Министерство образования Республики Беларусь

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

Факультет прикладной математики и информатики

**Кафедра математического моделирования и анализа данных**

Кражевский Алексей Игоревич

Отчет по лабораторным работам по курсу

“Математическое моделирование ”

студента 2 курса 15 группы

|  |  |
| --- | --- |
| Работа сдана 2022г. | **Преподаватель** |
| зачтена \_\_\_\_\_\_\_\_\_\_ 2022 г.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись преподавателя) | *Лобач Сергей Викторович*  Ассистент кафедры математического моделирования и анализ данных ФПМИ |
|  |  |

*Минск 2022*

**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* = 231 .
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 χ2-критерия Пирсона с уровнем значимости ε = 0.05.

**Вариант задания:**

8) a0 = β = 262 147, K = 256

**Теория:**

**Мультипликативный конгруэнтный метод:**

Псевдослучайная последовательность  строится по следующим рекуррентным формулам:

где  - параметры датчика:  - множитель (*<M*), *M* – модуль,  - стартовое значение (нечетное число).

В данной работе брались значения: *M*=2147483648, ==262 147.

**Метод Маклорена-Марсальи:**

Пусть  - псевдослучайные последовательности, порожденные независимо работающими датчиками;  - результирующая псевдослучайная последовательность реализация БСВ;

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

Процесс вычисления  включает следующие этапы:

- первоначальное заполнение таблицы

*V*: 

- случайный выбор из таблицы:



-обновление табличных значений:

.

В данной работе в качестве  бралась последовательность (из 1000 элементов), полученная мультипликативным конгруэнтным методом, описанным выше. В качестве , бралась последовательности (из 1000) элементов, полученная аналогичным способом с тем же M и . *K*=256.

** - критерий согласия Пирсона:**

Область возможных значений случайной величины разбивается на интервалы .

Рассматривается следующая статистика,

,

*n* – объем выборки,

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

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

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

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

**Критерий согласия Колмогорова:**

Рассматривается статистика:



где

,

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

**Код программы:**

*import* math

*from* random *import* random

*import* numpy *as* np

*#константы*

k = 256

a0 = 262147

b = a0

M = math.pow(2, 31)

now = a0

*def* *mult\_cong\_met*():

    global now

    now = (now \* b) % M

*return* now / M

v = []

*def* *first*():

*for* \_ *in* range(k):

        v.append(mult\_cong\_met())

*def* *maclaren*():

    indx = int(random() % k)

    num = v[indx]

    v[indx] = mult\_cong\_met()

*return* num

'''

v - объем выборки

n - количество разбиений (0,1)

c - критическое значение лямбда = 1.36 для 0,05

func - закон распределения СВ

sqrt(v) \* Dn <= LAMBDAa, где

Dn = max|Fn(x) - F(x)|

'''

*def* *kolmogorov*(*v*, *n*, *c*, *func*):

    print ('Working', *end* = '')

    cof = 0

*for* i *in* range (n):

        count = 0

*for* \_ *in* range (v):

*if* func() *<* float(i / n):

                count += 1

*#print (i, ': ', count)*

        print('.', *end* = '')

        f = float(count / v)

*if* cof *<* abs(f - i / n):

            cof = abs(f - i / n)

        '''

        if cof < max(i / n - f, f - (i - 1) / n):

            cof = max(i / n - f, f - (i - 1) / n)

        '''

*#print (math.sqrt(v) \* cof)*

*if* math.sqrt(v) \* cof *<* c:

*return* 'pass'

*else*:

*return* 'not pass'

'''

ans = (Oi - Ei)^2 / Ei

'''

*def* *pirson*(*v*, *n*, *c*, *func*):

    arr = np.zeros(n+1, *dtype* = int)

*#подсчет числа степеней свободы*

*for* \_ *in* range (v):

        a = round(func() \* n)

        arr[a] += 1

*#подсчет ответа по формуле*

    ans = 0

*for* i *in* range(n):

        ans += math.pow(arr[i] - v/n, 2) / (v/n)

*#print (ans)*

*if* ans *<* c:

*return* 'pass'

*else*:

*return* 'not pass'

*if* \_\_name\_\_ *==* '\_\_main\_\_':

    fout1 = open ('out1.txt', 'w')

    fout2 = open ('out2.txt', 'w')

    fout3 = open ('test\_res.txt', 'w')

    first()

*for* \_ *in* range (1000):

        fout1.write(str(mult\_cong\_met()))

        fout1.write('\n')

*for* \_ *in* range (1000):

        fout2.write(str(maclaren()))

        fout2.write('\n')

    fout1.close()

    fout2.close()

    fout3.write('Test Results:\n')

    fout3.write('\tKolmogorov for maclaren:\n')

    fout3.write(kolmogorov (20000, 250, 1.36, maclaren))

    fout3.write('\n\tPirson for maclaren:\n')

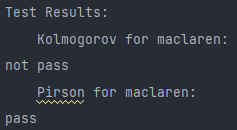
    fout3.write(pirson (1000, 25, 37.6525, maclaren))

    fout3.close()

**Результаты:**

В файле out1.txt находятся 1000 реализаций БСВ мультипликативно-конгуэртного метода, а в файле out2.txt – 1000 реализаций БСВ методом макларена-марсальи.

Результат критерия Колмогорова и Пирсона:



**Лабораторная работа 2.**

**Условие:**

Смоделировать дискретную случайную величину (задания на стр. 18-22). Исследовать точность моделирования.

1. Осуществить моделирование *n* = 1000 реализаций СВ из заданных дискретных распределений.
2. Вывести на экран несмещенные оценки математического ожидания и дисперсии, сравнить их с истинными значениями.
3. Для каждой из случайных величин построить свой χ2-критерием Пирсона с уровнем значимость ε=0.05. Проверить, что вероятность ошибки I рода стремится к 0.05.
4. Осуществить проверку каждой из сгенерированных выборок каждым из построенных критериев.

**Вариант задания:**

8) Отрицательное биномиальное – (r,p), r = 6, p = 0.25; Пуассона – П(λ), λ = 3

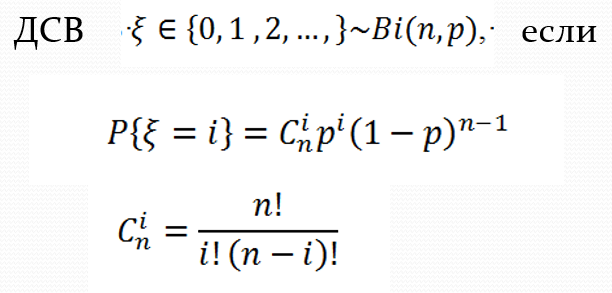
**Теория:**

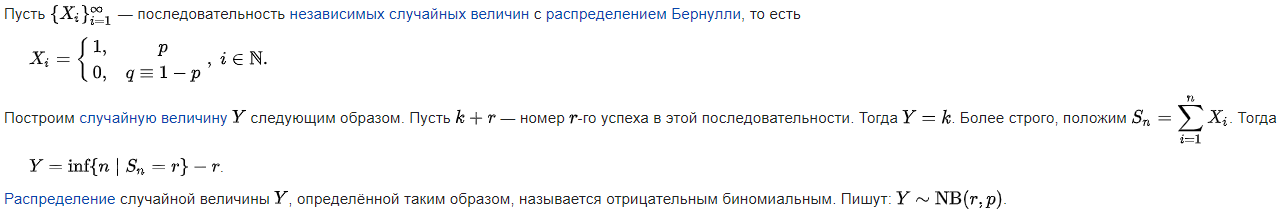
**Распределение Пуассона (с параметром ):**

Случайная величина  принимает только целые неотрицательные значения, причем 

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  проверялось, в какой интервал попадает элемент последовательности БСВ.

**Биномиальное распределение:**

****

**Отрицательное биномиальное распределение:**

**Код программы:**

**main.py**

*from* puasson *import* puasson\_gen, puasson\_e, puasson\_d

*from* binomial *import* bernoulli\_gen, binomial\_gen

*from* neg\_binomial *import* neg\_binomial\_e, neg\_binomial\_gen, neg\_binomial\_d

*from* mcg *import* mcg

*import* matplotlib.pyplot *as* plt

*if* \_\_name\_\_ *==* '\_\_main\_\_':

    g = puasson\_gen(3, mcg(2\*\*31, 16807, 16807))

    seq = [n *for* n *in* g]

*#print(seq)*

    plt.plot(seq)

    plt.show()

    Ep = puasson\_e(3)

    Dp = puasson\_d(3)

    print(Ep, Dp)

    '''

    g = binomial\_gen(4, 0.25, mcg(2\*\*31, 16807, 16807))

    seq = [n for n in g]

    plt.plot(seq)

    plt.show()

    #print(seq)

    '''

    g = neg\_binomial\_gen(6, bernoulli\_gen(0.25, mcg(2\*\*31, 16087, 16087, *count*=100000)))

    seq = [n *for* n *in* g]

*#print(seq)*

    plt.plot(seq)

    plt.show()

    e = neg\_binomial\_e(6, 0.25)

    d = neg\_binomial\_d(6, 0.25)

**mcg.py**

*def* *lcg*(*modulus*, *a*, *c*, *seed*, *count*=1000):

*for* \_ *in* range(count):

        seed = (a \* seed + c) % modulus

*yield* seed / modulus

*def* *mcg*(*modulus*, *a*, *seed*, *count*=1000):

*yield from* lcg(modulus, a, 0, seed, count)

**neg\_binomial.py**

*def* *neg\_binomial\_gen*(*r*, *random\_generator*, *count*=1000):

    ''' Y = inf{n | Sn = r} - r '''

*for* \_ *in* range(count):

        sn = 0

        indx = 0

        n = -1

*for* rand *in* random\_generator:

*#print (rand)*

            sn += rand

*if* sn *==* r:

                n = indx

*break*

            indx += 1

*yield* n - r

*def* *neg\_binomial\_d*(*r*, *p*):

*return* r \* (1-p) / (p \*\* 2)

*def* *neg\_binomial\_e*(*r*, *p*):

*return* r \* (1-p) / p

**puasson.py**

*import* math

*def* *puasson\_gen*(*lambd*, *random\_generator*):

*for* r *in* random\_generator:

        p = math.exp(-lambd)

        a = 1.0

        x = 0

*while* True:

            x += 1

            a \*= r

*if* a *<=* p:

*break*

*yield* x - 1

*def* *puasson\_e*(*lam*):

*return* lam

*def* *puasson\_d*(*lam*):

*return* lam

**binomial.py**

*def* *binomial\_gen*(*n*, *p*, *random\_generator*):

*for* r *in* random\_generator:

        x = 0

*for* \_ *in* range(n):

*if* r *<* p:

                x += 1

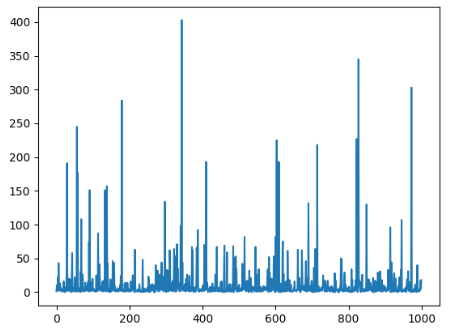
*yield* x

*def* *bernoulli\_gen*(*p*, *random\_generator*):

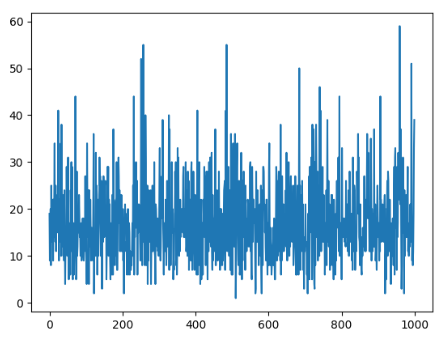
*yield from* binomial\_gen(1, p, random\_generator)

Результат:

Пуассон:



Отрицательное биномиальное:



Мат. ожидание и дисперсия:



**Лабораторная работа 3.**

**Условие:**

Смоделировать непрерывную случайную величину (задания на стр. 25-47). Исследовать точность моделирования.

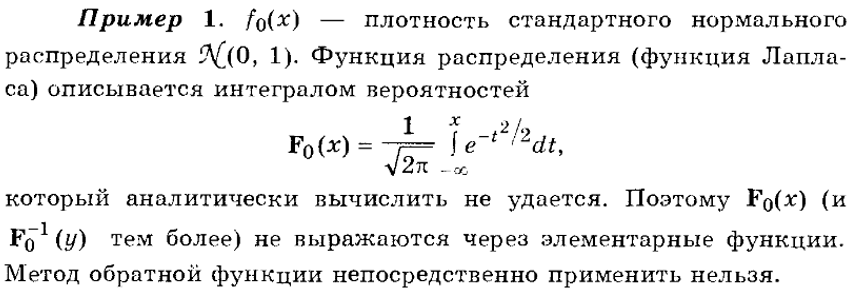
1. Осуществить моделирование *n* = 1000 реализаций СВ из нормального закона распределения *N*(*m*, *s*2) с заданными параметрами. Вычислить несмещенные оценки математического ожидания и дисперсии, сравнить их с истинными.
2. Смоделировать *n* = 1000 СВ из заданных абсолютно непрерывных распределений. Вычислить несмещенные оценки математического ожидания и дисперсии, сравнить их с истинными значениями (если это возможно).
3. Для каждой из случайных величин построить свой критерий Колмогорова с уровнем значимость ε=0.05. Проверить, что вероятность ошибки I рода стремится к 0.05.
4. Для каждой из случайных величин построить свой χ2-критерий Пирсона с уровнем значимость ε=0.05. Проверить, что вероятность ошибки I рода стремится к 0.05.
5. Осуществить проверку каждой из сгенерированных выборок каждым из построенных критериев.

**Условие задания:**

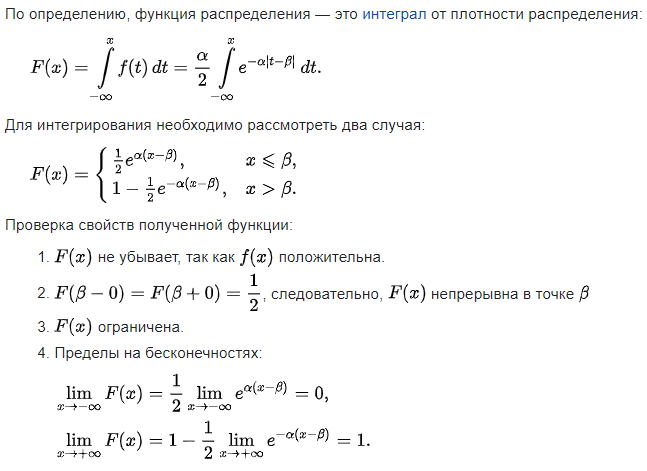
8) m = -5, s2 = 25; Лапласа L(a), a = 1; Экспоненциальное Е(a), a = 4.

**Теория:**

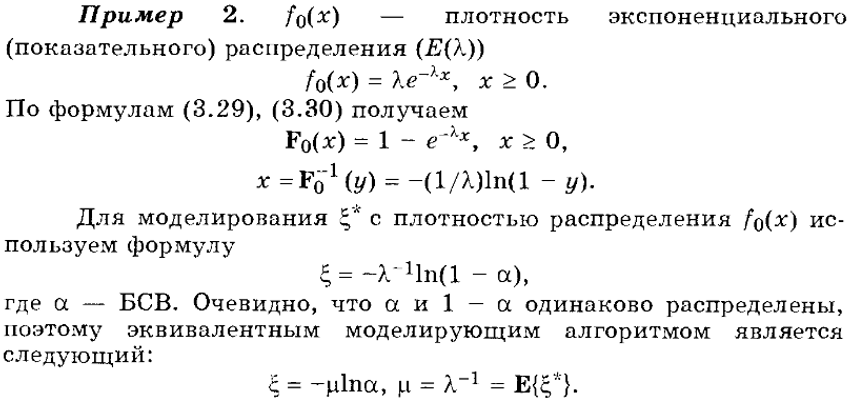
**Нормальный закон распределения:**

****

**Распределение Лапласа:**

****

**Экспоненциальное распределение:**

****

**Код программы:**

**main.py**

*import* math  
*from* normal *import* normal\_generator  
*from* mult\_met *import* mcg  
*from* laplace *import* laplace\_generator  
*from* exponential *import* exponential\_generator  
*import* matplotlib.pyplot *as* plt  
*from* criteria *import* pearson\_criterion, kolmogorov\_criterion  
*from* disp *import* expectation, dispersion  
  
  
*def* test\_criterion(criterion, params):  
 *try*:  
 *return* "PASS" *if* criterion(\*params) *else* "FAIL"  
 *except* (ArithmeticError, ValueError):  
 *return* "NOT PASS"  
  
  
*def* normal\_distribution(x, mu, sigma2):  
 sigma = math.sqrt(sigma2)  
 *return* math.exp(-0.5 \* ((x - mu) / sigma) \*\* 2) / (sigma \* math.sqrt(2 \* math.pi))  
  
  
*def* lognormal\_distribution(x, mu, sigma2):  
 *return* math.exp(normal\_distribution(x, mu, sigma2))  
  
  
*def* exponential\_distribution(x, a):  
 *return* a\*math.exp(-a\*x)  
  
  
*def* laplace\_distribution(x, a):  
 *return* math.copysign(1, -x) \* math.exp(-a \* x) / 2  
  
  
*def* weibull\_distribution(x, a, b):  
 *return* 1 - math.exp(-a \* x \*\* b)  
  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 *#print('Normal distribution:')* normally\_distributed = normal\_generator(mu=-5, sigma2=25,  
 modulus=2 \*\* 31, a=16807, seed=16807,  
 count=10000)  
 *#print(list(normally\_distributed))* seq = [g *for* g *in* normally\_distributed]  
 plt.plot(seq)  
 plt.show()  
  
 *#print('Laplace distribution:')* gen = mcg(modulus=2 \*\* 31, a=16807, seed=16807, count=1000)  
 laplace\_distributed = list(laplace\_generator(1, gen))  
 *#print(laplace\_distributed)* seq = [g *for* g *in* laplace\_distributed]  
 plt.plot(seq)  
 plt.show()  
  
 *#print('Exponential distribution:')* gen = mcg(modulus=2 \*\* 31, a=16807, seed=16807, count=1000)  
 exponentially\_distributed = list(exponential\_generator(4, gen))  
 *#print(exponentially\_distributed)* seq = [g *for* g *in* exponentially\_distributed]  
 plt.plot(seq)  
 plt.show()  
  
 mcg\_params = dict(modulus=2 \*\* 31, a=16807, seed=16807, count=1000)  
 distr = {"Normal distribution":  
 (normal\_generator(mu=5, sigma2=25,  
 \*\*mcg\_params), *lambda* x: normal\_distribution(x, mu=5, sigma2=25)),  
 "Exponential distribution":  
 (exponential\_generator(4,  
 mcg(\*\*mcg\_params)), *lambda* x: exponential\_distribution(x, 4)),  
 "Laplace distribution":  
 (laplace\_generator(1,  
 mcg(\*\*mcg\_params)), *lambda* x: laplace\_distribution(x, 1))}  
  
 *for* name, (generator, func) *in* distr.items():  
 print(name)  
 seq = list(generator)  
 *if* func:  
 print("Pearson criterion:", test\_criterion(pearson\_criterion, (seq, 25, func)))  
 print("Kolmogorov criterion:", test\_criterion(kolmogorov\_criterion, (seq, 25, func)))  
 m = expectation(seq)  
 print("Mean", m)  
 print("Disp", dispersion(seq, m))  
 print()

**criteria.py**

*import* math  
  
  
*def* empirical(x, seq):  
 res = 0  
 *for* item *in* seq:  
 *if* item < x:  
 res += 1  
 *return* res / len(seq)  
  
  
*def* kolmogorov\_criterion(seq, parts, distr\_func):  
 lo = min(seq)  
 step = (max(seq) - min(seq)) / parts  
 dn = 0  
 *for* i *in* range(1, parts + 1):  
 point = lo + i \* step  
 temp = math.fabs(empirical(point, seq) - distr\_func(point))  
 *if* dn < temp:  
 dn = temp  
 *return* math.sqrt(len(seq)) \* dn < 1.36  
  
  
*def* pearson\_criterion(seq, parts, distr\_func):  
 lo = min(seq)  
 *if* lo >= 0:  
 lo = 0  
 step = (max(seq) - lo) / parts  
 frequency = [0] \* parts  
 *for* item *in* seq:  
 index = int((item - lo) / step)  
 *if* index == parts:  
 index -= 1  
 frequency[index] += 1  
 res = 0  
 *for* i *in* range(1, parts + 1):  
 delta = len(seq) \* (distr\_func(lo + i \* step) - distr\_func(lo + (i - 1) \* step))  
 res += pow((frequency[i - 1] - delta), 2) / delta  
 *return* res < 37.7

**disp.py**

*def* expectation(vec):  
 res = 0  
 *for* item *in* vec:  
 res += item  
 *return* res / *len*(vec)  
  
  
*def* dispersion(vec, avg):  
 res = 0  
 *for* item *in* vec:  
 res += *pow*(item - avg, 2)  
 *return* res / (*len*(vec) - 1)

**exponential.py**

*import* math  
  
  
*def* exponential\_generator(a, generator):  
 *for* n *in* generator:  
 *yield* -math.log(n) / a

**laplace.py**

*from* mult\_met *import* mcg  
*import* math  
  
  
*def* laplace\_generator(a, generator):  
 *for* n *in* generator:  
 *if* n < 0.5:  
 *yield* math.log(2 \* n) / a  
 *else*:  
 *yield* -math.log(2 \* (1 - n)) / a

**mult\_meg.py**

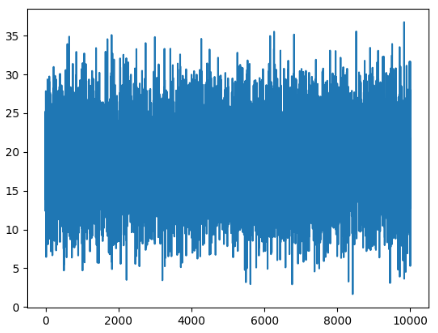
*def* lcg(modulus, a, c, seed, count=100):  
 *"""  
 Linear congruential generator.  
 """  
 for* i *in range*(count):  
 seed = (a \* seed + c) % modulus  
 *yield* seed / modulus  
  
  
*def* mcg(modulus, a, seed, count=100):  
 *"""  
 Multiplicative congruential generator.  
 """  
 yield from* lcg(modulus, a, 0, seed, count)

**normal.py**

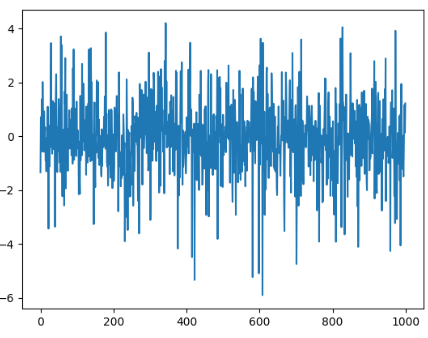
*from* mult\_met *import* mcg  
*import* math  
  
  
*def* normal\_generator(mu, sigma2, modulus, a, seed, count):  
 sequence = *list*(mcg(modulus, a, seed, count \* 12))  
 *for* i *in range*(count):  
 *yield* mu + math.sqrt(sigma2) \* *sum*(sequence[12 \* i + j] *for* j *in range*(12)) - 6

**Результат:**

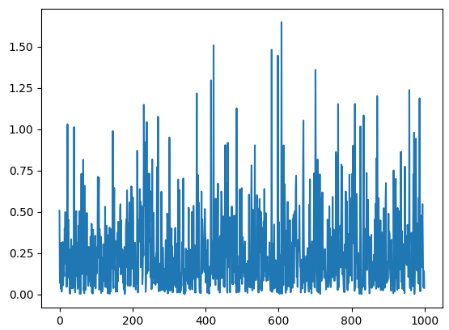
Нормальное распределение:



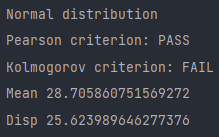
Распределение Лапласа:

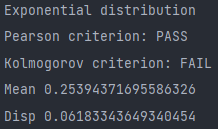


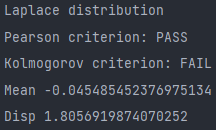
Экспоненциальное распределение:



Критерии для распределений:







**Лабораторная работа 4.**

**Условие:**

Вычислить значение интеграла, используя метод Монте-Карло. Оценить точность.

1. По методу Монте-Карло вычислить приближенное значения интегралов.
2. Сравнить полученное значение либо с точным значением (если его получится вычислить), либо с приближенным, полученным в каком-либо математическом пакете (например, в mathematica). Для этого построить график зависимости точности вычисленного методом Монте-Карло интеграла от числа итераций *n*.

**Вариант задания:**

|  |  |  |
| --- | --- | --- |
| 8 |  |  |

**Теория:**

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

Необходимо вычислить .

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

Пусть  Тогда 

В качестве приближенного значения *a* можно взять



В данной работе в качестве  бралась случайная величина, равномерно распределенная на отрезках, указанных в задании.

**Код программы:**

**main.py**

*from* MonteCarlo *import* monte\_carlo, monte\_carlo\_double, compare\_to\_solution  
*from* math *import* e, tan  
*import* matplotlib.pyplot *as* plt  
  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 integral1 = monte\_carlo(0, 2, *lambda* x: ((e \*\* (-x)) \* (1 + x) \*\* 0.5), 1000)  
 print(f"1. {integral1}")  
  
 integral2 = monte\_carlo\_double(0, 1, 0, 2, *lambda* x, y: x \*\* 2 + y \*\* 2, 1000)  
 print(f"2. {integral2}")  
  
 *# compare monte-carlo1 result to solution = 1.11007* seq1 = []  
 *for* i *in* range(1000, 10000, 50):  
 seq1.append(compare\_to\_solution(monte\_carlo(0, 2, *lambda* x: ((e \*\* (-x)) \* (1 + x) \*\* 0.5), i), 1.11007))  
  
 plt.plot(seq1)  
 plt.title("Monte-Carlo method 1")  
 plt.xlabel("Iterations (1000 + x\*50)")  
 plt.ylabel("Precision")  
 plt.show()  
  
 *# compare monte-carlo2 result to solution = 3.3333333333* seq2 = []  
 *for* i *in* range(1000, 10000, 50):  
 seq2.append(compare\_to\_solution(monte\_carlo\_double(0, 1, 0, 2, *lambda* x, y: x \*\* 2 + y \*\* 2, i), 3.3333333333))  
  
 plt.plot(seq2)  
 plt.title("Monte-Carlo method 2")  
 plt.xlabel("Iterations (1000 + x\*50)")  
 plt.ylabel("Precision")  
 plt.show()

**MonteCarlo.py**

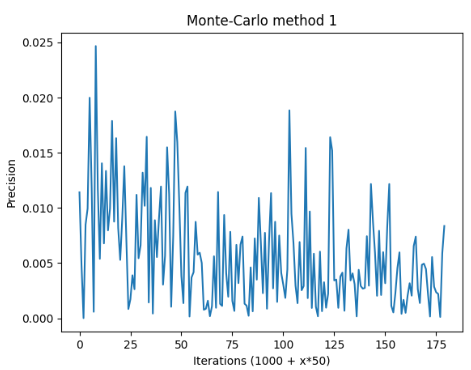
*import* numpy *as* np  
  
  
*def* monte\_carlo(left\_bound, right\_bound, function, n) -> float:  
 *"""  
 Monte-Carlo method implementation  
 returns the value of the integral  
 """* sum\_ = 0  
 *for* i *in* range(n):  
 sum\_ += function(np.random.uniform(left\_bound, right\_bound)) \* (right\_bound - left\_bound)  
  
 *return* sum\_ / n  
  
  
*def* monte\_carlo\_double(left\_bound1, right\_bound1, left\_bound2, right\_bound2, function, n) -> float:  
 *"""  
 Monte-Carlo method implementation for double integration  
 returns the value of the integral  
 """* sum\_ = 0  
  
 *for* i *in* range(n):  
 u1 = np.random.uniform(left\_bound1, right\_bound1)  
 u2 = np.random.uniform(left\_bound2, right\_bound2)  
  
 sum\_ += function(u1, u2) \* (right\_bound1 - left\_bound1) \* (right\_bound2 - left\_bound2)  
  
 *return* sum\_ / n  
  
  
*def* compare\_to\_solution(generated\_value, solution\_value) -> float:  
 *"""  
 Compare generated value with solution value  
 returns precision  
 """  
 return* abs(generated\_value - solution\_value)

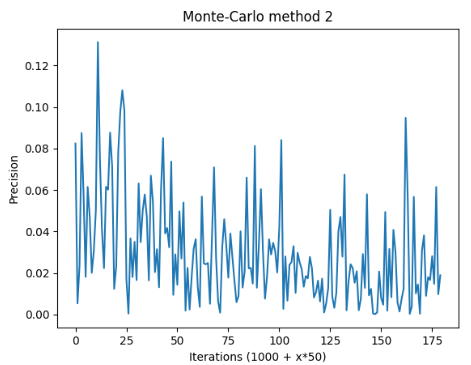
Результат:

Результаты вычисление интегралов:



Графики:





**Лабораторная работа 5.**

**Условие:**

Решить систему линейных уравнений, используя метод Монте-Карло.

1. Решить систему линейных алгебраических уравнений  методом Монте-Карло.
2. Сравнить с решением данного уравнения, полученным в произвольном математическом пакете.

Построить график зависимости точности решения от длины цепи маркова и числа смоделированных цепей маркова.

**Вариант задания:**

8) 

**Теория:**

**Метод Монте-Карло приближенного решения системы линейных алгебраических уравнений:**

Необходимо решить систему, представленную в виде , где , собственные значения *A* по модулю меньше 1.

Наша цель – вычислить скалярное произведение вектора решения  с некоторым вектором .

Рассмотрим цепь Маркова с параметрами  такими что





 если 

 если 

Положим



Выберем некоторое натуральное *N* и рассмотрим случайную величину



Где 🡪🡪…🡪 - траекторая цепи Маркова.

*Qm* опряделяется как:



Тогда скалярное произведение вектором *h* и *x* приблизительно равно .

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

**Код программы:**

**main.py**

*import* matplotlib.pyplot *as* plt  
*import* numpy *as* np  
*from* MonteCarlo *import* monte\_carlo\_solve, norm  
*import* sympy *as* sp  
  
  
*if* \_\_name\_\_ == '\_\_main\_\_':  
 *# data according to the task* a = [[1.2, -0.3, 0.4],  
 [0.4, 0.7, -0.2],  
 [0.2, -0.3, 0.9]]  
  
 f = [-4, 2, 0]  
  
 *# accurate solution = [-2.829, 5.143, 2.343]* n = 3  
 markov\_len = 50  
 markov\_cnt = 1000  
  
 *# x = Bx + f* b = [[-0.2, 0.3, -0.4],  
 [-0.4, 0.3, 0.2],  
 [-0.2, 0.3, 0.1]]  
 *# probability matrix* p = [[1 / 3, 1 / 3, 1 / 3],  
 [1 / 3, 1 / 3, 1 / 3],  
 [1 / 3, 1 / 3, 1 / 3]]  
  
 x = monte\_carlo\_solve(b, n, f, p, markov\_len, markov\_cnt)  
 print(f"[1] Monte-Carlo method: x = {x}")  
  
 *# get an accurate solution using sumpy* a1 = sp.Matrix(a)  
 b1 = sp.Matrix(f)  
 x1 = a1.inv() \* b1  
 print(f"[2] Accurate solution: x1 = {x1}")  
  
 *# difference between solutions* print(f"[3] Norm of x-x1: {norm(x, x1, n)}")  
  
 *# task 3 - plot the solution* X = np.loadtxt('Lab5Cnt.txt')  
 Y = np.loadtxt('Lab5Len.txt')  
 Z = np.loadtxt('Lab5Norms.txt')  
  
 fig = plt.figure()  
 ax = fig.gca(projection='3d')  
  
 ax.plot\_trisurf(X, Y, Z, linewidth=0, antialiased=*True*)  
 plt.show()

**MonteCarlo.py**

*from* numpy *import* random  
*from* math *import* sqrt, pow  
  
  
*def* norm(X, X1, n):  
 *return* sqrt(sum([pow(X[i] - X1[i], 2) *for* i *in* range(n)]))  
  
  
*def* markov\_rand(P, state, size):  
 rand = random.random()  
 *for* i *in* range(size):  
 rand -= P[state][i]  
 *if* rand <= 0:  
 *return* i  
 *return* size - 1  
  
  
*def* monte\_carlo\_solve(b, n, f, P, markov\_len=50, markov\_cnt=1000) -> list:  
 *"""  
 Solves the linear system x = Bx + f using the Monte-Carlo method  
 :return list: x  
 """* solution = []  
 *for* coordinate *in* range(n):  
 x = 0  
 *for* i *in* range(markov\_cnt):  
 prev\_m = coordinate  
 prev\_q = 1  
 *for* j *in* range(1, markov\_len):  
 next\_m = markov\_rand(P, prev\_m, n)  
 next\_q = 0  
 *if* P[prev\_m][next\_m] > 0:  
 next\_q = prev\_q \* b[prev\_m][next\_m] / P[prev\_m][next\_m]  
 x += next\_q \* f[next\_m]  
 prev\_q = next\_q  
 prev\_m = next\_m  
 solution.append(f[coordinate] + x / markov\_cnt)  
 *return* solution

**Результат:**

Вывод результата методом Монте-Карло, точного решения и нормы их разницы:

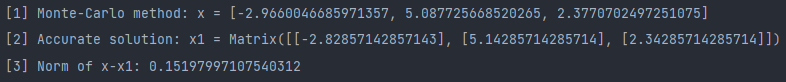
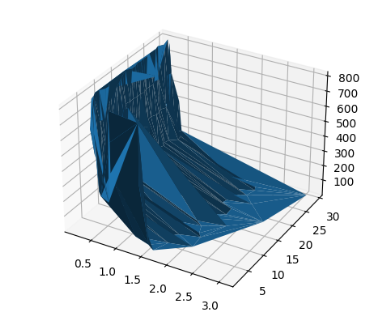


График:



**Литература**

1. Харин Ю.С., Малюгин В.И., Кирлица В.П., Лобач В.И., Хацкевич Г.А. Основы имитационного и статистического моделирования. Учебное пособие. Минск: ДизайнПРО, 1997 – 228 с.
2. Лобач В.И., Кирлица В.П., Малюгин В.И., Сталевская С.Н. Имитационное и статистическое моделирование. Практикум для студентов математических и экономических специальностей. Минск, БГУ, 2004 –189 с.