Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное
учреждение высшего образования

“Новосибирский государственный технический университет”

Кафедра теоретической и прикладной информатики

Лабораторная работа №2
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 равномерно распределённой случайной величины и проводить статистический анализ сгенерированных данных. Построить генератор, дающий для заданного вида генератора достаточно качественную псевдослучайную последовательность.

Исходные данные

Заданная формула для генерирования псевдослучайных чисел:
$x_{n+1} = (ax_n^3 + bx_n + cx_{n-1}^2)\ mod\ m$

Исследования

Для начала подберем параметры a, b, c для нашего генератора. Такие, чтобы период сгенерированной последовательности был более 2000. Случайным образом выберем значения параметров и составим сетку параметров. Затем, перебирая комбинации случайно выбранных значений параметров, генерируем последовательности длинной N > 2000 и вычисляем их периоды, если период вновь сгенерированной последовательности удовлетворяет нашим условиям, то запоминаем соответствующие параметры для следующей генерации последовательностей.

Проведем исследования со следующими параметрами:

Тест №2:	K = 20
Тест №3:	r = 4, K = 8

Результаты исследований:

Параметры генератора	Длина периода, T	Тест №1 (n = 40, n = 100)	Тест №2 (n = 40, n = 100)	Тест №3 (комбинированный)	Критерий χ^2 (n = 2000)	Критерий Андерсона
psr_generator N = 6000, m = 3000, a = 157, b = 246, c = 149, x0 = 3, x1 = 19	2100	+ +	- + + + + +	+	K = 23 P = 0.881212 S = 14.54 +	S = 1.543 P = 0.167 +
np.random N = 6000, low = 0, high = 3000, size = 6000	6000	+ +	- + + - + +	-	K = 23 P = 0.438585 S = 22.36 -	S = 0.918 P = 0.403 +

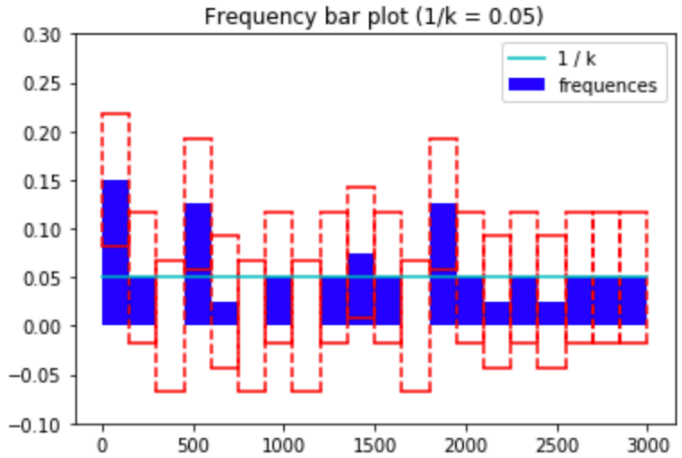
✦ Результаты теста №2 для последовательности, сгенерированной **нашим генератором**:

n = 40

```
-------- (FAILED) ---> Frequency interval test failed! (Take a look on bar plot)

-------- m/2 = 1500.0
-------- expectation interval: [1094.1369657252287 ; 1672.1130342747713]
-------- (PASSED) ---> Expectation interval test passed successfully!

-------- m^2/12 = 750000.0
-------- variance interval: [621464.6855802588 ; 1319875.3487779752]
-------- (PASSED) ---> Variance interval test passed successfully!
```

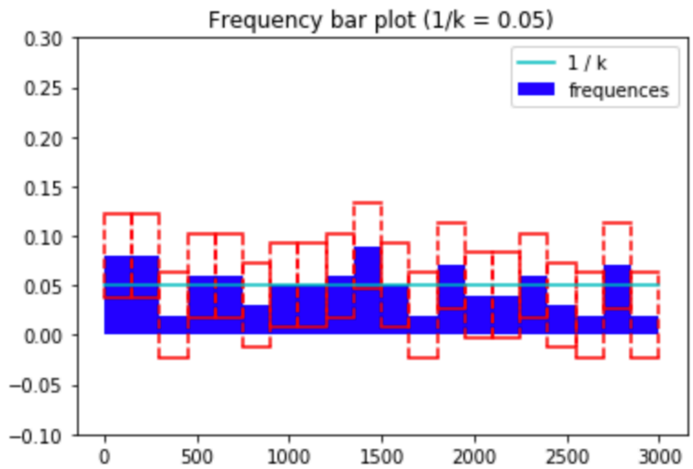


n = 100

```
-------- (PASSED) ---> Frequency interval test passed successfully!

-------- m/2 = 1500.0
-------- expectation interval: [1198.4150389486354 ; 1535.2849610513645]
-------- (PASSED) ---> Expectation interval test passed successfully!

-------- m^2/12 = 750000.0
-------- variance interval: [593340.3537048812 ; 948967.9094883726]
-------- (PASSED) ---> Variance interval test passed successfully!
```



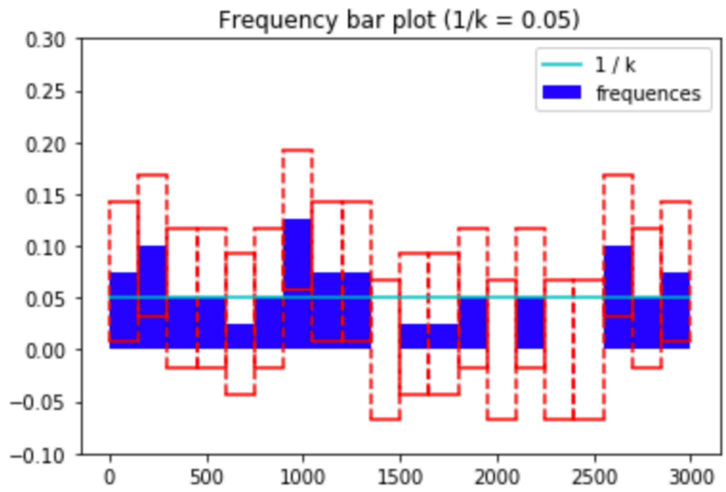
✦ Результаты теста №2 для последовательности, сгенерированной **встроенным генератором numpy.random.randint**:

n = 40

```
-------- (FAILED) ---> Frequency interval test failed! (Take a look on bar plot)

-------- m/2 = 1500.0
-------- expectation interval: [1046.2739014857807 ; 1626.1760985142191]
-------- (PASSED) ---> Expectation interval test passed successfully!

-------- m^2/12 = 750000.0
-------- variance interval: [625613.699543921 ; 1328687.0823799581]
-------- (PASSED) ---> Variance interval test passed successfully!
```

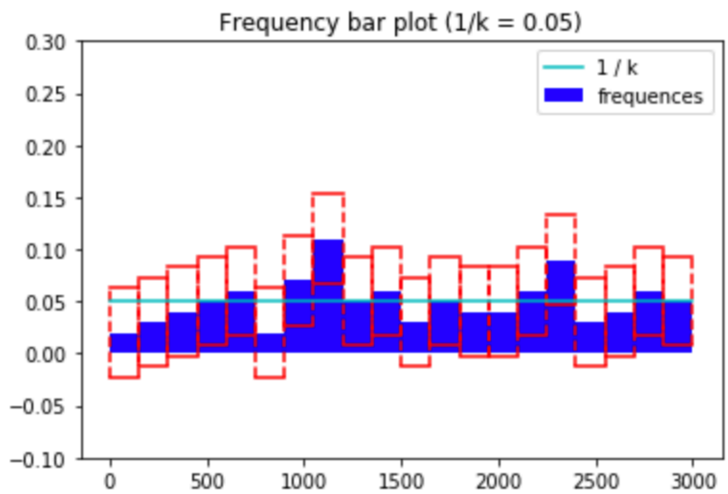


n = 100

```
-------- (FAILED) ---> Frequency interval test failed! (Take a look on bar plot)

-------- m/2 = 1500.0
-------- expectation interval: [1412.5131043728318 ; 1731.2668956271684]
-------- (PASSED) ---> Expectation interval test passed successfully!

-------- m^2/12 = 750000.0
-------- variance interval: [531239.2218087922 ; 849645.5206700232]
-------- (PASSED) ---> Variance interval test passed successfully!
```



✦ Результаты теста №3 для последовательности, сгенерированной **нашим генератором** с n = 200, m = 100:

```
- Permutation testing subsequence #1/4
- (PASSED) ---> Permutation test passed succesfully for subsequence #1/4
- Frequency testing subsequence #1/4
- (PASSED) ---> Frequency test passed succesfully for subsequence #1/4
--------------------------------------------------------------------------------

- Permutation testing subsequence #2/4
- (PASSED) ---> Permutation test passed succesfully for subsequence #2/4
- Frequency testing subsequence #2/4
- (PASSED) ---> Frequency test passed succesfully for subsequence #2/4
--------------------------------------------------------------------------------

- Permutation testing subsequence #3/4
- (PASSED) ---> Permutation test passed succesfully for subsequence #3/4
- Frequency testing subsequence #3/4
- (PASSED) ---> Frequency test passed succesfully for subsequence #3/4
--------------------------------------------------------------------------------

- Permutation testing subsequence #4/4
- (PASSED) ---> Permutation test passed succesfully for subsequence #4/4
- Frequency testing subsequence #4/4
- (PASSED) ---> Frequency test passed succesfully for subsequence #4/4
--------------------------------------------------------------------------------


True
```

✦ Результаты теста №3 для последовательности, сгенерированной **встроенным генератором numpy.random.randint** с n = 200, m = 100:

```
- Permutation testing subsequence #1/4
- (PASSED) ---> Permutation test passed succesfully for subsequence #1/4
- Frequency testing subsequence #1/4
- (PASSED) ---> Frequency test passed succesfully for subsequence #1/4
--------------------------------------------------------------------------------

- Permutation testing subsequence #2/4
- (PASSED) ---> Permutation test passed succesfully for subsequence #2/4
- Frequency testing subsequence #2/4
- (PASSED) ---> Frequency test passed succesfully for subsequence #2/4
--------------------------------------------------------------------------------

- Permutation testing subsequence #3/4
- (PASSED) ---> Permutation test passed succesfully for subsequence #3/4
- Frequency testing subsequence #3/4
- (PASSED) ---> Frequency test passed succesfully for subsequence #3/4
--------------------------------------------------------------------------------

- Permutation testing subsequence #4/4
- (PASSED) ---> Permutation test passed succesfully for subsequence #4/4
- Frequency testing subsequence #4/4
- (FAILED) ---> Frequency test failed for subsequence #4/4

False
```

Вывод

По итогам тестов встроенный генератор менее предпочтителен в силу того, что сгенерированная им последовательность не является равномерной в больших количествах тестов, чем последовательность, сгенерированная построенным нами генератором.

Текст программы

gen_res.py

```
class Psr_generator():
    """
    Adaptive pseudorandom generator
    """
    def __init__(self, N, m, x0, x1):
        """
        Inputs:
        - N: Integer size of generated sequence
        - m: Integer modulus of generator
        - x0: Integer first element of our sequence
        - x1: Integer second element of our sequence
        - params: Dictionary of ranges for generator's parameters
            - a: range of a-values
            - b: range of b-values
            - c: range of c-values
        - low_cost: Boolean; if set to false then perform grid search,
        otherwise find first suitable parameters for receiving
        desired_period
        - desired_perion: None, Integer; if low_cost is set to false
        then have to be integer value
        - verbose: Boolean; if set to true then print T for each set
        of params

        Outputs:
        - T: size of period
        - best_param: parameters for received size of period
        - seq: sequence with best parameters and biggest period
        """
        self.N = N
        self.m = m
        self.x0 = x0
        self.x1 = x1


    def _gen(self, a, b, c, N, m = None):
        """
        Generate pseudorandom sequence

        Inputs:
        - a: Integer first parameter of generator
        - b: Integer second parameter of generator
        - c: Integer third parameter of generator
        - N: Integer size of sequence
        - m: Integer modulus of generator

        Outputs:
        - seq: pseudorandom sequence
        """
        if m is None:
            m = self.m
        x0 = self.x0
```

```
        x1 = self.x1

        seq = [x0, x1]
        for _ in range(N - 2):
            x2 = (a * x1**3 + b * x1 + c * x0**2) % m
            seq.append(x2)
            x0 = x1
            x1 = x2
        return seq


    def _period_of_seq(self, seq, w_size, verbose=False):
        """
        Calculate period of sequence

        Inputs:
        - seq: Array of int
        - w_size: Integer window size.
        - verbose: Boolean; if set to true then print logs

        Outputs:
        - size: Integer size of period
        """
        window = seq[-w_size:]
        T = 0
        print(f'window : {window}')
        for i in range(len(seq) - w_size - 1, 0, -1):
            T += 1
            if i > len(seq) - w_size - 20 and verbose:
                print(f'our seq: {seq[i:i + w_size]}')
            if window == seq[i:i + w_size]:
                return T
        return len(seq)

    def set_params(self, params):
        """
        Explicitly set parameters of generator

        Inputs:
        - params:
            - a: Integer first parameter of generator
            - b: Integer second parameter of generator
            - c: Integer third parameter of generator
        """
        self._gen_params = params


    def fit(self, params, low_cost=False, desired_period=None, verbose=False):
        """
        Train our generator (select best parameters from random grid)

        Inputs:
        - params:
            - a: Array of integers; possible values of first parameter
            - b: Array of integers; possible values of second parameter
            - c: Array of integers; possible values of third parameter
        - low_cost: Boolean; If set to true then stop parameters selection
        when desired period is achieved
        - desired_period: Integer size of period that is desired
        - verbose: Boolean; If set to true then print logs

        Outputs:
        - nothing, but print logs
        """

        if low_cost and desired_period is None:
```

```
            raise ValueError('Check low_cost and desired_period values.')

        num_of_sets = len(params['a']) * len(params['b']) * len(params['c'])

        self._best_res = 0
        self._gen_params = {}
        self._best_seq = None
        self._iter_num = 0
        T = 0
        for ia in params['a']:
            for ib in params['b']:
                for ic in params['c']:
                    self._iter_num += 1
                    seq = self._gen(ia, ib, ic, self.N)
                    T = self._period_of_seq(seq, w_size=2)

                    if verbose:
                        print(f'ia: {ia} | ib: {ib} | ic: {ic} |---> T: {T}')
                    if T > self._best_res:
                        self._best_res = T
                        self._best_seq = seq
                        self._gen_params = {
                            'a' : ia,
                            'b' : ib,
                            'c' : ic,
                            'm' : self.m,
                        }

                    if low_cost and T >= desired_period:
                        print(f'Desired perios is reached \
                        by {self._iter_num} of {num_of_sets} iterations')
                        print(f'T equal to {T}\n')
                        return
        if low_cost:
            print('Desired period is not reached!')
        return


    def generate(self, N = None, m = None):
        """
        Generate pseudorandom sequence via fitted generator.

        Inputs:
        - N: Integer length of desirible sequence
        - m: Integer modulus

        Outputs:
        - seq: Integer array; Generated pseudorandom sequences
        """
        if self._gen_params is None:
            raise ValueError('Parameters of generator is not set. \
                                Use .train or .set_params before.')

        if m is None:
            m = self.m

        a = self._gen_params['a']
        b = self._gen_params['b']
        c = self._gen_params['c']

        if N is None:
            N = self.N

        return self._gen(a, b, c, N, m)
```

empirical_tests.py

```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt


def permutation_test(seq, alpha = 0.05, n = None, verbose=True):
    """
    Randomness test via number of sign permutations.

    Inputs:
    - seq: Array of integer
    - alpha:

    Outputs:
    - boolean: If permutation test is passed then true,
    otherwise false
    """
    if n is None:
        n = len(seq)
    elif len(seq) < n:
        raise ValueError(f'Length of sequence is smaller than \
        n = {n}')

    seq = seq[:n]

    # count number of sign permutations in sequence
    q = 0
    for i in range(len(seq) - 1):
        if seq[i] > seq[i + 1]:
            q += 1

    # compute confidence interval
    u_a = stats.norm.ppf(1 - alpha)

    eps = u_a * np.sqrt(n) / 2
    left_bound = q - eps
    right_bound = q + eps

    if verbose:
        print('Eps: %.3f \nq: %d \nn/2: %.1f \nBounds: [%.3f, %.3f]\n' \
            % (eps, q, n/2, left_bound, right_bound))

    return n/2 <= right_bound and n/2 >= left_bound


def frequency_test(seq,
    m = None,
    k = 20,
    n = None,
    verbose=True,
    alpha = 0.05):
    """
    Randomness test via bar chart

    Inputs:
    - seq: Array of integer
    - m: Integer modulus of generator
    - k: Integer number of regions
    - n: Integer length of sequence
    - verbose: Boolean; If set to true then print along the process
    - alphs: Float level of significance

    Outputs:
    - Boolean; If all tests passed successfully, then return true
```

```
and false otherwise
If verbose is set to true then print logs and bar plots
"""
if n is None:
    n = len(seq)

if len(seq) < n:
    raise ValueError(f'Length of sequence is smaller than n = {n}')
seq = np.array(seq[:n])

if m is None:
    m = max(seq) + 1e-5

# mean, var
mean = seq.mean()
var = seq.var()

# frequency vector
left_borders, widths = [], []
freq = []
region_size = m / k
for i in range(k):
    low = i * region_size
    high = i * region_size + region_size
    freq.append( np.logical_and(seq >= low, seq < high).sum()  )

    left_borders.append(low)
    widths.append(high - low)

freq = np.array(freq) / n

# freq interval tests
freq_int_test = False

u_a = stats.norm.ppf(1 - alpha/2)
eps = (u_a / k) * np.sqrt((k-1)/n)
freq_low = freq - eps
freq_high = freq + eps
freq_int_test = np.prod((1/k >= freq_low) * (1/k <= freq_high))

if verbose:
    if freq_int_test:
        print('-------- (PASSED) --->\
         Frequency interval test passed successfully!\n')
    else:
        print('-------- (FAILED) --->\
         Frequency interval test failed! (Take a look on bar plot)\n')

# expectation interval tests
mean_int_test = False

eps = u_a * np.sqrt(var/n)
mean_low = mean - eps
mean_high = mean + eps
mean_int_test = m/2 >= mean_low and m/2 <= mean_high

if verbose:
    print(f'-------- m/2 = {m/2}')
    print(f'-------- expectation interval: [{mean_low} ; {mean_high}]')
    if mean_int_test:
        print('-------- (PASSED) --->\
         Expectation interval test passed successfully!\n')
    else:
        print('-------- (FAILED) --->\
         Expectation interval test failed!\n')
```

```
    # variance interval tests
    var_int_test = False

    var_low = var * (n - 1) / stats.chi2.ppf(1 - alpha, n - 1)
    var_high = var * (n - 1) / stats.chi2.ppf(alpha, n - 1)
    var_int_test = m*m/12 >= var_low and m*m/12 <= var_high

    # print logs
    if verbose:
        print(f'-------- m^2/12 = {m*m/12}')
        print(f'-------- variance interval: [{var_low} ; {var_high}]')
        if var_int_test:
            print('-------- (PASSED) --->\
             Variance interval test passed successfully!')
        else:
            print('-------- (FAILED) --->\
             Variance interval test failed!')

    # show plot
    if verbose:
        plt.ylim(-0.1, 0.3)
        plt.bar(left_borders, height=freq,
                    width=widths, align='edge',
                    color='blue', label='frequences')
        for i, (l, h) in enumerate(zip(freq_low, freq_high)):
            x = [left_borders[i], left_borders[i] + widths[i]]
            plt.plot(x, [l, l], 'r')
            plt.plot(x, [h, h], 'r')
            plt.plot([x[0], x[0]], [l, h], 'r--')
            plt.plot([x[1], x[1]], [l, h], 'r--')
        plt.title(f'Frequency bar plot (1/k = {1/k})')
        plt.plot([left_borders[0], left_borders[-1] + widths[-1]], \
                    [1/k, 1/k], 'c-', markersize=20, label='1 / k')
        plt.legend()
        plt.show()

    return freq_int_test and mean_int_test and var_int_test


def complex_test(seq, r=4, K=8, low_cost=True, verbose=1):
    """
    Perform frequency and permutation tests for subsequences

    Inputs:
    - seq: Array of integers
    - r: Integer coefficient
    - K: Integer coefficient
    - verbose: Integer 0, 1, 2 or 3; level of verbose
    - low_cost: Boolean;

    Otputs:
    - Boolean; If tests for all subsequences passed successfully,
    then return true and false otherwise
    If verbose is set to true then print logs
    """
    seq = np.array(seq)
    n = len(seq)
    t = (n - r) // r
    mask = np.arange(t+1) * r

    # if verbose == 0
    _verbose = [None, None]

    if verbose == 1:
        _verbose = [False, False]
    elif verbose == 2:
```

```
        _verbose = [False, True]
    elif verbose == 3:
        _verbose = [True, True]

    for i in range(r):
        subseq = seq[mask]

        # permutation test
        if verbose > 0:
            print(f'- Permutation testing subsequence #{i + 1}/{r}')
        if permutation_test(subseq, verbose=_verbose[0], n=len(subseq)):
            if verbose > 0:
                print(f'- (PASSED) --->\
                Permutation test passed succesfully for \
                subsequence #{i + 1}/{r}')
        else:
            if verbose > 0:
                print(f'- (FAILED) --->\
                Permutation test failed for subsequence #{i + 1}/{r}')
            if low_cost:
                return False

        # frequency test
        if verbose > 0:
            print(f'- Frequency testing subsequence #{i + 1}/{r}')
        if frequency_test(subseq, verbose=_verbose[1], k=K):
            if verbose > 0:
                print(f'- (PASSED) --->\
                Frequency test passed succesfully for \
                subsequence #{i + 1}/{r}')
        else:
            if verbose > 0:
                print(f'- (FAILED) --->\
                 Frequency test failed for subsequence #{i + 1}/{r}')
            if low_cost:
                return False

        # print delimiter
        print(80*'-'+'\n')

        mask += 1

    return True


def _freqs(seq, lower_bound, upper_bound, n):
    """
    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() )

    return freqs, region_width
```

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
currentdir =
os.path.dirname(os.path.abspath(inspect.getfile(inspect.currentframe())))
parentdir = os.path.dirname(currentdir)
sys.path.insert(0,parentdir)
from empirical_tests import _freqs


def chisquare_uniform(seq,
                    lower_bound,
                    upper_bound,
                    alpha=0.05,
                    n = None,
                    verbose=True):
    """
    Statistical test applied to sets of categorical data to
    evaluate how likely it is that any observed difference
    between the sets arose by chance.

    Inputs:
    - seq: Array of integers
    - lower_bound: Integer lower bound of the domain of
    generator values
    - upper_bound: Integer upper bound of the domain of
    generator values
    - alpha: Float desirible level of significance
    - n: Integer number of regions (default is None)
    - verbose: Boolean; If set to true then print logs

    Outputs:
    - Boolean; If hypothesis is ejected
    """
    seq = np.array(seq)
    if n is None:
        n = len(seq)
    else:
        seq = np.random.choice(seq, n)

    k = int(5 * np.log(n))
    r = k - 1

    freqs, _ = _freqs(seq, lower_bound, upper_bound, k)

    freqs = np.array(freqs) / n
    p = 1 / k
    s = (np.square(freqs - p) / p).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

    if verbose:
        plt.hist(seq, bins=k, label='freq', color='blue')
        plt.hlines(n // k, lower_bound, upper_bound, 'r', label='n / k')
        plt.title(f'Frequency bar plot (n/k = {n//k})')
        plt.legend()
        plt.show()

        print(f'Number of interval k = {k}')
        print(f'Sequence length n = {n}')
        print('P = %f' % p)

    return is_rejected
```

noparam.py

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

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

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

    Outputs:
    - Boolean; If hypothesis is ejected
    """
    seq = np.array(seq)
    if n is None:
        n = len(seq)
    else:
        seq = np.random.choice(seq, n)

    seq_sorted = sorted(seq)

    s = 0
    for i in range(1, n + 1):
        f = F(seq_sorted[i - 1])
        s += (2*i - 1) * np.log(f) / (2*n) + \
        (1 - (2*i - 1) / (2*n)) * np.log(1 - f)
    s = -n - 2*s

    a2 = 0
    for j in range(15):
        a2 += (-1)**j * g(j + .5) * (4*j + 1) / g(.5) * g(j + 1) * \
        np.exp((4*j + 1)**2 * np.pi**2 / (-8 * s)) * \
        integrate.quad(lambda y: np.exp(s / (8 * (y**2 + 1)) - (4*j + 1)**2 \
                                * np.pi**2 * y**2 / (8 * s)), 0, np.inf)[0]

    a2 *= np.sqrt(2 * np.pi) / s
    p = 1 - a2

    is_rejected = p <= alpha

    if verbose:
```

```
    print(f'Significance level S = {s}')
    print(f'a2 = {a2}')
    print(f'p = {p}')
return is_rejected
```