Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное
учреждение высшего образования

“Новосибирский государственный технический университет”

Кафедра теоретической и прикладной информатики

Лабораторная работа №5
по дисциплине “Компьютерное моделирование”

Факультет:	ПМИ
Группа:	ПМи-51
Вариант	1
Студенты:	Фатыхов Т.М. Неупокоев М.В. Хахолин А.А.
Преподаватель:	Волкова В.М.

Новосибирск
2018

Цель работы

Научиться моделировать значения непрерывно распределённой случайной величины методом исключений и проводить статистический анализ сгенерированных данных.

Исходные данные

Генератор равномерно распределенной псевдослучайной последовательности для генерирования значений на осях X и Y был взят встроенный: numpy.random.rand.

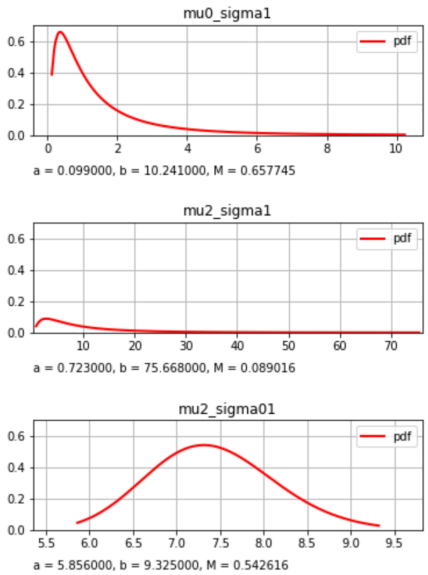
Исследуемые параметры лог-нормального распределения:
$\mu = 0, \sigma = 1$
$\mu = 2, \sigma = 1$
$\mu = 2, \sigma = 0.1$

Исследования

Параметры распределения зададим при помощи словаря, а длины исследуемых последовательностей с помощью массива следующим образом:

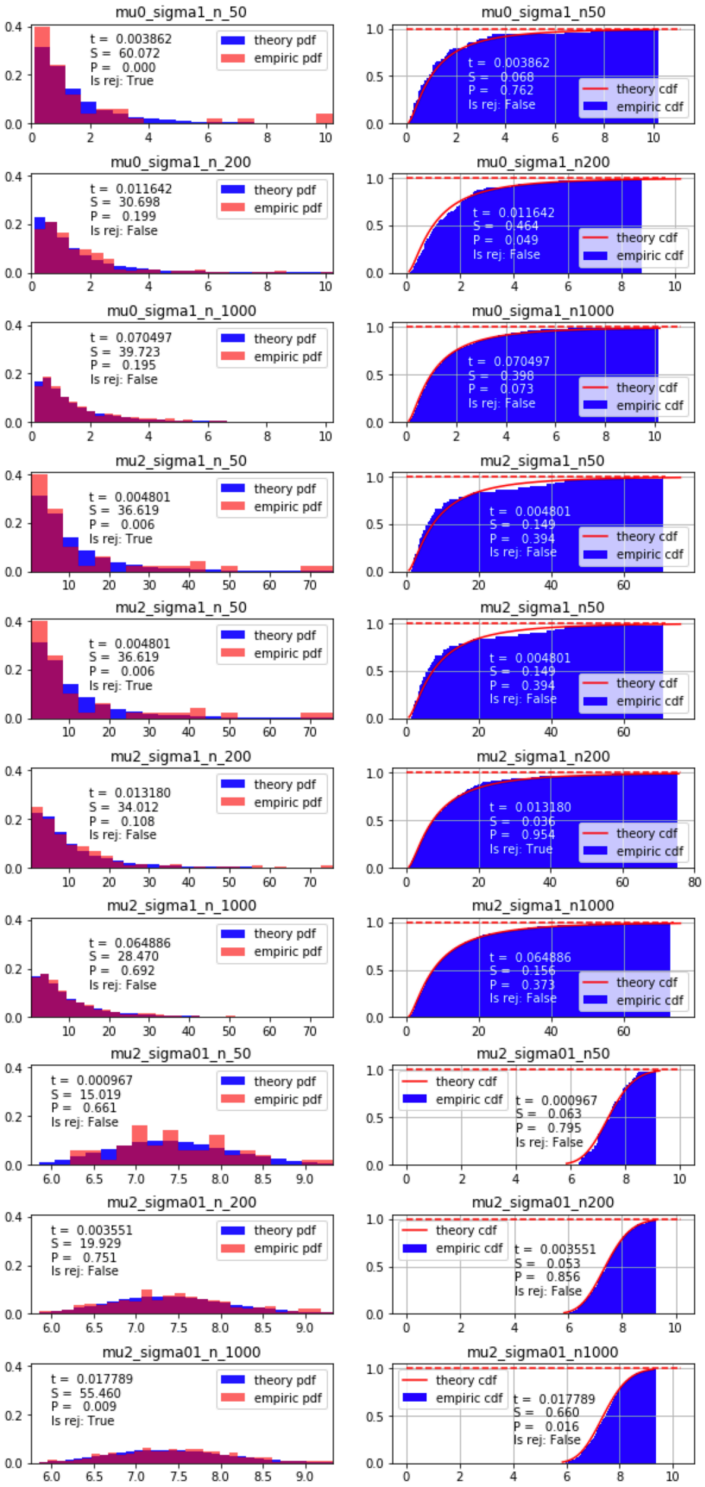
```
experiment_params = {
    'mu0_sigma1' : {'mu' : 0, 'sigma' : 1},
    'mu2_sigma1' : {'mu' : 2, 'sigma' : 1},
    'mu2_sigma01' : {'mu' : 2, 'sigma' : 0.1}
}

desirable_sizes = [50, 200, 1000]
```

Далее представлены графики плотности (pdf). Название каждого графика имеет вид:
mu{значение параметра μ}_sigma{значение параметра σ}



Далее представлены графики эмпирической и теоретической плотностей распределения и функций распределения, с результатами проверок гипотез по критерию хи-квадрат (левый столбец) и критерию Крамера-Мизеса (правый столбец).

Название каждого графика имеет вид:
mu{значение параметра μ}_sigma{значение параметра σ}_n{длина последовательности}



Вывод

Эмпирические значения плотности вероятности и функции распределения визуально приближаются к теоретическим значениям, при увеличении размера генерируемой выборки. Всего лишь несколько гипотез о том, что генерируемая выборка принадлежит лог-нормальному распределению было отклонено, что говорит о хорошем, но не отличном качестве генератора.

Текст программы

lognormal_gen.py

```
import numpy as np
from scipy.optimize import fmin
from scipy.special import erf

class Lognormal_gen():
    """
    Generate sequence from lognormal distribution
    """

    def __init__(self, mu, sigma, capacity = 0.98):
        """
        Inputs:
        - mu: Float parameter of distribution
        - sigma: Float parameter of distribution
        - capacity: Float value from interval [0, 1);
        what percent of cdf to use
        """
        self.mu = mu
        self.sigma = sigma

        if capacity is not None and capacity >= 0 and capacity < 1:
            self.capacity = capacity
        else:
            ValueError('capacity have to be from interval [0, 1)')

        self.a, self.b = self._get_bounds_(capacity)

        # find point where pdf is maximum
        self.max_x = fmin(lambda x: -self._pdf_(x), self.a, disp=False)
        self.max_y = self._pdf_(self.max_x)


    def _pdf_(self, x):
        """
        Probability density function of log-normal distribution
        Inputs:
        - x: Array, list or single value
        """
        if type(x) == list:
            x = np.array(x)

        res = None
        if type(x) is np.ndarray:
            # if x is equal to 0, then put 0 to result vector;
            # define mask of non-zero elements
            mask = x > 1e-6
            res = np.empty_like(x)

            res[~mask] = 0
            e = np.exp(-(np.log(x[mask]) - self.mu)**2/(2*self.sigma**2))
            denominator = x[mask] * self.sigma * np.sqrt(2 * np.pi)
            res[mask] = e / denominator
        else:
            if x == 0:
                res = 0
            else:
                e = np.exp(-(np.log(x) - self.mu)**2/(2*self.sigma**2))
                denominator = x * self.sigma * np.sqrt(2 * np.pi)
                res = e / denominator

        return res
```

```
def _cdf_(self, x):
    """
    Cummulative distribution function
    """
    return 1/2 * (1 + erf((np.log(x) - self.mu) / (self.sigma * np.sqrt(2)))


def _get_bounds_(self, capacity=None, dx=1e-3):
    """
    Calculate bounds according to capacity of genereretor

    Inputs:
    - capacity: Float value from [0, 1)
    - dx: Float step for numerical integral calculation

    Outputs:
    - a, b: Tupple of bounds of x
    """
    capacity = self.capacity if capacity is None else capacity
    if capacity is not None and capacity >= 0 and capacity < 1:
        self.capacity = capacity
    else:
        ValueError('capacity have to be from interval [0, 1)')

    offset = (1 - capacity) / 2

    x = 0
    cumm = 0
    while cumm < offset:
        s = self._pdf_(x) * dx
        cumm += s
        x += dx
    a = x

    while cumm < 1 - offset:
        s = self._pdf_(x) * dx
        cumm += s
        x += dx
    b = x

    return a, b


def generate(self, N):
    """
    Generate sequence from log-normal distribution

    Inputs:
    - N: Integer size of sequence

    Outputs:
    - sequence
    """
    self._N = N
    seq = []
    seq_size = 0
    iters = 0
    while seq_size < N:
        iters += 1
        x_value = self.a + np.random.rand(1)[0] * (self.b - self.a)
        y_value = np.random.rand(1) * self.max_y

        if y_value <= self._pdf_(x_value):
            seq.append(x_value)
            seq_size += 1
```

```
        self._iters = iters
        return np.array(seq)
```

no_param.py

```
import numpy as np
from scipy.stats import anderson
from scipy.special import gamma as g
import scipy.integrate as integrate
import matplotlib.pyplot as plt

def kramer_mizes(seq, F, alpha=.05):
    """
    Statistical test of whether a given sample of data is drawn from
    a given probability distribution.

    Inputs:
    - seq: Array
    - F: Cummulative distribution function (a.k.a cdf)
    - alpha: Level of significance

    Outputs:
    - is_rejected: Boolean; True if rejected
    - s: Value of statistic
    - p: Value of probability
    """
    sort_seq = sorted(seq)
    n = len(sort_seq)
    s = _getKramerStatistic_(sort_seq, n, F)
    p = _getKramerProbability_(s)
    is_rejected = s <= alpha
    return is_rejected, s, p


def _getKramerStatistic_(seq, n, F):
    """
    Calculate statistic for Kramer-Mizes criteria
    """
    S = 0.0
    for i in range(n):
        S += (F(seq[i]) - (2 * i - 1) / (2 * n)) ** 2
    return 1 / (12 * n) + S


def _getKramerProbability_(stat):
    """
    Calculate probability for Kramer-Mizes criteria
    """
    a1 = 0.0
    j = 0
    part1 = part2 = part3 = 1.0

    while j < 5 and part1 != 0.0 and part2 != 0.0 and part3 != 0.0:
        part1 = (gamma(j + 1 / 2) * sqrt(4 * j + 1)) / (gamma(1 / 2) * gamma(j +
1))
        part2 = exp(-(4 * j + 1) ** 2 / (16 * stat))
        part3 = iv(-1 / 4, (4 * j + 1) ** 2 / (16 * stat)) - iv(1 / 4, (4 * j +
1) ** 2 / (16 * stat))
        a1 += part1 * part2 * part3
        j += 1
    a1 *= 1 / sqrt(2 * stat)
    return 1 - a1
```

param.py

```
from scipy.stats import chisquare
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as integrate
from scipy.special import gamma

import os,sys,inspect

def _freqs(seq, lower_bound, upper_bound, n, normalized=False):
    """
    Return frequences of sequence values

    Inputs:
    - seq: Array of integers. Observable sequence
    - lower_bound: Integer lower bound of the domain
     of generator values
    - upper_bound: Integer upper bound of the domain
     of generator values
    - n: Integer number of regions

    Outputs:
    - freqs: Array of occurences of values in each region
    with region_width
    - region_width: Float width of regions
    """
    freqs = []
    region_width = (upper_bound - lower_bound) / n

    for i in range(n):
        low = lower_bound + i * region_width
        high = lower_bound + i * region_width + region_width
        freqs.append( np.logical_and(seq >= low, seq < high).sum() )

    # because last interval has '[a;b]' - bounds, not '[a,b)'
    freqs[-1] += 1

    if normalized:
        freqs = np.array(freqs) / len(seq)

    return np.array(freqs), region_width

def chisquare(empiric_freqs, probs, n, alpha):
    """
    Inputs:
    - empiric: Empirical frequencies
    - probs: Theoretical probabilities
    - n: Size of sequence
    """
    k = len(empiric_freqs)
    r = k - 1
    s = (np.square(empiric_freqs - probs) / probs).sum() * n
    p = integrate.quad( lambda x: x**(r/2 - 1) * np.exp(-x/2) , s, np.inf)[0] \
                                            / (2**(r/2) * gamma(r/2))
    is_rejected = p <= alpha

    return is_rejected, s, p
```