Федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский государственный технический университет имени Н.Э. Баумана» (национальный исследовательский университет) (МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ ФУНДАМЕНТАЛЬНЫЕ НАУКИ

КАФЕДРА ВЫЧИСЛИТЕЛЬНАЯ МАТЕМАТИКА И МАТЕМАТИЧЕСКАЯ ФИЗИКА (ФН11)

НАПРАВЛЕНИЕ ПОДГОТОВКИ МАТЕМАТИКА И КОМПЬЮТЕРНЫЕ НАУКИ (02.03.01)

О т ч е т

по лабораторной работе № 3

Название лабораторной работы:

Моделирование выборки из абсолютно непрерывного закона распределения методом обратных функций.

Вариант № 9

Дисциплина:

Теория вероятности и математическая статистика

Студент группы ФН11-52Б ____________ (Подпись, дата) **Очкин Н.В.** (И.О. Фамилия)

Преподаватель ____________ (Подпись, дата) **Облакова Т.В.** (И.О. Фамилия)

Москва, 2024

Содержание

1 Задание

1. Для данного n методом обратных функций смоделируйте выборку из закона распределения с заданной плотностью $p(x)$.

2. Для полученной выборки найдите гистограмму относительных частот. Постройте на одном рисунке графики теоретической плотности $p(x)$ и гистограмму относительных частот.

3. Вычислите выборочное среднее и выборочную дисперсию и сравните с истинными значениями этих характеристик.

4. Используя неравенство Dvoretzky-Kiefer-Wolfowitz, постройте 90% доверительный интервал для функции распределения $F(x)$.

Приведите графическую иллюстрацию

2 Исходные данные

Вариант: 9 $\quad n : 120$

$$p(x) = \frac{1}{\sqrt{0.4\pi x}} e^{-(\ln x - 2)^2/0.4}, \quad x > 0 \tag{1}$$

3 Решение

3.1 Часть 1

Для данного n методом обратных функций смоделируйте выборку из закона распределения с заданной плотностью $p(x)$.

3.1.1 Функция распределения

Найдем функцию распределения:

$$F_X(x) = \int_{-\infty}^{x} f_X(t)dt, \quad \text{где} \tag{2}$$

$f_X(x)$ - плотность распределения.

Подставим (1) в (2):

$$F_X(x) = \int_0^x \frac{1}{\sqrt{0.4\pi} y} e^{-(\ln y - 2)^2/0.4} dy =$$

$$= \left[\begin{array}{ll} t = \dfrac{\ln(y) - 2}{\sqrt{0.4}} & dt = \dfrac{1}{y\sqrt{0.4}} dy \\ \ln(y) - 2 = t\sqrt{0.4} & dy = y\sqrt{0.4} dt \\ \ln(y) = t\sqrt{0.4} + 2 & x : t = \dfrac{\ln(x) - 2}{\sqrt{0.4}} \\ y = \exp\left[t\sqrt{0.4} + 2\right] & 0 : t = -\infty \end{array} \right] =$$

$$= \frac{1}{\sqrt{0.4\pi}} \int_{-\infty}^{\frac{\ln(x)-2}{\sqrt{0.4}}} e^{\left[-t\sqrt{0.4} - 2\right]} \cdot e^{-t^2} \cdot e^{\left[t\sqrt{0.4} + 2\right]} \cdot \sqrt{0.4} dt =$$

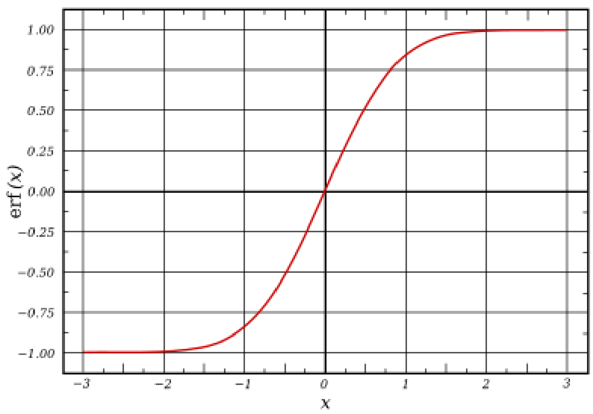
$$= \frac{1}{\sqrt{\pi}} \int_{-\infty}^{\frac{\ln(x)-2}{\sqrt{0.4}}} e^{-t^2} dt = \frac{1}{\sqrt{\pi}} \left(\int_{-\infty}^{0} e^{-t^2} dt + \int_{0}^{\frac{\ln(x)-2}{\sqrt{0.4}}} e^{-t^2} dt \right) =$$

$$= \frac{1}{\sqrt{\pi}} \left(\frac{\pi}{2} \mathrm{erf}(t) \Big|_{-\infty}^{0} + \frac{\sqrt{\pi}}{2} \cdot \mathrm{erf}\left(\frac{\ln(x) - 2}{\sqrt{0.4}} \right) \right) ⊜$$

— —

где erf(x) - **функция ошибок** (также называемая функция ошибок Гаусса).

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$



Примечание: из графика видно, что $\mathrm{erf}(0) = 0$, $\mathrm{erf}(-\infty) =$ -1

— —

$$= \frac{1}{\pi}\left(\frac{\pi}{2}\left(0-(-1)\right)+\frac{\pi}{2}\cdot\mathrm{erf}\left(\frac{\ln(x)-2}{\sqrt{0.4}}\right)\right) =$$

$$= \frac{1}{\pi}\left(\frac{\pi}{2}+\frac{\pi}{2}\cdot\mathrm{erf}\left(\frac{\ln(x)-2}{\sqrt{0.4}}\right)\right) =$$

$$= \frac{1}{2}+\frac{1}{2}\mathrm{erf}\left(\frac{\ln(x)-2}{\sqrt{0.4}}\right)$$

В конечном итоге, функция распределения имеет вид

$$F_X(x) = \frac{1}{2}+\frac{1}{2}\mathrm{erf}\left(\frac{\ln(x)-2}{\sqrt{0.4}}\right) \tag{3}$$

3.1.2 Обратная функция

Так как для нахождения обратной функции распределения требуется найти обратную функцию ошибок, что аналитически сделать сложно, воспользуемся численными методами.

3.1.2.1 Метод Ньютона

Для нахождения обратной функции воспользуемся методом касательных (Ньютона).
Рабочая формула

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Вообще говоря, метод используется для нахождения корня заданной функции. Так что для нахождения обратной функции $y = f(x)$, т.е. $x = f^{-1}(y)$ будем искать решение уравнения: $f(x) - y = 0$

$$x_{n+1} = x_n - \frac{f(x_n)-y}{(f(x_n)-y)'_x} = x_n - \frac{f(x_n)-y}{f'(x_n)} \tag{4}$$

Погрешность ε возьмем равной 1e-6.

3.1.2.2 Метод центральных разностей

Производные будем искать методом центральных разностей.
Рабочая формула

$$f'(x) \approx \frac{f(x+h)-f(x-h)}{2h} \tag{5}$$

Погрешность определяется как $O(h)$, h примем равной 1e-6.

Подставив (5) в (4), получим:

$$x_{n+1} = x_n - \frac{(f(x_n) - y) \cdot 2h}{f(x_n + h) - f(x_n - h)} \tag{6}$$

3.1.3 Реализация численного нахождения обратной функции

3.1.3.1 Реализация метода центральных разностей

Реализуем на языке программирования python метод центральных разностей (5):

Листинг 1: Реализация метода центральных разностей

```
class CDM:
  def __init__(self, h):
      self.h = h

  def diff(self, f, x):
      numerator = f(x + self.h) - f(x - self.h)
      denominator = 2 * self.h

      return numerator / denominator
```

3.1.3.2 Реализация метода Ньютона

Теперь реализуем метод Ньютона (4), используя метод центральных разностей (листинг 1):

Листинг 2: Реализация метода Ньютона

```
class Newton:
  def __init__(self, f, CDM_object, tol=1e-6, max_iter=1000):
      self.f = f
      self.CDM = CDM_object
      self.tol = tol
      self.max_iter = max_iter

  def solve(self, y, x0):
      x = x0
      for _ in range(self.max_iter):
          f_x = self.f(x) - y
          f_prime_x = self.CDM.diff(self.f, x)
          if abs(f_prime_x) < 1e-10:
              raise ValueError("Derivative is zero, method fails.")
          x_new = x - f_x / f_prime_x
          if abs(x_new - x) < self.tol:
              return x_new
          x = x_new
```

```
        raise ValueError(f"Method did not converge.({x_new})")
```

3.1.3.3 Реализация нахождения обратной функции

В конечном итоге получим:

Листинг 3: Реализация нахождения обратной функции

```
if __name__ == '__main__':
  def cdf(x): # F_X
    return float(1/2 + 1/2 * \
        scipy.special.erf((np.log(x) - 2)/(np.sqrt(0.4))))

  cdm    = CDM(h=1e-6)
  newton = Newton(cdf, cdm, tol=1e-6, max_iter=1000)

  def inverse(y, x0): # x = f^-1(y)
      return newton.solve(y, x0)
```

где
функция cdf - программная запись, найденной ранее функции распределения (3);
функция inverse - функция, возвращающее значение обратной функции к (3) в точке.

Примечание: Библиотеки scipy и numpy используются только для доступа к функции ошибок, натуральному логарифму и квадратному корню.

3.1.4 Генерация псевдослучайных чисел

3.1.4.1 Линейный конгруэнтный метод

Для генерации случайных величин воспользуемся одним из методов генерации псевдослучайных чисел - **Линейным конгруэнтным методом**.
Суть метода заключается в вычислении последовательности случайных чисел X_n, полагая

$$X_{n+1} = (aX_n + c) \bmod m, \quad \text{где} \tag{7}$$

m - модуль ($m \geq 2$);
a - множитель ($0 \leq a < m$);
c - приращение ($0 \leq c < m$);
X_0 - начальное значение ($0 \leq X_0 < m$).

За значениями параметров обратимся к [1].

$$m = 2^{(60)} - 93 \qquad a = 561860773102413563 \qquad c = 0. \tag{8}$$

В случае когда $c = 0$, метод называют **мультипликативным конгруэнтным методом**.

3.1.4.2 Реализация ЛКМ

Реализуем линейный конгруэнтный метод (7), используя параметры (8):

Листинг 4: Реализация ЛКМ

```
class LCG:
  def __init__(self,
               seed, a=561860773102413563, c=0, m=2**60-93):
      self.seed = seed
      self.a = a
      self.c = c
      self.m = m
      self.state = seed

  def next(self):
      self.state = (self.a * self.state + self.c) % self.m
      return self.state / self.m # Normalize to [0, 1)
```

3.1.4.3 Моделирование выборки

Наконец смоделируем 120 случайных величин в виде вектора линейным конгруэнтным методом:

```
n = 120
lcg = LCG(seed=340751464)

data = [lcg.next() for _ in range(n)]
print(data)
```

Начальное значение (seed) в ЛКМ выбирается так, чтобы $x_0 \neq 0$. Это необходимо для того, чтобы последовательность была полной длины, т.е. имела максимальную периодичность при генерации чисел. Обычно используют случайное или произвольно выбранное значение из множества $\{1, ..., m - 1\}$ [1].

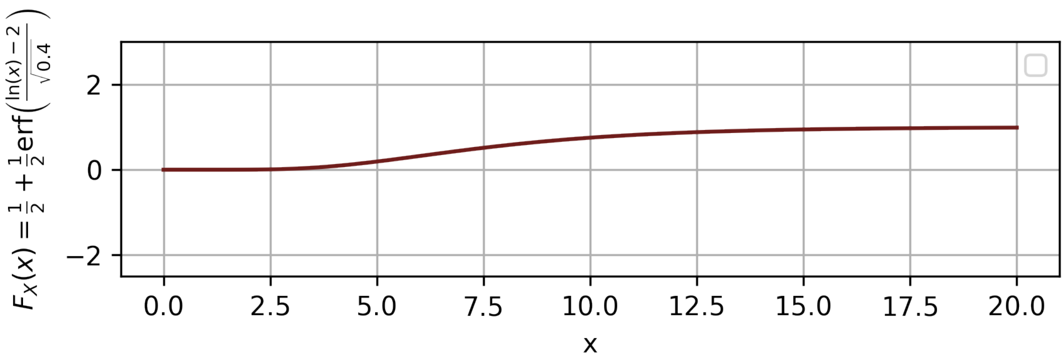
$$Y = [$$

$0.32949885091783276,\quad 0.9732846125910063,\quad 0.39434856188646605,\quad 0.8210789016402354,$
$0.20093003622010405,\quad 0.9707650441880256,\quad 0.4178790819080603,\quad 0.2974690498690837,$
$0.32632062605066997,\quad 0.8137561621450644,\quad 0.6418089688930682,\quad 0.72226998934102,$
$0.12543257092465954,\quad 0.39665152743167287,\quad 0.7205668938187388,\quad 0.18456086494051507,$

$$\dots$$

$$]$$

Теперь пересчитаем полученный вектор случайных величин, в соответствии с функцией inverse из листинга 3.
Однако сперва подеберем вектор начальных приближений, так как того требует метод Ньютона.



Из графика видно, что функция (3) приблизительно принимает значения $0 < x < 20$ при $0 < y < 1$. Исходя из этого подберем вектор начальных приближений:
[0, 3, 6, 9, 12, 15, 18, 21].

Итого имеем:

```
guesses = [0, 3, 6, 9, 12, 15, 18, 21]
for ind, el in enumerate(data):
    for attempt, guess in enumerate(guesses):
        try:
            inv_value = inverse(el, guess)
            data[ind] = inv_value
            break
        except:
            pass

        if attempt == len(guesses) - 1:
            raise Exception('Solution was not found')
```

$$X = [$$

$$\begin{array}{llll}
6.065674809818662, & 17.52728100897831, & 6.5544583429545265, & 11.147396579310449, \\
5.078922433676263, & 17.222193164730466, & 6.734763210632847, & 5.825351333431677, \\
6.041854304433931, & 11.010347184551701, & 8.692598700648851, & 9.618384853081634, \\
4.421534190647852, & 6.572007701239677, & 9.596593105982482, & 4.944860000874664,
\end{array}$$

$$\ldots$$

$$]$$

3.2 Часть 2

Для полученной выборки найдите гистограмму относительных частот. Постройте на одном рисунке графики теоретической плотности $p(x)$ и гистограмму относительных частот.

3.2.1 Первоначальная обработка полученных статистических данных

3.2.1.1 Крайние члены вариационного ряда и размах выборки

Найдем крайние члены вариационного ряда как минимальное и максимальное значения набора данных, а также размах выборки, как их разницу:

```
mini, maxi = min(data), max(data)
print(mini, maxi)

range_ = maxi - mini
print(range_)
```

Крайние члены: $2.1028, \ 23.4245$

Размах выборки: 21.3217

Примечание: Выводимые данные округлены до 4х знаков для удобства чтения.

3.2.1.2 Группировка данных

Для начала определим количество интервалов, воспользовавшись правилом Стерджеса:

$$k = 1 + \lfloor \log_2 n \rfloor,$$

где n — общее число наблюдений величины,
$\log_2$ — логарифм по основанию 2,
$\lfloor x \rfloor$ — обозначает целую часть числа x.

И определим шаг интервала разделив размах выборки на количество интервалов:

```
trunc = lambda x : int(str(x)[:str(x).index('.')])
k = 1 + trunc(np.log2(n))
h = range_ / k
```

Количество интервалов: 7

Шаг интервала: 3.046

Теперь сгруппируем данные:

```
grouped_data = []

begin = mini
for i in range(k):
    end = begin + h

    middle = (begin + end) / 2
    freq = sum(begin <= el < end for el in data)

    if i == k - 1:
        freq += 1

    relative_freq = freq / n

    grouped_element = {
        'interval numero': i,
        'interval': f'[{begin}, {end})',
        'middle': middle,
        'frequency': freq,
        'relative frequency': relative_freq
    }
    grouped_data.append(grouped_element)

    begin = end
```

Полученную группировку представим в виде таблицы:

номер интервала	интервал	середина интервала	частота	относительная частота
0	[2.1028, 5.1488)	3.6258	30	0.25
1	[5.1488, 8.1947)	6.6718	46	0.3833
2	[8.1947, 11.2407)	9.7177	24	0.2
3	[11.2407, 14.2867)	12.7637	11	0.09167
4	[14.2867, 17.3326)	15.8096	6	0.05
5	[17.3326, 20.3786)	18.8556	2	0.0167
6	[20.3786, 23.4245)	21.9016	1	0.00833

Таблица 1: Сгруппированные данные

3.2.1.3 Гистограмма относительных частот

Построим на одном рисунке графики теоретической плотности (1) и гистограмму относительных частот.
По оси абсцисс для гистограммы укажем середины интервалов,
по оси ординат - вектор относительных частот, разделенный на шаг интервала:

```
x_axis = [el['middle']                    for el in grouped_data]
y_axis = [el['relative frequency'] / h for el in grouped_data]
```

$$\text{int} : [3.6258 \quad 6.6718 \quad 9.7177 \quad 12.7637 \quad 15.8096 \quad 18.8556 \quad 21.9016]$$

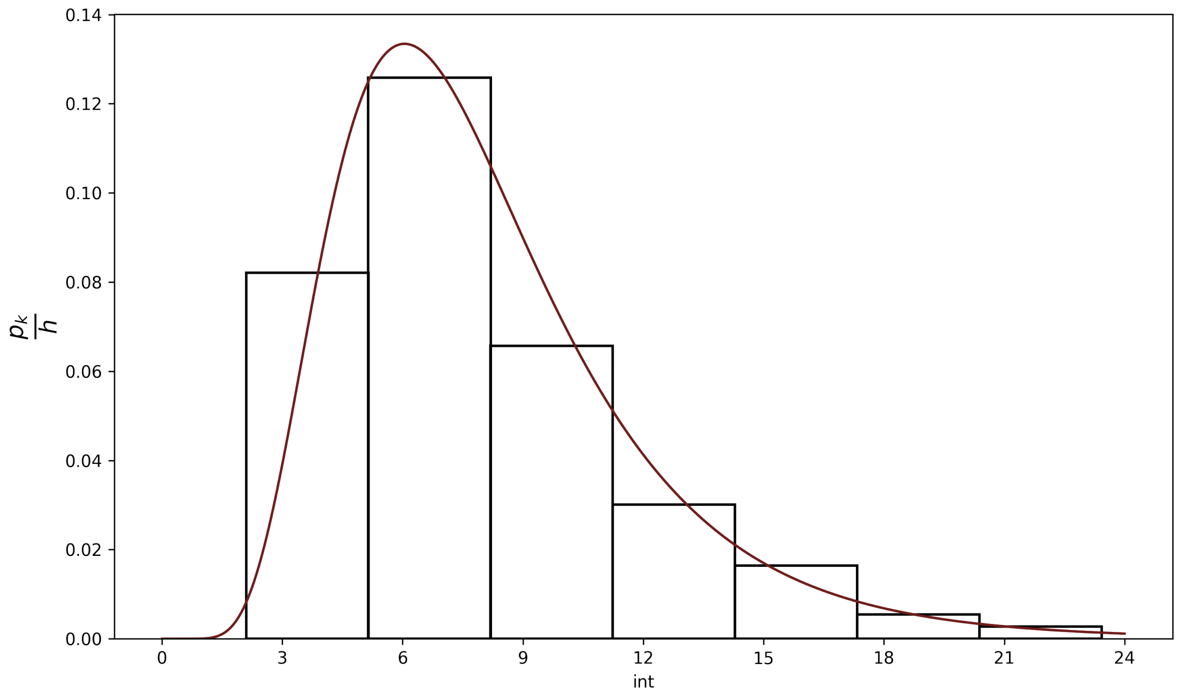
$$\frac{\mathrm{p}_k}{\mathrm{h}} : [0.0821 \quad 0.1258 \quad 0.0657 \quad 0.0301 \quad 0.0164 \quad 0.0055 \quad 0.0027]$$

Для построения графиков воспользуемся библиотекой MATPLOTLIB.

```
import matplotlib.pyplot as plt

def pdf(x):
    return 1 / (np.sqrt(0.4 * np.pi) * x) \
        * np.exp(-(np.log(x) - 2)**2 / 0.4)

def buildBar(x, y):
```

```
    # histogtamm
    plt.bar(x, y, color='white', edgecolor='black')

    # pdf
    x_values = np.linspace(0.01, trunc(maxi), 1000)
    y_values = pdf(x_values)
    plt.plot(x_values, y_values, color='red', linestyle='-',
                                             linewidth=1.5)

    plt.show()

buildBar(x_axis, y_axis)
```

Примечание: код был несколько упрощен, чтобы не загромождать текст, полный код см. в приложении.



3.3 Часть 3

Вычислите выборочное среднее и выборочную дисперсию и сравните с истинными значениями этих характеристик.

3.3.1 Эмпирические и теоретические характеристики

3.3.1.1 Математическое ожидание

Запишем формулу для математического ожидания:

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x f_X(x) dx, \qquad \text{где} \tag{9}$$

$f_X(x)$ - плотность распределения.

3.3.1.2 Метод интегрирования Монте-Карло

Для вычисления интеграла воспользуемся численным методом интегрирования Монте-Карло

$$\int_a^b f(x)dx \approx \frac{b-a}{N} \sum_{i=1}^{N} f(u_i), \quad \text{где} \tag{10}$$

u - равномерно распредленная на отрезке интегрирования $[a, b]$ случайная величина.

Геометрическая интерпретация данного метода похожа на известный детерминистический метод, с той разницей, что вместо равномерного разделения области интегрирования на маленькие интервалы и суммирования площадей получившихся «столбиков» мы забрасываем область интегрирования случайными точками, на каждой из которых строим такой же «столбик», определяя его ширину как $\frac{b-a}{N}$, и суммируем их площади.

Точность оценки данного метода зависит только от количества точек N.

3.3.1.3 Реализация метода Монте-Карло

Так как данный метод опирается на генерацию случайных чисел на промежутке, расширим функционал нашей реализации ЛКМ (листинг 4) и добавим следующий метод:

```
def next_in_range(self, a, b):
  return a + (b - a) * self.next()
```

Теперь реализуем интегрирование методом Монте-Карло, используя описаннный ЛКМ:

Листинг 5: Реализация метода Монте-Карло

```
class MonteCarlo:
  def __init__(self, N, PRNG_object):
      self.N = int(N)
      self.PRNG = PRNG_object

  def integrate(self, f, a, b):
      mult = (b - a) / self.N

      generatedValues = []
      for _ in range(self.N):
          randomArg = self.PRNG.next_in_range(a, b)
          randomFuncVal = f(randomArg)

          generatedValues.append(randomFuncVal)

      return mult * sum(generatedValues)
```

3.3.1.4 Реализация численного нахождения математического ожидания

Прежде чем реализовывать вычисление самого интеграла, заметим, что в пределах интегрирования (9) присутствует бесконечность, что затрудняет интегрирование методом Монте-Карло (10).
Воспользуемся заменой, чтобы свести бесконечные пределы в конечные:

$$\mathbb{E}[X] = \int_{-\infty}^{+\infty} x f_X(t)dt =$$

$$= \begin{bmatrix} x = \tan(t) \\ t = \arctan(x) \\ dx = \dfrac{1}{\cos^2(t)}dt \\ -\infty : t = \arctan(-\infty) = -\dfrac{\pi}{2} \\ +\infty : t = \arctan(+\infty) = \dfrac{\pi}{2} \end{bmatrix} =$$

$$= \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} \tan(t) \cdot f_X(\tan(t)) \cdot \frac{1}{\cos^2(t)}dt$$

Итого получим:

$$\mathbb{E}[X] = \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} g(t)dt, \qquad g(t) = \tan(t) \cdot f_X(\tan(t)) \cdot \frac{1}{\cos^2(t)} \tag{11}$$

Объединим теперь (11) и (10) и получим:

$$\mathbb{E}[X] = \int_{-\infty}^{+\infty} x f_X(x)dx = \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} g(t)dt \approx \frac{\pi}{N}\sum_{i=1}^{N} g(u_i), \quad \text{где} \tag{12}$$

$g(x) = \tan(x) \cdot f_X(\tan(x)) \cdot \dfrac{1}{\cos^2(x)}$,

u_i ищем в соответствии с (листинг 4).

Подставляя (1) в (12) и (8) в (7):

$$\mathbb{E}[X] = \int_{0}^{+\infty} x f_X(x)dx = \int_{0}^{\frac{\pi}{2}} \tan(t) f_X(\tan(t)) \frac{1}{\cos^2(t)} dt \approx$$

$$\approx \frac{\pi/2}{N}\sum_{i=1}^{N}\left[\tan(u_i) \cdot \frac{1}{\sqrt{0.4\pi}\tan(u_i)} e^{-(\ln(\tan(u_i))-2)^2/0.4} \cdot \frac{1}{\cos^2(u_i)}\right], \quad \text{где}$$

$u_i = (5618607731024135 63 \cdot u_{i-1}) \bmod 2^{60} - 93$

При программной реализации, как уже было сказано ранее, N отвечает за точность полученной оценки метода, так что чем оно больше, тем лучше.

```
monteCarlo = MonteCarlo(1e7, lcg)

def subs(t):
    return np.tan(t) * pdf(np.tan(t)) * (1 / np.cos(t)**2)

ExpectedValue = monteCarlo.integrate(subs, 0, np.pi/2)
```

где классы **LCG** и **MonteCarlo** представлены в листингах 4 и 5 соответственно.

Итого получаем:

$$\mathbb{E}[X] \approx 8.16$$

3.3.1.5 Дисперсия

Аналогично найдем дисперсию, как

$$\mathrm{D}[X] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$$

```
def subs2(t):
return np.tan(t)**2 * pdf(np.tan(t)) * (1 / np.cos(t)**2)

Var = monteCarlo.integrate(subs2, 0, np.pi/2) - \
  monteCarlo.integrate(subs, 0, np.pi/2)**2
```

Итого получаем:

$$\mathrm{D}[X] \approx 14.65$$

3.3.1.6 Выборочное среднее

$$\overline{X} = \frac{1}{n}\sum_{k=1}^{n} X_k$$

```
OverlineX = sum(data)/n
```

Итого получаем:

$$\overline{X} \approx 7.88$$

3.3.1.7 Выборочная дисперсия

$$S^2 = \frac{1}{n-1}\sum_{k=1}^{n} \left(X_k - \overline{X}\right)^2$$

```
S2 = 1 / (n - 1) * sum([(x - OverlineX)**2 for x in data])
```

Итого получаем:

$$S^2 \approx 15.36$$

3.3.1.8 Сравнение

$$\left|\mathbb{E}[X] - \overline{X}\right| \qquad \sqrt{\frac{\mathrm{D}[X]}{S^2}}$$

```
diff1 = abs(ExpectedValue - OverlineX)
diff2 = np.sqrt(Var/S2)
```

Итого имеем:

$$\mathbb{E}[X] = 8.16 \qquad \overline{X} = 15.36 \qquad \left|\mathbb{E}[X] - \overline{X}\right| = 0.2792$$

$$\mathrm{D}[X] = 14.65 \qquad S^2 = 15.36 \qquad \sqrt{\frac{\mathrm{D}[X]}{S^2}} = 0.9768$$

Поскольку абсолютная величина разности математического ожидания и выборочного среднего мала, а отношение выборочной дисперсии к ее теоретическому значению близко к единице, то результаты моделирования можно признать удовлетворительными.

3.4 Часть 4

Используя неравенство Dvoretzky-Kiefer-Wolfowitz, постройте 90% доверительный интервал для функции распределения $F(x)$.

3.4.0.1 Неравенство Dvoretzky-Kiefer-Wolfowitz

$$P\left(\sup_{x\in\mathbb{R}}\left|\hat{F}_n(x) - F(x)\right| > \varepsilon\right) \le 2e^{-2n\varepsilon^2}$$

Таким образом, если $2e^{-2n\varepsilon^2} = \alpha$, $\ln\left(\frac{2}{\alpha}\right) = 2n\varepsilon^2$, $\varepsilon = \sqrt{\frac{1}{2n}\ln\left(\frac{2}{\alpha}\right)}$, то с вероятностью $1-\alpha$

$$L(x) \le \hat{F}_n(x) \le R(x),$$

где

$$L(x) = \max\left\{\hat{F}_n(x) - \sqrt{\frac{1}{2n}\ln\left(\frac{2}{\alpha}\right)}, 0\right\} \qquad R(x) = \min\left\{\hat{F}_n(x) + \sqrt{\frac{1}{2n}\ln\left(\frac{2}{\alpha}\right)}, 1\right\}$$

3.4.0.2 Реализация функций

Так как требуется построить 90% доверительный интервал, α возьмем равной 0.1.

```
def Fempir(x):
  ind = lambda x : 1 if x > 0 else 0

  return sum([ind(x - X)/n for X in data])

alpha = 0.1
epsilon = np.sqrt(1/(2*n) * np.log(2/alpha))
```

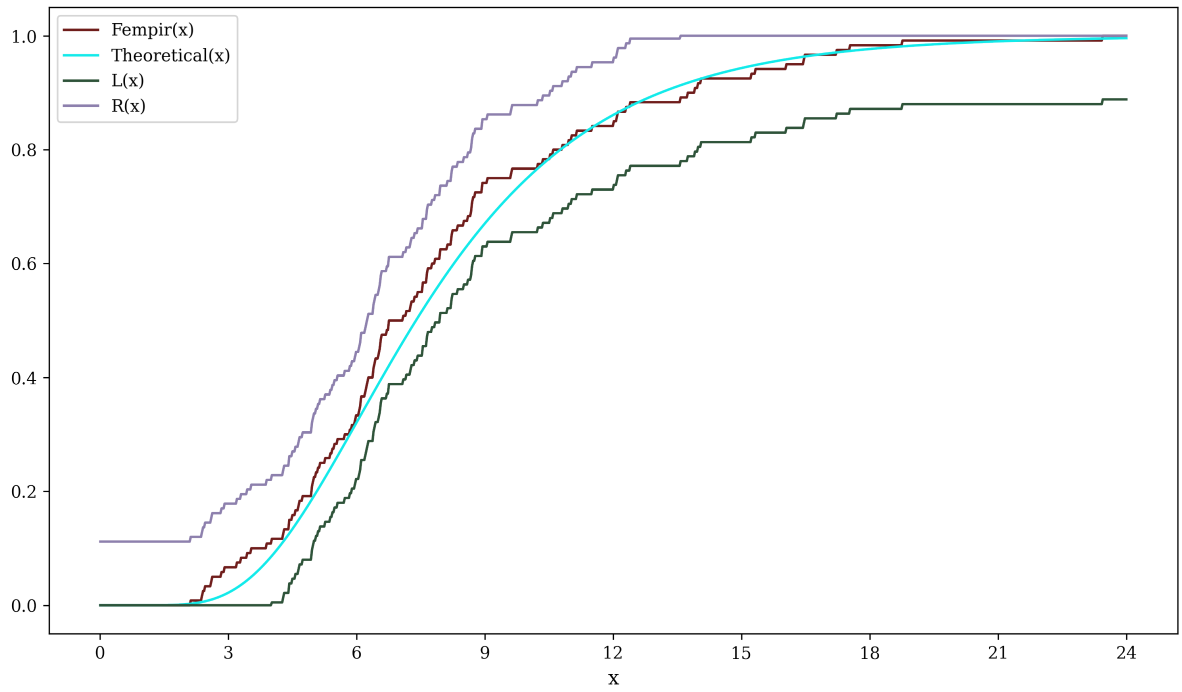
```
def L(x):
  return max(Fempir(x) - epsilon, 0)

def R(x):
  return min(Fempir(x) + epsilon, 1)
```

3.4.0.3 Графическая иллюстрация

Построим доверительный интервал уровня 0.9 для функции распределения на основе неравенства Дворецкого - Кифера - Волфовица.

```
def buildPlots():
  x_values = np.linspace(0.01, trunc(maxi) + 1, 1000)

  # empir
  empir_y_values = [Fempir(x) for x in x_values]
  plt.plot(x_values, empir_y_values)

  # theoretical
  cdf_y_values = [cdf(x) for x in x_values]
  plt.plot(x_values, cdf_y_values)

  # L
  L_y_values = [L(x) for x in x_values]
  plt.plot(x_values, L_y_values)

  # R
  R_y_values = [R(x) for x in x_values]
  plt.plot(x_values, R_y_values)

  # Show the plot
  plt.show()
```

Примечание: код был несколько упрощен, чтобы не загромождать текст, полный код см. в приложении.



4 Вывод

В ходе проделанной лабораторной работы было проведено моделирование выборки из логнормального распределения методом обратных функций, реализованы такие численные методы, как метод Ньютона, метод центральных разностей и метод Монте-Карло. Был реализован алгоритм генерации псевдослучайных чисел. На основе значений выборочного среднего и выборочной дисперсии был сделан вывод о степени качества моделирования. Также был построен доверительный интервал на основе неравенства Дворецкого - Кифера - Вольфовица.

5 Приложение

Программный код, с помощью которого была выполнена данная лабораторная работа.

```
import warnings
warnings.filterwarnings("ignore", category=RuntimeWarning)


class CDM:
    def __init__(self, h):
        self.h = h

    def diff(self, f, x):
        numerator = f(x + self.h) - f(x - self.h)
        denominator = 2 * self.h

        return numerator / denominator


class Newton:
    def __init__(self, f, CDM_object, tol=1e-6, max_iter=1000):
        self.f = f
        self.CDM = CDM_object
        self.tol = tol
        self.max_iter = max_iter

    def solve(self, y, x0):
        x = x0
        for _ in range(self.max_iter):
            f_x = self.f(x) - y
            f_prime_x = self.CDM.diff(self.f, x)
            if abs(f_prime_x) < 1e-10:
                raise ValueError("Derivative is zero, method fails.")
            x_new = x - f_x / f_prime_x
            if abs(x_new - x) < self.tol:
                return x_new
            x = x_new

        raise ValueError(f"Method did not converge.({x_new})")


class LCG:
    def __init__(self, seed, a=561860773102413563, c=0, m=2**60-93):
        self.seed = seed
        self.a = a
        self.c = c
        self.m = m
        self.state = seed

    def next(self):
        self.state = (self.a * self.state + self.c) % self.m
        return self.state / self.m  # Normalize to [0, 1)
```

```python
    def next_in_range(self, a, b):
        return a + (b - a) * self.next()


class MonteCarlo:
    def __init__(self, N, PRNG_object):
        self.N = int(N)
        self.PRNG = PRNG_object

    def integrate(self, f, a, b):
        mult = (b - a) / self.N

        generatedValues = []
        for _ in range(self.N):
            randomArg = self.PRNG.next_in_range(a, b)
            randomFuncVal = f(randomArg)

            generatedValues.append(randomFuncVal)

        return mult * sum(generatedValues)


import scipy.special
import numpy as np


if __name__ == '__main__':

    # --------------------------PART1--------------------------

    def cdf(x): # F_X
        return float(1/2 + 1/2 * \
            scipy.special.erf((np.log(x) - 2)/(np.sqrt(0.4))))

    cdm    = CDM(h=1e-6)
    newton = Newton(cdf, cdm, tol=1e-6, max_iter=1000)

    def inverse(y, x0): # x = f^-1(y)
        return newton.solve(y, x0)

    n = 120

    lcg = LCG(seed=340751464)

    # --------------------------PART2--------------------------

    data = [lcg.next() for _ in range(n)]
    # print(f'Y: {data}')

    guesses = [0, 3, 6, 9, 12, 15, 18, 21]
    for ind, el in enumerate(data):
        for attempt, guess in enumerate(guesses):
            try:
                inv_value = inverse(el, guess)
```

```
                data[ind] = inv_value
                break
        except:
            pass

        if attempt == len(guesses) - 1:
            raise Exception('Solution was not found')

# print(f'X: {data}')

# --------------------------PART3--------------------------

mini, maxi = min(data), max(data)
# print(f'min: {mini}, max: {maxi}')

range_ = maxi - mini
# print(f'range: {range_}')

# --------------------------PART4--------------------------

trunc = lambda x : int(str(x)[:str(x).index('.')])

k = 1 + trunc(np.log2(n))
# print(f'k: {k}')

h = range_ / k
# print(f'h: {h}')

# --------------------------PART5--------------------------

grouped_data = []

begin = mini
for i in range(k):
    end = begin + h

    middle = (begin + end) / 2
    freq = sum(begin <= el < end for el in data)

    if i == k - 1:
        freq += 1

    relative_freq = freq / n

    grouped_element = {
        'interval numero': i,
        'interval': f'[{np.round(begin, 4)}, {np.round(end, 4)})',
        'middle': np.round(middle, 4),
        'frequency': freq,
        'relative frequency': relative_freq
    }
    grouped_data.append(grouped_element)

    begin = end
```

```
# for element in grouped_data:
#     print(element['interval numero'],
#           element['interval'],
#           element['middle'],
#           element['frequency'],
#           element['relative frequency'])

# -------------------------PART6-------------------------

import matplotlib.pyplot as plt

def pdf(x):
    return 1 / (np.sqrt(0.4 * np.pi) * x) \
        * np.exp(-(np.log(x) - 2)**2 / 0.4)

def buildBar(x, y):
    # Define colors
    RED   = '#6F1D1B'

    # Define font sizes
    SIZE_TICKS   = 10

    # Create the figure and axis
    _, ax = plt.subplots(figsize=(10, 6))

    # histogtamm
    ax.bar(x, y, width=3.05, color='none',
                             edgecolor='black',
                             linewidth=1.5)

    # pdf
    x_values = np.linspace(0.01, trunc(maxi) + 1, 1000)
    y_values = pdf(x_values)
    ax.plot(x_values, y_values, color=RED,
                                linestyle='-',
                                linewidth=1.5)

    # axis names
    ax.set_xlabel('int')
    ax.set_ylabel('$\\frac{p_k}{h}$', fontsize=20)

    # ticks settings
    xticks = [i for i in range(0, trunc(maxi) + 2, 3)]
    ax.set_xticks(xticks)

    # Adjust the font size of the tick labels
    ax.tick_params(axis='both', which='major',
                                labelsize=SIZE_TICKS)

    # Update font settings
    plt.rcParams.update({'font.family': 'serif',
                         'font.size': 12})

    # Adjust layout
    plt.tight_layout()
```

```
    # Save the figure
    plt.savefig('histXpdf.png', dpi=300, transparent=True)

    # Show the plot
    plt.show()

x_axis = [el['middle']                      for el in grouped_data]
y_axis = [el['relative frequency'] / h for el in grouped_data]

# print(f'x: {np.round(x_axis, 4)}')
# print(f'y: {np.round(y_axis, 4)}')

# buildBar(x_axis, y_axis)

# --------------------------PART7--------------------------

monteCarlo = MonteCarlo(1e7, lcg)

def subs(t):
    return np.tan(t) * pdf(np.tan(t)) * (1 / np.cos(t)**2)

ExpectedValue = monteCarlo.integrate(subs, 0, np.pi/2)

# print(ExpectedValue)

# --------------------------PART8--------------------------

def subs2(t):
    return np.tan(t)**2 * pdf(np.tan(t)) * (1 / np.cos(t)**2)

Var = monteCarlo.integrate(subs2, 0, np.pi/2) - \
      monteCarlo.integrate(subs, 0, np.pi/2)**2

# print(Var)

# --------------------------PART9--------------------------

OverlineX = sum(data)/n

# print(f'OverlineX: {OverlineX}')

S2 = 1 / (n - 1) * sum([(x - OverlineX)**2 for x in data])

# print(f'S2: {S2}')

# --------------------------PART10--------------------------

diff1 = abs(ExpectedValue - OverlineX)
diff2 = np.sqrt(Var/S2)

# print(diff1)
# print(diff2)

# --------------------------PART11--------------------------
```

```
def Fempir(x):
    ind = lambda x : 1 if x > 0 else 0

    return sum([ind(x - X)/n for X in data])

alpha = 0.1
epsilon = np.sqrt(1/(2*n) * np.log(2/alpha))

def L(x):
    return max(Fempir(x) - epsilon, 0)

def R(x):
    return min(Fempir(x) + epsilon, 1)

# --------------------------PART12--------------------------

def buildPlots():
    # Define colors
    RED    = '#6F1D1B'
    BLUE   = '#12EAEA'
    GREEN  = '#2E5339'
    PURPLE = '#8D80AD'

    # Define font sizes
    SIZE_TICKS    = 10

    # Create the figure and axis
    _, ax = plt.subplots(figsize=(10, 6))
# Adjust the figure size as needed

    x_values = np.linspace(0.01, trunc(maxi) + 1, 1000)

    # empir
    empir_y_values = [Fempir(x) for x in x_values]
    ax.plot(x_values, empir_y_values, color=RED,
                                  linestyle='-',
                                  linewidth=1.5,
                                  label='Fempir(x)')

    # theoretical
    cdf_y_values = [cdf(x) for x in x_values]
    ax.plot(x_values, cdf_y_values, color=BLUE,
                                  linestyle='-',
                                  linewidth=1.5,
                                  label='Theoretical(x)')

    # L
    L_y_values = [L(x) for x in x_values]
    ax.plot(x_values, L_y_values, color=GREEN,
                                  linestyle='-',
                                  linewidth=1.5,
                                  label='L(x)')

    # R
```

```
    R_y_values = [R(x) for x in x_values]
    ax.plot(x_values, R_y_values, color=PURPLE,
                                  linestyle='-',
                                  linewidth=1.5,
                                  label='R(x)')

    # axis names
    ax.set_xlabel('x')

    # ticks settings
    xticks = [i for i in range(0, trunc(maxi) + 2, 3)]
    ax.set_xticks(xticks)

    # Adjust the font size of the tick labels
    ax.tick_params(axis='both', which='major',
                            labelsize=SIZE_TICKS)

    plt.legend(fontsize=10, loc='best')

    # Update font settings
    plt.rcParams.update({'font.family': 'serif',
                         'font.size': 12})

    # Adjust layout
    plt.tight_layout()

    # Save the figure
    plt.savefig('Dvoretzky-Kiefer-Wolfowitz.png',
                dpi=300, transparent=True)

    # Show the plot
    plt.show()

buildPlots()
```

6 Список использованных источников

1. L'Ecuyer, Pierre (January 1999). "Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure С. 256