Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

“Новосибирский государственный технический университет”

Кафедра теоретической и прикладной информатики

Лабораторная работа №4
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 обратной функции и проводить статистический анализ сгенерированных данных.

Исходные данные

Генератор равномерно распределенной псевдослучайной последовательности для генерирования вероятностей биномиального распределения был взят встроенный: numpy.random.uniform.

Исследуемые параметры распределения Вейбулла:

$\mu = 3.5,\ \nu = 6$

$\mu = 2.5,\ \nu = 1.2$

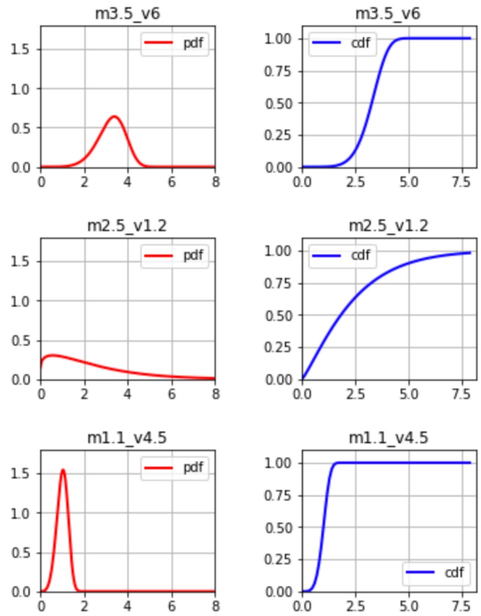
$\mu = 1.1,\ \nu = 4.5$

Исследования

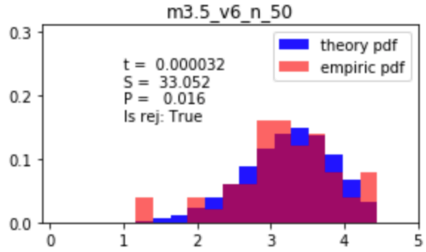
Параметры распределения зададим при помощи словаря, а длины исследуемых последовательностей с помощью массива следующим образом:

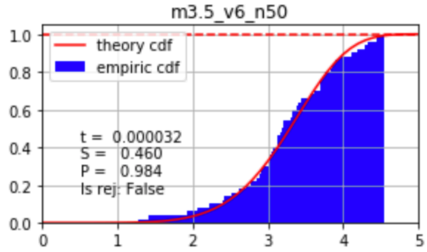
```
weibull_experiments = {
    'm3.5_v6' : {'m' : 3.5, 'v' : 6},
    'm2.5_v1.2' : {'m' : 2.5, 'v' : 1.2},
    'm1.1_v4.5' : {'m' : 1.1, 'v' : 4.5}
}


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

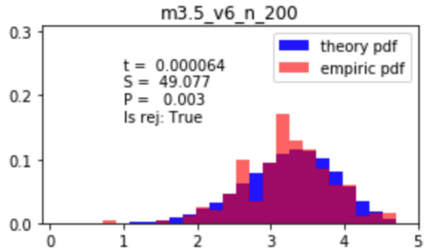
Далее представлены графики плотности (pdf) и функции распределения (cdf). Название каждого графика имеет вид: m{значение параметра μ}_v{значение параметра ν}

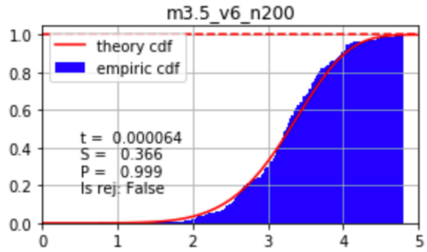


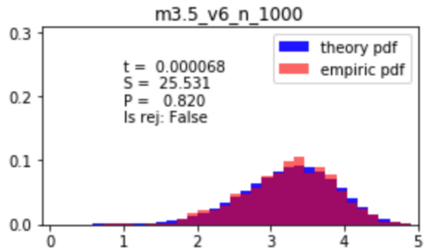
Далее представлены графики эмпирической и теоретической плотностей распределения и функций распределения, с результатами проверок гипотез по критерию хи-квадрат (левый столбец) и критерию Колмогорова (правый столбец). Название каждого графика имеет вид: m{значение параметра μ}_v{значение параметра ν}_n{длина последовательности}

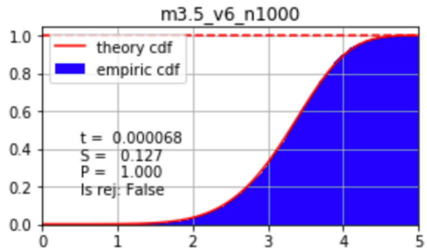


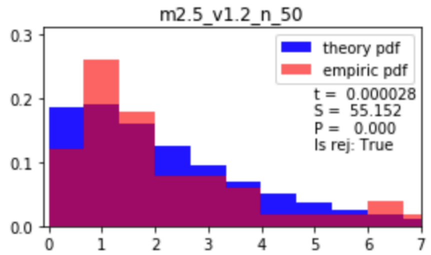


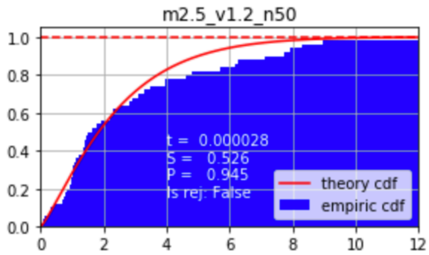


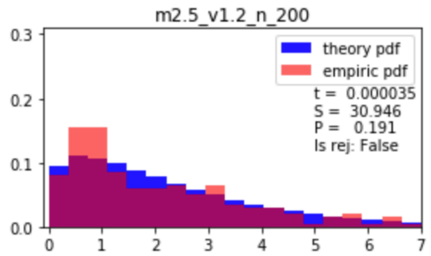


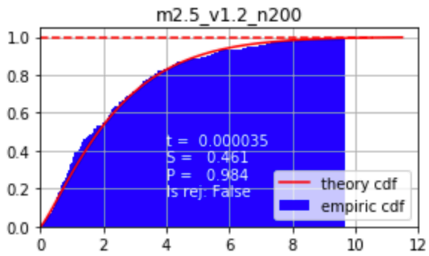


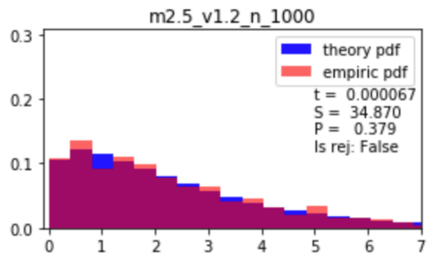


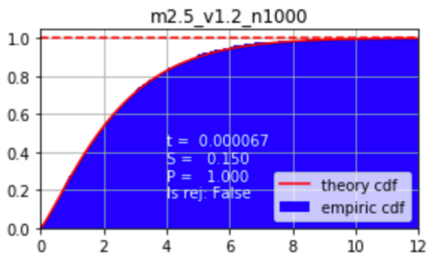


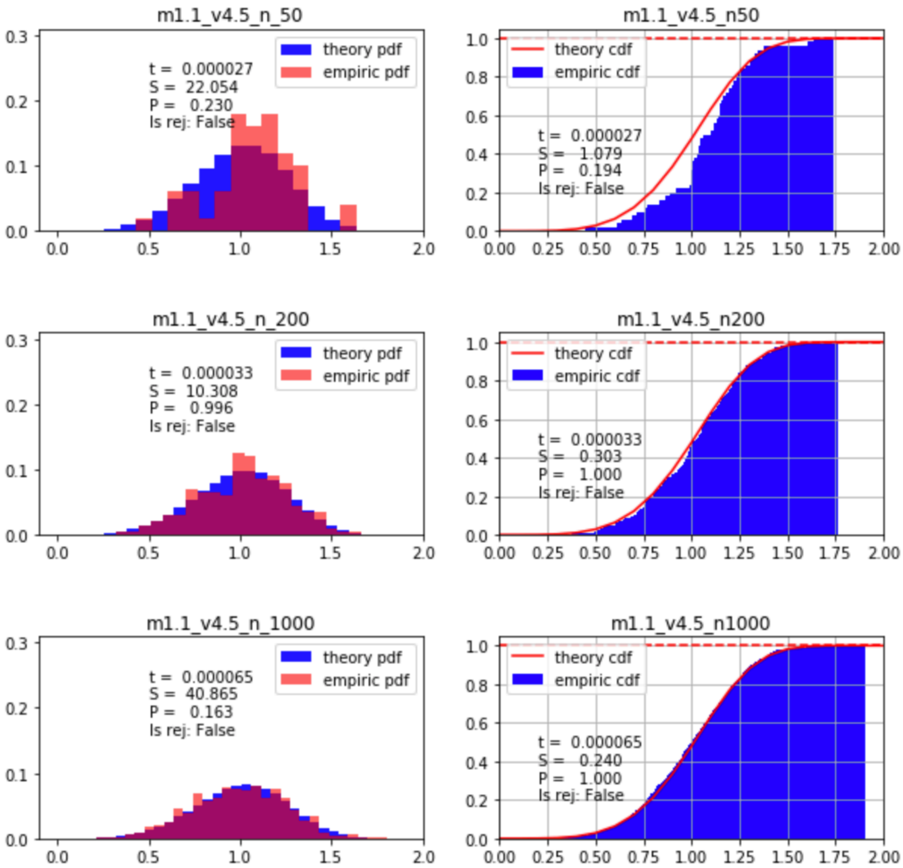












Сгенерированные последовательности (для n = 50, в порядке, соответствующем вышестоящим графикам):

```
1)1.92  3.02  3.69  3.13  3.94  2.51  1.40  3.33  3.75  3.74  1.28  4.18  3.57  3.13
2.39  2.89  2.41  3.12  2.93  4.44  3.90  3.01  2.90  3.20  3.06  3.20  2.07  3.37  4.32
2.71  4.11  3.33  3.18  3.83  4.29  3.71  3.61  2.61  3.44  3.16  3.85  2.22  3.39  4.45
3.70  2.86  3.02  3.30  2.77  3.04

2)2.22  0.92  0.79  4.81  6.29  1.38  0.09  0.28  3.10  1.01  0.38  1.13  8.30  0.97
1.52  0.81  0.02  1.37  1.63  0.12  5.70  2.47  7.73  3.97  1.39  1.45  2.92  1.93  1.03
0.68  1.05  1.01  2.27  12.65  7.13  1.20  2.31  0.98  4.59  1.36  8.95  3.27  6.16  3.50
2.80  3.94  0.32  0.73  7.92  1.77

3)1.12  1.04  1.31  0.77  0.69  1.01  1.15  0.44  1.14  1.14  1.22  1.34  0.99  1.15
1.20  1.00  1.13  1.10  0.76  0.88  1.16  1.33  0.61  1.64  1.04  1.61  1.00  1.23  1.00
1.00  1.04  1.02  1.18  1.15  1.25  1.00  0.83  0.63  1.04  1.03  1.31  0.92  0.72  1.06
1.21  1.37  1.12  1.24  1.28  0.86
```

Вывод

Эмпирические значения плотности вероятности и функции распределения визуально приближаются к теоретическим значениям, при увеличении размера генерируемой выборки. Ни одна гипотеза о том, что генерируемая выборка принадлежит распределению Вейбулла не была отклонена, что говорит о высоком качестве генератора.

Текст программы

weibull_gen.py

```
from .generator import Generator
import numpy as np

class Weibull_gen(Generator):
    def __init__(self, m, v):
        """
        Initialize Weibull generator

        Inputs:
        - m: Float first parameter (the most frequent element)
        - v: Float second parameter
        """
        self.v = v
        self.m = m


    def _pdf(self, x):
        """
        Probability density function of Weibull distribution (aka pdf)

        Inputs:
        - x: Float x-point

        Outputs:
        - y: Float y-point of pdf
        """
        numerator = self.v * np.power(x, self.v - 1) *
                                    np.exp( -np.power(x/self.m, self.v) )
        denominator = self.m ** self.v
        return numerator / denominator


    def _cdf(self, x):
        """
        Cummulative density function of Weibull distribution (aka cdf)

        Inputs:
        - x: Float x-point

        Outputs:
        - y: Float y-point of cdf
        """
        return 1 - np.exp(-(x/self.m) ** self.v)


    def generate(self, N):
        """
        Generate sequence

        Inputs:
        - N: Integer size of sequence

        Outputs:
        - seq: Array of elements from Weibull distribution
        """
        self._N = N
        self._uni_seq = np.random.uniform(size=N)
        # apply inverse distribution function (aka PPF)
        ln = np.log(1 - self._uni_seq)
        root = (-ln) ** (1 / self.v)
        return self.m * root
```

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 kolmogorov(seq, F, alpha=.05, k = None, verbose = True):
    """
    Statistical test of whether a given sample of data is drawn from
    a given probability distribution.

    Inputs:
    - seq: Array of values from distribution
    - F: Cumulative distribution function (aka cdf)
    - alpha: Float desirible level of significance
    - k: Integer number of regions (default is None)
    - verbose: Boolean; If set to true then print logs

    Outputs:
    - Boolean; If hypothesis is rejected
    - s; Value of statistic
    - p; Probability
    """
    # build empirical probability row
    n = len(seq)
    if k is None:
        k = int(5 * np.log(n))
    lower_bound = 0
    upper_bound = max(seq)
    interval_width = (upper_bound - lower_bound) / k
    sorted_seq = sorted(seq)
    left_bounds = np.arange(k) * interval_width

    # calculate value of statistic
    all_bounds = np.append(left_bounds, [(k + 1)*interval_width], -1)
    d_minus = []
    d_plus = []
    i = 1
    for el in sorted_seq:
        d_plus.append( float(i) / n - F(el) )
        d_minus.append( F(el) - float(i - 1) / n )
        i += 1

    d = max(d_minus + d_plus)
    s = (6 * k * d + 1) / (6 *np.sqrt(k) )
    if verbose:
        print('...\n... Dmax = %f\n...' % (d))

    p = 1 - _K(s)

    is_rejected = p <= alpha

    return is_rejected, s, p


def _freqs(seq, lower_bound, upper_bound, k, 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
    - k: Integer number of intervals

    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) / k

    for i in range(k):
        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 _K(s):
    """
    Auxiliary function for integral calculating within Kolmogorov's
    statistical test

    Inputs:
    - s: Float value of statistic

    Outputs:
    - p: Float probability of Kolmogorov distribution
    """
    p = 0
    for k in range(-10, 10, 1):
        p += (-1)**k * np.exp(-2 * k**2 * s**2)
    return p
```

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
```