Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное
учреждение высшего образования

“Новосибирский государственный технический университет”

Кафедра теоретической и прикладной информатики

Лабораторная работа №3
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 дискретно распределённой случайной величины и проводить статистический анализ сгенерированных данных.

Исходные данные

Генератор равномерно распределенной псевдослучайной последовательности для генерирования вероятностей биномиального распределения:
$x_{n+1} = (a x_n^3 + b x_n + c x_{n-1}^2) \ mod \ m$

Для распределения Пуассона использовался генератор *numpy.random.uniform*

Исследуемые параметры биномиального распределения:
$m = 4,\ p = 0.1$
$m = 4,\ p = 0.5$
$m = 4,\ p = 0.9$

Исследуемые параметры распределения Пуассона:
$\lambda = 2$

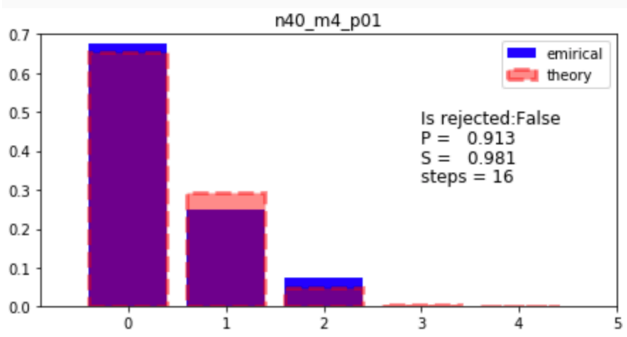
Исследования

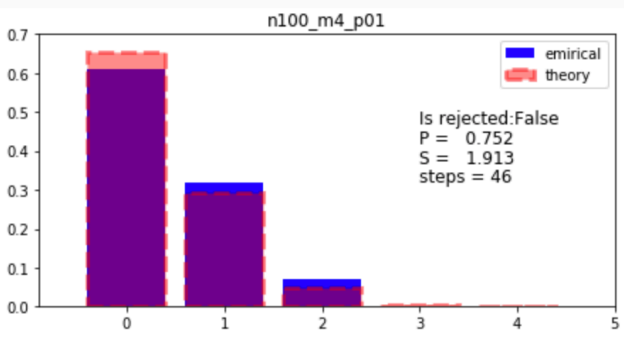
Биномиальное распределение

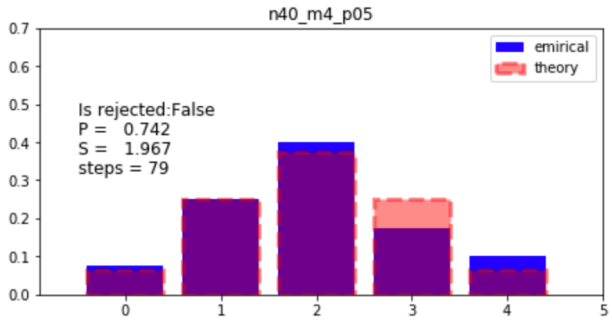
Параметры распределений будем задавать с помощью словарей, а длины исследуемых последовательностей с помощью массива следующим образом:

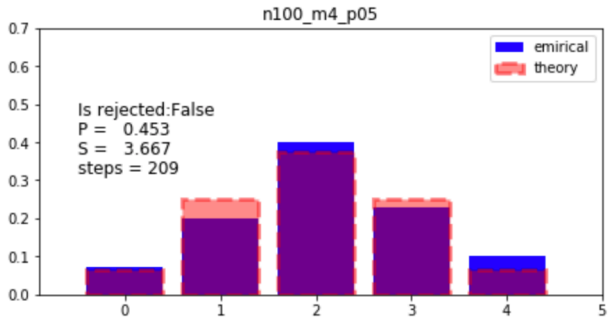
```
binom_experiments = {
    'm4_p01' : {'m' : 4, 'p' : 0.1},
    'm4_p05' : {'m' : 4, 'p' : 0.5},
    'm4_p09' : {'m' : 4, 'p' : 0.9},
}


desirable_sizes = [40, 100]
```

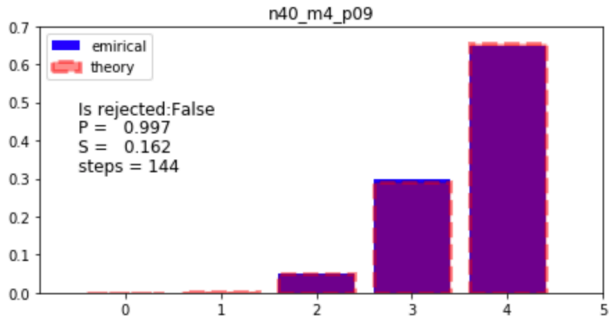
Далее представлены графики, отражающие теоретические и эмпирические частоты элементов генерируемых последовательностей. Название каждого графика имеет вид:
n{длина последовательности}_m{параметр распределения}_p{вероятность успеха}

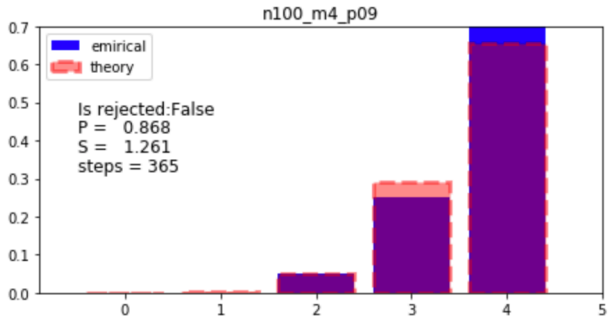












Сгенерированные последовательности (для n = 40, в порядке, соответствующем вышестоящим графикам):

1) 1 0 0 1 0 0 0 1 0 1 0 0 0 0 1 0 2 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 0 1 2 0 1 0 2

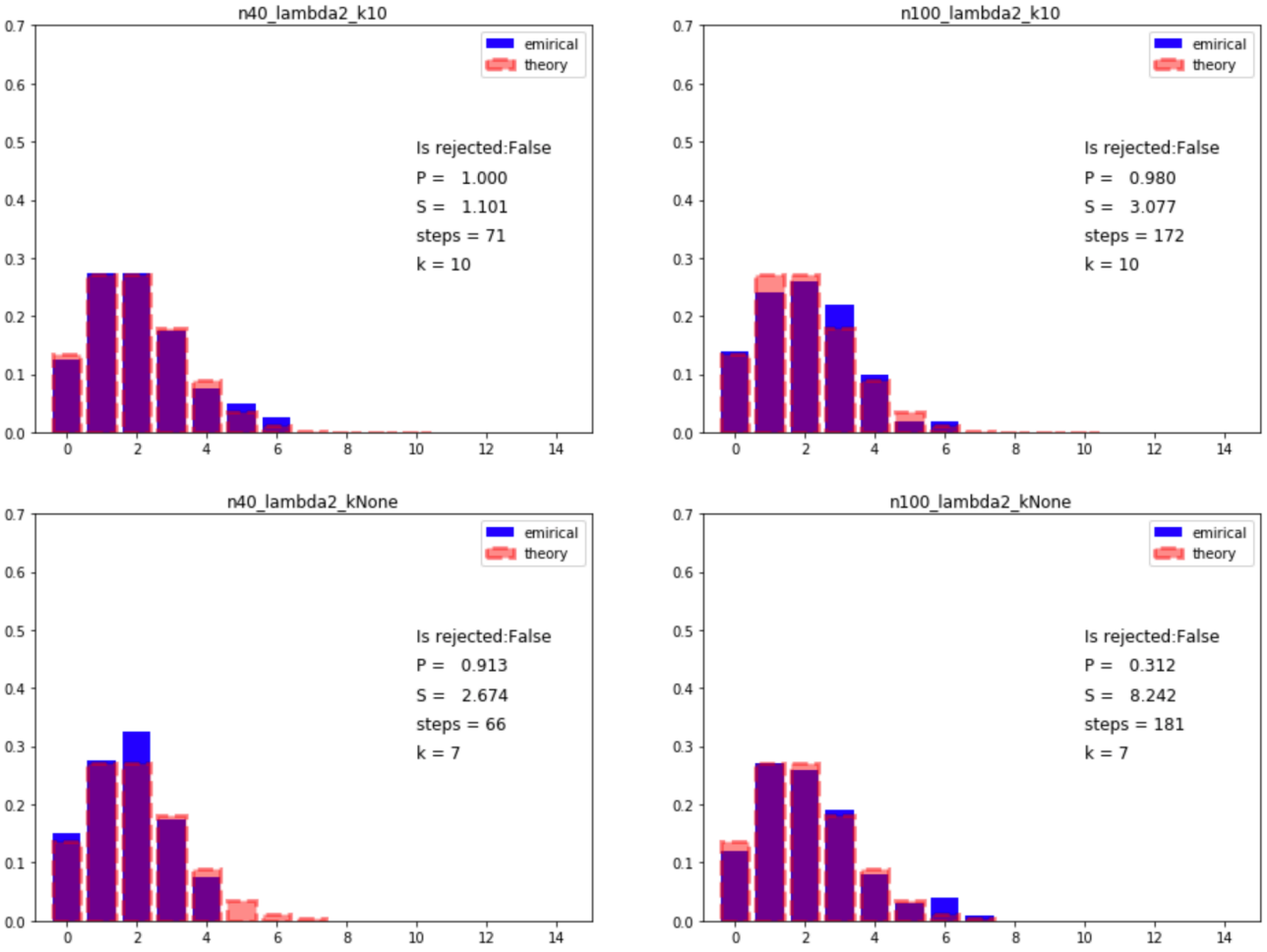
2) 3 0 2 3 2 2 2 4 1 3 2 2 2 1 3 2 4 1 1 0 3 1 1 2 2 1 1 0 1 2 2 3 2 2 2 4 2 3 1 4

3) 4 3 4 4 4 4 4 4 3 4 4 4 4 3 4 4 4 3 3 2 4 3 3 4 4 3 3 2 3 4 3 4 4 4 4 4 4 4 3 4

Распределение Пуассона

В данном распределении вероятностный ряд является бесконечным, и хоть длина ряда (*k*) не является параметром распределения, мы включим ее в словарь экспериментов. Специальное значение *None* для параметра *k* подразумевает эмпирическое вычисление значения этого параметра: ряд генерируется до тех пор, пока требуемая величина последовательности, умноженная на последний сгенерированный элемент ряда не будет равна единице. Попросту говоря, ряд генерируется до тех пор, пока есть теоретическая вероятность появления соответствующих значениям ряда элементов.

```
poisson_experiments = {
    'lambda2_k10' : {'lambda' : 2, 'k' : 10},
    'lambda2_kNone' : {'lambda' : 2, 'k' : None},
}


desirable_sizes = [40, 100]
```



Сгенерированные последовательности (для n = 40, в порядке, соответствующем вышестоящим графикам):

1) 2 1 0 4 3 1 4 2 2 0 1 3 1 1 3 2 4 1 1 3 6 2 2 3 1 1 0 0 0 5 3 3 2 1 1 2 2 2 5 2

2) 1 4 0 0 2 1 2 2 1 3 0 1 2 2 0 1 0 2 2 1 3 2 1 1 3 1 0 2 3 2 2 3 1 3 4 3 2 2 1 4

Вывод

Плотность (вероятность) элементов, сгенерированных последовательностей очень близка к теоретическим значениям. Количество шагов алгоритма при генерировании элементов из биномиального распределения существенно зависит от хода алгоритма (слева направо или справа налево) и от параметра *p (вероятности успеха)*.

Текст программы

binom_gen.py

```
import math
import numpy as np
from .uniform.uni_gen import Uni_generator

class Binom_gen():
    """
    Generate binomial distribution
    """

    def __init__(self, m, p):
        """
        Inputs:
        - m: Integer count of experiments
        - p: Float probability of success
        """
        self.m = m
        self.p = p


    def _combination_(self, n, k):
        """
        Calculate combination from 'n' by 'k'
        """
        res = 1
        for i in range(n - k + 1, n + 1):
            res *= i
        res /= math.factorial(k)
        return res


    def _prob(self, k):
        """
        Calculate probability for appropriate k
        """
        C = self._combination_(self.m, k)
        res = C * self.p**k * (1-self.p)**(self.m-k)
        return res

    def generate(self, N):
        """
        Generate binomial sequence
        """
        # create probability row
        self._prob_row = np.array([])
        for k in range(self.m + 1):
            self._prob_row = np.append(self._prob_row, self._prob(k))
        self._prob_cumsum = np.cumsum(self._prob_row)

        # create seq belong to uni(0, 1)
        uni = Uni_generator(1, 2)
        uni_seq = uni.generate(N, normalized=True)
```

```
        uni_seq = np.array(uni_seq)

        # calculation elements belong to binomial destribution

        # vanila algorithm
        steps = 0
        res = []
        for i in range(len(uni_seq) - 1, -1, -1):
            j = 0
            el = uni_seq[i]
            while el - self._prob_cumsum[j] > 0:
                j += 1
                steps += 1
            res.append(j)

        return np.array(res), steps
```

poisson_gen.py

```
import math
import numpy as np
from .uniform.uni_gen import Uni_generator

class Poisson_gen():
    def __init__(self, alpha, k=None, verbose=False):
        """
        Inputs:
        - alpha: Integer most popular element (aka lambda)
        - k: Integer size of probability row; if set to None
        then will be calculate durin generating process
        - verbose: Boolean; if set to true then print logs
        """
        self.alpha = alpha
        self.k = k
        self._verbose = verbose


    def _get_prob_row_(self):
        """
        Return probability row of poisson distribution
        """
        prob_row = []
        if self.k is not None:
            for i in range(self.k + 1):
                p = self.alpha**i * math.exp(-self.alpha) / math.factorial(i)
                prob_row.append(p)
        else:
            if self._N is None:
                raise ValueError('k is None, but desirable size /
                                  of sequence is not defined! /
                                  Try to set self._N manualy.')
            i = 0
            p = 1
            while self._N * p > 1:
                p = self.alpha**i * math.exp(-self.alpha) / math.factorial(i)
                prob_row.append(p)
                i += 1
            self._k = i

        return np.array(prob_row)


    def _print(self, string):
        """
```

```
    Print logs if self.verbose is equal true
    """
    if self._verbose:
        print(string)


def set_verbose_to(self, verbose):
    """
    self.verbose setter
    """
    if type(verbose) is not bool:
        raise ValueError('verbose must be bool, not ' + str(type(verbose)))
    self._verbose = verbose


def generate(self, N, set_verbose_to=None):
    """
    Inputs:
    - N: Integer length of sequence
    - set_verbose_to: Boolean; if set to None then
    do not change set value

    Outputs:
    - res: Array; generated sequence
    - steps: Integer number of steps required for genearation
    """
    # we have to know value of N if self.k is set to None
    self._N = N
    if set_verbose_to is not None:
        self._verbose = set_verbose_to

    self._uni_seq = np.random.uniform(size=N)

    self._Q = 0
    self._prob_row = self._get_prob_row_()
    for i in range(self.alpha + 1):
        self._Q += self._prob_row[i]

    steps = 0
    res = []
    self._print('. Q = % 5.2f\n'%(self._Q))
    for u in self._uni_seq:
        # calculate subtraction between generated probability
        # and cumulative summ of probabilities of first self._alpha
        # elements
        s = u - self._Q
        self._print('.. u = % 5.2f'%(u))
        self._print('.. u - Q = % 5.2f'%(s))

        # determine the direction of seek
        if s > 0:
            j = 0

            while s  > 1e-7:
                j += 1
                s -= self._prob_row[self.alpha + j]

                steps += 1
            res.append(self.alpha + j)
        elif s < 0:
            j = -1

            while s <= -1e-7:
                j += 1
                s += self._prob_row[self.alpha - j]
```

```
            steps += 1
        res.append(self.alpha - j)
    else:
        res.append(self.alpha)

return np.array(res), steps
```