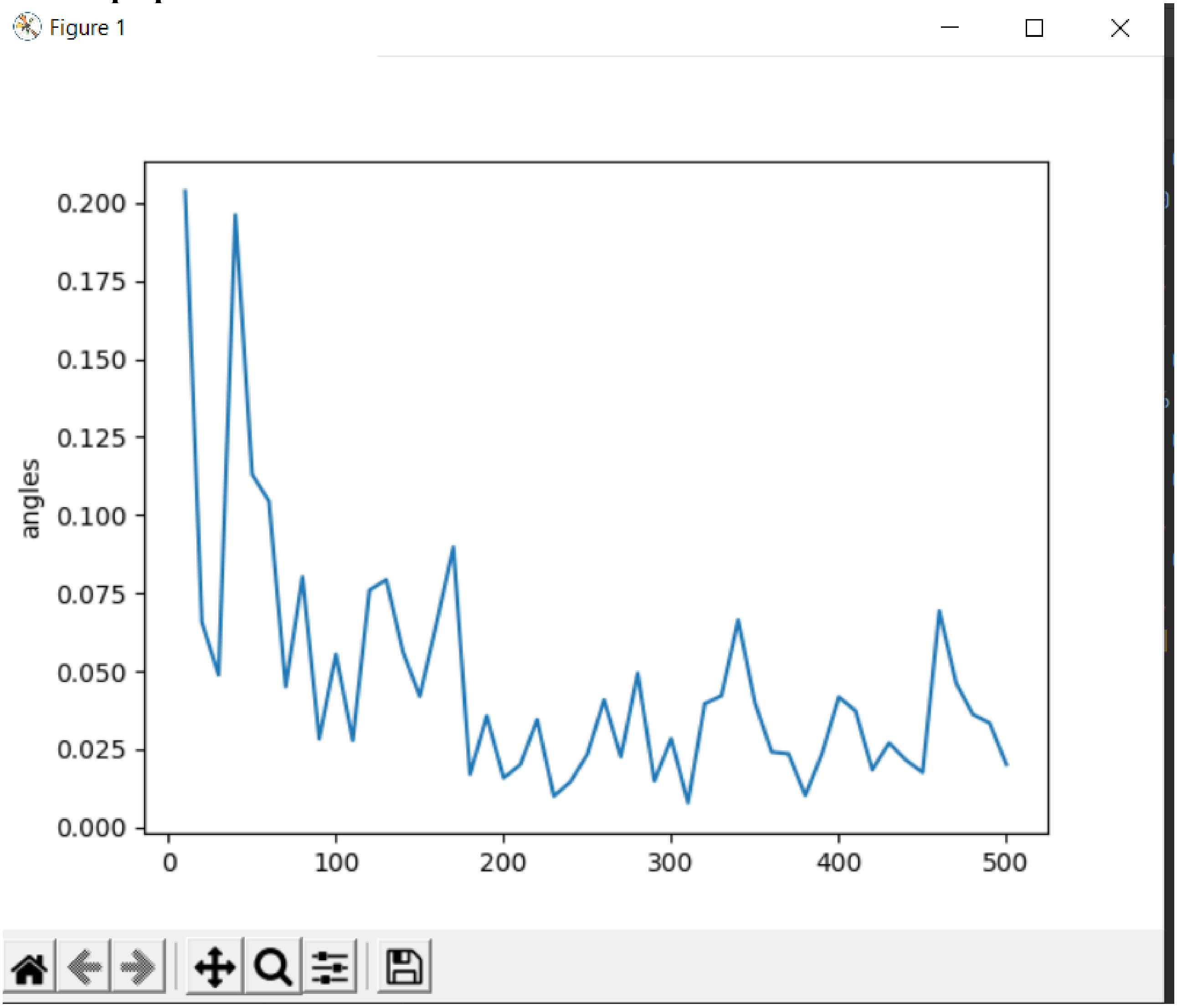
Код программы:

```
internal static class Program {

    private static Vector<double> MonteCarlo(Matrix<double> a,
                                             Vector<double> b, int m, int n) {
        var rnd = new Random();
        var res = Vector<double>.Build.Dense(3);
        var h = Matrix<double>.Build.DenseIdentity(3);
        var p = Matrix<double>.Build.Dense(3, 3, 1.0 / 3.0);
        var pi = Vector<double>.Build.Dense(3, 1.0 / 3.0);

        for (var i = 0; i < 3; i++){
            var chain = Vector<double>.Build.Dense(n+1);
            var q = Vector<double>.Build.Dense(n+1);
            var ksi = Vector<double>.Build.Dense(m);

            for (var j = 0; j < m; j++){
                for (var k = 0; k < n + 1; k++)
                {
                    var r = rnd.NextDouble();
                    chain[k] = (r < pi[0]) ? 0 : (r < pi[0] + pi[1] ? 1 : 2);
                }
                q[0] = pi[(int) chain[0]] > 0 ?
                       h[i, (int) chain[0]] / pi[(int) chain[0]] : 0;
                for (var k = 1; k < n + 1; k++) {
                    var row = (int) chain[k - 1];
                    var column = (int) chain[k];
                    q[k] = p[row, column] > 0 ?
                              q[k - 1] * a[row, column] / p[row, column] : 0;
                }
                for (var k = 0; k < n + 1; k++){
                    ksi[j] += q[k] * b[(int) chain[k]];
                }
            }
            for (var k = 0; k < m; k++) {
                res[i] += ksi[k];
            }
            res[i] /= m;
        }
        return res;
    }

    private static void Main(string[] args)
    {
        //Матрица заранее приведена к необходимому виду
        var A = Matrix<double>.Build.DenseOfRowArrays(
            new double[] {-0.1, 0.1, -0.2},
            new double[] {-0.1, 0.5, -0.3},
            new double[] {0.3, 0.1, -0.3}
        );
        var f = Vector<double>.Build.DenseOfArray(
            new double[] {-3.0, 1.0, 4.0}
        );
        var res = MonteCarlo(A, f, 100, 10000);
        Console.WriteLine("WolframAlpha exec: (-3.07, 1.14, 2.456)");
        Console.WriteLine(res);

    }
}
```

Результат выполнения программы:

```
WolframAlpha exec: (-3.07, 1.14, 2.456)
DenseVector 3-Double
-3,46226
 1,44105
 1,40091
```

График:



СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

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.