МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ
Кафедра технологий программирования

Толкун Кирилл Юрьевич

Отчёт по лабораторным работам по курсу “Имитационное и статистическое моделирование”

студента 4 курса 8 группы

Вариант 10

Преподаватель:
Лобач Сергей Викторович
ассистент кафедры ММАД

Работа сдана ________ 2020 г.

Зачтена ________ 2020 г.

____
(подпись преподавателя)

Минск
2020

1 Лабораторная 1

1.1 Условие

Согласно варианту: $\mathbf{X_0} = \alpha = \mathbf{16807}, \mathbf{K} = \mathbf{64}$.

Используя метод Маклерена-Марсальи построить датчик БСВ (1 датчик должен быть мультипликативно конгруэнтный, второй – на выбор). Исследовать точность построенной БСВ.

1. Осуществить моделирование $n = 1000$ реализаций БСВ с помощью мультипликативного конгруэнтного метода (МКМ) с параметрами $X0, \alpha, m = 231$;

2. Осуществить моделирование $n = 1000$ реализаций БСВ с помощью метода Макларена-Марсальи (один датчик должен быть мультипликативно конгруэнтный (п. 1), второй – на выбор). K – объем вспомогательной таблицы;

3. Проверить точность моделирования обоих датчиков (п. 1 и п. 2) с помощью критерия согласия Колмогорова и χ^2 - критерия Пирсона с уровнем значимости $\varepsilon = 0.05$.

1.2 Теория

1.2.1 Датчики БСВ

Для моделирования на ЭВМ реализаций ***Базовой случайной величины*** используются специальные программы, называемые программными датчиками БСВ.
В основе программных датчиков БСВ лежат рекуррентные формулы вида:

$$x_n = \varphi(x_{n-1}, \dots, x_{n-p}), n = 1, 2, \dots, \tag{1.1}$$

где $x_{1-p}, x_{2-p}, \dots, x_0$ ($p \geqslant 1$) - заданные стартовые значения. Описанное соотношение (1.1) описывает детерминированный алгоритм, однако при соответствующем подборе преобразования $\varphi(\cdot)$ получаемые на его основе псевдослучайные числа x_n по своим функциональным и числовым характеристикам близки к БСВ.

Алгоритмы моделирования вида (1.1) обладают общим недостатком: начиная с некоторого момента $\mathbf{t_0}$, последовательность псевдослучайных чисел образует цикл, который повторяется бесконечное число раз. Длина $\mathbf{T}$ циклически повторяющейся последовательности называется ***периодом датчика*** БСВ ($T \leq m - 1$).

Период $\mathbf{T}$ и ***коэффициент использования*** БСВ $\mathbf{k}$ являются основными показателями качества программных датчиков БСВ. Лучшим датчикам соответствуют большие значения $\mathbf{T}$ и $\mathbf{k}$.

1.2.2 Линейный конгруэнтный метод

Линейный конгруэнтный метод - один из методов генерации псевдослучайных чисел. Применяется в простых случаях и не обладает криптографической стойкостью. Входит в библиотеки различных компиляторов.

Суть метода заключается в вычислении последовательности случайных чисел X_n следующим образом:

$$X_{n+1} = \frac{\alpha X_n + c) \bmod m}{m}, \tag{1.2}$$

где:

$$\left.\begin{array}{l} 1.\ X_0 \text{ - начальное значение } (0 \leqslant X_0 < 1) \\ 2.\ \alpha \text{ - множитель } (0 \leqslant \alpha < m) \\ 3.\ c \text{ - приращение } (0 \leqslant c < m) \\ 4.\ m \geq 2 \text{ - модуль} \end{array}\right\} \text{ - } \textbf{\textit{параметры датчика}}.$$

Типовые значения параметров: $m = 2^{31}, x_0 = \alpha = 65539$.

1.2.3 Мультипликативный конгруэнтный метод

Метод генерации *линейной конгруэнтной последовательности* (раздел 1.2.2) при $\mathbf{c = 0}$ называют ***мультипликативным конгруэнтным методом***.

1.2.4 Метод Макларена-Марсальи

Генератор Макларена-Марсальи - криптографически стойкий генератор псевдослучайных чисел, который основан на комбинации двух конгруэнтных генераторов и вспомогательной матрице, с помощью которой происходит перемешивание двух последовательностей, полученных от двух генераторов.

Данный генератор псевдослучайных чисел оперирует с тремя объектами: двумя конгруэнтными генераторами, которые порождают последовательности $\langle \mathbf{X_n} \rangle, \langle \mathbf{Y_n} \rangle$, и массива $\mathbf{V}$, состоящей из $\mathbf{k}$ элементов, обычно $k \in \{64, 28, 256\}$. На выходе последовательность $\langle \mathbf{Z_n} \rangle$.

Генератор состоит из четырёх основных стадий:

1. Инициализация $\mathbf{V}$ и $\mathbf{Z}$ первыми $\mathbf{k}$ элементами последовательности $\langle \mathbf{X_n} \rangle$ - выполняется один раз;

2. Выборка $\mathbf{X}, \mathbf{Y}$ из $\langle \mathbf{X_n} \rangle, \langle \mathbf{Y_n} \rangle$, то есть $\mathbf{X}, \mathbf{Y}$ - очередные члены последовательностей $\langle \mathbf{X_n} \rangle, \langle \mathbf{Y_n} \rangle$;

3. Вычисление $\mathbf{j} = \lfloor \mathbf{k} \cdot \mathbf{Y} \rfloor$, где $\mathbf{j} \in [\mathbf{0}, \mathbf{k})$ - случайное число, определяемое Y;

4. Присвоение $\mathbf{Z_i} = \mathbf{V_i}$ и замена $\mathbf{V_j} = \mathbf{X}$.

Последние три стадии могут повторяться необходимое число раз.

Данный метод позволяет ослабить зависимость между членами последовательности $\mathbf{Z_n}$ и получить сколь угодно большие значения её периода T при условии, что периоды T_1, T_2 исходных датчиков являются взаимно простыми числами. Коэффициент использования БСВ для данного датчика $\mathbf{k} = \mathbf{\frac{1}{2}}$ (за исключением первой реализации, для моделирования которой используется $K + 1$ реализация).

1.2.5 χ^2 критерий согласия Пирсона

Критерий согласия Пирсона - это непараметрический метод, который позволяет оценить значимость различий между фактическим (выявленным в результате исследования) количеством исходов или качественных характеристик выборки, попадающих в каждую категорию, и теоретическим количеством, которое можно ожидать в изучаемых группах при справедливости нулевой гипотезы. Метод позволяет оценить статистическую значимость различий двух или нескольких относительных показателей (частот, долей).

Данный критерий применяют для проверки гипотезы о соответствии эмпирического распределения предполагаемому теоретическому распределению $F(X)$ при большом объёме выборки ($n \geqslant 100$). *Критерий применим для любых видов функции $F(x)$, даже при неизвестных значениях их параметров, что обычно имеет место при анализе результатов механических испытаний.*

Статистика критерия проверки гипотез имеет вид:

$$\chi^2 = \sum_{i=1}^{p} \frac{(n_i - n \cdot p_i)^2}{n \cdot p_i}, \tag{1.3}$$

где n_i - наблюдаемые частоты, $n \cdot p_i$ - ожидаемые частоты.

Чем больше χ^2, тем сильнее выборка X не согласуется с гипотезой H_0 (***нулевая гипотеза***: наблюдаемые частоты соответствуют ожидаемым).

Чтобы проверить гипотезу по *критерию Пирсона* необходимо сравнить *статистику критерия* с *критическим значения*, которой находится в таблице для соответствующего *уровня значимости* и количеству *степеней свободы*.

Пример: при уровне значимости $\alpha = 0.05$ и количестве степеней свободы $\nu = 9$ *критерий Пирсона* согласуется с *нулевой гипотезой* при $\chi^2 < 16.919$.

ν \ α	0,99	0,98	0,95	0,90	0,80	0,70	0,50	0,30	0,20	0,10	0,05	0,02	0,01	α / ν
1	0,00016	0,00628	0,00393	0,0158	0,0642	0,148	0,455	1,074	1,642	2,706	3,841	5,412	6,635	1
2	0,0201	0,0404	0,103	0,211	0,446	0,713	1,386	2,408	3,219	4,605	5,991	7,824	9,210	2
3	0,115	0,185	0,352	0,584	1,005	1,424	2,366	3,605	4,642	6,251	7,815	9,837	11,345	3
4	0,297	0,429	0,711	1,064	1,649	2,195	3,357	4,878	5,989	7,779	9,488	11,668	13,277	4
5	0,554	0,752	1,145	1,610	2,343	3,000	4,351	6,064	7,289	9,236	11,070	13,388	15,086	5
6	0,872	1,134	1,635	2,204	3,070	3,828	5,348	7,231	8,558	10,645	12,592	15,033	16,812	6
7	1,239	1,564	2,167	2,833	3,822	4,671	6,346	8,383	9,803	12,017	14,067	16,622	18,475	7
8	1,646	2,032	2,733	3,490	4,594	5,527	7,344	9,524	11,030	13,362	15,507	18,168	20,090	8
9	2,088	2,532	3,325	4,168	5,380	6,393	8,343	10,656	12,242	14,684	16,919	19,679	21,666	9
10	2,558	3,059	3,940	4,865	6,179	7,267	9,342	11,781	13,442	15,987	18,307	21,161	23,209	10
11	3,053	3,609	4,575	5,578	6,989	8,148	10,341	12,899	14,631	17,275	19,675	22,618	24,725	11
12	3,571	4,178	5,226	6,304	7,807	9,034	11,340	14,011	15,812	18,549	21,026	24,054	26,217	12
13	4,107	4,765	5,892	7,042	8,634	9,926	12,340	15,119	16,985	19,812	22,362	25,472	27,688	13
14	4,660	5,368	6,571	7,790	9,467	10,821	13,339	16,222	18,151	21,064	23,685	26,873	29,141	14
15	5,229	5,985	7,261	8,547	10,307	11,721	14,339	17,322	19,311	22,307	24,996	28,259	30,578	15
16	5,812	6,614	7,962	9,312	11,152	12,624	15,338	18,418	20,465	23,542	26,296	29,633	32,000	16
17	6,408	7,255	8,672	10,085	12,002	13,531	16,338	19,511	21,615	24,769	27,587	30,995	33,409	17
18	7,015	7,906	9,390	10,865	12,857	14,440	17,338	20,601	22,760	25,989	28,869	32,346	34,805	18
19	7,633	8,567	10,117	11,651	13,716	15,352	18,338	21,689	23,900	27,204	30,144	33,687	36,191	19
20	8,260	9,237	10,851	12,443	14,578	16,266	19,337	22,775	25,038	28,412	31,410	35,020	37,566	20

Рис. 1: Значения χ^2 при различных P_{χ^2} в зависимости от числа степеней свобод ν.

1.2.6 Критерий согласия Колмогорова

Критерий согласия Колмогорова предназначен для проверки гипотезы о принадлежности выборки некоторому закону распределения, то есть проверки того, что эмпирическое распределение соответствует предполагаемой модели.

Эмпирическая функция распределения $\mathbf{F_n}$, построенная по выборке $X = (X_1, \dots, X_n)$, имеет вид:

$$F_n(x) = \frac{1}{n}\sum_{i=1}^{n} I_{X_i \leqslant x} \tag{1.4}$$

где $I_{X_i \leqslant x}$ указывает, попало ли наблюдение X_i в область $(-\infty, x]$:

$$I_{X_i \leqslant x} = \begin{cases} 1, X_i \leqslant x \\ 0, X_i > x \end{cases} \tag{1.5}$$

Статистика критерия для эмпирической функции распределения $F_n(x)$ определяется следующим образом:

$$D_n = \sup_{x \in R} |F_n(x) - F(x)| \tag{1.6}$$

Принятие решения по критерию Колмогорова: В случае справедливости *нулевой гипотезы* (H_0) при $n \to +\infty$ статистика D_n имеет распределение Колмогорова:

$$\lim_{n\to\infty} P(\sqrt{n}D_n < x) = K(x) \tag{1.7}$$

здесь

$$K(x) = \sum_{k=-\infty}^{+\infty} (-1)^k e^{-2k^2x^2} \approx 1 - 2e^{-2x^2}, x \geqslant 0 \tag{1.8}$$

- функция Колмогорова.

При ***уровне значимости*** α пороговое значение C_α, находится из соотношения:

$$K(C_\alpha) = 1 - \alpha \tag{1.9}$$

Таким образом, для проверки гипотезы о виде распределения получаем:

$$\rho(X) = \begin{cases} H_0, \sqrt{n}D_n \leqslant \alpha, \\ H_1, \sqrt{n}D_n > \alpha. \end{cases} \tag{1.10}$$

C_α	α	C_α	α	C_α	α	C_α	α	C_α	α	C_α	α	C_α	α
$\alpha \leq 0,29$	1,00000	**0,62**	0,8368	**0,95**	0,3275	**1,28**	0,0755	**1,61**	0,0112	**1,94**	0,0011	**2,27**	0,0001
0,30	0,99999	**0,63**	0,8222	**0,96**	0,3154	**1,29**	0,0717	**1,62**	0,0105	**1,95**	0,0010	**2,28**	0,0001
0,31	0,99998	**0,64**	0,8073	**0,97**	0,3036	**1,30**	0,0681	**1,63**	0,0098	**1,96**	0,0009	**2,29**	0,0001
0,32	0,99995	**0,65**	0,7920	**0,98**	0,2921	**1,31**	0,0646	**1,64**	0,0092	**1,97**	0,0009	**2,30**	0,0001
0,33	0,99991	**0,66**	0,7764	**0,99**	0,2809	**1,32**	0,0613	**1,65**	0,0086	**1,98**	0,0008	**2,31**	0,000046
0,34	0,99993	**0,67**	0,7604	**1,00**	0,2700	**1,33**	0,0582	**1,66**	0,0081	**1,99**	0,0007	**2,32**	0,000042
0,35	0,9997	**0,68**	0,7442	**1,01**	0,2594	**1,34**	0,0551	**1,67**	0,0076	**2,00**	0,0007	**2,33**	0,000038
0,36	0,9995	**0,69**	0,7278	**1,02**	0,2492	**1,35**	0,0522	**1,68**	0,0071	**2,01**	0,0006	**2,34**	0,000035
0,37	0,9992	**0,70**	0,7112	**1,03**	0,2392	**1,36**	0,0495	**1,69**	0,0066	**2,02**	0,0006	**2,35**	0,000032
0,38	0,9987	**0,71**	0,6945	**1,04**	0,2296	**1,37**	0,0469	**1,70**	0,0062	**2,03**	0,0005	**2,36**	0,000030
0,39	0,9981	**0,72**	0,6777	**1,05**	0,2202	**1,38**	0,0444	**1,71**	0,0058	**2,04**	0,0005	**2,37**	0,000027
0,40	0,9972	**0,73**	0,6609	**1,06**	0,2111	**1,39**	0,0420	**1,72**	0,0054	**2,05**	0,0004	**2,38**	0,000024
0,41	0,9960	**0,74**	0,6440	**1,07**	0,2024	**1,40**	0,0397	**1,73**	0,0050	**2,06**	0,0004	**2,39**	0,000022
0,42	0,9945	**0,75**	0,6272	**1,08**	0,1939	**1,41**	0,0375	**1,74**	0,0047	**2,07**	0,0004	**2,40**	0,000020
0,43	0,9926	**0,76**	0,6104	**1,09**	0,1857	**1,42**	0,0354	**1,75**	0,0044	**2,08**	0,0004	**2,41**	0,000018
0,44	0,9903	**0,77**	0,5936	**1,10**	0,1777	**1,43**	0,0335	**1,76**	0,0041	**2,09**	0,0003	**2,42**	0,000016
0,45	0,9874	**0,78**	0,5770	**1,11**	0,1700	**1,44**	0,0316	**1,77**	0,0038	**2,10**	0,0003	**2,43**	0,000014
0,46	0,9840	**0,79**	0,5605	**1,12**	0,1626	**1,45**	0,0298	**1,78**	0,0035	**2,11**	0,0003	**2,44**	0,000013
0,47	0,9800	**0,80**	0,5441	**1,13**	0,1555	**1,46**	0,0282	**1,79**	0,0033	**2,12**	0,0002	**2,45**	0,000012
0,48	0,9753	**0,81**	0,5280	**1,14**	0,1486	**1,47**	0,0266	**1,80**	0,0031	**2,13**	0,0002	**2,46**	0,000011
0,49	0,9700	**0,82**	0,5120	**1,15**	0,1420	**1,48**	0,0250	**1,81**	0,0029	**2,14**	0,0002	**2,47**	0,000010
0,50	0,9639	**0,83**	0,4962	**1,16**	0,1356	**1,49**	0,0236	**1,82**	0,0027	**2,15**	0,0002	**2,48**	0,000009
0,51	0,9572	**0,84**	0,4806	**1,17**	0,1294	**1,50**	0,0222	**1,83**	0,0025	**2,16**	0,0002	**2,49**	0,000008
0,52	0,9497	**0,85**	0,4653	**1,18**	0,1235	**1,51**	0,0209	**1,84**	0,0023	**2,17**	0,0002	**2,50**	0,0000075
0,53	0,9415	**0,86**	0,4503	**1,19**	0,1177	**1,52**	0,0197	**1,85**	0,0021	**2,18**	0,0001	**2,55**	0,0000044
0,54	0,9325	**0,87**	0,4355	**1,20**	0,1122	**1,53**	0,0185	**1,86**	0,0020	**2,19**	0,0001	**2,60**	0,0000026
0,55	0,9228	**0,88**	0,4209	**1,21**	0,1070	**1,54**	0,0174	**1,87**	0,0019	**2,20**	0,0001	**2,65**	0,0000016
0,56	0,9124	**0,89**	0,4067	**1,22**	0,1019	**1,55**	0,0164	**1,88**	0,0017	**2,21**	0,0001	**2,70**	0,0000010
0,57	0,9013	**0,90**	0,3927	**1,23**	0,0970	**1,56**	0,0154	**1,89**	0,0016	**2,22**	0,0001	**2,75**	0,0000006
0,58	0,8896	**0,91**	0,3791	**1,24**	0,0924	**1,57**	0,0145	**1,90**	0,0015	**2,23**	0,0001	**2,80**	0,0000003
0,59	0,8772	**0,92**	0,3657	**1,25**	0,0879	**1,58**	0,0136	**1,91**	0,0014	**2,24**	0,0001	**2,85**	0,00000018
0,60	0,8643	**0,93**	0,3527	**1,26**	0,0836	**1,59**	0,0127	**1,92**	0,0013	**2,25**	0,0001	**2,90**	0,00000010
0,61	0,8508	**0,94**	0,3399	**1,27**	0,0794	**1,60**	0,0120	**1,93**	0,0012	**2,26**	0,0001	**2,95**	0,00000006

Рис. 2: Значения C_α при различных α.

Пример: при уровне значимости $\alpha = 0.05$ и пороговое значение из соотношений (1.8), (1.9) $\mathbf{C}_\alpha \approx \mathbf{1.359}$.

Критерий согласия Колмогорова для непрерывного равномерное распределения

Непрерывное равномерное распределение - распределение случайной вещественной величины, принимающей значения, принадлежащие некоторому промежутку конечно длины, характеризующая тем, что плотность вероятности на этом промежутке почти всюду постоянна.

Функция распределения:

$$F_X(x) \equiv P(X \leqslant x) = \begin{cases} 0, x < a \\ \frac{x-a}{b-a}, a \leqslant x < b \\ 1, x \geqslant b \end{cases} \tag{1.11}$$

Значения теоретическая функция распределения для интервала $[0, 1)$:

$$F(x) = \frac{x - 0}{1 - 0} = x \tag{1.12}$$

Значения эмпирической функции распределения.

Для x_i из выборки X, значение эмпирической функции распределения:

$$F_n(x) = \frac{n_i}{n} \tag{1.13}$$

где n - количество элементов в выборке, n_i - количество элементов в выборке меньших x_i.

1.3 Код программы

```
import math

import matplotlib.pyplot as plt


def linear_congruential_generator(x, alpha, c, m):
    while (True):
        x = (alpha * x + c) % m
        yield x / m


def multiplexial_congruential_generator(x, alpha, m):
    generator = linear_congruential_generator(x, alpha, 0, m)
    while (True):
        yield next(generator)


def mclaren_marsaglia_generator(x_generator, y_generator, k):
    V = [next(x_generator) for _ in range(k)]
    while (True):
        X = next(x_generator)
        Y = next(y_generator)
        j = math.floor(k * Y)
        yield V[j]
        V[j] = X


def hi_squared_test(values, k, critical_value):
    nu = [0] * k
    for value in values:
        nu[math.floor(value * k)] += 1
    p_k = len(values) / k
    hi_squared = 0
    for value in nu:
        hi_squared += ((value - p_k) ** 2) / p_k
    # Дляуровнязначимости      0.05 прити 9- степеняхсвободы .
    return hi_squared < critical_value, hi_squared
```

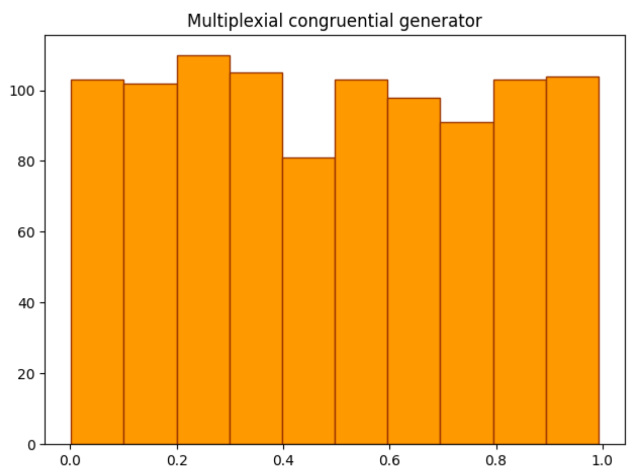
```


def kolmogorov_test(values, critical_value):
    values.sort()
    Dn = 0
    i = 0
    n = len(values)
    for value in values:
        i += 1
        # F(X) = (x-a)/(b-a) = для[ a = 0 иb = 1] = x.
        theoretical_func_res = value
        # колво- значениеввыборкеменьшихтекущегозначенияизвыборки      .
        empirical_function_result = i / n
        Dn = max(Dn, abs(theoretical_func_res - empirical_function_result))
    Dn *= math.sqrt(n)
    return Dn < critical_value, Dn


x0 = 16807
alpha0 = 16807
K = 64

x1 = 8195
alpha1 = 8195
c = 46
k = 64

m = 2 ** 31

hi_squared_critical_value = 16.919
kolmogorov_critical_value = 1.359

mult_congr_gen = multiplexial_congruential_generator(x0, alpha0, m)
x = [next(mult_congr_gen) for _ in range(1000)]
# print('\n'.join(map(str, x)))
hi_squa_test1 = hi_squared_test(x, 10, hi_squared_critical_value)
kolm_test1 = kolmogorov_test(x, kolmogorov_critical_value)

print('Multiplexial congruential generator:')
print('Hi Squared Pirson criteria: ' + str(hi_squa_test1[1]) + ' <= '
      + str(hi_squared_critical_value) if hi_squa_test1[0] else
      'Zero hypothesis fails by Hi Squared Pirson criteria.')
print('Kolmogorov criteria: ' + str(kolm_test1[1]) + ' <= '
      + str(kolmogorov_critical_value) if kolm_test1[0]
      else 'Zero hypothesis fails by Kolmogorov criteria.')

plt.hist(x, 10, ec='#993300', facecolor='#ff9900')
plt.title('Multiplexial congruential generator')
plt.show()

x = linear_congruential_generator(x0, alpha0, 0, m)
y = linear_congruential_generator(x1, alpha1, c, m)
mclar_mars_gen = mclaren_marsaglia_generator(x, y, k)
z = [next(mclar_mars_gen) for _ in range(1000)]
# print('\n'.join(map(str, z)))
```

```python
hi_squa_test2 = hi_squared_test(z, 10, hi_squared_critical_value)
kolm_test2 = kolmogorov_test(z, kolmogorov_critical_value)

print('\nMcLaren marsaglia generator:')
print('Hi Squared Pirson criteria: ' + str(hi_squa_test2[1]) + ' <= '
      + str(hi_squared_critical_value) if hi_squa_test2[0]
      else 'Zero hypothesis fails by Hi Squared Pirson criteria.')
print('Kolmogorov criteria: ' + str(kolm_test2[1]) + ' <= '
      + str(kolmogorov_critical_value) if kolm_test2[0]
      else 'Zero hypothesis fails by Kolmogorov criteria.')

plt.hist(z, 10, ec='#666633', facecolor="#99ff33")
plt.title('McLaren marsaglia generator')
plt.show()
```

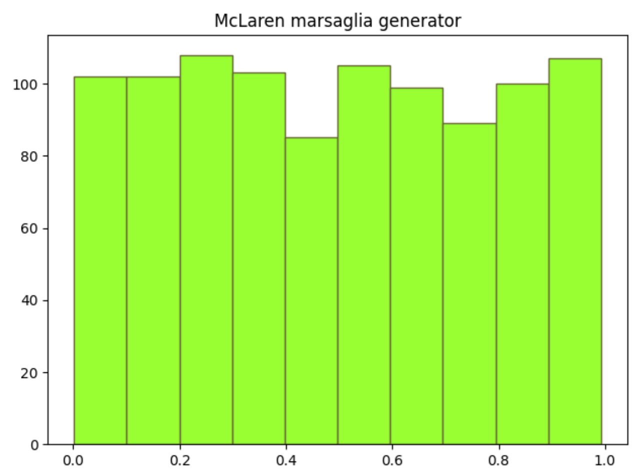
1.4 Результат выполнения

```
Multiplexial congruential generator:
Hi Squared Pirson criteria: 7.300000000000001 <= 16.919
Kolmogorov criteria: 0.12043782997824995 <= 1.359

McLaren marsaglia generator:
Hi Squared Pirson criteria: 4.840000000000001 <= 16.919
Kolmogorov criteria: 0.1608008491673147 <= 1.359
```

Рис. 3: Результат выполнения программы: проверка критерием согласия Пирсона и критерием согласия Колмогорова.



(a) Диаграмма выборки, полученной мультипликативным конгруэнтным методом.



(b) Диаграмма выборки, полученной методом Макларена-Марсальи.

2 Лабораторная 2

2.1 Условие

Согласно варианту 10:

1. Пуассона – $\Pi(\lambda), \lambda = 0.7$; Геометрическое – $G(p), p = 0.2$;

2. Бернулли – $Bi(1, p), p = 0.75$; Пуассона – $\Pi(\lambda), \lambda = 1$;

Смоделировать дискретную случайную величину. Исследовать точность моделирования.

1. Осуществить моделирование $n = 1000$ реализаций СВ из заданных дискретных распределений;

2. Вывести на экран несмещённые оценки математического ожидания и дисперсии, сравнить их с истинными значениями;

3. Для каждой из случайных величин построить свой χ^2-критерий Пирсона с уровнем значимости $\varepsilon = 0.05$. Проверить, что вероятность ошибки I рода стремится к 0.05;

4. Осуществить проверку каждой из сгенерированных выборок каждым из построенных критериев.

2.2 Теория

2.2.1 Датчик случайной величины распределения Пуассона

Распределение Пуассона - распределение дискретного типа случайной величины, представляющей собой число событий, произошедших за фиксированное время, при условии, что данные события происходят с некоторой фиксированной средней интенсивностью и независимо друг от друга.

Функция распределения:

$$\frac{\Gamma(k+1,\lambda)}{k!}. \tag{2.1}$$

Функция вероятности:

$$\frac{e^{-\lambda}\lambda^k}{k!}. \tag{2.2}$$

Математическое ожидание: λ.

Дисперсия: λ.

Алгоритм моделирования:

При моделировании будем использовать свойство пуассоновского процесса, состоящего в том, что время ожидания появления события имеет показательное распределение:

$$F_\tau(t) = 1 - e^{-\lambda t}. \tag{2.3}$$

Следовательно, последовательность наступления событий в пуассоновском процессе можно задать через *последовательность времён ожидания* этих событий. При этом надо проверять, чтобы суммарное время *суммарное время ожидания событий в цепочке не превышала единицы.*

Последовательность времён ожидания можно получить методом обратных функций:

$$\tau_i = -\frac{1}{\lambda} \cdot \ln(1 - u_i), \tag{2.4}$$

где $u_i = rnd(1)$ - *случайные числа*, т.е. значения случайной величины (СВ), равномерно распределённой на [0, 1].

Последовательность (2.4) следует продолжать, пока не нарушается условие:

$$\sum_{i=1}^{j} \tau_i = \sum_{i=1}^{j} (-\frac{1}{\lambda} \cdot \ln(1 - u_i \leqslant 1. \tag{2.5}$$

Максимально возможное количество слагаемых в сумме (2.5) и будет равно числу появления событий в данной серии, т.е. эти числа и есть значения случайной величины, имеющей распределение Пуассона.

1. Во-первых, в (2.5) заменим выражение $1 - u_i$ просто на u_i, поскольку они имеют один и тот же закон распределения;

2. Во-вторых, избавимся от операций логарифмирования в каждом слагаемом, для чего пропотенциируем выражение (2.5).

Таким образом, определим случайную величину:

$$\xi = max \left\{ j : \prod_{i}^{j} u_i \geqslant e^{-\lambda} \right\}, \lambda > 0, \tag{2.6}$$

которая описывается распределением Пуассона. Элемент выборки можно получить последовательно увеличивая число членов (j) в произведении до тех пор, пока не нарушится условие:

$$\prod_{i}^{j} u_i \geqslant e^{-\lambda}, \tag{2.7}$$

максимальное значение (j), удовлетворяющее этому условию и есть очередное значение случайной величины.

Программа создания выборки:

$$D(\lambda, N) := \left|\begin{array}{l} d \leftarrow \exp(-\lambda) \\ j \leftarrow 0 \\ for\ \ k \in 0..N \\ \quad \left|\begin{array}{l} x \leftarrow rnd(1) \\ while\ x > d \\ \quad \left|\begin{array}{l} x \leftarrow x \cdot rnd(1) \\ j \leftarrow j + 1 \end{array}\right. \\ P_k \leftarrow j \\ j \leftarrow 0 \end{array}\right. \\ P \end{array}\right.$$

Рис. 5: Псевдоалгоритм генерации СВ распределения Пуассона.

2.2.2 Датчик случайной величины геометрического распределения

Под **Геометрическим распределением** в теории вероятностей подразумевают одно из двух распределений дискретной случайной величины:

- распределение вероятностей случайной величины X равное номеру первого "успеха" в серии испытания Бернулли и принимающей значения $n = 1, 2, 3, \ldots$;
- распределение вероятностей случайной величины $Y = X - 1$, равное количеству "неудач" до первого "успеха" и принимающей значения $n = 0, 1, 2, \ldots$.

Функция распределения:

$$1 - q^{n+1}. \tag{2.8}$$

Функция вероятности:

$$q^n p. \tag{2.9}$$

Математическое ожидание:

$$\frac{q}{p}. \tag{2.10}$$

Дисперсия:

$$\frac{q}{p^2}. \tag{2.11}$$

Алгоритм моделирования:

1. Моделирование реализации α БСВ;

2. Принятие решения о том, что реализация ξ является значением x, определяемым соотношением:

$$x = \left[\frac{\ln \alpha}{\ln q}\right], \tag{2.12}$$

где $[z]$ - округление числа z в большую сторону до ближайшего целого значения.

2.2.3 Датчик случайной величины распределения Бернулли

Распределение Бернулли - дискретное распределение вероятностей, моделирующее случайный эксперимент произвольной природы, при заранее известной вероятности успеха или неудачи.:

Функция распределения:

$$\begin{cases} 0, k < 0 \\ q, 0 \leqslant k < 1 \\ 1, k \geqslant 1 \end{cases}. \tag{2.13}$$

Функция вероятности:

$$\begin{cases} q, k = 0 \\ p, k = 1 \end{cases}. \tag{2.14}$$

Математическое ожидание:

$$p. \tag{2.15}$$

Дисперсия:

$$pq. \tag{2.16}$$

Алгоритм моделирования:

1. Моделирование реализации α БСВ;

2. Принятие решения о том, что реализация ξ является значением x, определяемым по правилу:

$$x = \begin{cases} 1, \alpha \leqslant p \\ 0, \alpha > p \end{cases}. \tag{2.17}$$

2.3 Код программы

```
import math
from collections import Counter
from functools import partial

import matplotlib.pyplot as plt


def linear_congruential_generator(x, alpha, c, m):
    while True:
```

```
        x = (alpha * x + c) % m
        yield x / m


def poisson_generator(l, linear_gen):
    d = math.exp(-l)
    while True:
        x = 1
        j = 0
        while x > d:
            x *= next(linear_gen)
            j += 1
        yield j - 1


def geometric_generator(p, linear_gen):
    while True:
        yield math.floor(math.log(next(linear_gen)) / math.log(1 - p))


def bernoulli_generator(p, linear_gen):
    while True:
        yield int(next(linear_gen) <= p)


def hi_squared_test(values, distribution_func, critical_value):
    distinct_map = Counter(values).most_common()
    exampling_size = len(values)
    hi_squared = 0
    for pair in distinct_map:
        empiric_freq = pair[1]
        random_value = pair[0]
        theoretic_freq = math.ceil(
            exampling_size * distribution_func(random_value))
        hi_squared += ((empiric_freq - theoretic_freq) ** 2) / theoretic_freq
    # Дляуровнязначимости      0.05 прити 9- степеняхсвободы .
    return hi_squared < critical_value, hi_squared


def empirical_expectation_func(values):
    return sum(values) / len(values)


def empirical_dispersion_func(values):
    expectation = empirical_expectation_func(values)
    result = 0
    for value in values:
        result += (value - expectation) ** 2
    return result / len(values) - 1


def poisson_distribution_func(l, value):
    return l ** value * math.exp(-l) / math.factorial(value)
```

```
def geometric_distribution_func(p, unique_x_geometric, value):
    return (1 - p) ** unique_x_geometric.index(value) * p


def bernoulli_distribution_func(p, value):
    return p if value == 1 else 1 - p


x0 = 79507
alpha0 = 79507
m = 2 ** 31

# POISSON SAMPLE, LAMBDA = 0.7.
l = 0.7
poisson_gen = poisson_generator(l, linear_congruential_generator(x0, alpha0,
                                                           0, m))
x_poisson = [next(poisson_gen) for _ in range(1000)]
# print('\n'.join(map(str, x_poisson)))

unique_x_poisson = sorted(list(Counter(x_poisson).keys()))
# Колво- степенейсвободыдля ( 10 варианта 6 - 1 = 5 степенейсвободы)
k_poisson = len(unique_x_poisson)
critical_value_poisson = 11.07
hi_squa_test1 = hi_squared_test(x_poisson,
                             partial(poisson_distribution_func, l),
                             critical_value_poisson)
print('Poisson generator, lambda = 0.7:')
print('Hi Squared Pirson criteria: ' + str(hi_squa_test1[1]) + ' <= '
     + str(critical_value_poisson) if hi_squa_test1[0] else
     'Zero hypothesis fails by Hi Squared Pirson criteria.')

theoretical_expectation = l
empirical_dispersion = empirical_dispersion_func(x_poisson)
theoretical_dispersion = l
empirical_expectation = empirical_expectation_func(x_poisson)
print('theoretical expectation: ', theoretical_expectation)
print('empirical expectation: ', empirical_expectation)
print('theoretical dispersion: ', theoretical_dispersion)
print('empirical dispersion: ', empirical_dispersion)
print('')

unique_x_poisson.append(unique_x_poisson[k_poisson - 1] + 1)
plt.hist(x_poisson, bins=unique_x_poisson, ec='#666633',
        facecolor="#99ff33")
plt.title('Poisson generator, $\lambda = 0.7$')
plt.show()

# POISSON SAMPLE, LAMBDA = 1
l = 1
poisson_gen = poisson_generator(l, linear_congruential_generator(x0, alpha0,
                                                           0, m))
x_poisson = [next(poisson_gen) for _ in range(1000)]
# print('\n'.join(map(str, x_poisson)))

unique_x_poisson = sorted(list(Counter(x_poisson).keys()))
```

```
# Колво- степенейсвободыдля ( 10 варианта 6 - 1 = 5 степенейсвободы)
k_poisson = len(unique_x_poisson)
critical_value_poisson = 11.07
hi_squa_test2 = hi_squared_test(x_poisson,
                                partial(poisson_distribution_func, 1),
                                critical_value_poisson)
print('Poisson generator, lambda = 1:')
print('Hi Squared Pirson criteria: ' + str(hi_squa_test2[1]) + ' <= '
      + str(critical_value_poisson) if hi_squa_test2[0] else
      'Zero hypothesis fails by Hi Squared Pirson criteria.')

theoretical_expectation = 1
empirical_dispersion = empirical_dispersion_func(x_poisson)
theoretical_dispersion = 1
empirical_expectation = empirical_expectation_func(x_poisson)
print('theoretical expectation: ', theoretical_expectation)
print('empirical expectation: ', empirical_expectation)
print('theoretical dispersion: ', theoretical_dispersion)
print('empirical dispersion: ', empirical_dispersion)
print('')

unique_x_poisson.append(unique_x_poisson[k_poisson - 1] + 1)
plt.hist(x_poisson, bins=unique_x_poisson, ec='#666633',
         facecolor="#99ff33")
plt.title('Poisson generator, $\lambda = 1$')
plt.show()

# GEOMETRIC SAMPLE.
p = 0.2
geometric_gen = geometric_generator(p, linear_congruential_generator(x0, alpha0,
                                                                     0, m))
x_geometric = [next(geometric_gen) for _ in range(1000)]
# print('\n'.join(map(str, x_geometric)))

unique_x_geometric = sorted(list(Counter(x_geometric).keys()))
# Колво- степенейсвободыдля ( 10 варианта 27 - 1 = 26 степенейсвободы)
k_geometric = len(unique_x_geometric)
critical_value_geometric = 38.89
hi_squa_test3 = hi_squared_test(x_geometric,
                                partial(geometric_distribution_func, p,
                                        unique_x_geometric),
                                critical_value_geometric)
print('Geometric generator, p = 1:')
print('Hi Squared Pirson criteria: ' + str(hi_squa_test3[1]) + ' <= '
      + str(critical_value_geometric) if hi_squa_test3[0] else
      'Zero hypothesis fails by Hi Squared Pirson criteria.')
theoretical_expectation = 1 / p
empirical_dispersion = empirical_dispersion_func(x_geometric)
theoretical_dispersion = (1 - p) / p ** 2
empirical_expectation = empirical_expectation_func(x_geometric)
print('theoretical expectation: ', theoretical_expectation)
print('empirical expectation: ', empirical_expectation)
print('theoretical dispersion: ', theoretical_dispersion)
print('empirical dispersion: ', empirical_dispersion)
print('')
```

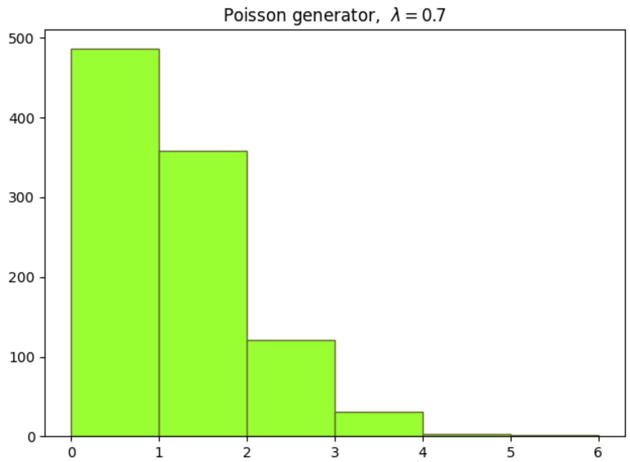
```

unique_x_geometric.append(unique_x_geometric[k_geometric - 1] + 1)
plt.hist(x_geometric, bins=sorted(list(unique_x_geometric)), ec='#666633',
        facecolor="#99ff33")
plt.title('Geometric generator, $p = 0.2$')
plt.show()

# BERNOULLI SAMPLE.
p = 0.75
bernoulli_gen = bernoulli_generator(p, linear_congruential_generator(x0, alpha0,
                                                                    0, m))
x_bernoulli = [next(bernoulli_gen) for _ in range(1000)]
# print('\n'.join(map(str, x_bernoulli)))

critical_x_bernoulli = 10
unique_x_bernoulli = sorted(list(Counter(x_bernoulli).keys()))
# Колво- степенейсвободыдля ( 10 варианта 2 - 1 = 1 степенейсвободы)
k_bernoulli = len(unique_x_bernoulli)
critical_value_bernoulli = 3.841
hi_squa_test4 = hi_squared_test(x_bernoulli,
                                partial(bernoulli_distribution_func, p),
                                critical_value_bernoulli)
print('Geometric generator, p = 1:')
print('Hi Squared Pirson criteria: ' + str(hi_squa_test4[1]) + ' <= '
      + str(critical_value_bernoulli) if hi_squa_test4[0] else
      'Zero hypothesis fails by Hi Squared Pirson criteria.')

theoretical_expectation = p
empirical_dispersion = empirical_dispersion_func(x_poisson)
theoretical_dispersion = p * (1 - p)
empirical_expectation = empirical_expectation_func(x_poisson)
print('theoretical expectation: ', theoretical_expectation)
print('empirical expectation: ', empirical_expectation)
print('theoretical dispersion: ', theoretical_dispersion)
print('empirical dispersion: ', empirical_dispersion)

unique_x_bernoulli.append(unique_x_bernoulli[k_bernoulli - 1] + 1)
plt.hist(x_bernoulli, bins=unique_x_bernoulli, ec='#666633',
        facecolor="#99ff33")
plt.title('Bernoulli generator, $p = 0.75$')
plt.show()
```

2.4 Результат выполнения

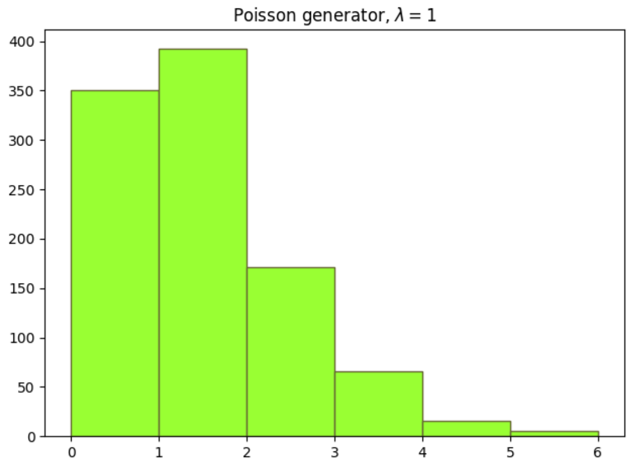
```
Poisson generator, lambda = 0.7:
Hi Squared Pirson criteria: 1.4769448422208398 <= 11.07
theoretical expectation:  0.7
empirical expectation:  0.71
theoretical dispersion:  0.7
empirical dispersion:  -0.31010000000000315

Poisson generator, lambda = 1:
Hi Squared Pirson criteria: 3.8721949509116405 <= 11.07
theoretical expectation:  1
empirical expectation:  1.021
theoretical dispersion:  1
empirical dispersion:  0.008559000000012418
```

```
Geometric generator, p = 1:
Hi Squared Pirson criteria: 20.886693627653305 <= 38.89
theoretical expectation:  5.0
empirical expectation:  4.121
theoretical dispersion:  19.999999999999996
empirical dispersion:  20.328358999999953

Geometric generator, p = 1:
Hi Squared Pirson criteria: 0.08533333333333333 <= 3.841
theoretical expectation:  0.75
empirical expectation:  1.021
theoretical dispersion:  0.1875
empirical dispersion:  0.008559000000012418
```

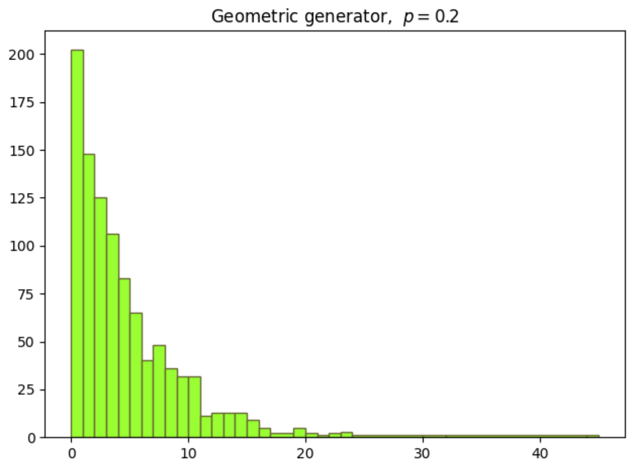
Рис. 6: Результат выполнения программы: проверка критерием согласия Пирсона и подсчёт несмещённых оценок математического ожидания и дисперсии.



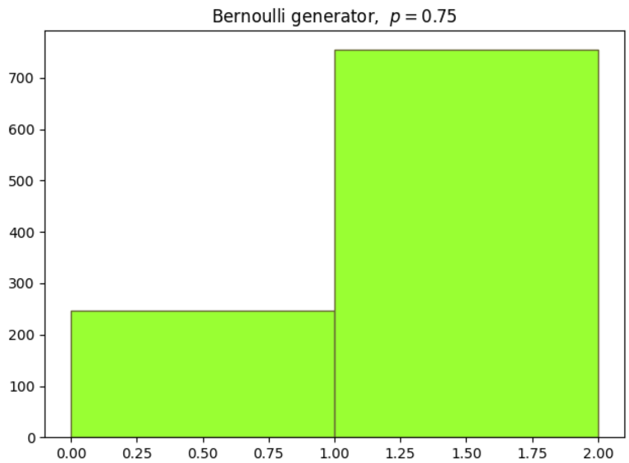
(a) Диаграмма выборки, полученной генератором распределения Пуассона при $\lambda = 0.7$.



(b) Диаграмма выборки, полученной генератором распределения Пуассона при $\lambda = 1$.



(c) Диаграмма выборки, полученной генератором геометрического распределения.



(d) Диаграмма выборки, полученной генератором распределения Бернулли.

3 Лабораторная 3

3.1 Условие

Согласно варианту 10:

1. Нормальное $N(m, s^2), m = 1, s^2 = 9$; Логонормальное $LN(m, s^2), m = 1, s^2 = 9$; Экспоненциальное $E(\alpha), \alpha = 2$;

2. Нормальное $N(m, s^2), m = 0, s^2 = 1$; Лапласа $L(\alpha), \alpha = 0.5$; Вейбула $W(a, b), a = 1, b = 0.5$;

Смоделировать непрерывную случайную величину. Исследовать точность моделирования.

1. Осуществить моделирование $n = 1000$ реализаций СВ из нормального закона распределения $N(m, s^2)$ с заданными параметрами. Вычислить несмещённые оценки математического ожидания и дисперсии, сравнить их с истинными;

2. Смоделировать $n = 1000$ СВ из заданных абсолютно непрерывных распределений. Вычислить несмещённые оценки математического ожидания и дисперсии, сравнить их с истинными значениями (если это возможно);

3. Для каждой из случайных величин построить свой критерий Колмогорова с уровнем значимости $\varepsilon = 0.05$. Проверить, что вероятность ошибки I рода стремится к 0.05;

4. Для каждой из случайных величин построить свой χ^2-критерий Пирсона с уровнем значимости $\varepsilon = 0.05$. Проверить, что вероятность ошибки I рода стремится к 0.05;

5. Осуществить проверку каждой из сгенерированных выборок каждым из построенных критериев.

3.2 Теория

3.2.1 Датчик случайной величины нормального распределения

Нормальное распределение - распределение вероятностей, которое в одномерном случае задаётся функцией плотности вероятности, совпадающей с функцией Гаусса.

Функция распределения:

$$\frac{1}{2}(1 + erf\left[\frac{x - \mu}{\sqrt{2\sigma^2}}\right]. \tag{3.1}$$

Функция плотности:

$$\frac{1}{\sigma\sqrt{2\pi}} \cdot \exp\left[\frac{(x - \mu)^2}{2\sigma^2}\right]. \tag{3.2}$$

Математическое ожидание: μ.

Дисперсия: σ^2.

Алгоритм моделирования:

При моделировании будем использовать **преобразование Бокса-Мюллера**. Пусть r b ϕ - независимые случайные величины, равномерно распределённые на интервале $(0, 1]$. Вычислим z_0, z_1 по формулам:

$$\begin{aligned} z_0 &= \cos(2\pi\phi)\sqrt{-2\ln(r)}, \\ z_1 &= \sin(2\pi\phi)\sqrt{-2\ln(r)}. \end{aligned} \tag{3.3}$$

Тогда z_0, z_1 будут независимы и распределены нормально с математическим ожиданием 0 и дисперсией 1.

3.2.2 Датчик случайной величины логнормального распределения

Логнормальное распределение - это двухпараметрическое семейство абсолютно непрерывных распределений. Если СВ имеет логнормальное распределение, то её логарифм имеет нормальное распределение.

Функция распределения:

$$\frac{1}{2} + \frac{1}{2}Erf\left[\frac{\ln(x) - \mu}{\sigma\sqrt{2}}\right]. \tag{3.4}$$

Функция плотности:

$$\exp\left[\frac{\left(\frac{\ln(x)-\mu}{\sigma}\right)^2}{2}\right] / \left(x\sigma\sqrt{2\pi}\right). \tag{3.5}$$

Математическое ожидание: $e^{\mu+\sigma^2/2}$.

Дисперсия: $(e^{\sigma^2} - 1) \cdot e^{2\mu+\sigma^2}$.

Алгоритм моделирования:

Для моделирования обычно используется связь с нормальным распределением. Поэтому, достаточно сгенерировать нормально распределённую СВ, например, используя **преобразование Бокса-Мюллера**, и вычислить её экспонент.

3.2.3 Датчик случайной величины экспоненциального распределения

Экспоненциальное распределение - абсолютно непрерывное распределение, моделирующее время между двумя последовательными свершениями одного и того же события.

Функция распределения:

$$1 - e^{\lambda x}. \tag{3.6}$$

Функция плотности:

$$\lambda e^{-\lambda x}. \tag{3.7}$$

Математическое ожидание:

$$\lambda^{-1}. \tag{3.8}$$

Дисперсия:

$$\lambda^{-2}. \tag{3.9}$$

Алгоритм моделирования:

1. Моделирование реализации α БСВ;
2. Вычисление экспоненциально распределённой СВ:

$$x = -\frac{1}{\lambda}\ln(\alpha) \tag{3.10}$$

3.2.4 Датчик случайной величины распределения Лапласа

Распределение Лапласа - в теории вероятностей это непрерывное распределение случайной величины, при котором плотность вероятности есть $f(x) = \frac{\alpha}{2} \cdot e^{-\alpha|x-\beta|}$, $-\infty < x < +\infty$, где $\alpha > 0$ - параметр масштаба, $-\infty < \beta < +\infty$ - параметр сдвига.

Функция распределения:

$$\begin{cases} \frac{1}{2}e^{\alpha(x-\beta)} \\ 1 - \frac{1}{2}e^{\alpha(x-\beta)} \end{cases}. \tag{3.11}$$

Функция плотности:

$$\frac{\alpha}{2}e^{-\alpha|x-\beta|}. \tag{3.12}$$

Математическое ожидание:

$$\beta. \tag{3.13}$$

Дисперсия:

$$\frac{2}{\alpha^2}. \tag{3.14}$$

Алгоритм моделирования:

Алгоритм моделирования $\xi \sim L(\lambda)$ основан на методе обратной функции. Обратная для функции распределения $F_\xi(x)$ функция имеет вид:

$$\begin{cases} x = F_\xi^{-1}(y) = \frac{1}{\lambda}\ln(2y) < 0, y \in [0, 0.5) \\ x = F_\xi^{-1}(y) = \frac{1}{\lambda}\ln(2(1-y)) < 0, y \in [0.5, 1) \end{cases} \tag{3.15}$$

Для моделирования реализация x СВ $\xi \sim L(\lambda)$ выполняются следующие действия:

1. Моделирование реализации α БСВ;
2. Принимается решение о том, что реализацией СВ ξ является величина x, вычисляемая по формулам (3.15) согласно отрезку, которому принадлежит y.

3.2.5 Датчик случайной величины распределения Вейбула

Распределение Вейбула - в теории вероятностей это двухпараметрическое семейство абсолютно непрерывных распределений.

Функция распределения:

$$1 - e^{-(x/\lambda)^k}. \tag{3.16}$$

Функция плотности:

$$\frac{k}{\lambda} \cdot \left(\frac{x}{\lambda}\right)^{k-1} \cdot e^{-(x/\lambda)^k}. \tag{3.17}$$

Математическое ожидание:

$$\lambda\Gamma\left(1 + \frac{1}{k}\right). \tag{3.18}$$

Дисперсия:

$$\lambda^2\Gamma\left(1 + \frac{2}{k}\right) - \mu^2. \tag{3.19}$$

Алгоритм моделирования:

Алгоритм моделирования $\xi \sim WG(\lambda, k)$ основан на методе обратной функции. Обратная для функции распределения $F_\xi(x)$ функция имеет вид:

$$x = F_\xi^{-1}(y) = \left(-\frac{1}{\lambda}\ln(y)\right)^{1/k}. \tag{3.20}$$

Для моделирования реализация x СВ $\xi \sim L(\lambda)$ выполняются следующие действия:

1. Моделирование реализации α БСВ;
2. Принимается решение о том, что реализацией СВ ξ является величина x, вычисляемая по формулам (3.20), где $y = \alpha$.

3.3 Код программы

```
import math
import random
from functools import partial

import matplotlib.pyplot as plt
from scipy.stats import norm, chi2, kstwobign, lognorm, expon, laplace, \
    weibull_min


def linear_congruential_generator(x, alpha, c, m):
    while True:
        x = (alpha * x + c) % m
        yield x / m


def normal_generator(mu, sigma, linear_gen):
    while True:
        u1 = next(linear_gen)
        u2 = next(linear_gen)
        z0 = math.sqrt(-2.0 * math.log(u1)) * math.cos(2 * math.pi * u2)
        # z1 = math.sqrt(-2.0 * math.log(u1)) * math.sin(2 * math.pi * u2)
        yield mu + z0 * sigma


def exponential_generator(l, linear_gen):
    while True:
        yield -1 / l * math.log(next(linear_gen))


def lognormal_generator(mu, sigma, linear_gen):
    normal_gen = normal_generator(mu, sigma, linear_gen)
    while True:
        yield math.exp(next(normal_gen))


def laplace_generator(alpha, linear_gen):
    while True:
        y = next(linear_gen)
        if 0 <= y < 0.5:
            yield 1 / alpha * math.log(2 * y)
        else:
            yield -1 / alpha * math.log(2 * (1 - y))


def weibull_generator(l, k, linear_gen):
    while True:
        yield l * ((-math.log(next(linear_gen))) ** (1 / k))


def hi_squared_test(frequencies, borders, distribution_func, p_value):
    exampling_size = sum(frequencies)
    hi_squared = 0
```

```
    for i in range(1, len(frequencies) + 1):
        empiric_freq = frequencies[i - 1]
        theoretic_freq = (distribution_func(borders[i]) - distribution_func(
            borders[i - 1])) * exampling_size
        if theoretic_freq:
            hi_squared += ((
                                     empiric_freq - theoretic_freq) ** 2) / theoretic_freq
    degrees_of_freedom = len(frequencies) - 1
    critical_value = chi2.ppf(1 - p_value, degrees_of_freedom)
    return hi_squared < critical_value, hi_squared, critical_value


def kolmogorov_test(values, distribution_func, p_value):
    values.sort()
    Dn = 0
    i = 0
    n = len(values)
    for value in values:
        i += 1
        theoretical_func_res = distribution_func(value)
        empirical_function_res = i / n
        Dn = max(Dn, abs(theoretical_func_res - empirical_function_res))
    Dn *= math.sqrt(n)
    critical_value = kstwobign.ppf(1 - p_value)
    return Dn < critical_value, Dn, critical_value


def empirical_expectation_func(values):
    return sum(values) / len(values)


def empirical_dispersion_func(values):
    expectation = empirical_expectation_func(values)
    result = 0
    for value in values:
        result += (value - expectation) ** 2
    return result / len(values) - 1


def cumulative_norm_distrib_func(mu, sigma, value):
    return norm.cdf(value, mu, sigma)


def cumulative_lognorm_distrib_func(mu, sigma, value):
    return lognorm.cdf(value, scale=math.exp(mu), s=sigma)


def cumulative_exponential_distrib_func(l, value):
    return expon.cdf(value, scale=1 / l)


def cumulative_laplace_distrib_func(alpha, betta, value):
    return laplace.cdf(value, scale=1 / alpha, loc=betta)
```

```python
def cumulative_weibull_distrib_func(l, k, value):
    return weibull_min.cdf(value, k, scale=l)


def built_in_random():
    while True:
        yield random.random()


x0 = 79507
alpha0 = 79507
c = 63
m = 2 ** 31
p_value = 0.05

generator = linear_congruential_generator(x0, alpha0, c, m)

# NORMAL SAMPLE, MU = 1, SIGMA^2 = 9
mu = 1
sigma = 3

normal_gen = normal_generator(mu, sigma, generator)
x_normal = [next(normal_gen) for _ in range(1000)]
# print('\n'.join(map(str, x_normal)))

freq_normal, borders_normal, _ = plt.hist(x_normal, bins='auto',
                                          ec='#666633',
                                          facecolor="#99ff33")
plt.title('Normal generator, $\mu = 1, \sigma^2=9$')
plt.show()

hi_squa_test1 = hi_squared_test(freq_normal,
                             borders_normal,
                             partial(cumulative_norm_distrib_func,
                                     mu, sigma),
                             p_value)
print('Normal generator, mu = 1, sigma^2 = 9:')
print('Hi Squared Pirson criteria: ' + str(hi_squa_test1[1]) + ' <= '
      + str(hi_squa_test1[2]) if hi_squa_test1[0] else
      'Zero hypothesis fails by Hi Squared Pirson criteria.')
kolm_test1 = kolmogorov_test(x_normal,
                          partial(cumulative_norm_distrib_func, mu, sigma),
                          p_value)
print('Kolmogorov criteria: ' + str(kolm_test1[1]) + ' <= '
      + str(kolm_test1[2]) if kolm_test1[0]
      else 'Zero hypothesis fails by Kolmogorov criteria.')

theoretical_expectation = mu
empirical_dispersion = empirical_dispersion_func(x_normal)
theoretical_dispersion = sigma ** 2
empirical_expectation = empirical_expectation_func(x_normal)
print('theoretical expectation: ', theoretical_expectation)
print('empirical expectation: ', empirical_expectation)
print('theoretical dispersion: ', theoretical_dispersion)
print('empirical dispersion: ', empirical_dispersion)
```

```
print('')

# NORMAL SAMPLE, MU = 0, SIGMA^2 = 1
mu = 0
sigma = 1

normal_gen = normal_generator(mu, sigma, generator)
x_normal = [next(normal_gen) for _ in range(1000)]
# print('\n'.join(map(str, x_normal)))

freq_normal, borders_normal, _ = plt.hist(x_normal, bins='auto',
                                          ec='#666633',
                                          facecolor="#99ff33")
plt.title('Normal generator, $\mu = 0, \sigma^2=1$')
plt.show()

hi_squa_test2 = hi_squared_test(freq_normal,
                                borders_normal,
                                partial(cumulative_norm_distrib_func,
                                        mu, sigma),
                                p_value)
print('Normal generator, mu = 0, sigma^2 = 1:')
print('Hi Squared Pirson criteria: ' + str(hi_squa_test2[1]) + ' <= '
      + str(hi_squa_test2[2]) if hi_squa_test2[0] else
      'Zero hypothesis fails by Hi Squared Pirson criteria.')
kolm_test2 = kolmogorov_test(x_normal,
                             partial(cumulative_norm_distrib_func, mu, sigma),
                             p_value)
print('Kolmogorov criteria: ' + str(kolm_test2[1]) + ' <= '
      + str(kolm_test2[2]) if kolm_test2[0]
      else 'Zero hypothesis fails by Kolmogorov criteria.')

theoretical_expectation = mu
empirical_dispersion = empirical_dispersion_func(x_normal)
theoretical_dispersion = sigma ** 2
empirical_expectation = empirical_expectation_func(x_normal)
print('theoretical expectation: ', theoretical_expectation)
print('empirical expectation: ', empirical_expectation)
print('theoretical dispersion: ', theoretical_dispersion)
print('empirical dispersion: ', empirical_dispersion)
print('')

# LOGNORMAL SAMPLE, MU = 1, SIGMA^2 = 9
mu = 1
sigma = 3

lognormal_gen = lognormal_generator(mu, sigma, generator)
x_lognormal = [next(lognormal_gen) for _ in range(1000)]
# print('\n'.join(map(str, x_lognormal)))

freq_lognormal, borders_lognormal, _ = plt.hist(x_lognormal, bins='auto',
                                                ec='#666633',
                                                facecolor="#99ff33")
plt.title('Lognormal generator, $\mu = 1, \sigma^2=9$')
plt.show()
```

```
hi_squa_test3 = hi_squared_test(freq_lognormal,
                                borders_lognormal,
                                partial(cumulative_norm_distrib_func,
                                        mu, sigma),
                                p_value)

print('Lognormal generator, mu = 1, sigma^2 = 9:')
print('Hi Squared Pirson criteria: ' + str(hi_squa_test3[1]) + ' <= '
      + str(hi_squa_test3[2]) if hi_squa_test3[0] else
      'Zero hypothesis fails by Hi Squared Pirson criteria.')
kolm_test3 = kolmogorov_test(x_lognormal,
                             partial(cumulative_lognorm_distrib_func, mu,
                                     sigma),
                             p_value)
print('Kolmogorov criteria: ' + str(kolm_test3[1]) + ' <= '
      + str(kolm_test3[2]) if kolm_test3[0]
      else 'Zero hypothesis fails by Kolmogorov criteria.')

theoretical_expectation = math.exp(mu + sigma ** 2 / 2)
empirical_dispersion = empirical_dispersion_func(x_lognormal)
theoretical_dispersion = (math.exp(sigma ** 2) - 1) * math.exp(
    2 * mu + sigma ** 2)
empirical_expectation = empirical_expectation_func(x_lognormal)
print('theoretical expectation: ', theoretical_expectation)
print('empirical expectation: ', empirical_expectation)
print('theoretical dispersion: ', theoretical_dispersion)
print('empirical dispersion: ', empirical_dispersion)
print('')

# EXPONENTIAL SAMPLE, LAMBDA = 2
l = 2

exponential_gen = exponential_generator(l, generator)
x_exponential = [next(exponential_gen) for _ in range(1000)]
# print('\n'.join(map(str, x_exponential)))

freq_exponential, borders_exponential, _ = plt.hist(x_exponential, bins='auto',
                                                    ec='#666633',
                                                    facecolor="#99ff33")
plt.title('Exponential generator, $\lambda= 2$')
plt.show()

hi_squa_test4 = hi_squared_test(freq_exponential,
                                borders_exponential,
                                partial(cumulative_exponential_distrib_func, l),
                                p_value)

print('Exponential generator, lambda = 2:')
print('Hi Squared Pirson criteria: ' + str(hi_squa_test4[1]) + ' <= '
      + str(hi_squa_test4[2]) if hi_squa_test4[0] else
      'Zero hypothesis fails by Hi Squared Pirson criteria.')
kolm_test4 = kolmogorov_test(x_exponential,
                             partial(cumulative_exponential_distrib_func, l),
                             p_value)
```

```python
print('Kolmogorov criteria: ' + str(kolm_test4[1]) + ' <= '
      + str(kolm_test4[2]) if kolm_test4[0]
      else 'Zero hypothesis fails by Kolmogorov criteria.')

theoretical_expectation = 1 / l
empirical_dispersion = empirical_dispersion_func(x_exponential)
theoretical_dispersion = 1 / l ** 2
empirical_expectation = empirical_expectation_func(x_exponential)
print('theoretical expectation: ', theoretical_expectation)
print('empirical expectation: ', empirical_expectation)
print('theoretical dispersion: ', theoretical_dispersion)
print('empirical dispersion: ', empirical_dispersion)
print('')

# LAPLACE SAMPLE, ALPHA = 0.5, BETTA = 0
alpha = 0.5
beta = 0

laplace_gen = laplace_generator(alpha, generator)
x_laplace = [next(laplace_gen) for _ in range(1000)]
# print('\n'.join(map(str, x_laplace)))

freq_laplace, borders_laplace, _ = plt.hist(x_laplace, bins='auto',
                                            ec='#666633',
                                            facecolor="#99ff33")
plt.title(r'Laplace generator, $\alpha = 0.5, \beta = 0$')
plt.show()

hi_squa_test5 = hi_squared_test(freq_laplace,
                                borders_laplace,
                                partial(cumulative_laplace_distrib_func, alpha,
                                        beta),
                                p_value)

print('Laplace generator, alpha = 0.5, betta = 0:')
print('Hi Squared Pirson criteria: ' + str(hi_squa_test5[1]) + ' <= '
      + str(hi_squa_test5[2]) if hi_squa_test5[0] else
      'Zero hypothesis fails by Hi Squared Pirson criteria.')
kolm_test5 = kolmogorov_test(x_laplace,
                             partial(cumulative_laplace_distrib_func, alpha,
                                     beta),
                             p_value)
print('Kolmogorov criteria: ' + str(kolm_test5[1]) + ' <= '
      + str(kolm_test5[2]) if kolm_test5[0]
      else 'Zero hypothesis fails by Kolmogorov criteria.')

theoretical_expectation = beta
empirical_dispersion = empirical_dispersion_func(x_laplace)
theoretical_dispersion = 2 / alpha ** 2
empirical_expectation = empirical_expectation_func(x_laplace)
print('theoretical expectation: ', theoretical_expectation)
print('empirical expectation: ', empirical_expectation)
print('theoretical dispersion: ', theoretical_dispersion)
print('empirical dispersion: ', empirical_dispersion)
print('')
```

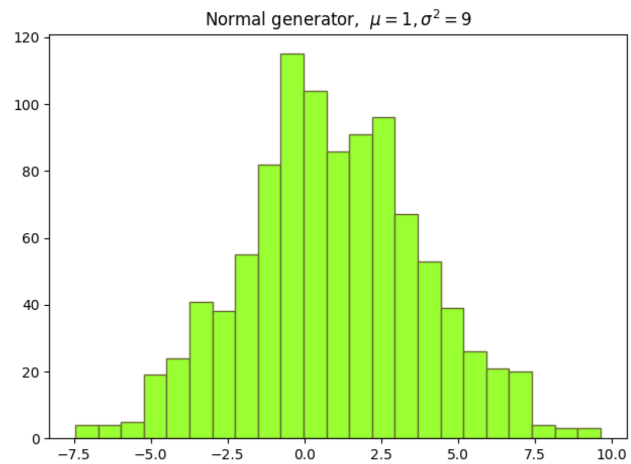
```python

# WEIBULL SAMPLE, LAMBDA = 1, K = 0.5
l = 1
k = 0.5

weibull_gen = weibull_generator(l, k, generator)
x_weibull = [next(weibull_gen) for _ in range(1000)]
# print('\n'.join(map(str, x_weibull)))

freq_weibull, borders_weibull, _ = plt.hist(x_weibull, bins='auto',
                                            ec='#666633',
                                            facecolor="#99ff33")
plt.title(r'Weibull generator, $\lambda = 1, k = 0.5$')
plt.show()

hi_squa_test = hi_squared_test(freq_weibull,
                               borders_weibull,
                               partial(cumulative_weibull_distrib_func, l, k),
                               p_value)

print('Weibull generator, lambda = 1, k = 0.5:')
print('Hi Squared Pirson criteria: ' + str(hi_squa_test[1]) + ' <= '
      + str(hi_squa_test[2]) if hi_squa_test[0] else
      'Zero hypothesis fails by Hi Squared Pirson criteria.')
kolm_test = kolmogorov_test(x_weibull,
                            partial(cumulative_weibull_distrib_func, l, k),
                            p_value)
print('Kolmogorov criteria: ' + str(kolm_test[1]) + ' <= '
      + str(kolm_test[2]) if kolm_test[0]
      else 'Zero hypothesis fails by Kolmogorov criteria.')

theoretical_expectation = weibull_min.mean(k, scale=l)
empirical_dispersion = empirical_dispersion_func(x_weibull)
theoretical_dispersion = weibull_min.var(k, scale=l)
empirical_expectation = empirical_expectation_func(x_weibull)
print('theoretical expectation: ', theoretical_expectation)
print('empirical expectation: ', empirical_expectation)
print('theoretical dispersion: ', theoretical_dispersion)
print('empirical dispersion: ', empirical_dispersion)
print('')
```

3.4 Результат выполнения

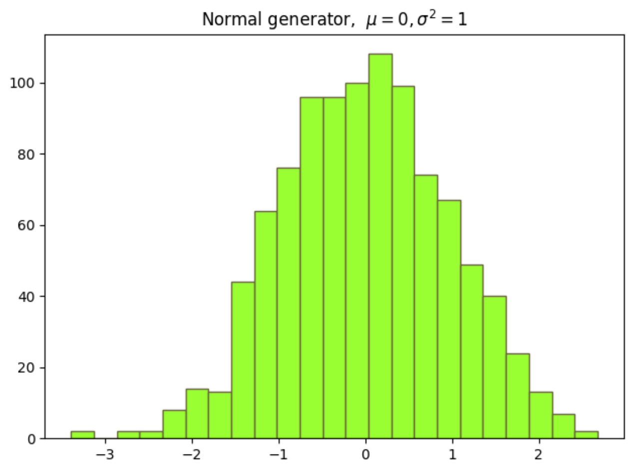
```
Normal generator, mu = 1, sigma^2 = 9:
Hi Squared Pirson criteria: 28.358119998727172 <= 33.92443847144381
Kolmogorov criteria: 0.9679439050529556 <= 1.3580986393225505
theoretical expectation:  1
empirical expectation:  0.9294316472943027
theoretical dispersion:  9
empirical dispersion:  7.4959786131342

Normal generator, mu = 0, sigma^2 = 1:
Hi Squared Pirson criteria: 21.488365965347455 <= 33.92443847144381
Kolmogorov criteria: 0.729460343122114 <= 1.3580986393225505
theoretical expectation:  0
empirical expectation:  -0.0021933952374124295
theoretical dispersion:  1
empirical dispersion:  -0.0879810652662405

Lognormal generator, mu = 1, sigma^2 = 9:
Zero hypothesis fails by Hi Squared Pirson criteria.
Kolmogorov criteria: 0.45996235601524604 <= 1.3580986393225505
theoretical expectation:  244.69193226422038
empirical expectation:  214.6402940382709
theoretical dispersion:  485105321.26807505
empirical dispersion:  6708849.993968649
```

```
Exponential generator, lambda = 2:
Zero hypothesis fails by Hi Squared Pirson criteria.
Kolmogorov criteria: 0.865617143544657 <= 1.3580986393225505
theoretical expectation:  0.5
empirical expectation:  0.4964390006492622
theoretical dispersion:  0.25
empirical dispersion:  -0.7272060875253951

Laplace generator, alpha = 0.5, betta = 0:
Hi Squared Pirson criteria: 42.81270564557594 <= 70.99345283378227
Kolmogorov criteria: 0.8620742035777464 <= 1.3580986393225505
theoretical expectation:  0
empirical expectation:  0.012242333617161668
theoretical dispersion:  8.0
empirical dispersion:  6.039302085513919

Weibull generator, lambda = 1, k = 0.5:
Hi Squared Pirson criteria: 118.4822098553722 <= 118.75161175336736
Kolmogorov criteria: 0.4340890407142095 <= 1.3580986393225505
theoretical expectation:  2.0
empirical expectation:  1.9964551645252555
theoretical dispersion:  20.0
empirical dispersion:  17.104799371406074
```

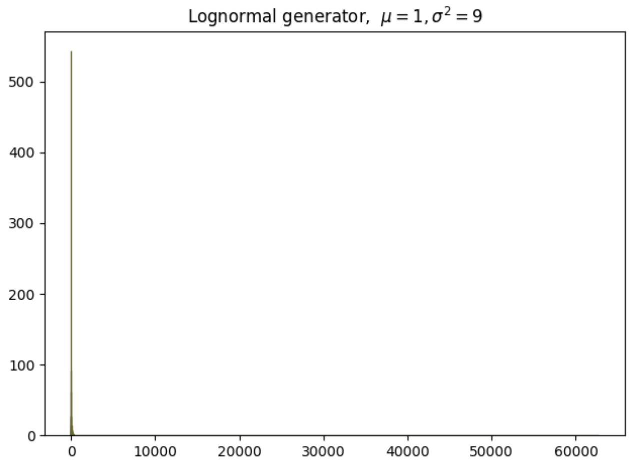
Рис. 8: Результат выполнения программы: проверка критерием согласия Пирсона и Колмогорова. Вывод теоретических и подсчёт эмпирических математических ожиданий и дисперсийё.



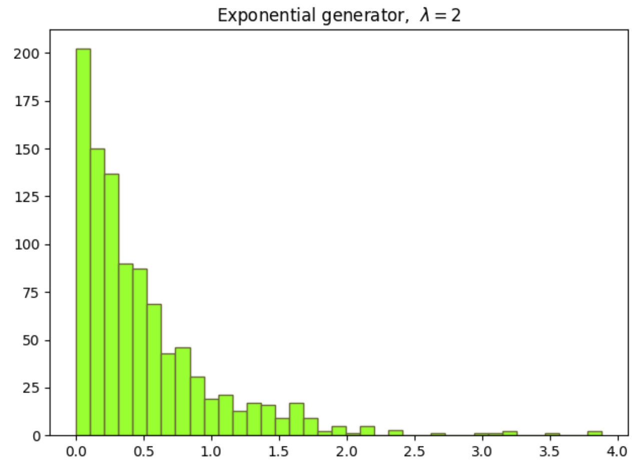
(a) Диаграмма выборки, полученной генератором нормального распределения при $\mu = 1, \sigma^2 = 9$.



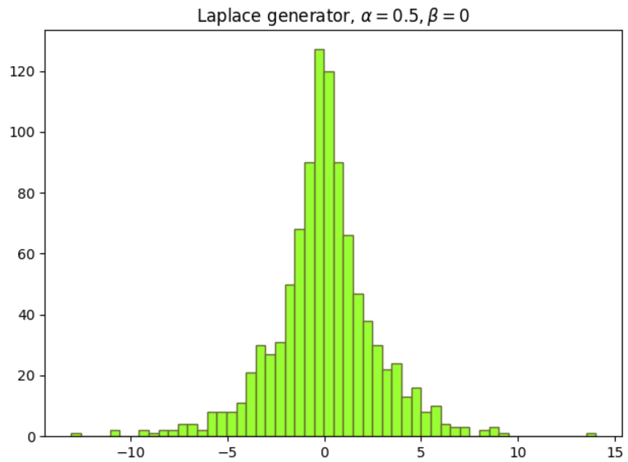
(b) Диаграмма выборки, полученной генератором нормального распределения при $\mu = 0, \sigma^2 = 1$.



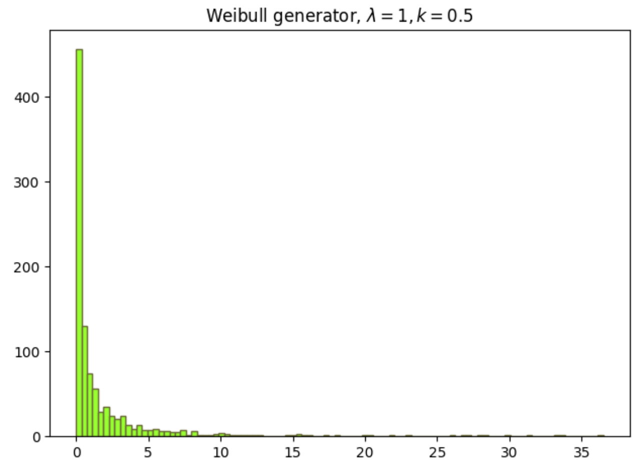
(a) Диаграмма выборки, полученной генератором логнормального распределения при $\mu = 1, \sigma^2 = 9$.



(b) Диаграмма выборки, полученной генератором экспоненциального распределения при $\lambda = 2$.



(c) Диаграмма выборки, полученной генератором распределения Лапласа при $\alpha = 0.5, \beta = 0$.



(d) Диаграмма выборки, полученной генератором распределения Вейбула при $\lambda = 1, k = 0.5$.

4 Лабораторная 4

4.1 Условие

Согласно варианту 10:

1. $I_1 = \int\limits_{-\infty}^{\infty} e^{-x^4\sqrt{1+x^4}} dx$;

2. $I_2 = \iint\limits_{1\leqslant x^2+y^2\leqslant 4} \frac{dxdy}{x^2+y^2}$.

Вычислить значение интеграла, используя метод Монте-Карло. Оценить точность.

1. По методу Монте-Карло вычислить приближённое значение интегралов;

2. Сравнить полученное значение либо с точным значением (если его получится вычислить), либо с приближённым, полученным в каком-либо математическом пакете (например, в mathematica). Для этого построить график зависимости точности вычисленного методом Монте-Карло интеграла от числа итераций n.

4.2 Теория

4.2.1 Метод Монте-Карло для вычисления интегралов

В основе метода лежит нахождение такой случайно величины ξ, математическое ожидание которой совпадает с искомым интегралом:

$$\int\limits_a^b f(x)dx = E(\xi) = \int\limits_a^b x\rho_\xi(x)dx. \tag{4.1}$$

Для этого выбирается такая СВ ξ_1 с плотностью $\rho_{\xi_1}(x)$, определённая на той же области, что и интеграл, тогда ξ определяется, как

$$\xi = g(\xi_1) = \frac{f(\xi_1)}{\rho_{\xi_1}(\xi_1)}. \tag{4.2}$$

Тогда

$$E(\xi) = E(g(\xi_1)) = \int\limits_a^b g(x)\rho_{\xi_1}(x)dx = \int\limits_a^b \frac{f(x)}{\rho_{\xi_1}(x)}\rho_{\xi_1}(x)dx = \int\limits_a^b f(x)dx. \tag{4.3}$$

Для нахождения математического ожидания необходимо смоделировать n реализаций x_i СВ ξ:

$$E(\xi) \approx \frac{1}{n}\sum_{i=0}^{n} x_i. \tag{4.4}$$

Алгоритм моделирования:

1. Для вычисления I_1 использовалось нормальное распределение $N(0, 1)$, область которого совпадает с областью интегрирования;

2. При вычислении I_2 производился переход к полярной системе координат и использовать равномерное распределение на отрезке [0, 2].

4.3 Код программы

```
import math

import matplotlib.pyplot as plt


def linear_congruential_generator(x, alpha, c, m):
    while True:
        x = (alpha * x + c) % m
        yield x / m


def normal_generator(mu, sigma, linear_gen):
    while True:
        u1 = next(linear_gen)
        u2 = next(linear_gen)
        z0 = math.sqrt(-2.0 * math.log(u1)) * math.cos(2 * math.pi * u2)
        z1 = math.sqrt(-2.0 * math.log(u1)) * math.sin(2 * math.pi * u2)
        yield mu + z0 * sigma, mu + z1 * sigma


def integral_function(x):
    return (math.exp(-x ** 4) * math.sqrt(1 + x ** 4))


def double_integral_function(x, y):
    return (2 * math.pi) / ((1 + x) * (math.cos(2 * math.pi * y) ** 2 + (
            1 + x) ** 2 * math.sin(2 * math.pi * y) ** 4))


def cumulative_norm_distrib_func(value, mu, sigma):
    return 1 / (sigma * math.sqrt(2 * math.pi)) * math.exp(
        - (value - mu) ** 2 / (2 * sigma ** 2))


def calc_integral(x_sample, from_num, to_num):
    return sum(
        integral_function(x) / cumulative_norm_distrib_func(x, mu, sigma) for x
        in x_sample[from_num:to_num]) / (to_num - from_num)


def calc_double_integral(xy_sable, from_num, to_num):
    return sum(
        double_integral_function(x, y) for x, y in
```

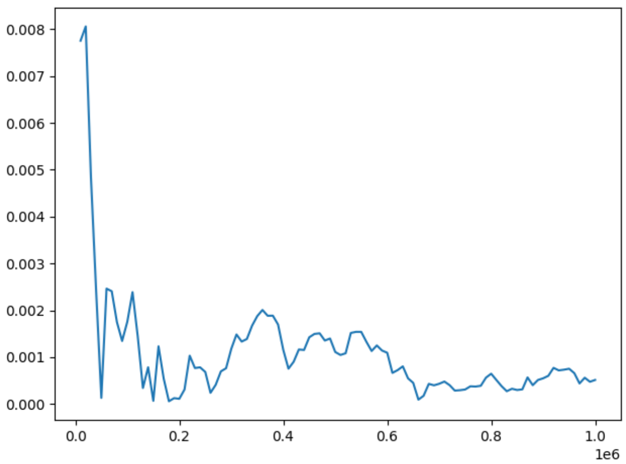
```
        xy_sample_in_area[from_num: to_num]) / (to_num - from_num)


x0 = 79507
alpha0 = 79507
c = 63
m = 2 ** 31

mu = 0
sigma = 1

linear_gen = linear_congruential_generator(x0, alpha0, c, m)
normal_gen = normal_generator(mu, sigma, linear_gen)

# Task 1
exact_result = 2.000057
exampling_size = 1000000
x_sample = [next(normal_gen)[0] for _ in range(exampling_size)]

step = exampling_size // 100
steps = []
results = []

sum_res = 0
for size in range(step, exampling_size + 1, step):
    sum_res += calc_integral(x_sample, size - step, size)
    results.append(sum_res)
    steps.append(size)

results = [x / (i + 1) for x, i in zip(results, range(0, len(results)))]

discrepancy = [abs(x - exact_result) for x in results]
plt.plot(steps, discrepancy)
plt.show()
print("Task 1: " + str(results[len(results) - 1]))

# Task2
exact_result = 3.8579
exampling_size = 1000000
x_from = 0
x_to = 2 * math.pi
y_from = 1
y_to = 2

xy_sample = [next(normal_gen) for _ in range(exampling_size)]
xy_sample_in_area = list(
    filter(lambda xy: 0 < xy[0] < 1 and 0 < xy[1] < 1, xy_sample))

step = len(xy_sample_in_area) // 100 # 10 segments
steps = []
results = []

sum_res = 0
for size in range(step, len(xy_sample_in_area) + 1, step):
    sum_res += calc_double_integral(xy_sample_in_area, size - step, size)
```

```python
    results.append(sum_res)
    steps.append(size)

results = [x / (i + 1) for x, i in zip(results, range(0, len(results)))]

discrepancy = [abs(x - exact_result) for x in results]
plt.plot(steps, discrepancy)
plt.show()
print("Task 2: " + str(results[len(results) - 1]))
```

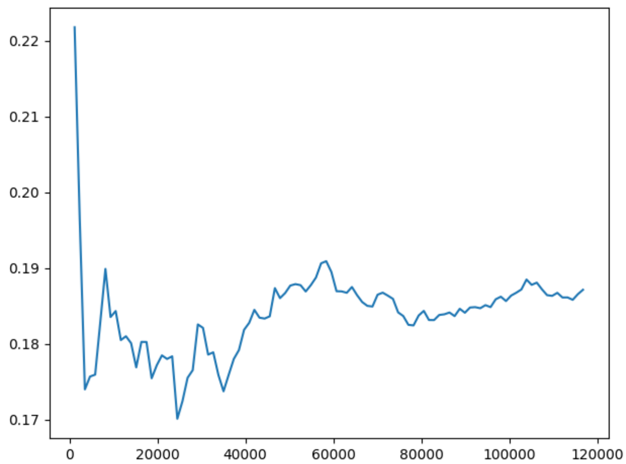
4.4 Результат выполнения

```
Task 1: 2.0005711732875366
Task 2: 4.045023607968606
```

Рис. 11: Результат выполнения программы: значения интегралов I_1, I_2 соответственно.



(a) График зависимости точности вычисленного методом Монте-Карло интеграла I_1 от числа итераций n.



(b) График зависимости точности вычисленного методом Монте-Карло интеграла I_2 от числа итераций n.

5 Лабораторная 5

5.1 Условие

Согласно варианту 10:

$$A = \begin{pmatrix} 1.2 & 0.1 & -0.3 \\ -0.3 & 0.9 & -0.2 \\ 0.4 & 0.5 & 1.0 \end{pmatrix}, f = \begin{pmatrix} 2 \\ 3 \\ 3 \end{pmatrix}. \tag{5.1}$$

Решить систему линейных уравнений, используя метод Монте-Карло.

1. Решить систему линейных алгебраических уравнений $Ax = f$ методом Монте-Карло;
2. Сравнить с решением данного уравнения, полученным в произвольном математическом пакете;
3. Построить график зависимости точности решения от длины цепи Маркова и числа смоделированных цепей Маркова.

5.2 Теория

5.2.1 Метод Монте-Карло для решения СЛАУ

Для необходимо привести СЛАУ к виду

$$x = Ax + f. \tag{5.2}$$

Предположим, что наибольшее по модулю характеристическое число матрицы A меньше единицы, так что сходиться метод последовательных приближений:

$$x^{(k)} = Ax^{(k-1)} + f, k = 1, 2, \ldots \tag{5.3}$$

Достаточным условием для того, чтобы все характеристические числа матрицы A лежали внутри единичного круга на комплексной плоскости, то есть $|\lambda_i| < 1, i = \overline{1, n}$ может служить одно неравенств:

$$\begin{gathered} \sum_{i,j=1}^{n} a_{ij}^2 < 1 \\ \max_{1 \leqslant i \leqslant n} \sum_{j=1}^{m} |a_{ij}| < 1 \end{gathered} \tag{5.4}$$

Пусть размерность вектора x равна n. Для нахождения компоненты x_i вектора x определим вектор:

$$h = \begin{cases} h_j = 0, j \neq i \\ h_i = 1, j = \overline{1, n} \end{cases} . \tag{5.5}$$

Моделирование цепи Маркова выглядит следующим образом:

$$i_0 \to i_1 \to \ldots \to i_{N-1}, i_k \in \overline{1,n}. \tag{5.6}$$

Вектор вероятностей начальных состояний цепи Маркова:

$$\pi = \left\{\pi_i = \frac{1}{n}, i = \overline{1,n}\right\}. \tag{5.7}$$

Матрица переходных вероятностей имеет вид:

$$P = \left\{P_{ij} = \frac{1}{n}, i,j = \overline{1,n}\right\}. \tag{5.8}$$

Каждому состоянию цепи Маркова приписываем веса, которые вычисляются по формулам:

$$\begin{aligned} &Q_{i_0} = g_{i_0} = \begin{cases} \frac{h_{i_0}}{\pi_{i_0}}, \pi_{i_0} > 0, \\ 0, \pi_{i_0} = 0 \end{cases} \\ &\ldots \\ &Q_{i_k} = Q_{i_{k-1}} g_{i_{k-1}} = \begin{cases} \frac{a_{i_{k-1}i_k}}{p_{i_{k-1}i_k}}, p_{i_{k-1}i_k} > 0, \\ 0, p_{i_{k-1}i_k} = 0 \end{cases} \end{aligned}. \tag{5.9}$$

Алгоритм моделирования:

Моделировать СВ ξ_N будем по формуле:

$$\xi_N^{(l)} = \sum_{n=0}^{N} = Q_{i_n} f_{i_n} \tag{5.10}$$

где $l = \overline{1,L}$ - номер реализации цепи Маркова. Тогда приближённое решение вычисляется по формуле:

$$x_i \approx \frac{1}{L} \sum_{l=1}^{L} \xi_N^{(l)}, i = \overline{1,n}. \tag{5.11}$$

5.3 Код программы

```
import math
import random

import matplotlib.pyplot as plt
import numpy as np

# ДлинацепиМаркова
min_chain_length = 100
max_chain_length = 1000
# Шаг
length_step = 100
# КоличествореализацийцепиМаркова
```

```python
min_chain_count = 1000
max_chain_count = 10000
# Шаг
count_step = 1000

# Исходнаяматрица
g_A_real = ((1.2, 0.1, -0.3), (-0.3, 0.9, -0.2), (0.4, 0.5, 1),)
# Преобразованнаяматрица
g_A = ((-0.2, -0.1, 0.3), (0.3, 0.1, 0.2), (-0.4, -0.5, 0),)
# Праваячастьсистемы
g_f = (2, 3, 3)


def built_in_random():
    while True:
        yield random.random()


generator = built_in_random()


def solve(A, f, chain_length, chain_count):
    # Размерностьсистемы
    n = len(f)
    # Решениесистемы
    X = [0.0] * n

    # Векторнач . вероятностейцепиМаркова
    pi = [1 / n] * n
    # МатрицапереходныхсостоянийцепиМаркова
    P = [[1 / n] * n] * n

    # ВесасостоянийцепиМаркова
    Q = [0.0] * (chain_length + 1)

    # СВ
    ksi = [0.0] * chain_count
    # БСВ
    alpha = 0

    for k in range(n):
        h = [0.0] * n
        h[k] = 1

        for j in range(chain_count):
            chain = [math.floor(next(generator) * 3) for _ in
                     range(chain_length + 1)]
            Q[0] = h[chain[0]] / pi[chain[0]]
            for i in range(1, chain_length + 1):
                Q[i] = Q[i - 1] * A[chain[i - 1]][chain[i]] / P[chain[i - 1]][
                    chain[i]]
            ksi[j] = sum(q * f[state] for q, state in zip(Q, chain))

        X[k] = sum(ksi) / chain_count

```

```
    return X


X_real = np.linalg.solve(np.array(g_A_real), np.array(g_f))

R = np.array([[np.linalg.norm(np.array(solve(g_A, g_f, length, count)) - X_real)
               for length in
               range(min_chain_length, max_chain_length + 1, length_step)]
              for count in
              range(min_chain_count, max_chain_count + 1, count_step)])
x, y = np.meshgrid(range(min_chain_length, max_chain_length + 1, length_step),
                   range(min_chain_count, max_chain_count + 1, count_step), )
plt.figure()
plt.title('||R||')
p = plt.pcolormesh(x, y, R, shading='nearest')
plt.colorbar(p)
plt.show()
```

5.4 Результат выполнения

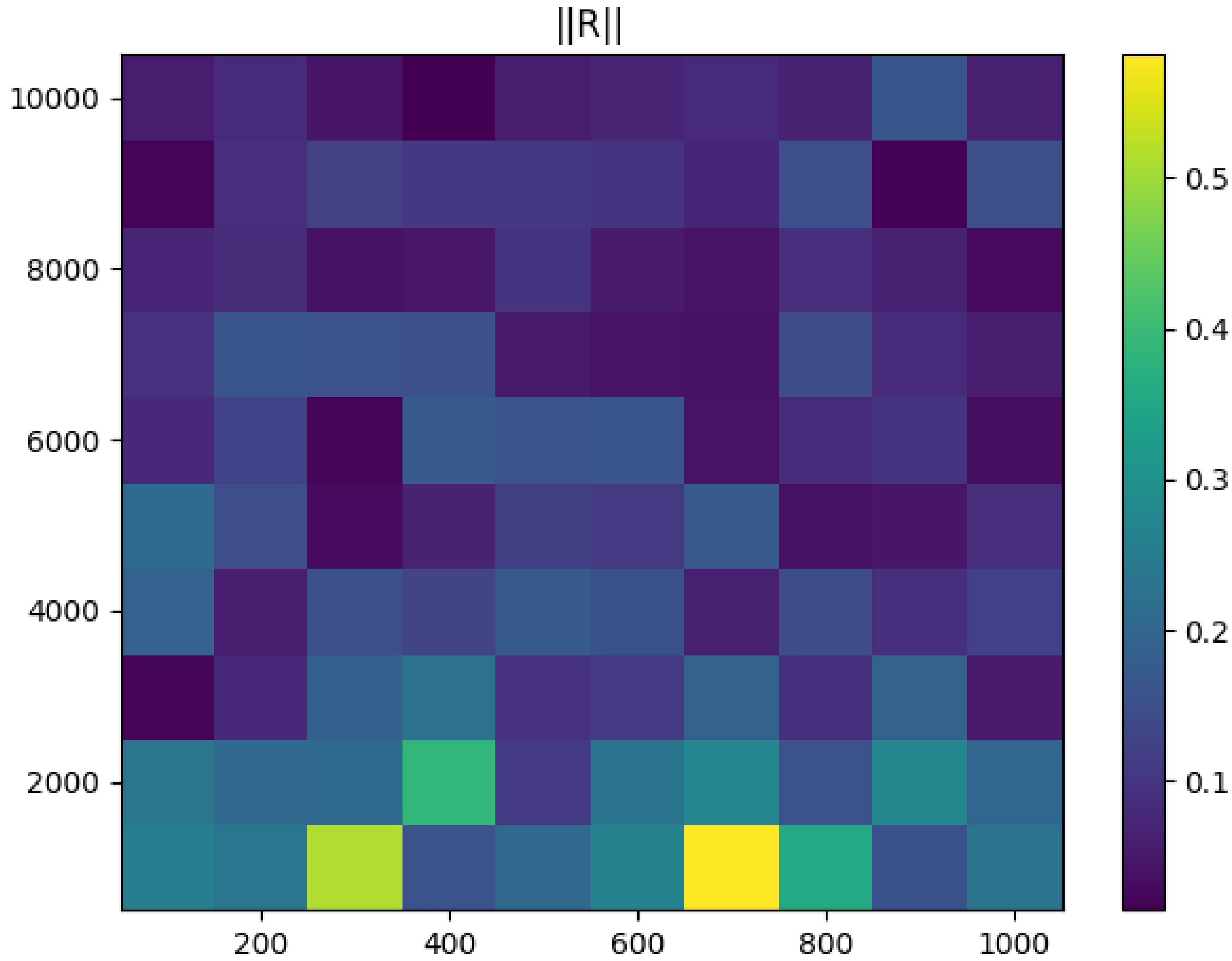


Рис. 13: График зависимости точности решения от длины цепи маркова - ось Ox и числа цепей Маркова - ось Oy.

6 Лабораторная 6

6.1 Условие

Согласно варианту 10: Моделирование процесса функционирования вычислительного центра.

Исходные данные:

1. Вычислительный центр, оснащенный тремя однотипными ЭВМ, обслуживает сеть активных терминалов;

2. Задачи пользователей образуют пуассоновский поток с λ зад/сек, а время выполнения задачи в ЭВМ имеет экспоненциальное распределение с математическим ожиданием μ сек;

3. Программа-диспетчер обрабатывает задачу, выбирая для нее свободную ЭВМ. Время обработки равномерно распределено на интервале $[a \pm \delta]$. Если все ЭВМ заняты, то задача направляется в очередь, которая на данный момент является минимальной;

4. После выполнения в ЭВМ, задача возвращается на соответствующий терминал, причем 30% задач обслуживается в АЦПУ $[b \pm \varepsilon]$ сек.

Цель: Разработать GPSSV-модель для анализа процесса функционирования вычислительного центра в течение одного часа.
Первоначальный перечень экспериментов: $\lambda = 0.2, \mu = 12, a = 2, \delta = 1, b = 12$.

6.2 Код программы

```
          GENERATE (EXPONENTIAL(1,0,5))
          ADVANCE 2,1

          TEST LE Q$QUE1,Q$QUE2,LBL_Q2Q3
          TEST LE Q$QUE1,Q$QUE3,LBL_QUE3
LBL_QUE1  QUEUE QUE1
          SEIZE PC1
          DEPART  QUE1
          ADVANCE (EXPONENTIAL(1,0,12))
          RELEASE PC1
          TRANSFER ,LBL_CHECK_APD

LBL_QUE3  QUEUE QUE3
          SEIZE PC3
          DEPART  QUE3
          ADVANCE (EXPONENTIAL(1,0,12))
          RELEASE PC3
          TRANSFER ,LBL_CHECK_APD

LBL_Q2Q3  TEST LE Q$QUE2,Q$QUE3,LBL_QUE3
```

```
LBL_QUE2   QUEUE QUE2
           SEIZE PC2
           DEPART  QUE2
           ADVANCE (EXPONENTIAL(1,0,12))
           RELEASE PC2
           TRANSFER ,LBL_CHECK_APD

LBL_CHECK_APD TRANSFER .3,TTT,APD

APD           ADVANCE    12,8
TTT           TERMINATE  0
           GENERATE 3600
           TERMINATE  1
           START   1
```

6.3 Результат выполнения

```
              START TIME           END TIME  BLOCKS  FACILITIES  STORAGES
                   0.000           3600.000    28         3          0


                 NAME                       VALUE
              APD                           25.000
              LBL_CHECK_APD                 24.000
              LBL_Q2Q3                      17.000
              LBL_QUE1                       5.000
              LBL_QUE2                      18.000
              LBL_QUE3                      11.000
              PC1                        10003.000
              PC2                        10004.000
              PC3                        10005.000
              QUE1                       10000.000
              QUE2                       10001.000
              QUE3                       10002.000
              TTT                           26.000

 LABEL              LOC  BLOCK TYPE     ENTRY COUNT CURRENT COUNT RETRY
                    1    GENERATE           728             0       0
                    2    ADVANCE            728             0       0
                    3    TEST               728             0       0
                    4    TEST               423             0       0
LBL_QUE1            5    QUEUE              276             0       0
                    6    SEIZE              276             0       0
                    7    DEPART             276             0       0
                    8    ADVANCE            276             1       0
                    9    RELEASE            275             0       0
                   10    TRANSFER           275             0       0
LBL_QUE3           11    QUEUE              184             0       0
                   12    SEIZE              184             0       0
                   13    DEPART             184             0       0
                   14    ADVANCE            184             1       0
                   15    RELEASE            183             0       0
                   16    TRANSFER           183             0       0
LBL_Q2Q3           17    TEST               305             0       0
LBL_QUE2           18    QUEUE              268             1       0
                   19    SEIZE              267             0       0
                   20    DEPART             267             0       0
                   21    ADVANCE            267             1       0
                   22    RELEASE            266             0       0
                   23    TRANSFER           266             0       0
LBL_CHECK_APD      24    TRANSFER           724             0       0
APD                25    ADVANCE            197             0       0
TTT                26    TERMINATE          724             0       0
                   27    GENERATE             1             0       0
                   28    TERMINATE            1             0       0
FACILITY         ENTRIES  UTIL.   AVE. TIME AVAIL. OWNER PEND INTER RETRY DELAY
 PC1                276    0.931      12.147  1      728    0    0     0     0
 PC2                267    0.844      11.377  1      725    0    0     0     1
 PC3                184    0.672      13.156  1      727    0    0     0     0
QUEUE              MAX CONT. ENTRY ENTRY(0) AVE.CONT. AVE.TIME   AVE.(-0) RETRY
 QUE1                6    0    276     51     1.591     20.747     25.450   0
 QUE2                6    1    268     59     1.199     16.101     20.646   0
 QUE3                6    0    184     52     0.960     18.775     26.171   0
FEC XN   PRI         BDT      ASSEM  CURRENT  NEXT  PARAMETER    VALUE
   725    0       3604.389    725     21      22
   730    0       3606.605    730      0       1
   728    0       3610.951    728      8       9
   727    0       3617.698    727     14      15
   731    0       7200.000    731      0      27
```

Рис. 14: Результат работы программы.