Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ *Информатика и системы управления*

КАФЕДРА *Программное обеспечение ЭВМ и информационные технологии*

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ:

Метод генерации псевдослучайных чисел на основе клеточного автомата «Жизнь»

Студент ИУ7-41М (Группа) _____ (Подпись, дата) **Мирзоян С.А.** (И.О.Фамилия)

Руководитель ВКР _____ (Подпись, дата) **Ковтушенко А.П** (И.О.Фамилия)

Консультант _____ (Подпись, дата) _____ (И.О.Фамилия)

Нормоконтролер _____ (Подпись, дата) _____ (И.О.Фамилия)

2023 г.

РЕФЕРАТ

Расчетно-пояснительная записка 174 с., 38 рис., 4 табл., 15 источников, 4 прил.

Объектом разработки является метод для генерации псевдослучайных чисел.

Цель работы – продолжить исследования в методе генерации псевдослучайных чисел на основе клеточного автомата, изучить новые способы применения, а также предложить модификации для улучшения качественно-скоростных характеристик.

Поставленная цель достигается за счет решения следующих задач:

- изучение методов генерации случайных и псевдослучайных чисел;
- изучение новых методик представления карты вселенной игры «Жизнь»;
- Модернизация правил игры «Жизнь»;
- Модернизация последних реализаций данного метода генерации.
- Исследование качественных характеристик приведенного метода генерации псевдослучайных чисел;
- Сравнительный анализ модернизированного метода генерации с изначальной итерацией на предмет улучшения качественно-скоростных характеристик;
- Сравнительный анализ приведенного метода с существующими, в том числе на основе технологии клеточных автоматов;
- разработка приложения, визуально демонстрирующая работу предложенного метода, а также выдающего на выход последовательность псевдослучайных чисел.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Окружающий нас мир полон неизведанного. Каждый день миллионы ученых и просто любопытных людей трудятся во благо науки. Одни изучают живые организмы, других химические соединения. Результатом многовековых испытаний и исследований всего и вся путем проб, ошибок, изобретений новых и все более совершенных методик изучения природы окружающего мира является современный уровень развития человечества, науки и прочих институтов жизни.

От пещеры до космоса человека вела череда великих открытий и изобретений. Как когда-то человечество изменило навсегда изобретение колеса, так и изобретение компьютера расширило человеческие возможности во всех сферах жизни до небывалых широт, границы которых не изведаны до сих пор.

Компьютер помог завершить Вторую Мировую Войну, отправить человека в космос, реализовать в виртуальной реальности все самые невероятные теории и изучать их.

В многих научных исследованиях, будь то исследования в сфере медицины, моделирования, защиты информации и прочих особую роль занимает поиск случайных чисел или последовательностей случайных чисел. Есть разные способы получения таких чисел, со своими достоинствами и недостатками.

В данной работе предложен метод генерации псевдослучайных чисел, в основе которого лежит технология клеточных автоматов. Проведен обзор существующих генераторов псевдослучайных чисел, их преимуществ и недостатков, выбраны критерии оценки качества генераторов псевдослучайных чисел и проведен сравнительный анализ предложенного генератора с уже существующими, а именно с линейно-конгруэнтным генератором и генератором, также основанным на технологии клеточных автоматов.

1 Аналитический раздел

В аналитическом разделе проведен обзор и сравнительный анализ существующих методов генерации псевдослучайных чисел с выделением их преимуществ и недостатков. Сформулирована постановка задачи для разработки метода генерации псевдослучайных чисел.

1.1 Генераторы случайных и псевдослучайных чисел

Генератор псевдослучайных чисел (ГПСЧ) — это алгоритм, порождающий последовательность чисел, которая подчиняется заданному распределению и элементы которой почти независимы друг от друга.

В прошлом, для получения последовательности случайных чисел исследователям приходилось прибегать к неудобным и времязатратным способам, таким как перемешивание костей в мешочке и вынимание их оттуда по очереди. Однако с развитием технологии стали возможны способы получения истинно случайных чисел, используя микрофон для захвата окружающего шума или распад радиоактивного вещества в течение определенного времени.

Несмотря на очевидные достоинства, такие методы требуют наличия специального оборудования, содержащего генераторы энтропии, которые могут генерировать не коррелированные и статистически независимые числа, что является нетривиальной задачей. К тому же, для использования истинно случайных чисел в криптографии они должны быть изолированы от внешних влияний, что может быть проблематично.

Кроме аппаратного метода генерации случайных чисел, можно также составить таблицы, которые будут содержать некоррелированные последовательности чисел. Этот метод называется табличным. Однако у такого метода есть недостатки, такие как ограниченность последовательности, занимаемая память, предопределенность значений в случае утечки таблиц и т.д.

Существует также третий тип генераторов - алгоритмический. В основном он представляет собой комбинацию физического генератора и детерминированного алгоритма. Он использует ограниченный набор входных данных от физического генератора и преобразует его в новое значение согласно заданному алгоритму. Хотя этот метод работает быстро, не занимает много памяти и не требует специального оборудования, он генерирует псевдослучайные последовательности чисел, которые имеют свои недостатки, такие как цикличность и зависимость каждого последующего числа от предыдущих.

Стоит отметить, что в России существует ГОСТ для генераторов псевдослучайных чисел. В стандарте установлены методы генерации случайных чисел, подчиняющихся равномерному распределению и другим законам распределения, используемых при применении метода Монте-Карло. Однако криптографические методы генерации в нем отсутствуют. [6]

Генераторы псевдослучайных чисел

Генераторы псевдослучайных чисел широко используются в различных приложениях, таких как игры, криптография, научные и инженерные вычисления, моделирование и другие программы, где требуется случайный элемент.

Криптостойкие генераторы псевдослучайных чисел используются в криптографических протоколах для обеспечения безопасности коммуникации и защиты данных от несанкционированного доступа.

В текущей подразделе рассматриваются различные виды методов генерации псевдослучайных чисел. Представленные ниже методы – это лишь малая часть от общего пантеона ГПСЧ.

В качестве примера можно рассмотреть следующие генераторы псевдослучайных чисел:

- Линейный конгруэнтный;
- Регистр сдвига с обратной линейной связью (РСЛОС);
- Вихрь Мерсенна;
- ГПСЧ на базе клеточного автомата (правило 30);
- ГПСЧ на базе клеточного автомата (NESW)

Линейный конгруэнтный

Линейная конгруэнтная схема генерации псевдослучайных чисел была предложена Д. Г. Лехмером в 1949 году и считается наиболее популярной. Согласно этой схеме, последовательность случайных чисел $< X_n >$ можно получить, если положить, что

$$X_{n+1} = (aX_n + c) \, mod \, m, \; n \geq 0, \; (1)$$

где

$$m - \text{модуль}, 0 < m,$$

$$a - \text{множитель}, 0 \leq a < m,$$

$$c - \text{приращение}, 0 \leq \text{с} < m,$$

$$X_0 - \text{начальное значение}, 0 \leq X_0 < m,$$

В приведенной выше формуле особое значение имеет выбор параметров. Например, для $X_0 = 7, a = 8, c = 9, m = 10$ получится последовательность:

7, 5, 9, 1, 7, 5, 9, 1, …

Как можно заметить, данную последовательность с трудом можно назвать «случайной». В этом примере кроется проблема данного метода – линейный конгруэнтный метод всегда выдает последовательность, которая зацикливается.

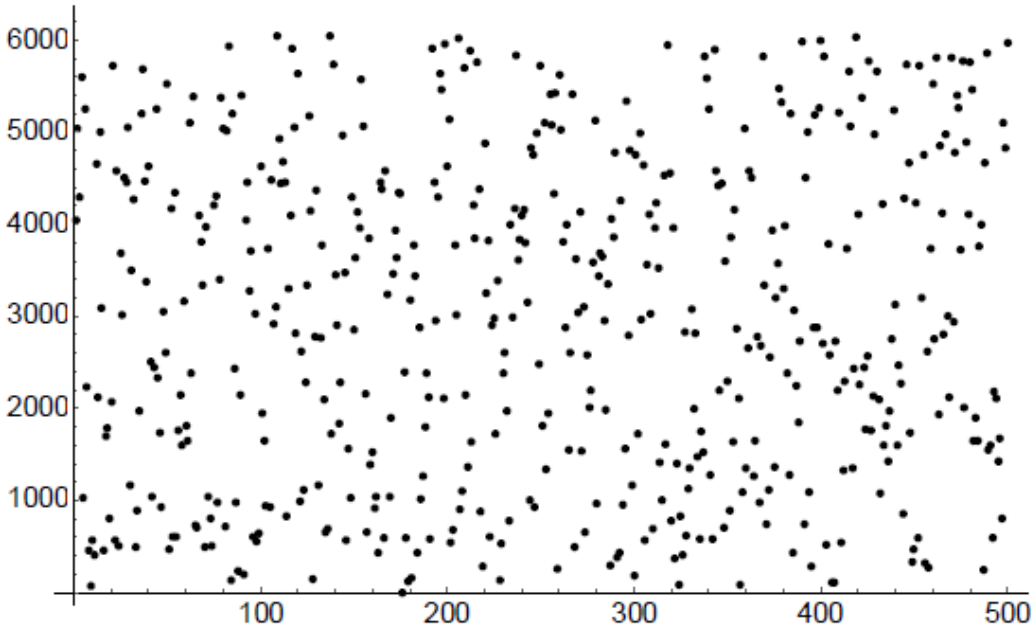

Рисунок 1.1. График ЛКП для X¬0= 7,a=106,c= 1283,m = 6075.

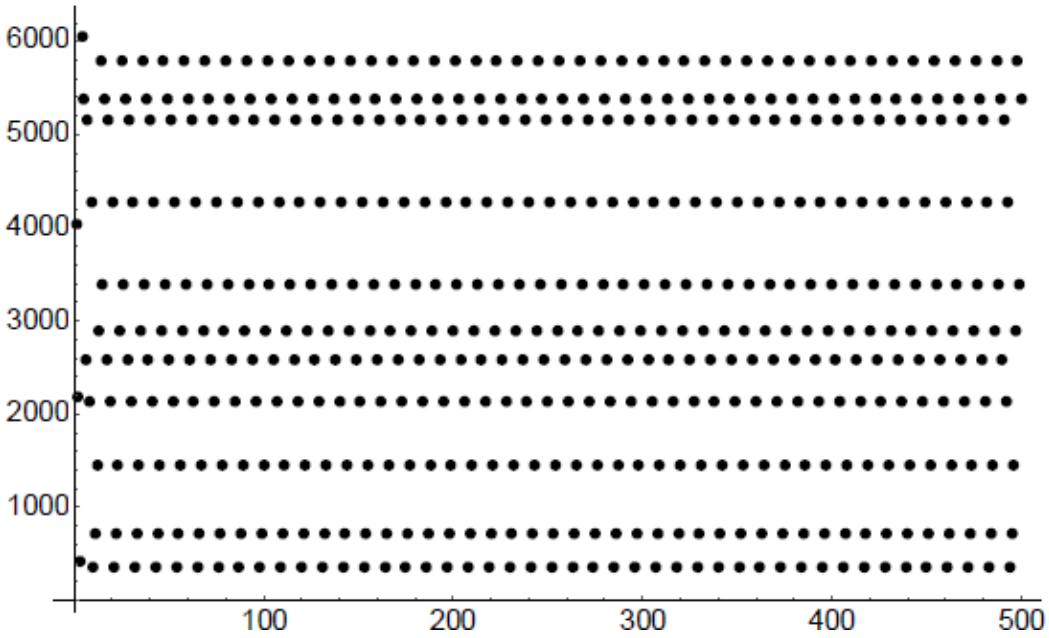

Рисунок 1.2. График ЛКП для X¬0= 7,a=105,c= 1283,m = 6075.

Также, ввиду свойств операций в конечном поле, применяемых в формуле (1), возникает «решетчатая структура» в последовательностях, что также является недостатком. Данный эффект можно наблюдать на рисунке 1.2.1.3

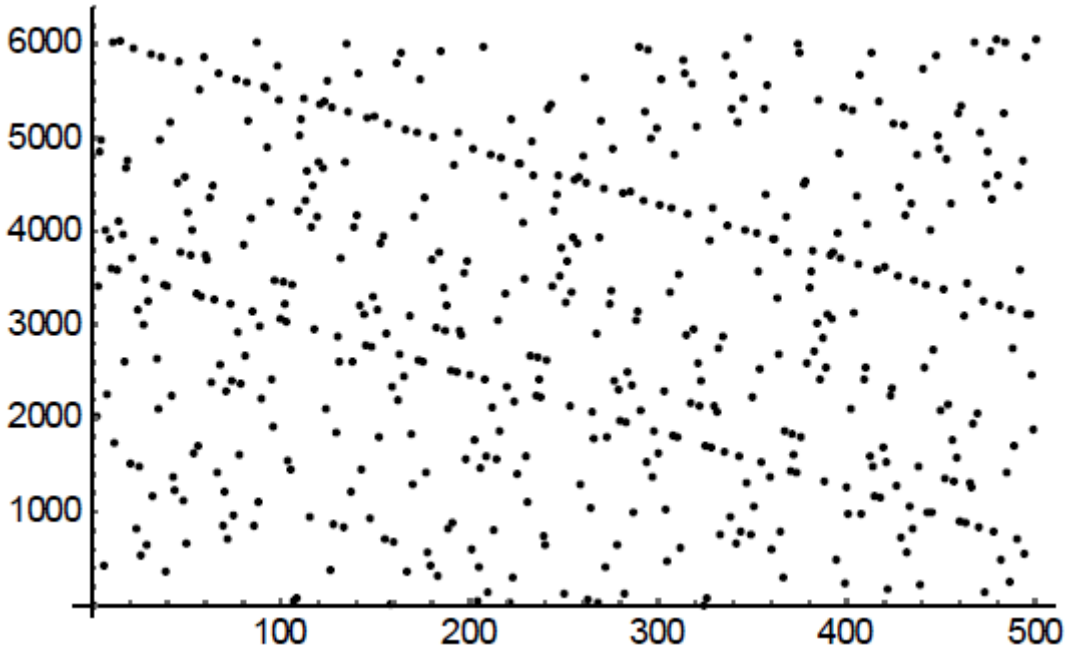

Рисунок 1.3. График ЛКП для X¬0= 7,a=106,c= 1284,m = 6075.

Ниже приведена таблица констант для линейных конгруэнтных генераторов. [4]

Таблица 1.1. Параметры ЛКГ для формулы 1.

(a, c, m)	(a, c, m)	(a, c, m)
(106,1283,6075)	(625,6571,31104)	(1277,24749,117128)
(211,1663,7875)	(1541,2957,14000)	(2041,25673,121500)
(421,1663,7875)	(1741,2731,12960)	(2311,25367,120050)
(430,2531,11979)	(1291,4621,21870)	(1597,51749,244944)
(936,1399,6655)	(205,29573,139968)	(2661,36979,175000)
(1366,1283,6075)	(421, 17117,81000)	(4081,25673,121500)
(171,11213,53125)	(1255,6173,29282)	(3661,30809,145800)
(859,2531,11979)	(281,28411,134456)	(3613,45289,214326)
(419,6173,29282)	(1093,18257,86436)	(1366,150889,714025)
(967,3041,14406)	(421,54773,259200)	(8121,28411,134456)
(141,28411,134456)	(1021,24631,116640)	(4561,51349,243000)

Если подытожить, то можно в качестве преимуществ выделить простоту реализации и скорость работы данного генератора. Недостатком же будет являться цикличность значений последовательности.

Регистр сдвига с обратной линейной связью

Данный класс ГСПЧ основан на преобразовании бинарного представления некоторого числа. Такие генераторы имеют некоторые преимущества, как, например, скорость генерации чисел, хорошие статистические свойства псевдослучайных чисел, а также возможность простой реализации на аппаратном уровне.

Регистр сдвига с обратной линейной связью (РСЛОС) – регистр сдвига битовых слов, у которого входной (вдвигаемый) бит является линейной функцией остальных битов. Вдвигаемый вычисленный бит заносится в ячейку с номером 0. Количество ячеек p называют длиной регистра. [4]

Для натурального числа p и $a_1, a_2, \dots, a_{p-1}$, принимающих значения 0 или 1, определяют рекуррентную формулу

$$X_{n+p} = a_{p-1}X_{n+p-1} + a_{p-2}X_{n+p-2} + \dots + a_1X_{n+1} + X_n,$$

Из формулы можно заключить, что для РСЛОС функция обратной связи является линейной булевой функцией от состояний всех или некоторых битов регистра.

Одна итерация алгоритма, генерирующего последовательность, состоит

из следующих шагов:

1. Содержимое ячейки $p - 1$ формирует очередной бит ПСП битов.

2. Содержимое ячейки 0 определяется значением функции обратной связи, являющейся линейной булевой функцией с коэффициентами $a_1, a_2, \dots, a_{p-1}$. Его вычисляют по вышеприведенной формуле.

3. Содержимое каждого i-го бита перемещается в $(i + 1)$-й, $0 \le i < p - 1$.

4. В ячейку 0 записывается новое содержимое, вычисленное на шаге 2.

Наименьшее положительное целое N, такое, что $X_{n+N} = X_N$ для всех значений n называют *периодом последовательности*. Эту последовательность называют М-последовательностью.

Вихрь Мерсенна

В 1997 году японскими учеными Макото Мацумото и Такудзи Нимисура был предложен метод генерации случайных чисел, основанный на свойствах простых чисел Мерсенна. Данный метод получил название «Вихревой генератор». «Вихрь» – это преобразование, которое обеспечивает равномерно распределение ПСЧ.

Числом Мерсенна называется натуральное число M_n, определяемое формулой

$$M_n = 2^n - 1$$

Пример. Первые 17 чисел последовательности: 1, 3, 7, 15, 31, 63, 127, 255, 511,1023, 2047, 4095, 8191, 16 383, 32 767, 65 535, 131 071. [4]

Часто числами Мерсенна называют числа с простыми индексами n. Важным свойством чисел Мерсенна является то, что если M_n является простым, то значит, что и n – также простое число. В общем случае обратное не верно, что не дает возможности просто и эффективно генерировать простые числа, но зато позволяет эффективно проверять число на простоту. Данное свойство лежит в основе теста на простоту Люка-Лемера. [1]

Существует несколько вариантов этого ГПСЧ. Мы рассмотрим наиболее распространенный, который имеет обозначение MT19937. По сути, данный ГПСЧ является РСЛОС, состоящим из 624 ячеек по 32 бита. Метод Вихрь Мерсенна позволяет генерировать последовательность двоичных псевдослучайных целых w-битовых чисел в соответствии со следующей рекуррентной формулой

$$X_{n+p} = X_{n+p} \oplus (X_n^r | X_{n+1}^l)A, \qquad n = 0,1,2,3\ldots,$$

где p, q, r – целые константы, p – степень рекуррентности, $1 \leq q \leq p$;

X_n – w-битовое двоичное целое число;

$X_n^r | X_{n+1}^l$ – двоичное целое число, полученное конкатенацией чисел X_n^r и X_{n+1}^l, когда первые (w-r) битов взяты из X_n, а последние r битов из X_{n+1} в том же порядке;

A – матрица размера w×w, состоящая из нулей и единиц, определенная по-

средством a;

XA – произведение, при вычислении которого сначала выполняют операцию $X \gg 1$ (сдвига битов на одну позицию вправо), если последний бит X равен

0, а затем, когда последний бит $X = 1$, вычисляют $XA = (X \gg 1) \oplus a$.

Вихрь Мерсенна имеет огромный период, равный числу Мерсенна ($2^{19937} - 1$). Этот период достаточен для большинства возможных применений

алгоритма.

Метод обеспечивает равномерное распределение генерируемых псевдослучайных чисел в 623 измерениях. Поэтому корреляция между последовательными значениями в выходной последовательности Вихря Мерсенна пренебрежимо мала. Метод также хорошо проходит статистические тесты на «случайность».

Однако данный ГПСЧ не предназначен для получения криптографически стойких последовательностей случайных чисел. [12]

ГПСЧ на базе клеточного автомата («правило 30»)

В работах [7],[8],[9] и [10] описан метод генерации псевдослучайных чисел, предложенный Стивеном Вольфрамом. В его основе лежит клеточный автомат «правило 30».

Клеточный автомат – это устройство, состоящее из n-мерного массива ячеек и правил изменения значений ячеек. Каждая из ячеек имеет начальное состояние и изменяет свое состояние в дискретные моменты времени.

Правило, в соответствии с которым ячейка изменяет состояние, – это рекуррентная формула, в которой новое значение ячейки определяется исходя из предыдущих значений этой и соседних ячеек. В простом одномерном случае, когда массив ячеек состоит из n битов $a_1, a_2, \dots, a_n$, правило выглядит так:

$$a_k(t + 1) = a_{k-1}(t) \oplus (a_k(t) \vee a_{k+1}(t))$$

Этот генератор показывает хорошие статистические свойства. Однако он сильно зависим от начальных значений ячеек: при неудачном выборе начального состояния, клеточный автомат порождает циклические структуры. Кроме того, для него существует успешное вскрытие с известным открытым текстом. В [8, 9] показано, что вскрытие выполнимо на персональном компьютере с размером клеточного автомата вплоть до 500 битов.

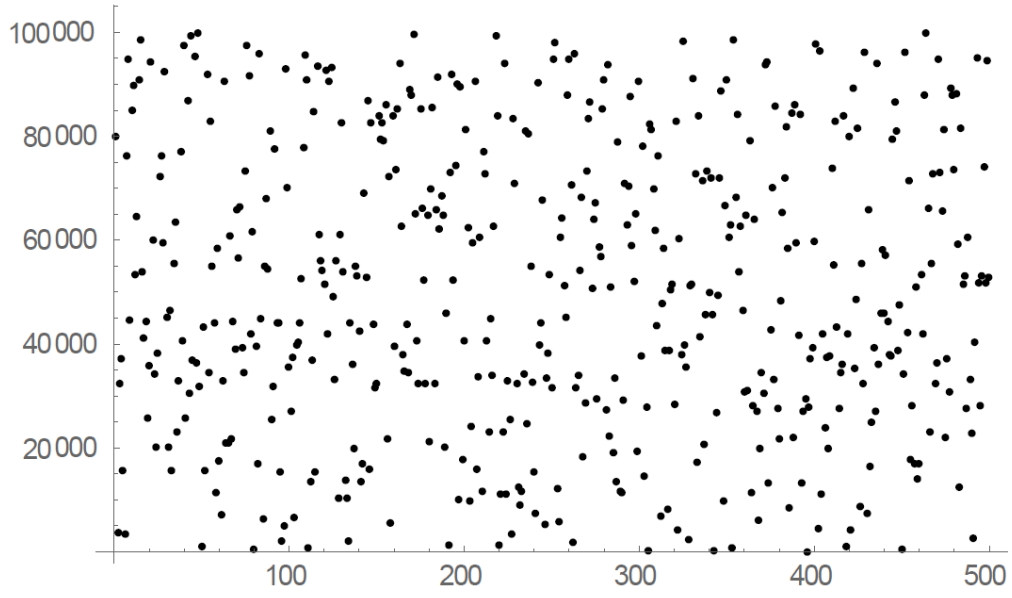

Рисунок 1.4. График ГПСЧ на базе клеточного автомата

ГПСЧ на базе клеточного автомата (NESW)

Одним из относительно молодых методов генерации псевдослучайных чисел, также основанном на технологии клеточного автомата, является метод, представленный университетом ИТМО за авторством Д. Д. Мухамеджанова, А. Б. Левиной. В сути своей данный метод является глубокой модификацией предыдущего метода. Сетка имеет размеры p и q , которые являются простыми числами (для улучшения периодичности). Изменения начинаются прямо с этого этапа: деление всей сетки на $m \geq 2$ блоков, которые формируются как прямоугольники b_i с размерами l_{b_i} и w_{b_i}, а каждый блок состоит из lbi wbi □ ячеек (рис. 1.5).

Выбрав произведение сторон прямоугольников блоков lbi □ wbi как начальную конфигурацию, необходимо заполнить ячейки таким образом, чтобы в каждой ячейке располагался один бит (состояние 0 или 1).

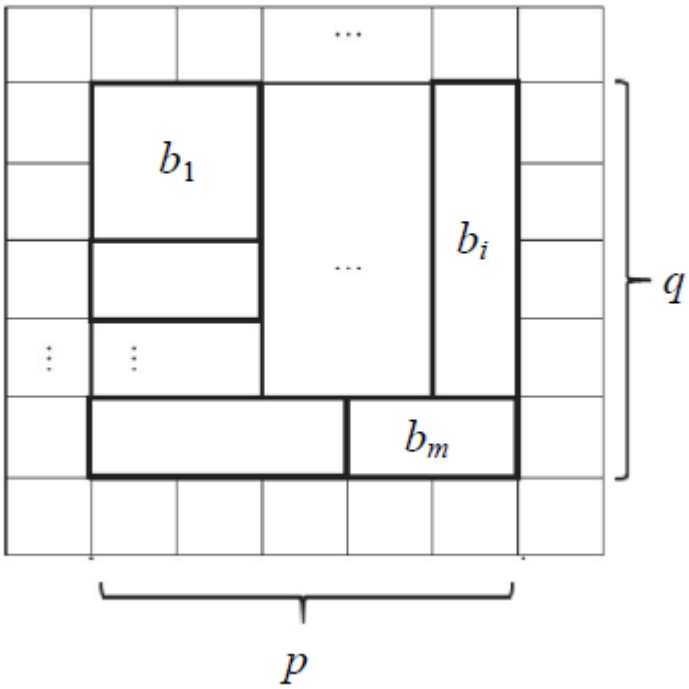

Рисунок 1.5. Разделение сетки клеточного автомата на блоки

Затем применяется алгоритм под названием NESW (North, East, South, West). Метод такого движения носит название по сторонам света (рис. 1.6). Суть

способа в том, чтобы заполнять сетку блока согласно направлениям сторон света, т.е. мы циклически заполняем битами полученного числа сетку, начиная с левого нижнего угла, двигаясь сначала вверх, затем вправо, вниз и влево до конца сетки (или до заполненной ячейки).

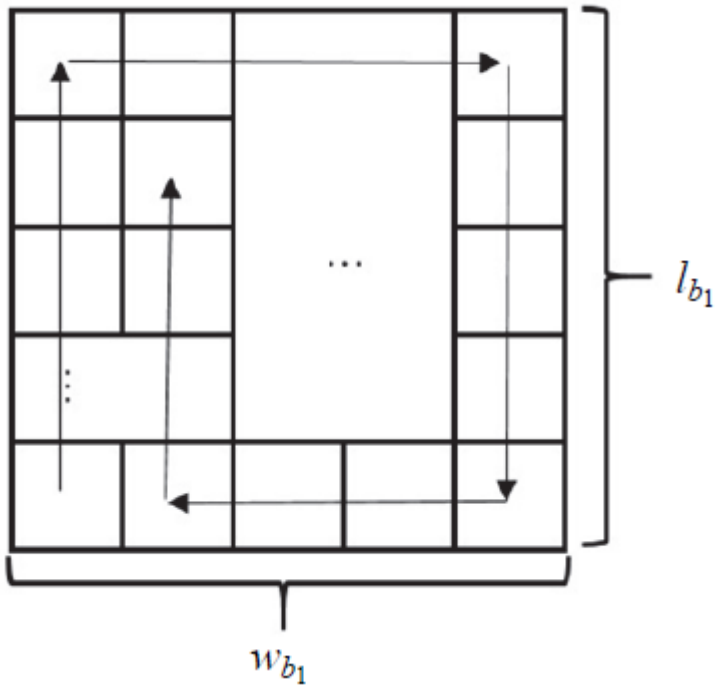

Рисунок 1.6. Метод NESW

Каждый блок может иметь свое собственное правило, что делает КА неоднородным и приводит к лучшим показателям критериев статистических свойств и лучшим периодическим свойствам. [11]

1.2 Криптографически стойкие методы генерации псевдослучайных чисел

Криптостойкие генераторы псевдослучайных чисел (КГПСЧ) используются в криптографии для генерации псевдослучайных чисел, которые могут использоваться для шифрования данных и защиты от несанкционированного доступа. КГПСЧ обладают особенными свойствами, которые делают их надежными и безопасными при использовании в криптографии.

Существует несколько типов криптостойких генераторов псевдослучайных чисел, которые различаются по состоянию, которое они используют для генерации чисел.

Односторонние хэш-функции

Односторонние хэш-функции (ОХФ) являются одним из наиболее широко распространенных типов криптостойких генераторов псевдослучайных чисел. Они генерируют псевдослучайные числа, используя входной ключ и текущее состояние.

Ключ и текущее состояние используются для обновления состояния генератора. При этом применяется односторонняя хэш-функция, которая выполняет преобразование текущего состояния в новое состояние. Новое состояние используется для генерации следующего числа.

Среди преимуществ данного метода можно выделить следующее:

- Несмотря на то, что исходное сообщение может быть произвольной длины, хеш-функция генерирует фиксированный хеш-код заданной длины. Это позволяет использовать хеш-функции для определения целостности данных и проверки наличия изменений;
- В зависимости от выбранной функции, хеш-значение может быть очень сложно предсказать. Это делает хеш-функции полезными для генерации псевдослучайных чисел;
- Хеши являются необратимыми - т.е. по значению хеша невозможно восстановить исходное сообщение. Это свойство является ключевым для обеспечения безопасной передачи данных и аутентификации клиентов.

Среди же недостатков следует отметить следующее:

- По определению хеш-функций, несколько сообщений могут иметь одинаковый хеш-значение — это называется коллизия. Чем

сложнее функция хеширования, тем меньше вероятность возникновения коллизий, но на практике они могут происходить. Это может стать уязвимостью для систем, которые зависят от уникальности хеш-кода для каждого сообщения;

- Хеш-функции также могут быть подвержены "атакам по словарю" - методу перебора множества возможных входных данных до нахождения их хеш-значения. Для устранения этого недостатка может использоваться "соление" - добавление случайного секретного значения в исходное сообщение перед хешированием;
- Некоторые современные криптографические алгоритмы могут быть уязвимыми к атакам с использованием специальных аппаратных средств, которые могут находить коллизии с высокой скоростью.

Один из примеров криптостойкой ОХФ – алгоритм SHA-2.

Генераторы на основе математических задач

Генераторы псевдослучайных чисел на основе сложных математических задач используют сложные вычисления для создания чисел, которые будут выглядеть случайными и не поддадутся криптоанализу. Примерами таких генераторов являются алгоритм Блюма-Блюма-Шуба и алгоритм Блюма-Микали. Рассмотрим детальнее один из этих генераторов.

Алгоритм Блюм-Блюма-Шуба (BBS) является одним из наиболее известных криптографических генераторов псевдослучайных чисел. Он основывается на трудной задаче факторизации больших целых чисел и является стойким к атакам по крайней мере на сегодняшний день.

Применение BBS имеет следующие преимущества:

1. Простота: алгоритм BBS достаточно прост в реализации, а входные данные легко регулируемы, в связи с чем алгоритм может быть удобен для различного рода исследований;

2. Псевдослучайность: числа, которые генерируются BBS, не могут быть предсказаны заранее, что делает их настоящими случайными числами.
3. Быстродействие: BBS генерирует большое количество случайных чисел за короткое время;

Однако, есть и недостатки BBS:

1. Значительные вычислительные ресурсы: генерация чисел происходит путем выполнения сложных математических операций, что требует большого количества вычислительных ресурсов.
2. Низкая скорость генерации чисел: время генерации каждого числа зависит от количества бит в последовательности. Также для генерации большого количества случайных чисел требуется много времени.
3. Неудобство использования: BBS может быть неудобным для использования в реальном времени, поскольку время генерации каждого числа может достигать нескольких секунд.

Итак, преимущества BBS включают безопасность, псевдослучайность и эффективность, а недостатки - значительные вычислительные ресурсы, низкую скорость генерации чисел и неудобство использования.[15]

Энтропийные коллекторы

Энтропийные коллекторы – это специализированные устройства или программы, которые собирают случайные данные из различных источников и используют их для генерации псевдослучайных чисел.

Один из примеров энтропийных коллекторов – алгоритм Fortuna.

Fortuna основывается на хэш-функции SHA-256 (Secure Hash Algorithm) и имеет структуру, состоящую из нескольких независимых генераторов, работающих параллельно. Каждый генератор использует отдельный вектор и периодически обновляется новыми случайными данными. Такая структура

делает Fortuna стойким к атакам и обеспечивает высокую скорость генерации случайных чисел.

У данного алгоритма есть несколько преимуществ:

1. Стойкость к атакам: многие криптографические алгоритмы могут быть взломаны при наличии достаточной вычислительной мощности. Fortuna, однако, является стойким к атакам благодаря использованию нескольких независимых генераторов, работающих параллельно.
2. Высокая скорость генерации: благодаря параллельной работе нескольких генераторов Fortuna может генерировать большие объемы случайных чисел за короткое время.
3. Автоматическое обновление ключей: Fortuna автоматически обновляет свои ключи после каждой генерации случайных чисел, что улучшает безопасность генерируемых данных.
4. Энтропия: Fortuna использует множество источников энтропии для генерации случайных чисел, что улучшает их качество.

Однако, есть и некоторые недостатки Fortuna:

1. Низкая скорость восстановления: если входной источник энтропии прекращает свою работу, то требуется определенное время для восстановления необходимого уровня энтропии.
2. Сложность реализации: Fortuna является сложным и трудоемким для реализации алгоритмом, что может сделать его менее привлекательным для использования в некоторых приложениях.

Каждый из вышеупомянутых типов КСГПСЧ имеет свои преимущества и недостатки, которые могут помочь при выборе соответствующего генератора в зависимости от конкретного случая использования.

1.3 Клеточные автоматы

Клеточные автоматы – это дискретные динамические системы, поведение которых полностью определяется в терминах локальных зависимостей, в значительной в значительной степени так же обстоит дело для большого класса непрерывных динамических систем, определенных уравнениями в частных производных. [2]

КА можно формально описать как четверку

$$\sigma = (Z^k, E_n, V, \varphi), \qquad (1)$$

где Z^k – множество k мерных векторов, $E_n = \{0; 1; 2; \dots, n-1\}$ - множество состояний одной ячейки в φ, $V = (\alpha_1, \alpha_2, \dots, \alpha_{n-1})$ – окрестность или шаблон соседства (упорядоченное множество различных k-мерных векторов из Z^k, $\varphi(x_0, x_1, \dots, x_{h-1})$, φ: $E_n^h \to E_n$ – локальная функция переноса.[3-4]

Из данного выше описания становится понятно, что на очередном шаге состояние каждой клетки зависит от состояния клеток вокруг нее – ее окрестности. Обычно различают два вида окрестностей: окрестность Мура и окрестность фон Неймана (рисунок 1.7). [3][13]

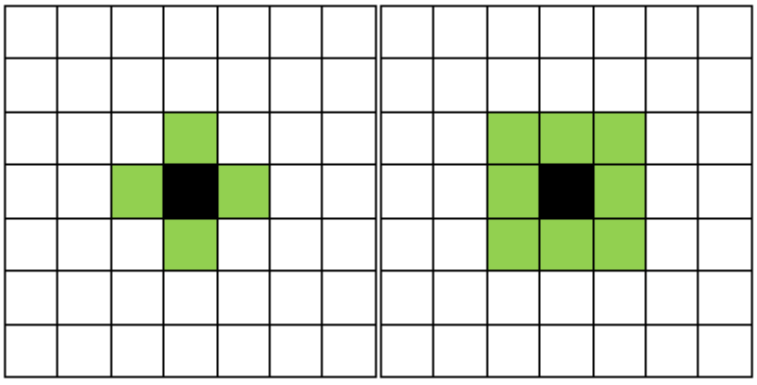

Рисунок 1.7 Окрестности фон Неймана (слева) и Мура (справа)

Игра «Жизнь»

Игра «Жизнь» — это клеточный автомат, изобретенный в 1970 году английским математиком Джоном Конвеем, в котором соблюдается ряд определенных правил поведения системы.

Правила игры таковы:

- «Вселенная» игры — это размеченная на клетки поверхность или плоскость. Она может быть безграничная, ограниченная или замкнутая.
- Каждая клетка на этой поверхности может находиться в двух состояниях: быть «мёртвой» или «живой». Клетка имеет восемь соседей, окружающих её.
- Распределение живых клеток в начале игры называется первым поколением. Каждое следующее поколение рассчитывается на основе предыдущего по следующим правилам:
    - Если рядом с мёртвой клеткой рядом находятся ровно три живые клетки, то в данной клетке зарождается жизнь;
    - если вокруг живой клетки два или три живых соседки, то эта клетка не умирает;
    - клетка умирает, если соседей меньше двух («от одиночества») или больше трёх («от перенаселённости»);
- Игра прекращается, если
    - Во «вселенной» не останется ни одной живой клетки;
    - складывается периодическая конфигурация;
    - складывается стабильная конфигурация;

Благодаря этим простым правилам можно добиться большого разнообразия «форм жизни».

Стоит также отметить, что игрок занимает позицию наблюдателя и не принимает прямого участия в игре. Он может лишь формировать начальную конфигурацию живых клеток.

В результате выполнения классических условий игры через некоторое количество ходов по выбору методики генерирования получается последовательность псевдослучайных чисел из нулей и единиц. Эта особенность данного клеточного автомата и будет применяться в предложенном методе.

1.4 Вывод

В разделе дано описание генераторов случайных и псевдослучайных чисел, а также клеточных автоматов. Рассмотрены особенности и проблемы генерации истинно случайных чисел, разновидности методов генерации псевдослучайных чисел. Среди разновидностей рассмотренных методов приведены два метода, также основанных на технологии клеточных автоматов. Рассмотрены обобщенный клеточный автомат, а также частный случай его реализации – игра «Жизнь», которая лежит в основе предложенного метода. Приведены правила игры «Жизнь». Определены ограничения и требования.

2 Конструкторский раздел

В конструкторском разделе проведена формализация задачи – рассмотрены проблемы и способы их решения. Проведен разбор используемых алгоритмов, предложены модификации для улучшения их работы. Представлены IDEF0 диаграммы, рисунки, изображающие ход работы клеточного автомата и игры «Жизнь», а также тот или иной выбор конфигурации клеточного автомата и последующие результаты спустя поколения.

2.1 IDEF0 диаграмма

На рисунках 2.1 – 2.3 представлена IDEF0 диаграмма.

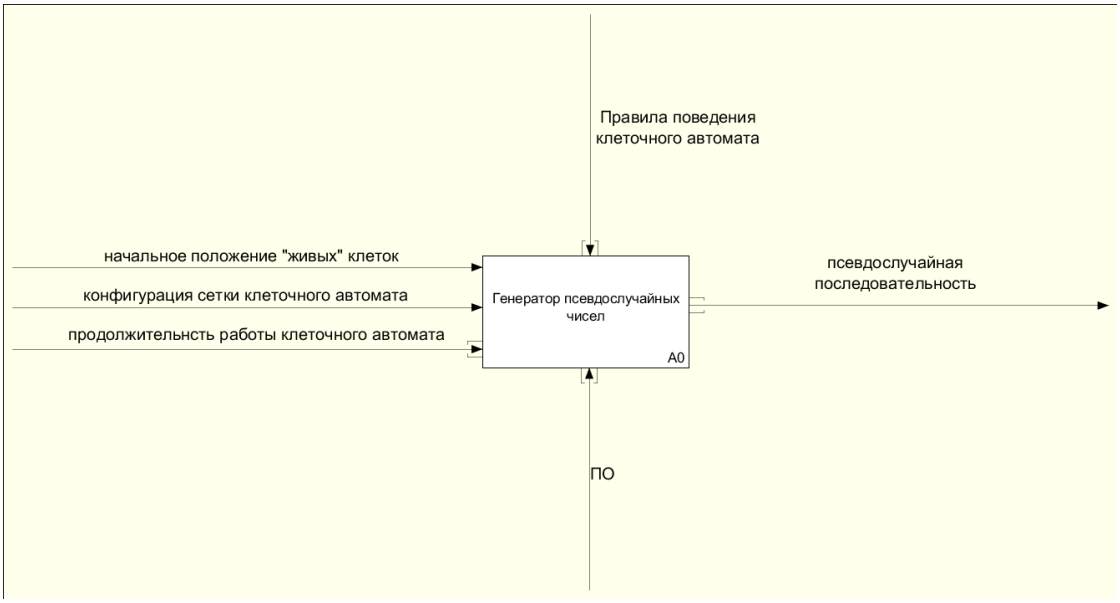

Рисунок 2.1 – Модель работы генератора псевдослучайных чисел, уровень 0

Генератор псевдослучайных чисел представляет собой клеточный автомат, правила поведения для которого заданы изначально и работают без вмешательства человека.

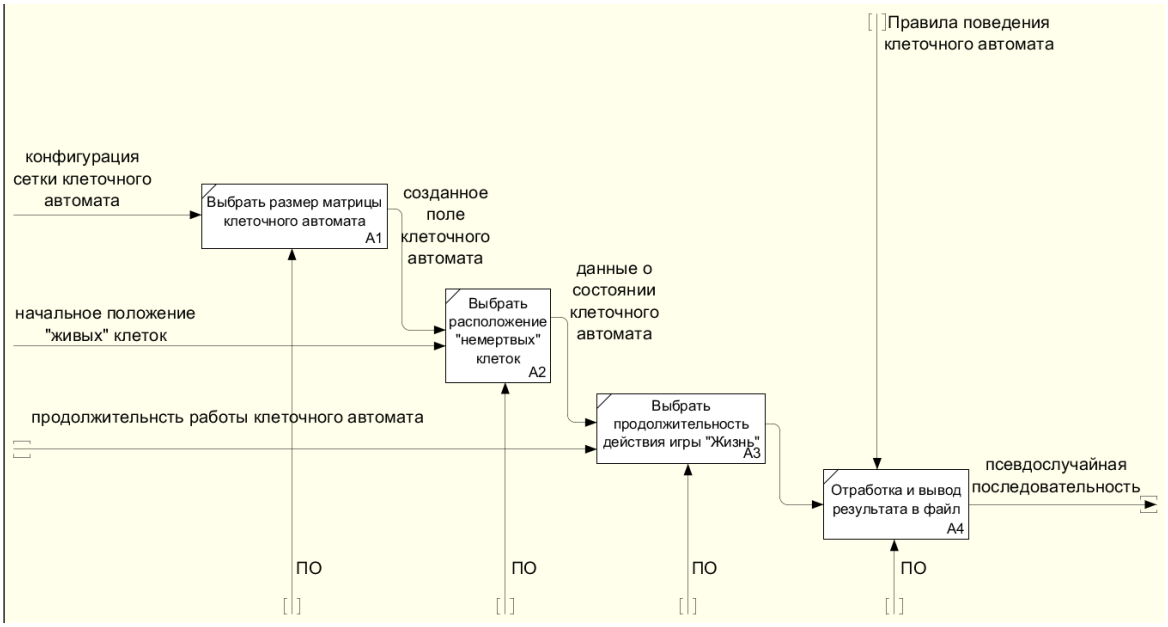

Рисунок 2.2 – Модель работы генератора псевдослучайных чисел, уровень 1

Для того, чтобы генератор выдал результат, клеточному автомату требуется задать ряд параметров, это:

- Размеры клеточного автомата;
- Положение ненулевых значений в клеточном автомате;
- Поведение клеточного автомата, представленного игрой «Жизнь»;
- Продолжительность игры

Клеточный автомат представляет собой матрицу, где каждый элемент представляет является отдельной клеткой. Правила клеточного автомата задаются условной конструкцией, рассмотренной в подразделе 2.3.2.

В результате работы программы выбирается одна строчка из матрицы, ее значение в виде тернарной последовательности записывается в текстовый файл формата .txt.

2.2 Сложности при проектировании

При создании предложенного метода возник ряд проблем, требующих решения, а именно:

- Выбор начального положения не мертвых клеток. В рамках классической игры «Жизнь» эта проблема также актуальна, ибо начальное положение должно быть подобрано так, чтобы наблюдаемая «вселенная» прожила достаточное большое количество поколений. С точки зрения игры, это нужно, дабы добиться устойчивых форм жизни. Для предложенного же метода это нужно, чтобы клеточный автомат не был заполнен преимущественно нулями, тем самым улучшая результирующую случайность;
- Продолжительность жизни «вселенной». Добившись правильного начального состояния клеточного автомата, при котором система может существовать достаточно долго, нужно также и ограничить продолжительность существования «вселенной», так как для результата в виде псевдослучайного числа или последовательности из таких чисел не требуется ожидать завершения полного цикла ее «вселенной». Помимо этого, также это не позволяет получать результат быстро. Поэтому требуется находить такую конфигурацию клеток и времени работы генератора, при котором случайность может быть получена как можно быстрее;
- Видимая случайность. При всей, казалось бы, случайности результата игры «Жизнь», тем не менее, зачастую, конечная матрица клеточного автомата остается заполненной нулями. Не хватает «шума», который повысил бы случайность результирующей картины. Необходимо пересмотреть правила игры «Жизнь».

Несмотря на порядок выделенных проблем рассматриваться их решения будут в обратном порядке. Решение третьей проблемы будет приведено в разделе 2.3

Для решения второй и первой проблемы требуется ряд экспериментов, в ходе которых можно выделить нужную начальную конфигурацию всей системы, что и будет рассматриваться в разделе 2.4.

2.3 Модификация правил игры «Жизнь»

В данном подразделе будут рассматриваться изменения, примененные к классической игре «Жизнь», сравнение модифицированной версии и классической. Благодаря приведенным ниже модификациям предложенный метод и добивается более «качественной» случайности, чем классическая игра «Жизнь». Продемонстрировано, как правила игры влияют на развитие «вселенной» игры.

Классические правила игры «Жизнь»

Как уже отмечалось в аналитическом разделе, классическая игра имеет следующий ряд правил:

- «Вселенная» игры — это размеченная на клетки поверхность или плоскость. Она может быть безграничная, ограниченная или замкнутая.
- Каждая клетка на этой поверхности может находиться в двух состояниях: быть «мёртвой» или «живой». Клетка имеет восемь соседей, окружающих её.
- Распределение живых клеток в начале игры называется первым поколением. Каждое следующее поколение рассчитывается на основе предыдущего по следующим правилам:
    - Если рядом с мёртвой клеткой рядом находятся ровно три живые клетки, то в данной клетке зарождается жизнь;
    - если вокруг живой клетки два или три живых соседки, то эта клетка не умирает;
    - клетка умирает, если соседей меньше двух («от одиночества») или больше трёх («от перенаселённости»);
- Игра прекращается, если
    - Во «вселенной» не останется ни одной живой клетки;
    - складывается периодическая конфигурация;

- складывается стабильная конфигурация;

Результат работы этого клеточного автомата, работающего по таким правилам, можно наблюдать в рисунках 2.3-2.4

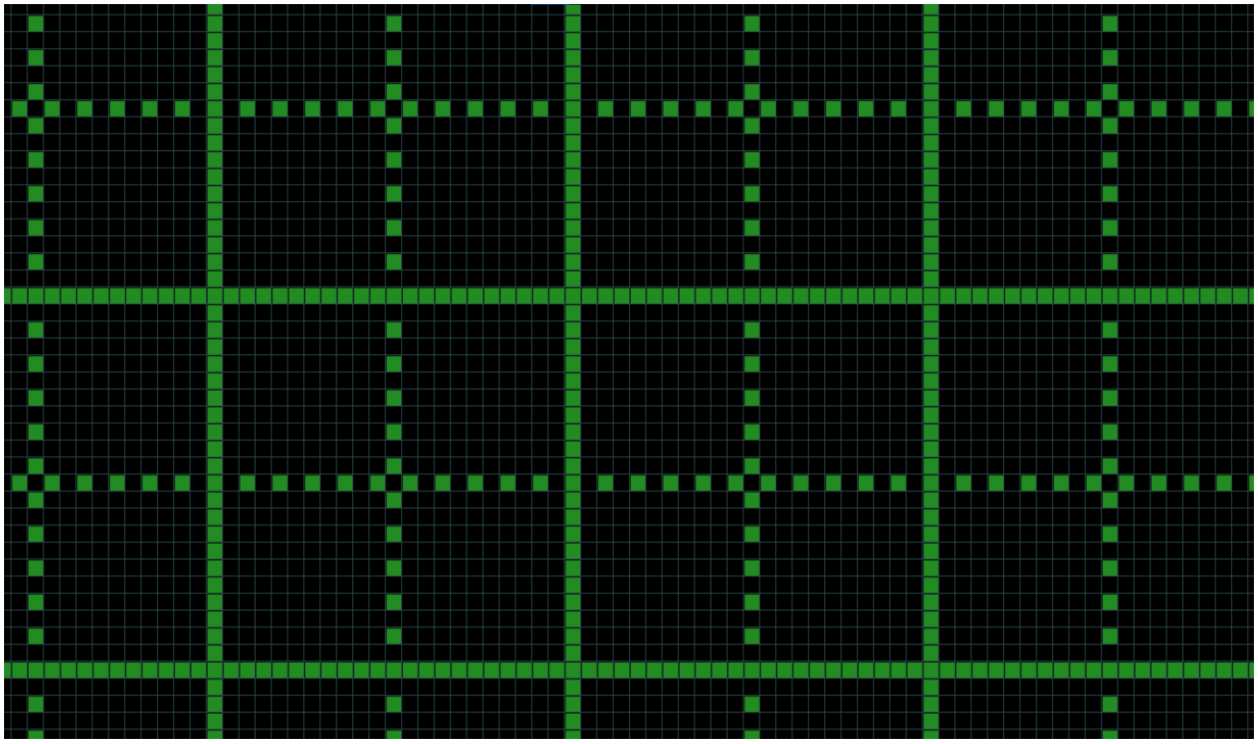

Рисунок 2.3 – Начальное состояние «вселенной»

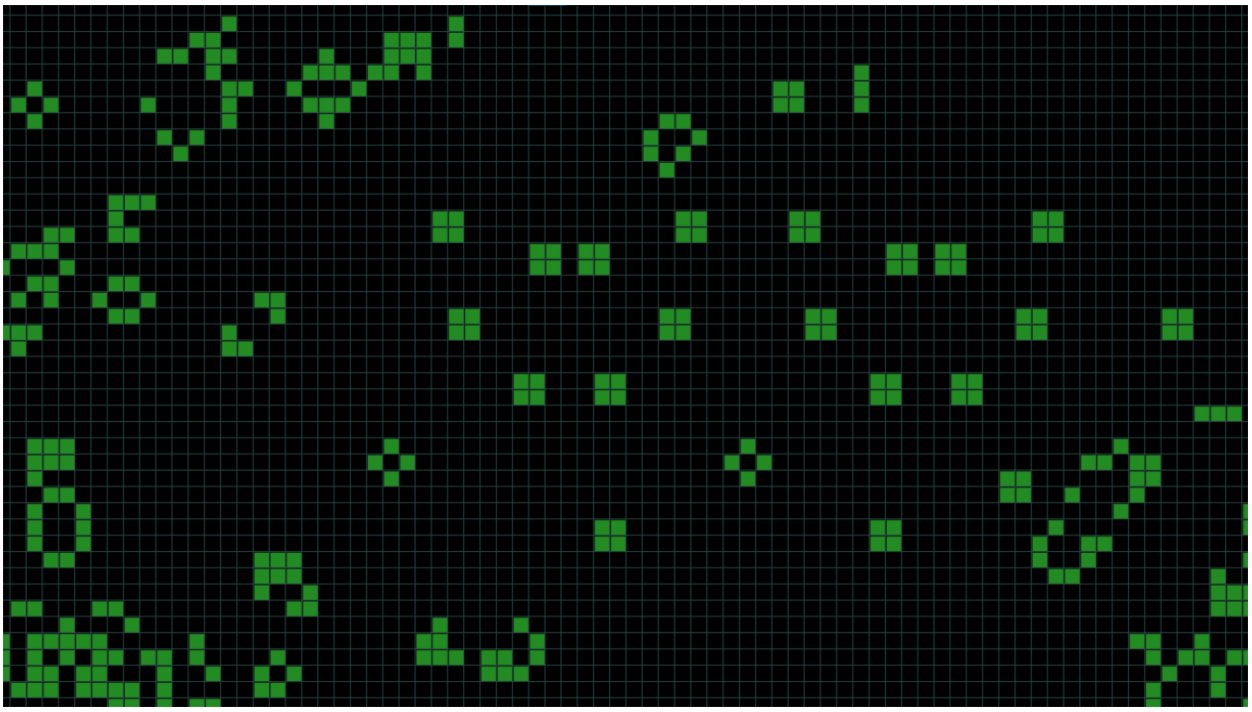

Рисунок 2.4 – Состояние «вселенной» спустя 100 поколений

Как можно заметить, в ходе развития популяции «живых» клеток, возникают устойчивые формы, такие как блок, ящик, каравай. Их расположение, равно как и положение просто «живых» клеток видится случайным, однако вряд ли с таким большим количеством нулевых клеток можно добиться хороших показателей ГПСЧ.

Модифицированная игра «Жизнь»

В предложенном методе генерации псевдослучайных чисел произведен следующий набор модификаций для игры «Жизнь»:

- Введены новые персонажи игры – «Зомби»;
- Введены новые правила игры с учетом новых персонажей

Начнем с первого – с «зомби» клеток. «Зомби» клетки имеют состояние равное 2. В сути своей они являются противниками «живых» клеток и между ними начинается «война». В ходе этой «войны» противники не только захватывают новые «территории» в виде клеток, но и убивают или вербуют клетки соперников.

Благодаря такому нехитрому способу можно наблюдать интересную картину:

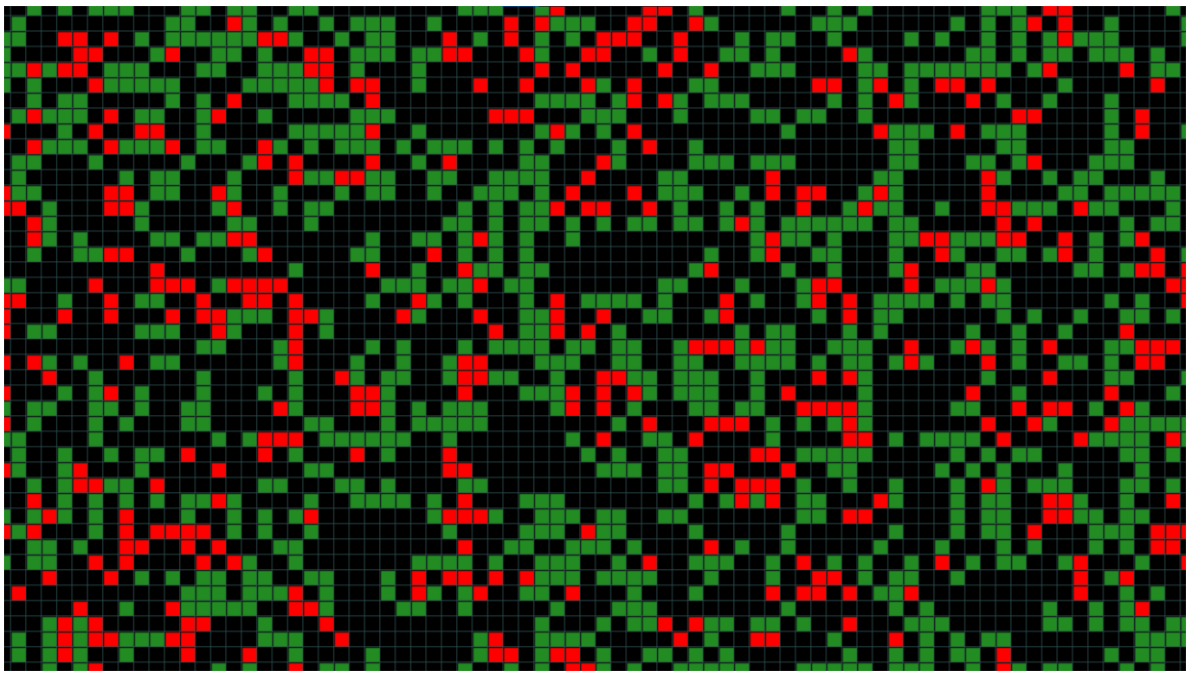

Рисунок 2.5 – Состояние «вселенной» спустя 100 поколений согласно новым условиям игры

При прочих равных, включая и правила поведения «живых» клеток, которые также остались неизменными, изменилось общее состояние клеточного автомата: уменьшилось количество «мертвых», нулевых клеток и увеличилось количество клеток, занятых каким-либо значением (1 или 2), даже визуально полученный рисунок кажется более случайным и определить какую-либо закономерность становится куда сложнее.

По итогу, такими простыми манипуляциями из такого клеточного автомата можно получать последовательности не бинарные, но тернарные.

2.4 Выбор начального положения клеток

От выбора начально положения немертвых клеток зависит то, как долго будет жить «вселенная», как быстро будет получена достаточно случайный результирующий рисунок и насколько случайным он окажется. К сожалению, данная проблема не решается теоретически, а потому выбор будет производиться исходя из результатов эмпирических исследований.

Далее будут показаны некоторые из вариантов начальной конфигурации клеточного автомата и ход их «жизни» от зарождения вселенной до ее «развития».

1. Расположение «живых» клеток крестом:

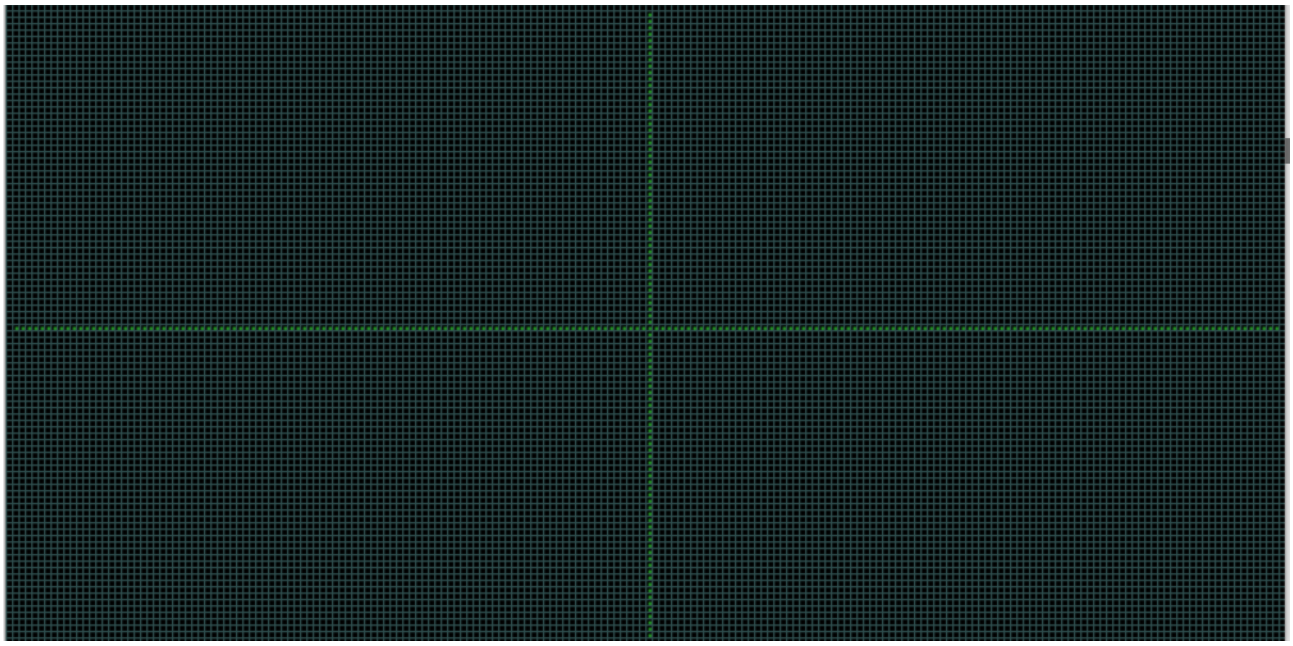

Рисунок 2.5 – Вариант 1, поколение 0

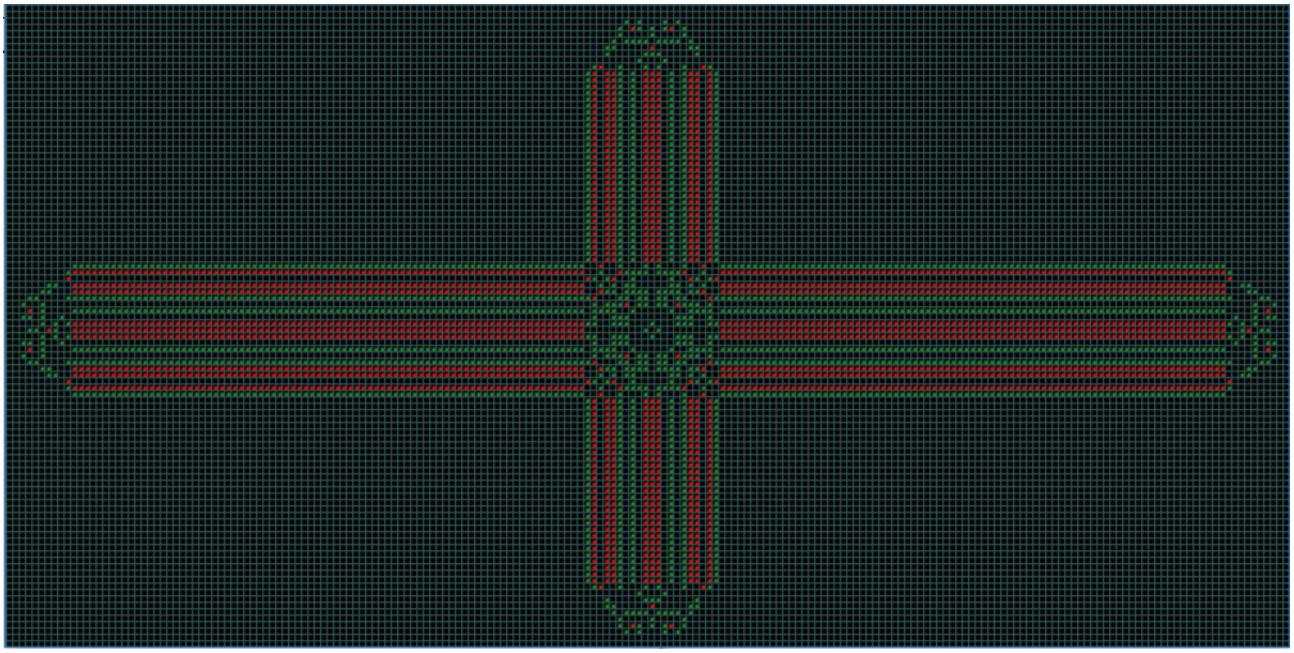

Рисунок 2.6 – Вариант 1, поколение 10

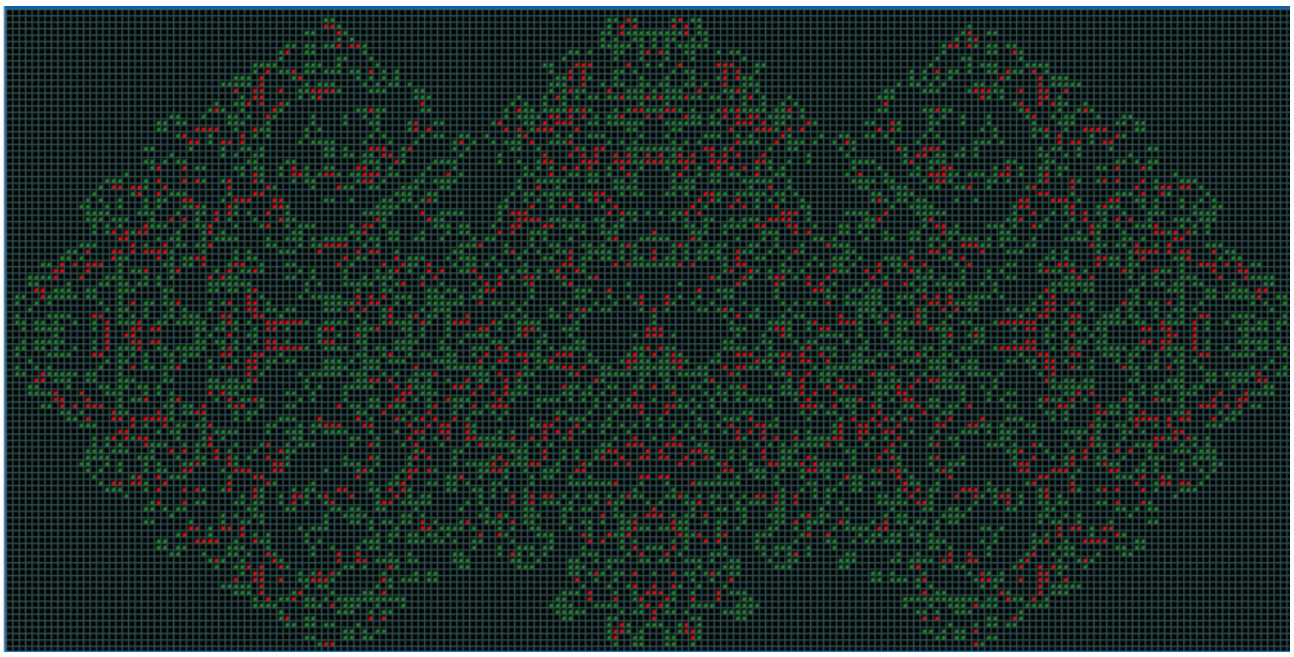

Рисунок 2.7 – Вариант 1, поколение 50

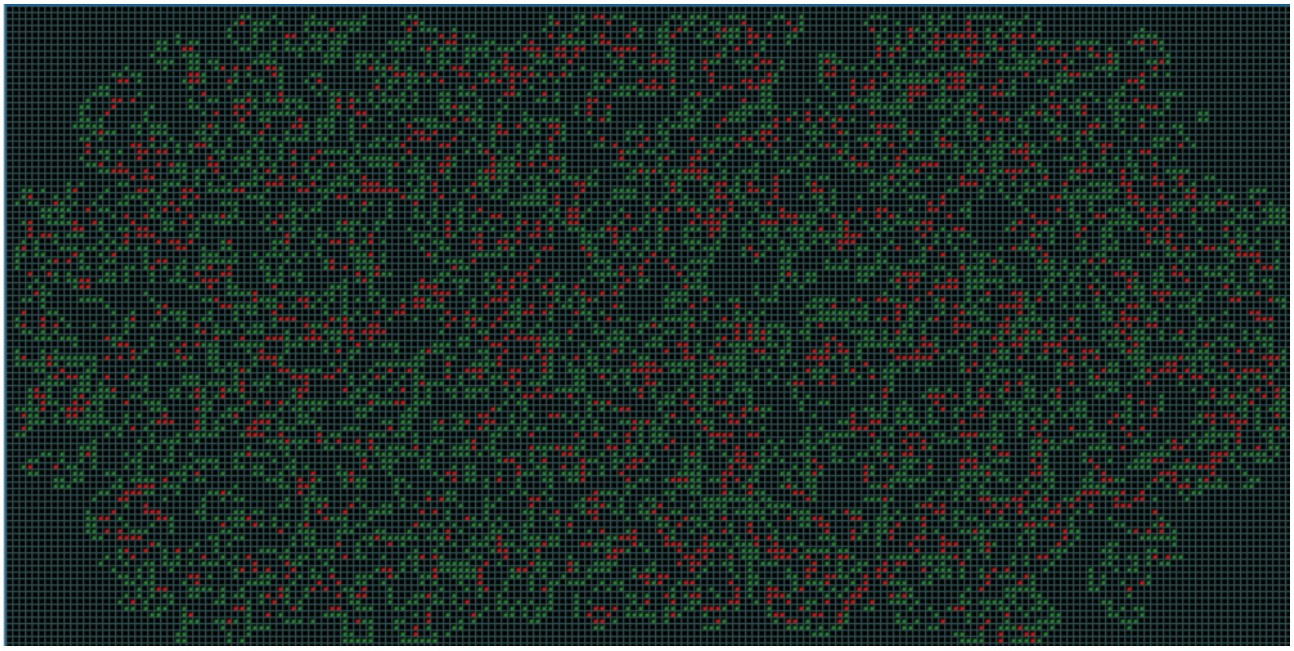

Рисунок 2.8 – Вариант 1, поколение 100

В рисунках 2.5 – 2.8 видно, как по мере развития «вселенная» становится все менее симметричной и все более случайной. Однако для того, чтобы дождаться несимметричной картины, понадобилось около 100 поколений, что является достаточно затратным процессом.

2. Расположение «сеткой»

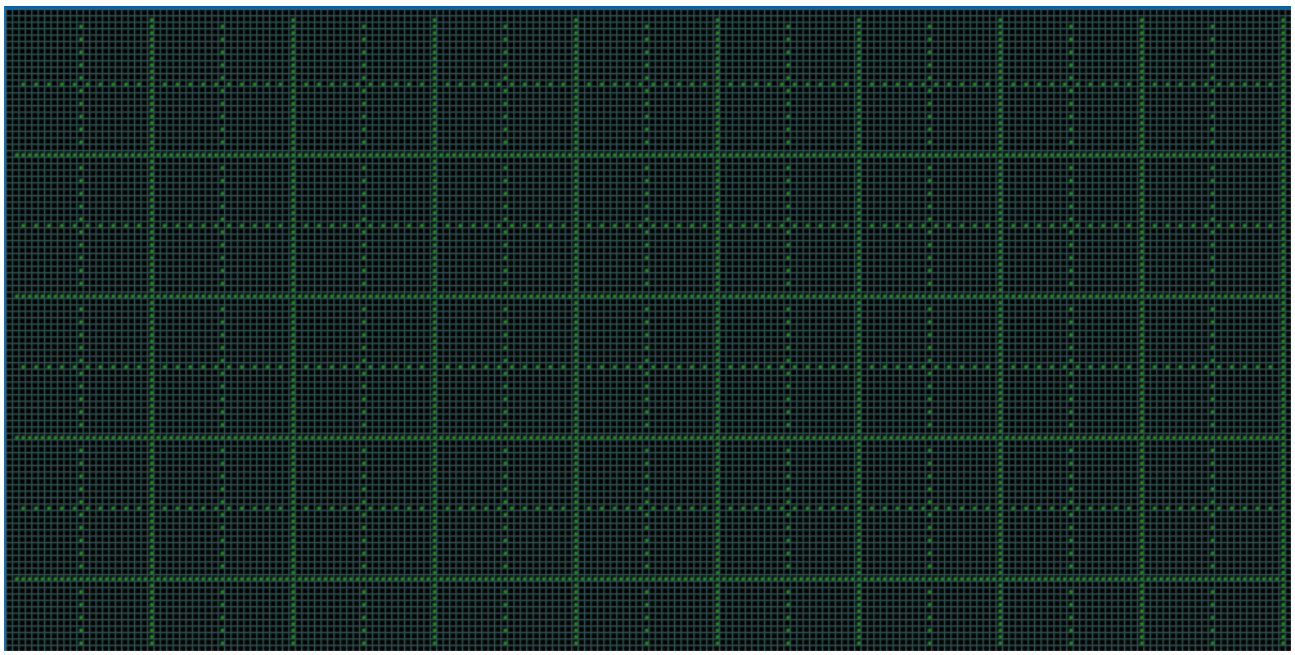

Рисунок 2.9 – Вариант 2, поколение 0

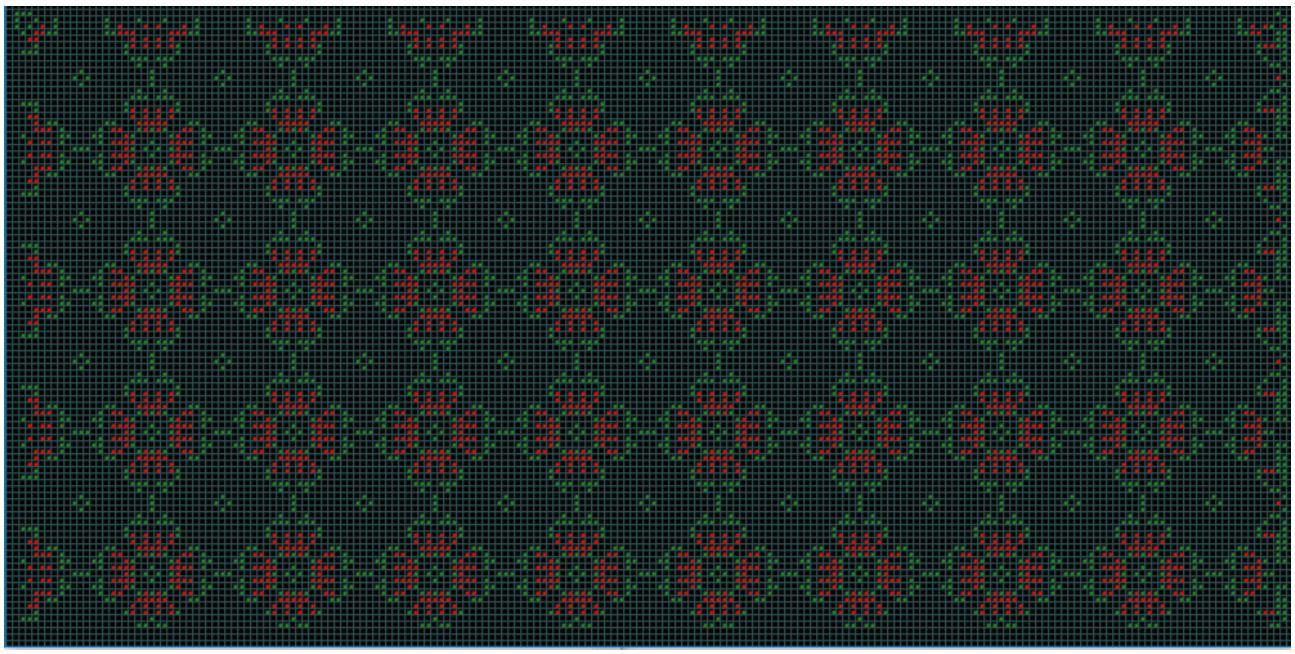

Рисунок 2.10 – Вариант 2, поколение 5

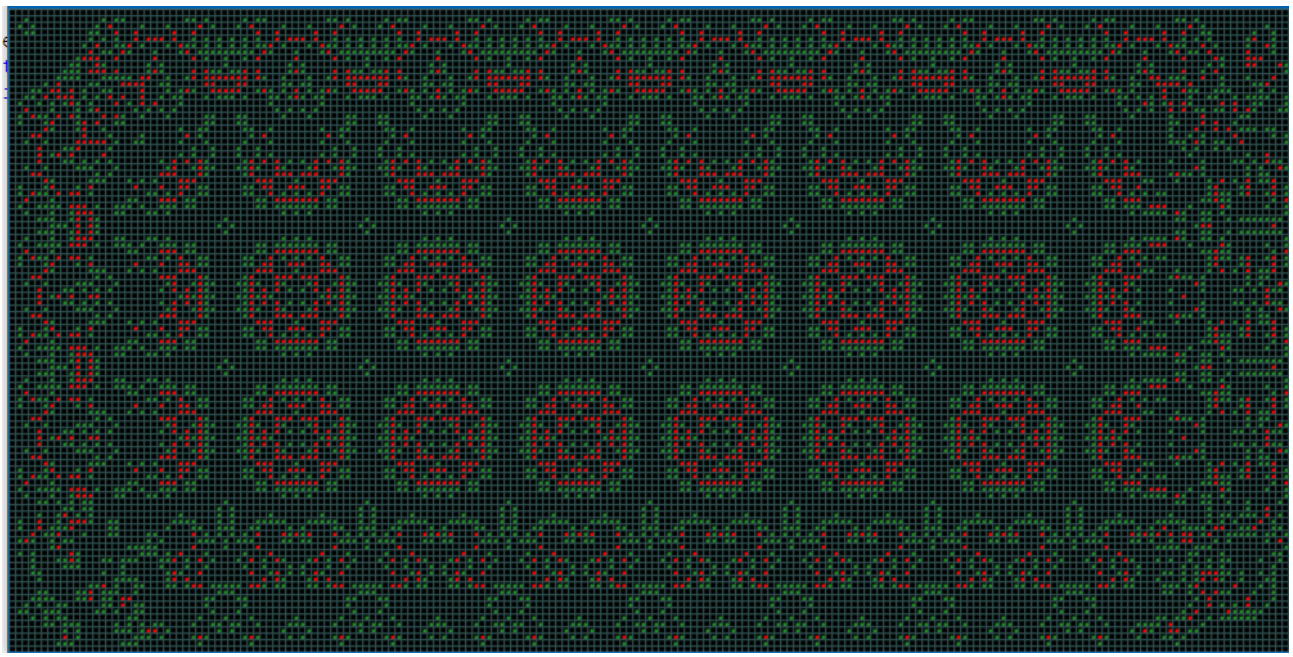

Рисунок 2.11 – Вариант 2, поколение 25

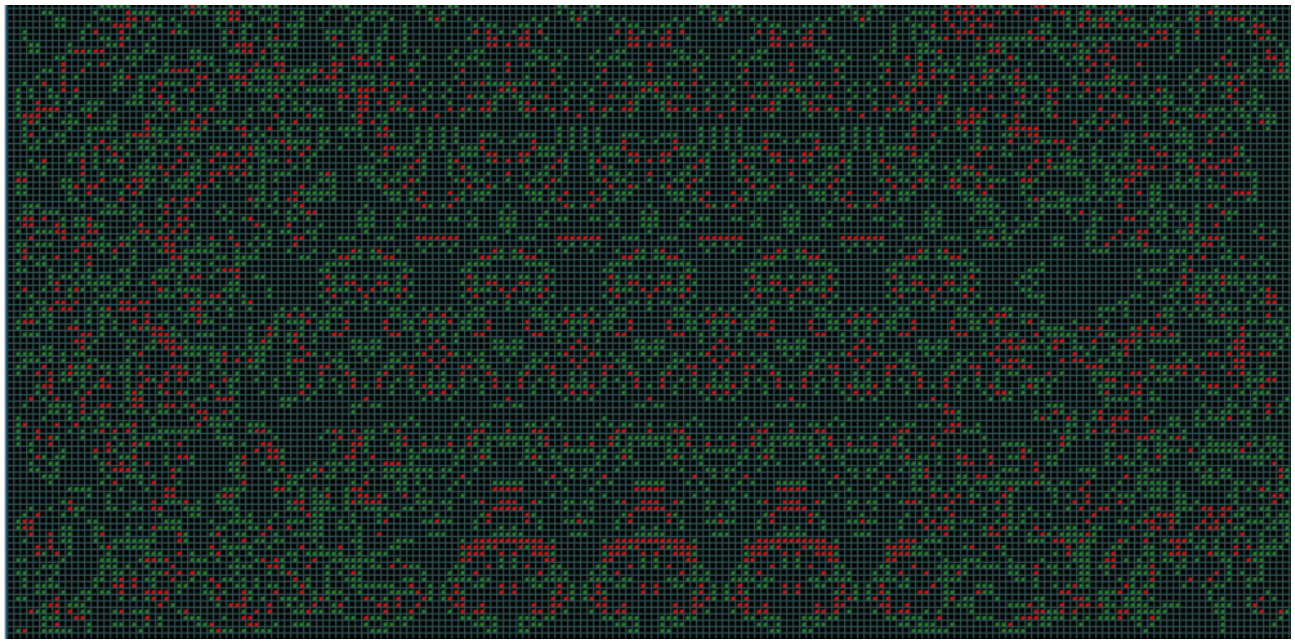

Рисунок 2.12 – Вариант 2, поколение 65

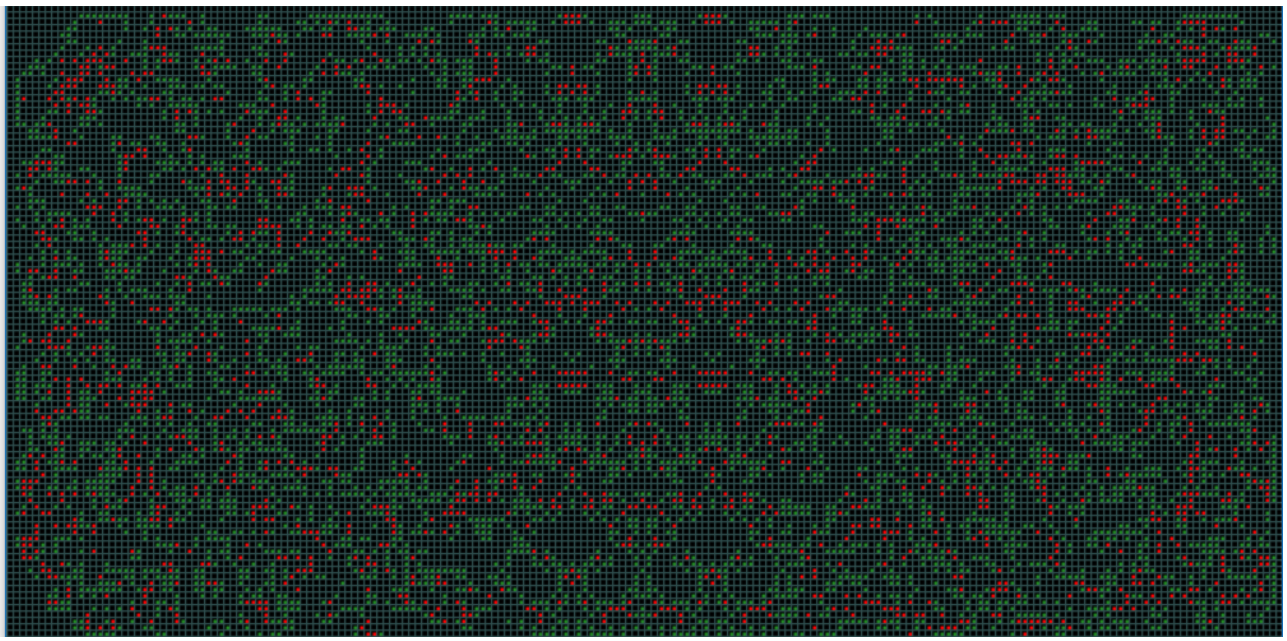

Рисунок 2.13 – Вариант 2, поколение 85

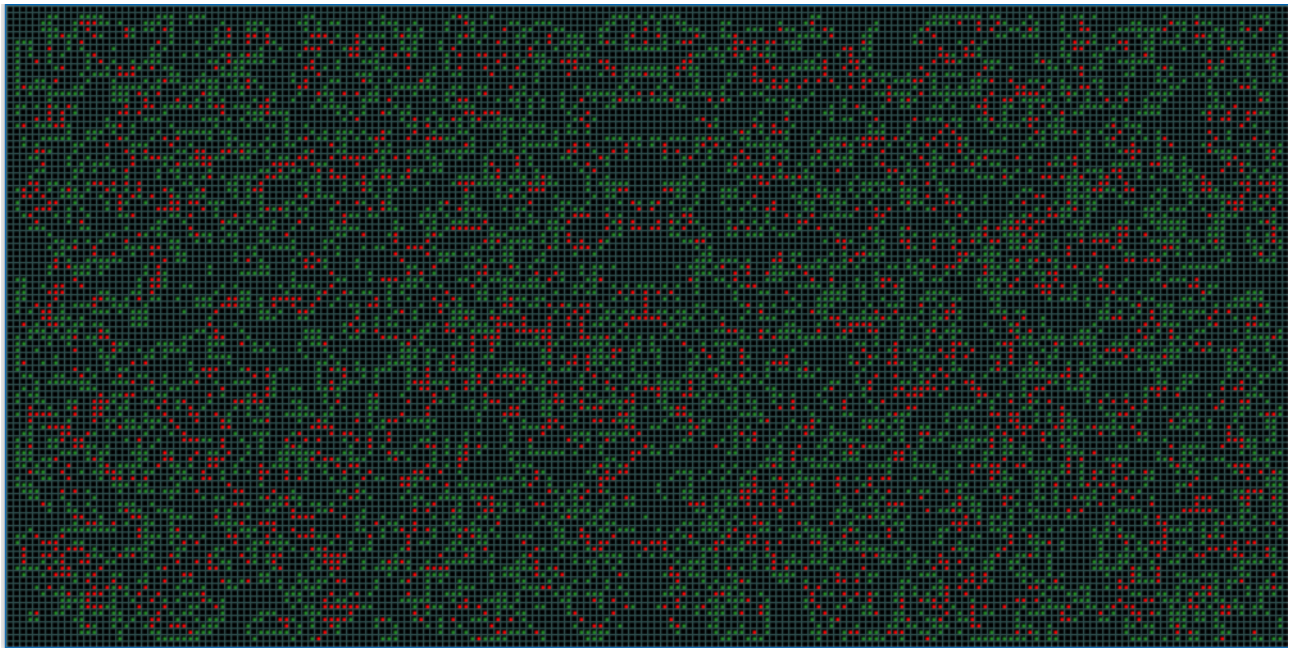

Рисунок 2.13 – Вариант 2, поколение 100

Второй вариант из симметричного рисунка, напоминающего персидский ковер, с каждым поколением все больше и больше превращался в полотно случайно расставленных «живых» и «зомби» персонажей, причем случайность возникает от краев поля к ее центру. Тем не менее если приглядеться, то вплоть до 85 поколения можно наблюдать в некоторых участках симметричное изображение, и даже в сотом поколении в некоторых малых участках также симметричность присутствует. Тем не менее, в отличии от первого варианта,

данная конфигурация покрывает все поле, в то время как первый вариант, включая такое же сотое поколение, оставляет пустыми все четыре угла «вселенной».

3. «Живые» клетки на четных позициях

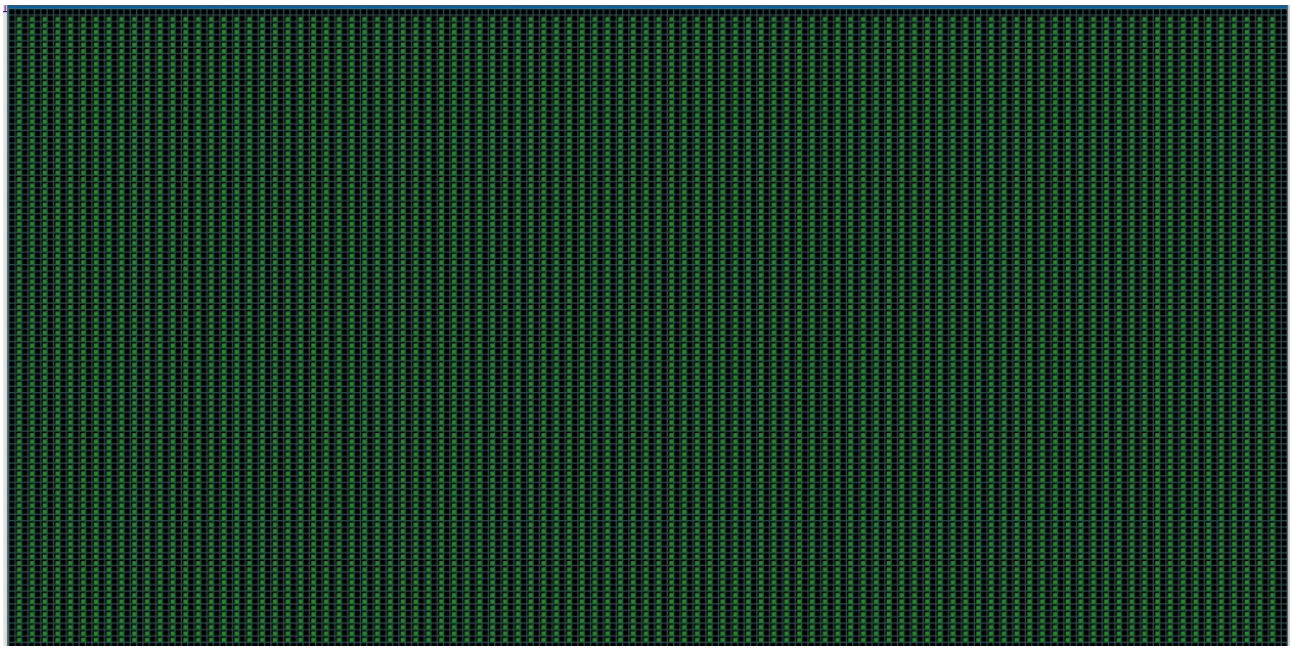

Рисунок 2.14 – Вариант 3, поколение 0

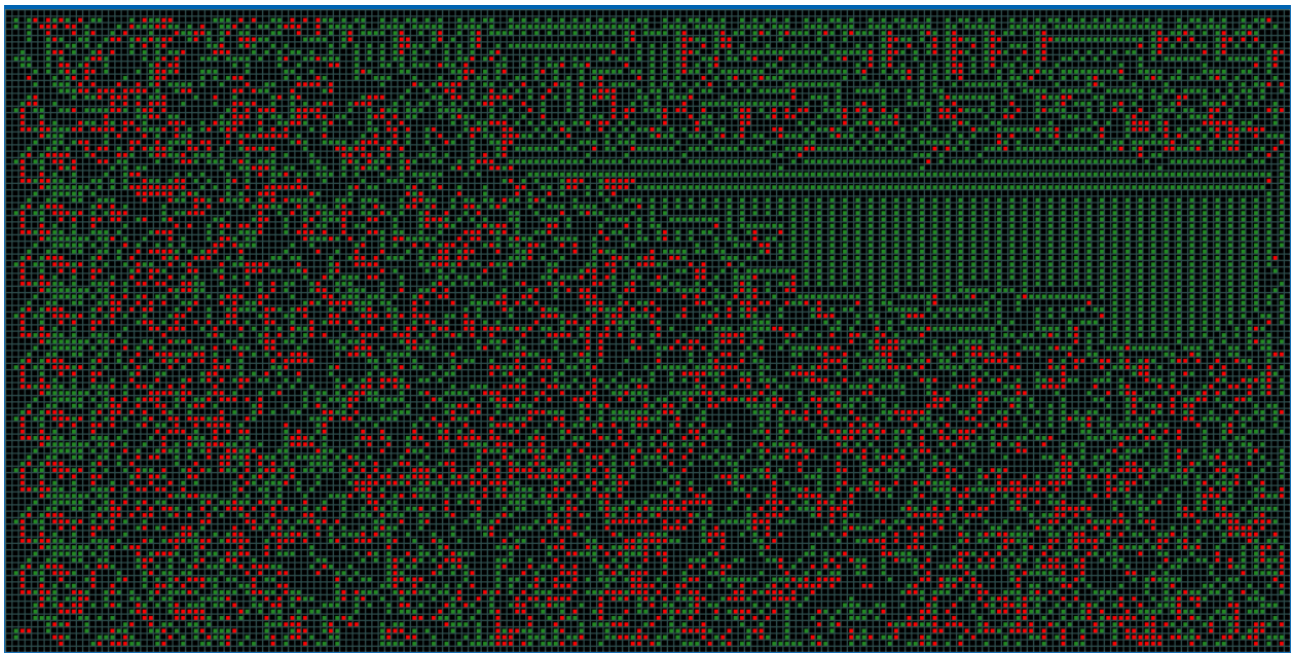

Рисунок 2.15 – Вариант 3, поколение 5

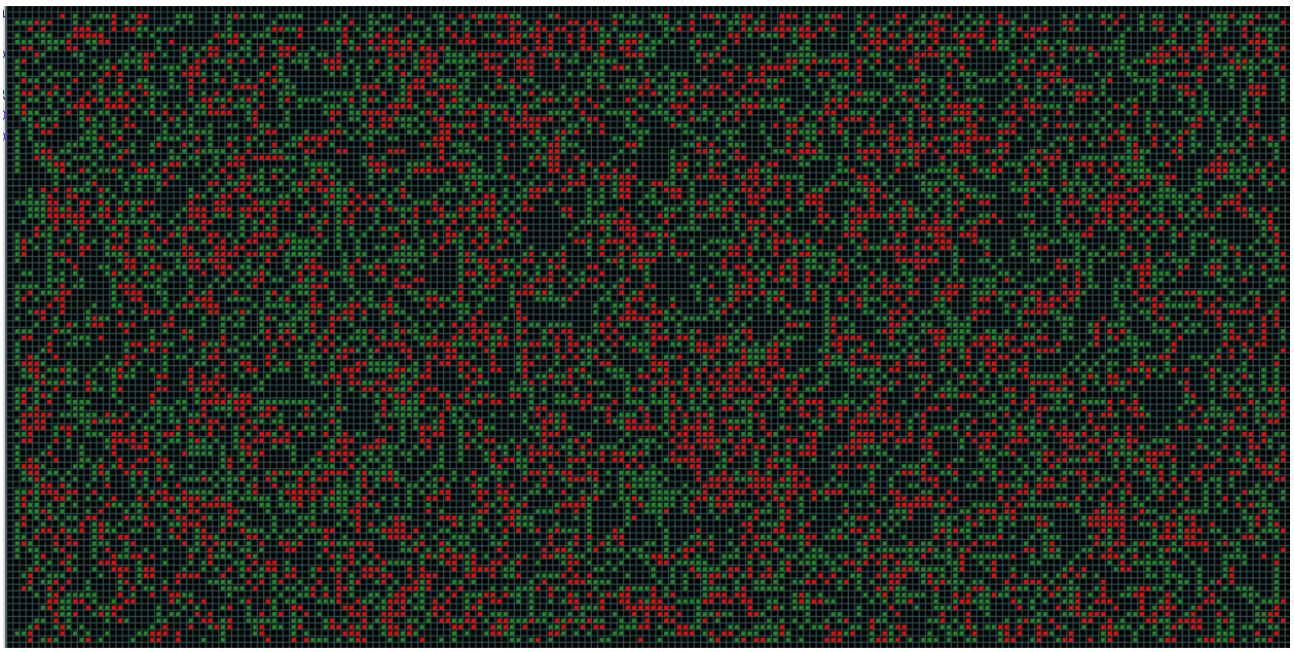

Рисунок 2.16 – Вариант 3, поколение 15

Здесь можно увидеть значительное сокращение времени, при котором можно получить случайную картину. Уже к 5 поколению большая часть полотна клеточного автомата заполнена случайно расположенными значениями, а к 15 поколению и вовсе все полотно. Однако можно еще немного ускорить данный процесс.

4. «Зомби» клетки на четных позициях

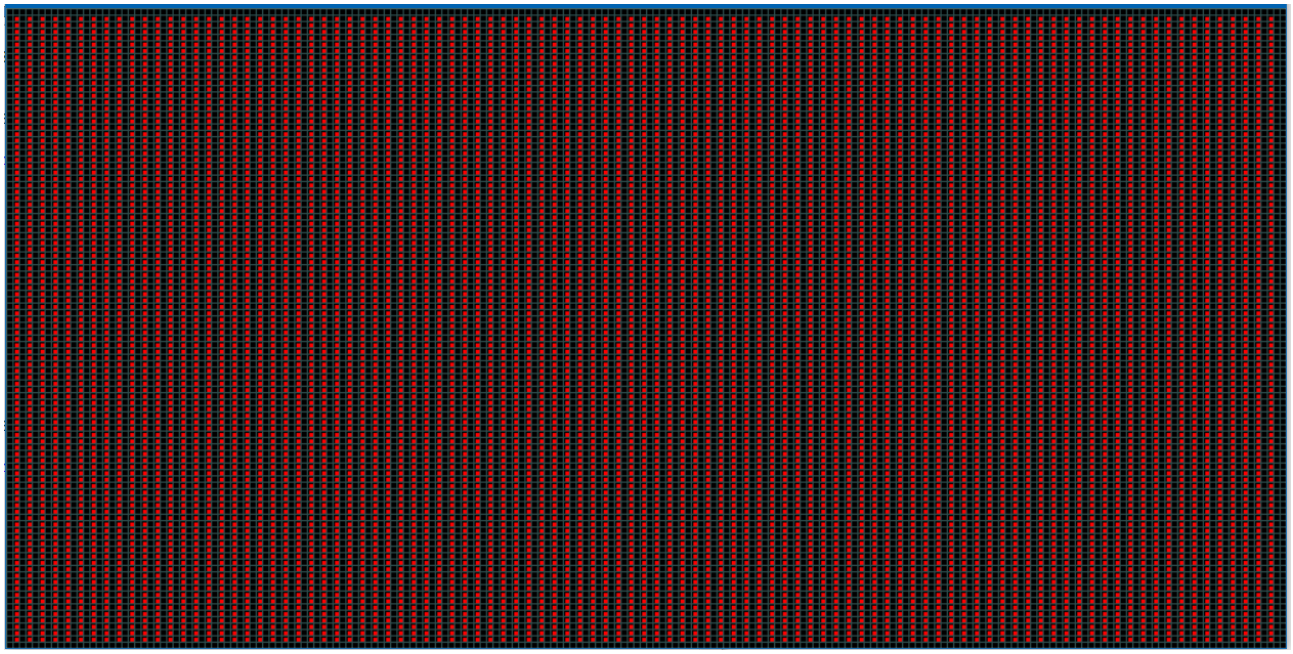

Рисунок 2.17 – Вариант 4, поколение 0

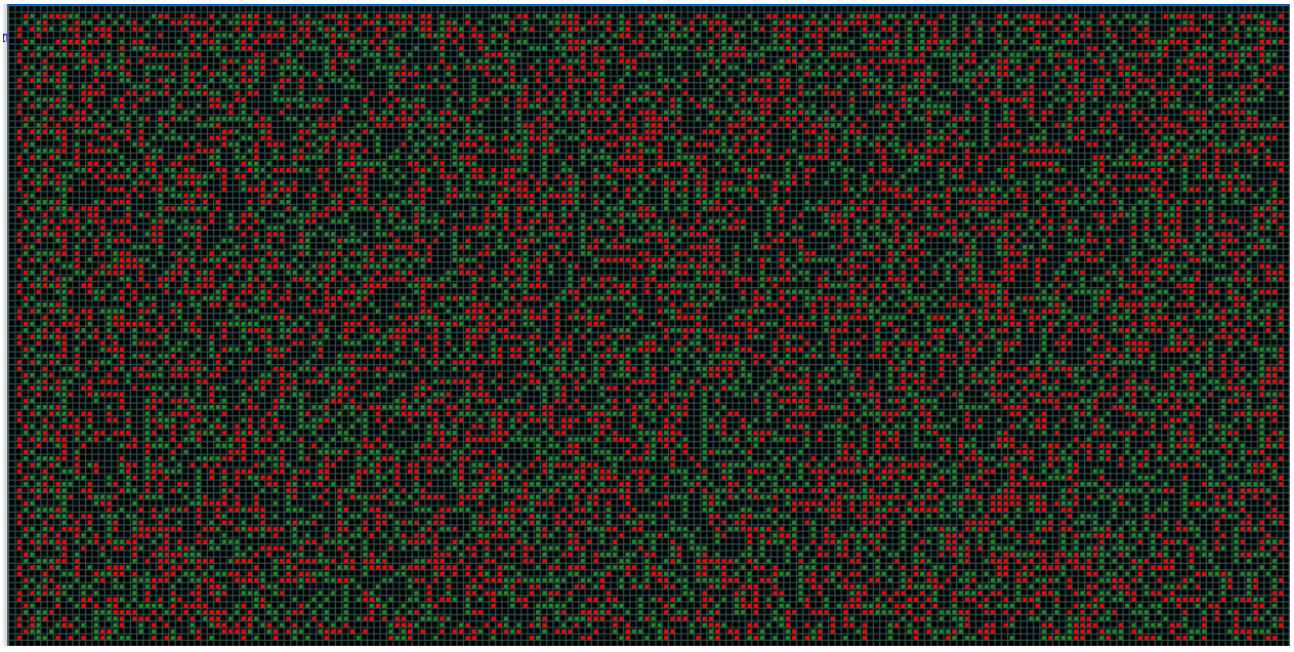

Рисунок 2.18 – Вариант 4, поколение 3

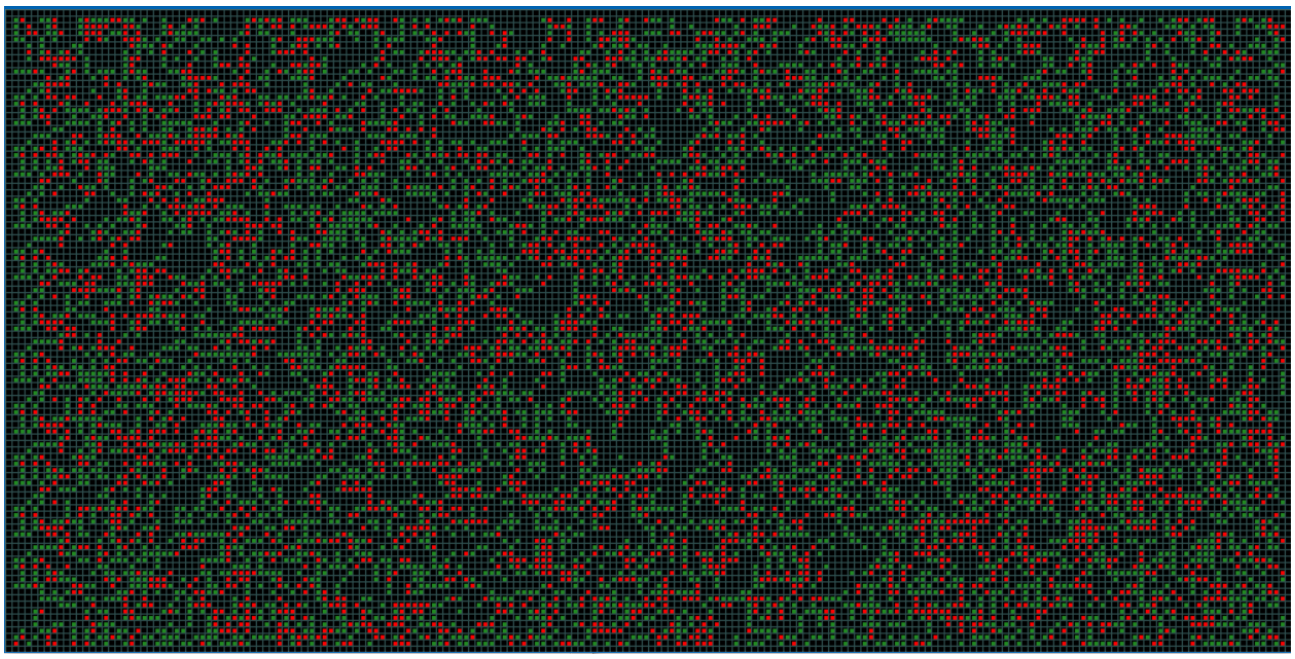

Рисунок 2.16 – Вариант 3, поколение 5

Заменив, «живые» клетки «мертвыми», удалось еще больше ускорить время, за которое достигается видимая случайность. Уже к 3 поколению получается достаточно хороший с визуальной точки результат.

Данную конфигурацию начального состояния клеточного автомата и будем рассматривать в качестве исследуемой.

2.5 Выбор продолжительности «Жизни»

Исходя из предыдущего пункта, можно понять, что при разных конфигурациях, а именно при разных размерах клеточного автомата и разных начальных состояниях клеточного автомата, требуется разная продолжительность «жизни». Поэтому при выборе стоит опираться на то, как быстро может быть достигнута хотя бы видимая случайность. Так как был выбран 4 вариант конфигурации, то в дальнейшем в исследовании будет рассматриваться результат после 5 поколений.

2.6 Вывод

В разделе проведена формализация задачи. Были представлены диаграммы, уточняющие суть представляемого метода. Были приведены основные элементы, лежащие в основе предлагаемого метода генерации псевдослучайных чисел, их модификация. Дано сравнение модифицированной и классической игр «Жизнь» и объяснение причин модификации. Также приведено описание проблем, которые возникли и возникают появляются при работе с данным генераторами псевдослучайных чисел, как например выбор начального заполнения клеточного автомата, выбор числа поколений, предложены способы их решения.

3 Технологический раздел

В технологическом разделе описан выбор способа реализации метода генерации псевдослучайных чисел на основе клеточного автомата, а также выбор средств реализации. Описаны форматы входных и выходных данных, используемые библиотеки и реализация подсистем. Приведены примеры реализации структур данных и описаны правила описания модулей для загрузки в систему. Описан предоставляемый функционал и интерфейс.

3.1 Обоснование выбора языка, среды программирования

Для реализации метода генерации псевдослучайных чисел на основе клеточного автомата «Жизнь» был выбран язык программирования Python 3.8, так как для него существует огромное количество различных библиотек, от математических до графических, облегчающих работу программиста и расширяющих его возможности. Данный язык имеет ряд преимуществ, которые и повлияли на выбор:

- Широкое распространение. Python используется во многих областях, включая научные и инженерные вычисления, веб-приложения и анализ данных.
- Простота и удобство в использовании. Python предлагает чистый и понятный синтаксис, что делает код легким для чтения и изменения.
- Большое количество библиотек. Python имеет широкий выбор библиотек для научных вычислений, включая numpy и scipy, которые предоставляют инструменты для работы с матрицами, числами, массивами и другими типами данных.
- Открытый исходный код. Python – это программное обеспечение с открытым исходным кодом, что означает, что сообщество

разработчиков может вносить свой вклад в улучшение и расширение языка.

- Динамическая типизация данных ускоряет разработку.
- Python содержит встроенный сборщик мусора, благодаря чему отсутствует утечка памяти. Также он используется в качестве источника энтропии.

Python является интерпретируемым языком, поэтому ПО запустится везде, где есть этот интерпретатор и необходимые библиотеки. В качестве среды разработки в данной работе использовалась стандартная IDLE Python. Эта среда предоставляется вместе с интерпретатором и в этой связи использование указанного инструмента является достаточным для эффективной разработки систем с невысоким уровнем сложности, при этом обеспечивая возможность независимого отслеживания разработчиком.

3.2 Описание используемых модулей

3.2.1 Используемые библиотеки

Как уже говорилось, язык программирования Python богат на различные библиотеки, расширяющие возможности программиста. Так и в реализации предложенного генератора псевдослучайных чисел используется ряд библиотек, описание которых и будет приведено в этой главе.

- tkinter - стандартный интерфейс Python для инструментария Tk GUI. Как Tk, так и tkinter доступны на большинстве платформ Unix, а также в системах Windows. Tkinter обеспечивает создание и работу графического интерфейса.

- os – модуль, обеспечивающий портативный способ использования функций, зависящих от операционной системы. Используется для открытия файла, содержащего результат работы генератора
- random – реализует генераторы псевдослучайных чисел для различных дистрибутивов. Данная библиотека не используется для в работе предложенного метода генерации случайных чисел, однако она используется в исследовании, в частности используется подкласс randint, который выдает псевдослучайное число в заданном диапазоне.
- pygame – набор модулей (библиотек), предназначенный для написания компьютерных игр и мультимедиа-приложений. Pygame базируется на мультимедийной библиотеке SDL. Имеет большие графические возможности и используется для реализации графического представления работы генератора псевдослучайных чисел
- math – стандартный модуль, обеспечивающий доступ к математическим функциям, определенным стандартом C. Не поддерживает комплексные числа.

3.2.2 Используемые модули

В этой реализации программного обеспечения имеется два модуля с исходным кодом: один обеспечивает интерфейс пользователя, в то время как другой реализует работу генератора псевдослучайных чисел. Поэтому далее мы будем сосредотачиваться на функциях и блоках кода, отвечающих за работу генератора псевдослучайных чисел.

Для работы метода генерации псевдослучайных чисел в разработанном ПО реализованы следующие модули:

- Заполнение карты клеточного автомата;

- Воспроизведение игры «Жизнь»;
- Отбор строки;
- Вывод результатов.

3.2.3 Реализация модулей

На листингах 3.1 – 3.3 представлена программная реализация клеточного автомата «Жизнь» с введенными модификациями и новыми правилами игры.

Заполнение карты клеточного автомата

Листинг 3.1 – Обход клеток мира клеточного автомата.

```
def life(W, H, next_field, current_field, tries):
    for k in range(tries):
        for x in range(1, W - 1):
            for y in range(1, H - 1):
                next_field[y][x] = check_cell(current_field, x, y,W,H)

        current_field = [*next_field]

    return current_field
```

Листинг 3.2 – Заполнение карты клеточного автомата. Проверка каждой ячейки согласно модифицированным правилам игры «Жизнь»

```
def fill (W, H):
    current_field = next_field = [[0 for i in range(W)] for j in range(H)]
    entrp = 0
    while entrp == 0:
        entrp = round((time.process_time_ns() * timeit.timeit()))#% (W*H)/2
        entrp = entrp % 10000
    print(entrp)
    current_field = [[2 if i == W // entrp or j == H // entrp else 0 for i in range(W)] for j in range(H)]
    for j in range (W):
        for i in range(H):
            if i != 0 and j!= 0:
                if entrp % i == 0 or entrp % j== 0:
                    current_field[i][j] = i % 3
                elif i > j and not (2 * i + j) % 4:
                    current_field[i][j] = 2
                else:
                    current_field[i][j-i] = 1
    return next_field, current_field
```

Воспроизведение игры «Жизнь»

Листинг 3.3 – Проверка каждой ячейки согласно модифицированным правилам игры «Жизнь»

```
def check_cell(current_field, x, y,W, H):
    count = 0

    for j in range(y - 1, y + 2):
        for i in range(x - 1, x + 2):
            if current_field[j % H][i % W] != 0:
                count += 1
    # Zombie
    if current_field[y][x] == 2:
        count -= 1
        if count == 2 or count == 4:
            return 2
        return 0
    else:
        if count == 6:
            return 2

    # Alive
    if current_field[y][x] == 0:
        count -= 1
        if count == 2 or count == 3:
            return 1
        return 0
    else:
        if count == 3:
            return 1
        return 0
```

Данные модули представлены на листингах приложения А.

Отбор строки

Ввиду того, что клеточный автомат представлен в виде матрицы, то в ходе работы реализованного программного обеспечения матрица заполняется цифрами 0, 1 и 2.

В зависимости от требований к применяемым задачам, а также в связи с тем, что клеточный автомат можно масштабировать и рассматривать разными способами, как например собирать все строки в одну или работать со столбцами.

Кроме того, мир игры «Жизнь» можно представить и как тор, и как шар. Последнее и реализовано в разработанном программном обеспечении.

Результатом работы ГПСЧ является двоичная последовательность, сформированная из отдельно взятой строки матрицы. Выбор данной строки основан на сравнении сериальной корреляции каждой из строк матрицы «Жизни», после чего выбирается та, у которой значение наиболее приближенное к нулю.

Так как критерий сериальной корреляции работает с бинарными значениями, а каждая из строк матрицы представлена в виде последовательности тритов, то для разрешения этой проблемы используется функция конвертации последовательности тритов в последовательность битов.

Листинг 3.4 – Модуль выбора строки

```
def choise_line(current_field):
    summ = 0
    summax = 0
    masmax = []
    cmin = 1
    x = []
    masK = []
    masnum = []
    KK = 0
    for k in range(1,len(current_field)-1):
        x = current_field[k]
        masK.append(k)
        num = ''
        num3 = ''
        for j in x:
            num += str(j)
            num3 += str(j)

        if sum(x) == 0:
            continue
        num = convert_base(num, 2, 3)
        num = list(num)

        c = Knut_test(num, 0)
        if c == '!!!':
            continue
        if abs(c) < cmin:
            masmax = num
            masnum = num3
            cmin = c
            KK = k
    return KK, masmax, masnum
```

Листинг 3.5 – Конвертер систем счислений

```
def convert_base(num, to_base, from_base):
    # first convert to decimal number
    n = int(num, from_base) if isinstance(num, str) else num
    # now convert decimal to 'to_base' base
    alphabet = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    res = ""
    while n > 0:
        n,m = divmod(n, to_base)
        res += alphabet[m]
    return res[::-1]
```

Листинг 3.6 – Тест сериальной корреляции

```
def Knut_test(U, flag):
    n = len(U)
    u2 = 0
    u_sum2 = 0
    # print(U)
    if isinstance(U[0], str):
        for i in range(len(U)):
            U[i] = int(U[i])

    u_sum2 = sum(U)**2
    U1U2 = 0
    for i in range(n-1):
        U1U2 += U[i]*U[i+1]
        u2 += U[i]**2
    C = (n*U1U2 - u_sum2)/(n*u2 - u_sum2)
    un = (-1)/(n-1)
    sigma = sqrt((n**2)/(((n-1)**2)*(n-2)))
    if flag == 1:
        if un - 2*sigma < C < un + 2*sigma:
            print('Хорошее значение - ', C)

        else:
            print('Нехорошее значение - ', C)

    return C
```

Данные модули представлены на листингах приложения Б.

Вывод результатов

Выходными данными данной реализации метода генерации псевдослучайных чисел является документ формата .txt, в котором находится бинарная последовательность псевдослучайных битов.

Листинг 3.6 – Тест сериальной корреляции

```
def print_field(field, W, H, pref):
     if H!= 0:
        f = open('nums'+pref+'.txt','w')
        for i in range(len(field)):
            buf = ''
            for j in range(len(field[i])):
                buf += str(field[i][j])
            f.write(buf+'\n')
        f.close()
    else:
        f = open('nums'+pref+'.txt','w')
        buf = ''
        for i in range(W):
            buf += str(field[i])+''
        f.write(buf)
        f.close()
```

Данные модули представлены на листингах приложения Б.

3.3 Реализация критериев для исследования

Подробно о критериях для исследования качественных характеристик будет сказано в исследовательском разделе. Здесь же листинг модулей, связующих реализации критериев для тестов NIST и DIEHARDER. Стоит отметить, что некоторые тесты обоих пакетов тестирования совпадают, в связи с чем был сокращен список тестов DIEHARDER, оставив только уникальные тесты.

Листинг 3.7 – модуль sp800_22_tests для пакета тестов NIST.

```
from __future__ import print_function

import argparse
import sys
import pickle


def read_bits_from_file(filename,bigendian):
    bitlist = list()
    if filename == None:
        f = sys.stdin
    else:
        f = open(filename, "rb")
    while True:
        bytes = f.read(16384)
        if bytes:
            for bytech in bytes:
                if sys.version_info > (3,0):
                    byte = bytech
```

```
                else:
                    byte = ord(bytech)
                for i in range(8):
                    if bigendian:
                        bit = (byte & 0x80) >> 7
                        byte = byte << 1
                    else:
                        bit = (byte >> i) & 1
                    bitlist.append(bit)
        else:
            break
    f.close()
    return bitlist

import argparse
import sys
parser = argparse.ArgumentParser(description='Test data for distinguishability
form random, using NIST SP800-22Rev1a algorithms.')
parser.add_argument('filename', type=str, nargs='?', help='Filename of binary
file to test')
parser.add_argument('--be', action='store_false',help='Treat data as big
endian bits within bytes. Defaults to little endian')
parser.add_argument('-t', '--testname', default=None,help='Select the test to
run. Defaults to running all tests. Use --list_tests to see the list')
parser.add_argument('--list_tests', action='store_true',help='Display the list
of tests')

args = parser.parse_args()

bigendian = args.be
filename = args.filename

testlist = [
        'monobit_test',
        'frequency_within_block_test',
        'runs_test',
        'longest_run_ones_in_a_block_test',
        'binary_matrix_rank_test',
        'dft_test',
        'non_overlapping_template_matching_test',
        'overlapping_template_matching_test',
        'maurers_universal_test',
        'linear_complexity_test',
        'serial_test',
        'approximate_entropy_test',
        'cumulative_sums_test',
        'random_excursion_test',
        'random_excursion_variant_test']

print("Tests of Distinguishability from Random")
if args.list_tests:
    for i,testname in zip(range(len(testlist)),testlist):
        print(str(i+1).ljust(4)+": "+testname)
    exit()

f = open('../nums.txt','r')
bits = ''

for i in f:
    bits += i
bits = list(bits)

for i in range(len(bits)):
    bits[i] = int(bits[i])
```

```
gotresult=False
if args.testname:
    if args.testname in testlist:
        m = __import__ ("sp800_22_"+args.testname)
        func = getattr(m,args.testname)
        print("TEST: %s" % args.testname)
        success,p,plist = func(bits)
        gotresult = True
        if success:
            print("PASS")
        else:
            print("FAIL")

        if p:
            print("P="+str(p))

        if plist:
            for pval in plist:
                print("P="+str(pval))
    else:
        print("Test name (%s) not known" % args.ttestname)
        exit()
else:
    results = list()

    for testname in testlist:
        print("TEST: %s" % testname)
        m = __import__ ("sp800_22_"+testname)
        func = getattr(m,testname)

        (success,p,plist) = func(bits)

        summary_name = testname
        if success:
            print("  PASS")
            summary_result = "PASS"
        else:
            print("  FAIL")
            summary_result = "FAIL"

        if p != None:
            print("  P="+str(p))
            summary_p = str(p)

        if plist != None:
            for pval in plist:
                print("P="+str(pval))
            summary_p = str(min(plist))

        results.append((summary_name,summary_p, summary_result))

    print()
    print("SUMMARY")
    print("-------")

    for result in results:
        (summary_name,summary_p, summary_result) = result
        print(summary_name.ljust(40),summary_p.ljust(18),summary_result)
```

Листинг 3.8 – модуль Dieharder_tests для пакета тестов Dieharder.

```
import math
from scipy.special import *
import numpy as np
import sys
sys.set_int_max_str_digits(0)


tests =    [
      rank_test,
      decoder_test,
      OSPOTest,
      LempelZivTest,
      IRWTest,
      RWTest,
      LCGTest,
      PiZeroTest,
      PiOneTest,
      PiTwoTest,
      PiThreeTest,
      SigmaZeroTest,
      SigmaOneTest,
      sigma_two_test,
      sigma_three_test]
i = 1
for test in tests:
    result = test(bits)
    print(result)
```

Данный модуль представлены на листингах приложения В.

3.4 Вывод

В разделе были приведены элементы конкретной реализации генератора псевдослучайных чисел. Было приведено описание работы функций и логических единиц приведенной реализации. Также приведено техническое описание разработанных подсистем, причины их использования, предоставлены примеры из исходного кода разработанного программного обеспечения.

4 Исследовательский раздел

В исследовательском разделе приведено исследование эффективности предложенного метода генерации псевдослучайных чисел. Проведен анализ качественно-временных характеристик в сравнении с альтернативными методами генерации псевдослучайных чисел. Также исследован вопрос криптографической стойкости предложенного метода и применимости предложенного метода к задачам криптографии, моделирования и прочих.

В текущем разделе также приведено описание проведенных экспериментов, тестовых результатов и пояснения к ним.

4.1 Описание среды и устройства для исследования

В данном разделе рассмотрены характеристики устройства и характеристики среды, в которых будет проходить дальнейшее исследование. Стоит отметить, что в действительности, ввиду просто ты реализации предложенного метода генерации псевдослучайных чисел, само программное обеспечение генератора не требует каких-либо серьезных характеристик запускающего его ЭВМ.

Характеристики устройства:

- операционная система Windows 11 Pro (сборка 22621);
- Процессор Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 1801 МГц, ядер: 4, логических процессоров: 8;
- ОЗУ 8Gb DDR3.

Характеристики среды:

- интерпретатор Python 3.11.2 x64;

Библиотеки, необходимые для запуска:

- pygame – для визуализации работы ГПСЧ;

- sys – используется для снятия ограничений на длину строкового представления целого числа. [5]

4.2 Пример работы

Устройства и программные обеспечения для генерации псевдослучайных чисел подразумеваются как средства, минимально использующие дополнительные ресурсы, библиотеки. От них не требуется визуальное представление, а лишь возможность вызова соответствующего модуля по запросу.

Тем не менее для демонстрации работы предложенного метода, создан графический интерфейс, с помощью которого можно рассмотреть зависимость результата от ввода входных параметров.

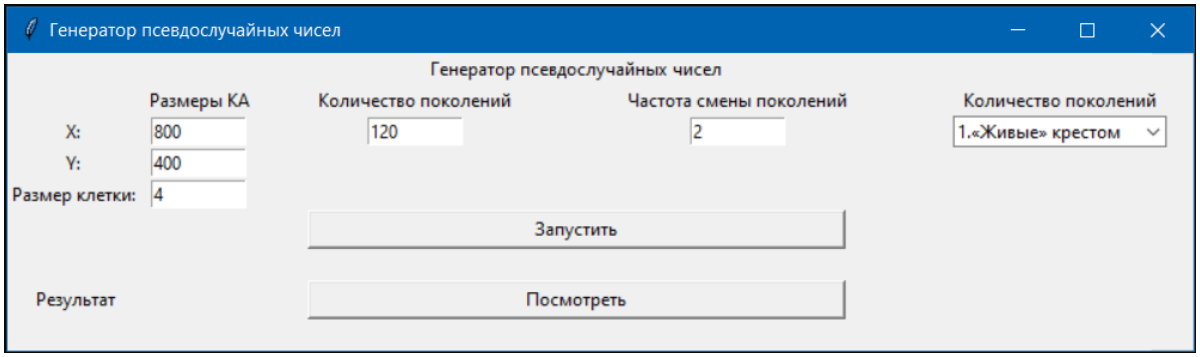

Рисунок 4.1 – Начальное окно демонстрационного приложения

Также реализована демонстрация самой модифицированной игры «Жизнь». Пример демонстрации представлен на рисунке 4.2. Можно видеть карту игры «Жизнь» в виде сетчатого поля. Клетки поля закрашены в 3 цвета: зеленые клетки отображают «живые» клетки, черные - «мертвых», а красные - это «зомби» клетки.

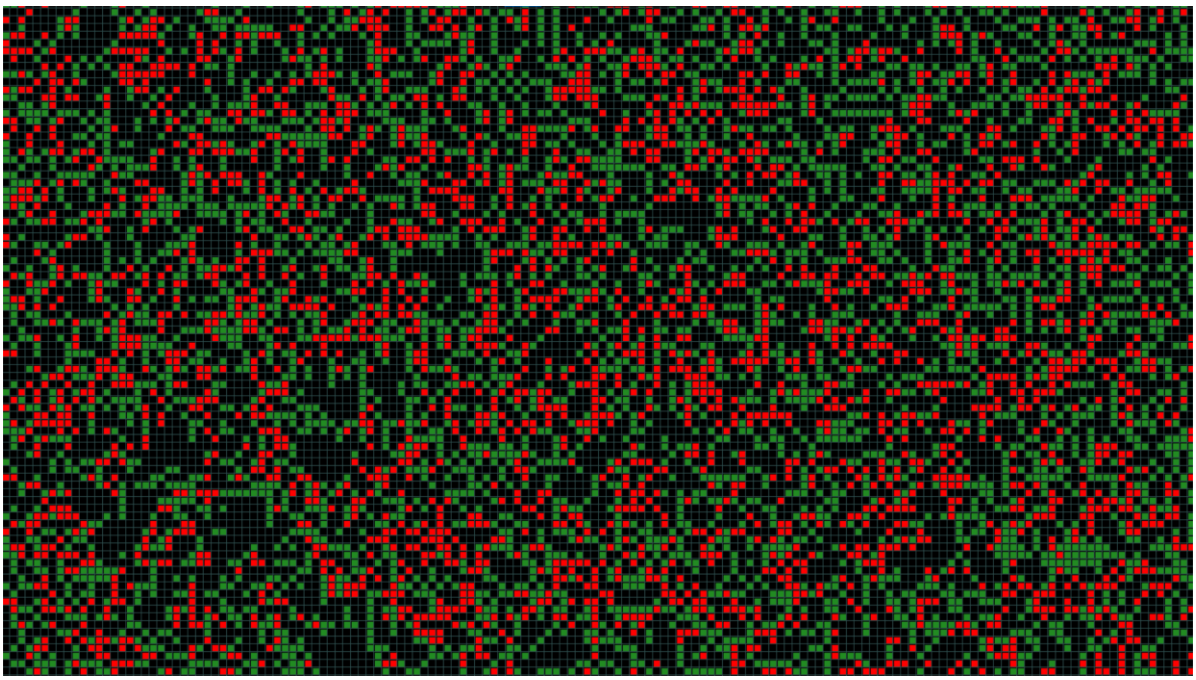

Рисунок 4.2 – Модифицированная игра «Жизнь»

Выходным результатом программного обеспечения является значение одной из строк клеточного автомата, которое записывается в текстовый файл формата .txt с названием nums. Для просмотра файла предусмотрена кнопка «посмотреть» в начальном окне приложения.

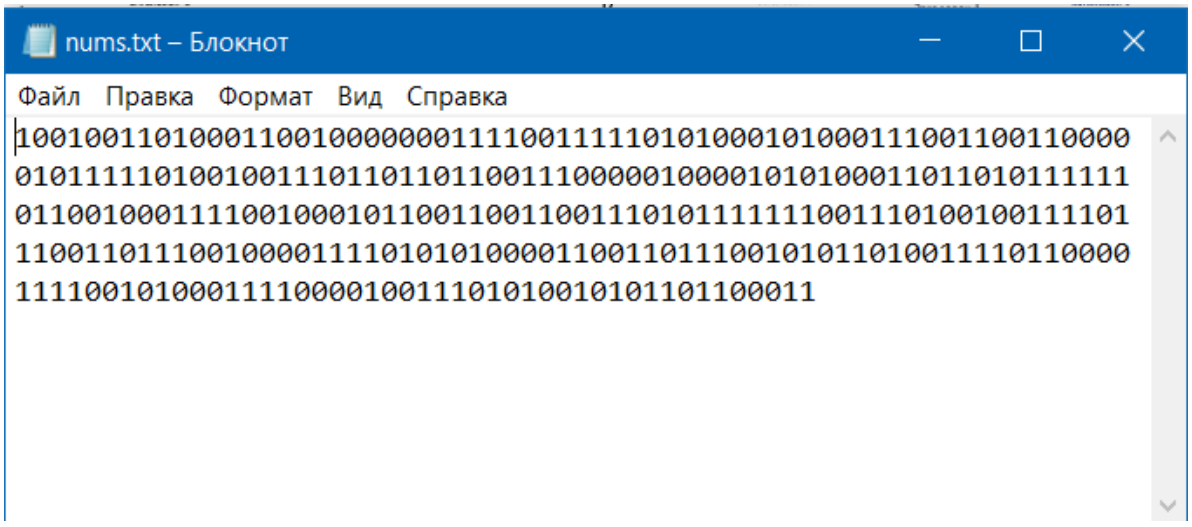

Рисунок 4.3 – Результат работы клеточного автомата

Данные в файле представлены в виде последовательности битов, а не тритов, как должно быть исходя из особенностей спроектированного клеточного автомата. Это сделано с целью дальнейшего исследования на оценку качества. Критерии качества описаны в следующих подразделах.

4.3 Постановка экспериментов

Для исследования предложенного метода генерации псевдослучайных чисел, основанного на технологии клеточных автоматов, следует прежде всего ответить на несколько вопросов:

- Как проверить случайность выходных данных предложенного метода?
- Если выходные данные предложенного метода действительно случайны, являются ли выдаваемые ею результаты криптографически стойкими?
- Для каких задач применим предложенный метод генерации псевдослучайных чисел?

Для проверки случайности результатов разрабатываемых методов генерации пседовслучайных чисел используются статистические тесты, которые и проверяют качество работы ГПСЧ. Среди таких можно выделить пакеты тестирования NIST и Diehard.

Однако классические вариации вышеупомянутых тестов не подходят для проверки криптографических свойств ГПСЧ, в связи чем данные пакеты тестирований были расширены и усложнены и получили названия NIST SP-800-22 и Dieharder. Ввиду того, что тесты для криптостойких ГПСЧ включают себя тесты для простых ГПСЧ, то в рамках изучения свойств предложенного метода будут использоваться пакеты тестов NIST SP-800-22 и Dieharder.

4.1.1 Критерии оценки качества

В данном подразделе описываются критерии оценки качества разработанного метода генерации псевдослучайных чисел. Отдельно каждый из данных критериев уже дает оценку случайности последовательности значений. Считается, что генератор псевдослучайных чисел, который успешно проходит

все критерии внутри соответствующих пакетов тестирования может считаться методом генерации псевдослучайных чисел.

4.1.2 NIST SP-800-22

NIST SP-800-22 - это публикация от Национального института стандартов и технологий, которая предоставляет руководство по выполнению статистических тестов на генераторы случайных чисел. Цель этих тестов - определить, насколько выходные данные от генератора случайных чисел действительно случайны, или есть ли какие-либо закономерности или предвзятости в сгенерированных числах, которые могут указывать на уязвимость безопасности. Эта публикация обычно используется в криптографии и кибербезопасности для оценки качества и прочности криптографических алгоритмов и протоколов.

NIST SP-800-22 состоит из 15 независимых статистических тестов:

1. Частотный побитовый тест;
2. Частотный блочный тест;
3. Сериальный тест;
4. Тест на самую длинную последовательность из единиц в блоке;
5. Тест рангов бинарных матриц;
6. Спектральный тест;
7. Тест на встречающиеся непересекающиеся шаблоны;
8. Тест на встречающиеся пересекающиеся шаблоны;
9. Универсальный тест Мауэра;
10. Тест на линейную сложность;
11. Тест на периодичность;
12. Тест приблизительной энтропии;
13. Тест кумулятивных сумм;
14. Тест на произвольные отклонения;
15. Другой тест на произвольные отклонения.

Частотный побитовый тест

Основная идея теста заключается в проверке числа единиц и нулей в последовательности бит. Идеально случайная последовательность должна иметь одинаковое количество единиц и нулей. Если последовательность содержит сильное отклонение от этого равновесия, то это может быть знаком наличия закономерности в последовательности.

Для выполнения теста сначала определяется общее число единиц и нулей в последовательности. Затем сравнивается их отношение друг к другу, чтобы определить, насколько оно близко к 1/2. Это отношение должно быть близко к нормальному распределению с математическим ожиданием равным 1/2 и стандартным отклонением равным корню из (n/4), где n - длина последовательности.

Если отношение отклоняется от этого нормального распределения более, чем на определенную критическую значимость, то тест не проходится, что может означать, что последовательность не случайна.

Рекомендуемыми параметрами для тестирования являются бинарная последовательность из 1.000.000 бит и 100 последовательностей для принятия гипотез.

Листинг 4.1 – модуль monobit_test

```
from __future__ import print_function

import math

def count_ones_zeroes(bits):
    ones = 0
    zeroes = 0
    for bit in bits:
        if (bit == 1):
            ones += 1
        else:
            zeroes += 1
    return (zeroes,ones)

def monobit_test(bits):
    n = len(bits)
```

```
    zeroes,ones = count_ones_zeroes(bits)
    s = abs(ones-zeroes)
    print("  Ones count   = %d" % ones)
    print("  Zeroes count = %d" % zeroes)

    p = math.erfc(float(s)/(math.sqrt(float(n)) * math.sqrt(2.0)))

    success = (p >= 0.01)
    return (success,p,None)
```

Частотный блочный тест

Основная идея теста заключается в проверке того, насколько хорошо последовательность бит может быть разбита на равные блоки заданной длины. Идеально случайная последовательность будет вести себя одинаково в каждом блоке, то есть количество нулей и единиц в каждом блоке должно быть примерно одинаковым.

Для выполнения теста последовательность делится на блоки фиксированной длины, и количество нулей и единиц в каждом блоке подсчитывается. Затем сравнивается количество единиц и нулей в каждом блоке, чтобы определить, насколько оно близко к 1/2. Это отношение должно быть близко к нормальному распределению с математическим ожиданием равным 1/2 и стандартным отклонением по формуле sqrt(n/4), где n - длина блока.

Если отношение отклоняется от этого нормального распределения на определенную критическую значимость, то тест не проходится, что может свидетельствовать о наличии закономерности в последовательности бит.

Рекомендуемыми параметрами для тестирования являются последовательность не менее 100 бит и блок равный 20 бит.

Листинг 4.2 – модуль frequency_within_block_test

```
from __future__ import print_function

import math
from fractions import Fraction
#from scipy.special import gamma, gammainc, gammaincc
```

```
from gamma_functions import *

#ones_table = [bin(i)[2:].count('1') for i in range(256)]
def count_ones_zeroes(bits):
    ones = 0
    zeroes = 0
    for bit in bits:
        if (bit == 1):
            ones += 1
        else:
            zeroes += 1
    return (zeroes,ones)

def frequency_within_block_test(bits):
    # Compute number of blocks M = block size. N=num of blocks
    # N = floor(n/M)
    # miniumum block size 20 bits, most blocks 100
    n = len(bits)
    M = 20
    N = int(math.floor(n/M))
    if N > 99:
        N=99
        M = int(math.floor(n/N))

    if len(bits) < 100:
        print("Too little data for test. Supply at least 100 bits")
        return False,1.0,None

    print("  n = %d" % len(bits))
    print("  N = %d" % N)
    print("  M = %d" % M)

    num_of_blocks = N
    block_size = M #int(math.floor(len(bits)/num_of_blocks))
    #n = int(block_size * num_of_blocks)

    proportions = list()
    for i in range(num_of_blocks):
        block = bits[i*(block_size):((i+1)*(block_size))]
        zeroes,ones = count_ones_zeroes(block)
        proportions.append(Fraction(ones,block_size))

    chisq = 0.0
    for prop in proportions:
        chisq += 4.0*block_size*((prop - Fraction(1,2))**2)

    p = gammaincc((num_of_blocks/2.0),float(chisq)/2.0)
    success = (p >= 0.01)
    return (success,p,None)
```

Сериальный тест

Тест используется для проверки случайности в последовательности бинарных значений (например, 0 и 1) или символьных значений. Он оценивает, насколько очередной элемент в последовательности зависит от предыдущего.

В рамках теста анализируется количество серий - последовательно идущих одинаковых значений в последовательности. Далее, на основе количества серий строится статистика, которая используется для проверки гипотезы о равномерном распределении значений в последовательности.

Если статистика попадает в критический интервал, то делается вывод, что последовательность не случайна и имеет систему. Если статистика попадает в допустимый интервал, то последовательность считается случайной.

Длина последовательности 100 и более бит. [4]

Листинг 4.3 – модуль runs_test

```
from __future__ import print_function

import math
from fractions import Fraction
from scipy.special import gamma, gammainc, gammaincc
from gamma_functions import *
import numpy
import cmath
import random

#ones_table = [bin(i)[2:].count('1') for i in range(256)]
def count_ones_zeroes(bits):
    ones = 0
    zeroes = 0
    for bit in bits:
        if (bit == 1):
            ones += 1
        else:
            zeroes += 1
    return (zeroes,ones)

def runs_test(bits):
    n = len(bits)
    zeroes,ones = count_ones_zeroes(bits)

    prop = float(ones)/float(n)
    print("  prop ",prop)

    tau = 2.0/math.sqrt(n)
    print("  tau ",tau)

    if abs(prop-0.5) > tau:
        return (False,0.0,None)

    vobs = 1.0
    for i in range(n-1):
        if bits[i] != bits[i+1]:
            vobs += 1.0

    print("  vobs ",vobs)
```

```
    p = math.erfc(abs(vobs - (2.0*n*prop*(1.0-
prop)))/(2.0*math.sqrt(2.0*n)*prop*(1-prop) ))
    success = (p >= 0.01)
    return (success,p,None)
```

Тест на самую длинную последовательность из единиц в блоке

Тест случайности, который используется для проверки гипотезы о том, что длина максимальной серии единиц в последовательности случайна.

Тест заключается в разбиении последовательности на блоки заданного размера и подсчете наибольшей серии из единиц в каждом блоке. Затем на основе этих значений вычисляется статистика теста, которая проверяет равномерность распределения наибольшей серии единиц в блоке.

Если значение статистики попадает в критический интервал, то считается, что тестовая последовательность не является случайной. Если значение статистики находится в допустимом интервале, то можно принять гипотезу о случайности последовательности.

Тест для самой длинной серии "1" в блоке применяется в криптографии, генерации случайных чисел и других областях, где требуется проверка случайности последовательностей.

NIST рекомендует несколько опорных значений для разбития последовательности на блоки:

Общая длина, n	Длина блока, M
128	8
6272	128
750000	10000

Листинг 4.4 – модуль longest_run_ones_in_a_block_test

```
from __future__ import print_function

import math
from scipy.special import gamma, gammainc, gammaincc
from gamma_functions import *

import random

def probs(K,M,i):
    M8 =      [0.2148, 0.3672, 0.2305, 0.1875]
    M128 =    [0.1174, 0.2430, 0.2493, 0.1752, 0.1027, 0.1124]
    M512 =    [0.1170, 0.2460, 0.2523, 0.1755, 0.1027, 0.1124]
    M1000 =   [0.1307, 0.2437, 0.2452, 0.1714, 0.1002, 0.1088]
    M10000 =  [0.0882, 0.2092, 0.2483, 0.1933, 0.1208, 0.0675, 0.0727]
    if (M == 8):        return M8[i]
    elif (M == 128):    return M128[i]
    elif (M == 512):    return M512[i]
    elif (M == 1000):   return M1000[i]
    else:               return M10000[i]

def longest_run_ones_in_a_block_test(bits):
    n = len(bits)

    if n < 128:
        return (False,1.0,None)
    elif n<6272:
        M = 8
    elif n<750000:
        M = 128
    else:
        M = 10000

    # compute new values for K & N
    if M==8:
        K=3
        N=16
    elif M==128:
        K=5
        N=49
    else:
        K=6
        N=75

    # Table of frequencies
    v = [0,0,0,0,0,0,0]

    for i in range(N): # over each block
        #find longest run
        block = bits[i*M:((i+1)*M)] # Block i

        run = 0
        longest = 0
        for j in range(M): # Count the bits.
            if block[j] == 1:
                run += 1
                if run > longest:
                    longest = run
```

```
            else:
                run = 0

        if M == 8:
            if longest <= 1:    v[0] += 1
            elif longest == 2:  v[1] += 1
            elif longest == 3:  v[2] += 1
            else:               v[3] += 1
        elif M == 128:
            if longest <= 4:    v[0] += 1
            elif longest == 5:  v[1] += 1
            elif longest == 6:  v[2] += 1
            elif longest == 7:  v[3] += 1
            elif longest == 8:  v[4] += 1
            else:               v[5] += 1
        else:
            if longest <= 10:   v[0] += 1
            elif longest == 11: v[1] += 1
            elif longest == 12: v[2] += 1
            elif longest == 13: v[3] += 1
            elif longest == 14: v[4] += 1
            elif longest == 15: v[5] += 1
            else:               v[6] += 1

    # Compute Chi-Sq
    chi_sq = 0.0
    for i in range(K+1):
        p_i = probs(K,M,i)
        upper = (v[i] - N*p_i)**2
        lower = N*p_i
        chi_sq += upper/lower
    print("  n = "+str(n))
    print("  K = "+str(K))
    print("  M = "+str(M))
    print("  N = "+str(N))
    print("  chi_sq = "+str(chi_sq))
    p = gammaincc(K/2.0, chi_sq/2.0)

    success = (p >= 0.01)
    return (success,p,None)
```

Тест рангов бинарных матриц

Тест рангов бинарных матриц (Rank Test for Binary Matrices) используется для оценки случайности бинарной матрицы (матрицы, состоящей из 0 и 1). Он проверяет гипотезу о том, что рассматриваемая матрица была сгенерирована случайно.

Для этого матрица разбивается на подматрицы заданного размера, и для каждой подматрицы вычисляется ранг. Ранг - это количество линейно-

независимых строк или столбцов в матрице. На основе рангов подматриц вычисляется статистика теста, которая сравнивается с теоретическими значениями.

Если статистика попадает в критический интервал, то считается, что матрица не случайна. Если статистика находится в допустимом интервале, то можно принять гипотезу о случайности матрицы.

Тест рангов бинарных матриц широко используется в криптографии, статистике, компьютерной графике и других областях, где требуется оценка случайности бинарных матриц.

Листинг 4.5 – модуль binary_matrix_rank_test

```
from __future__ import print_function

import math
import copy
import gf2matrix

def binary_matrix_rank_test(bits,M=32,Q=32):
    n = len(bits)
    N = int(math.floor(n/(M*Q))) #Number of blocks
    print("  Number of blocks %d" % N)
    print("  Data bits used: %d" % (N*M*Q))
    print("  Data bits discarded: %d" % (n-(N*M*Q)))

    if N < 38:
        print("  Number of blocks must be greater than 37")
        p = 0.0
        return False,p,None

    # Compute the reference probabilities for FM, FMM and remainder
    r = M
    product = 1.0
    for i in range(r):
        upper1 = (1.0 - (2.0**(i-Q)))
        upper2 = (1.0 - (2.0**(i-M)))
        lower = 1-(2.0**(i-r))
        product = product * ((upper1*upper2)/lower)
    FR_prob = product * (2.0**((r*(Q+M-r)) - (M*Q)))

    r = M-1
    product = 1.0
    for i in range(r):
        upper1 = (1.0 - (2.0**(i-Q)))
        upper2 = (1.0 - (2.0**(i-M)))
        lower = 1-(2.0**(i-r))
        product = product * ((upper1*upper2)/lower)
    FRM1_prob = product * (2.0**((r*(Q+M-r)) - (M*Q)))

    LR_prob = 1.0 - (FR_prob + FRM1_prob)

    FM = 0      # Number of full rank matrices
```

```
    FMM = 0     # Number of rank -1 matrices
    remainder = 0
    for blknum in range(N):
        block = bits[blknum*(M*Q):(blknum+1)*(M*Q)]
        # Put in a matrix
        matrix = gf2matrix.matrix_from_bits(M,Q,block,blknum)
        # Compute rank
        rank = gf2matrix.rank(M,Q,matrix,blknum)

        if rank == M: # count the result
            FM += 1
        elif rank == M-1:
            FMM += 1
        else:
            remainder += 1

    chisq =  (((FM-(FR_prob*N))**2)/(FR_prob*N))
    chisq += (((FMM-(FRM1_prob*N))**2)/(FRM1_prob*N))
    chisq += (((remainder-(LR_prob*N))**2)/(LR_prob*N))
    p = math.e **(-chisq/2.0)
    success = (p >= 0.01)

    print("  Full Rank Count  = ",FM)
    print("  Full Rank -1 Count = ",FMM)
    print("  Remainder Count = ",remainder)
    print("  Chi-Square = ",chisq)

    return (success, p, None)
```

Спектральный тест

Спектральный тест — это тест случайности, анализирующий спектральные свойства последовательностей псевдослучайных чисел. Его используют для проверки того, что последовательности псевдослучайных чисел являются случайными и не имеют скрытых структур.

Суть теста заключается в том, что последовательность псевдослучайных чисел преобразуется с помощью алгоритма Фурье в частотный спектр. Спектр анализируется на наличие определенных свойств - например, равномерности распределения или частот на частотной оси. Затем, на основе этих свойств вычисляется статистика теста.

Если статистика попадает в критический интервал, то считается, что проверяемая последовательность не случайна, а подвержена влиянию каких-либо скрытых структур. Если статистика находится в допустимом интервале, то можно принять гипотезу о случайности последовательности.

Спектральный тест широко используется в криптографии, компьютерной безопасности и генерации случайных чисел для проверки равномерности распределения значений в последовательностях псевдослучайных чисел. [4]

Листинг 4.6 – модуль dft_test

```
from __future__ import print_function

import math
import numpy
import sys

def dft_test(bits):
    n = len(bits)
    if (n % 2) == 1:        # Make it an even number
        bits = bits[:-1]

    ts = list()             # Convert to +1,-1
    for bit in bits:
        ts.append((bit*2)-1)

    ts_np = numpy.array(ts)
    fs = numpy.fft.fft(ts_np)  # Compute DFT

    if sys.version_info > (3,0):
        mags = abs(fs)[:n//2] # Compute magnitudes of first half of sequence
    else:
        mags = abs(fs)[:n/2] # Compute magnitudes of first half of sequence

    T = math.sqrt(math.log(1.0/0.05)*n) # Compute upper threshold
    N0 = 0.95*n/2.0
    print("  N0 = %f" % N0)

    N1 = 0.0   # Count the peaks above the upper theshold
    for mag in mags:
        if mag < T:
            N1 += 1.0
    print("  N1 = %f" % N1)
    d = (N1 - N0)/math.sqrt((n*0.95*0.05)/4) # Compute the P value
    p = math.erfc(abs(d)/math.sqrt(2))

    success = (p >= 0.01)
    return (success,p,None)
```

Тест с неперекрывающимися непериодическими шаблонами

Тест на встречающиеся непересекающиеся шаблоны (Test for the Non-overlapping Template Matching) – это тест случайности, который используется для проверки гипотезы о том, что последовательность данных случайна.

Тест заключается в поиске заранее заданного набора шаблонов в последовательности. Шаблоны могут быть любой длины и могут содержать

любые символы. Шаблоны ищутся в последовательности без перекрытий - то есть каждая найденная последовательность не должна перекрываться с предыдущей найденной последовательностью.

Затем, на основе количества найденных шаблонов, вычисляется статистика теста, которая проверяет, насколько вероятно, что найденные шаблоны случайны.

Если значение статистики теста попадает в критический интервал, то считается, что тестовая последовательность не является случайной. Если значение статистики находится в допустимом интервале, то можно принять гипотезу о случайности последовательности.

Тест на встречающиеся непересекающиеся шаблоны широко используется в криптографии и генерации случайных чисел для проверки степени случайности последовательностей символов. [4]

Листинг 4.7 – модуль non_overlapping_template_matching_test_test

```
from __future__ import print_function

import math
from scipy.special import gamma, gammainc, gammaincc
from gamma_functions import *
import random

def non_overlapping_template_matching_test(bits):
    # The templates provdided in SP800-22rev1a
    templates = [None for x in range(7)]
    templates[0] = [[0,1],[1,0]]
    templates[1] = [[0,0,1],[0,1,1],[1,0,0],[1,1,0]]
    templates[2] =
[[0,0,0,1],[0,0,1,1],[0,1,1,1],[1,0,0,0],[1,1,0,0],[1,1,1,0]]
    templates[3] =
[[0,0,0,0,1],[0,0,0,1,1],[0,0,1,0,1],[0,1,0,1,1],[0,0,1,1,1],[0,1,1,1,1],

[1,1,1,0,0],[1,1,0,1,0],[1,0,1,0,0],[1,1,0,0,0],[1,0,0,0,0],[1,1,1,1,0]]
    templates[4] =
[[0,0,0,0,0,1],[0,0,0,0,1,1],[0,0,0,1,0,1],[0,0,0,1,1,1],[0,0,1,0,1,1],
                    [0,0,1,1,0,1],[0,0,1,1,1,1],[0,1,0,0,1,1],
                    [0,1,0,1,1,1],[0,1,1,1,1,1],[1,0,0,0,0,0],
                    [1,0,1,0,0,0],[1,0,1,1,0,0],[1,1,0,0,0,0],
                    [1,1,0,0,1,0],[1,1,0,1,0,0],[1,1,1,0,0,0],
                    [1,1,1,0,1,0],[1,1,1,1,0,0],[1,1,1,1,1,0]]
    templates[5] =
[[0,0,0,0,0,0,1],[0,0,0,0,0,1,1],[0,0,0,0,1,0,1],[0,0,0,0,1,1,1],

[0,0,0,1,0,0,1],[0,0,0,1,0,1,1],[0,0,0,1,1,0,1],[0,0,0,1,1,1,1],

[0,0,1,0,0,1,1],[0,0,1,0,1,0,1],[0,0,1,0,1,1,1],[0,0,1,1,0,1,1],
```

```
[0,0,1,1,1,0,1],[0,0,1,1,1,1,1],[0,1,0,0,0,1,1],[0,1,0,0,1,1,1],

[0,1,0,1,0,1,1],[0,1,0,1,1,1,1],[0,1,1,0,1,1,1],[0,1,1,1,1,1,1],

[1,0,0,0,0,0,0],[1,0,0,1,0,0,0],[1,0,1,0,0,0,0],[1,0,1,0,1,0,0],

[1,0,1,1,0,0,0],[1,0,1,1,1,0,0],[1,1,0,0,0,0,0],[1,1,0,0,0,1,0],

[1,1,0,0,1,0,0],[1,1,0,1,0,0,0],[1,1,0,1,0,1,0],[1,1,0,1,1,0,0],

[1,1,1,0,0,0,0],[1,1,1,0,0,1,0],[1,1,1,0,1,0,0],[1,1,1,0,1,1,0],

[1,1,1,1,0,0,0],[1,1,1,1,0,1,0],[1,1,1,1,1,0,0],[1,1,1,1,1,1,0]]
    templates[6] =
[[0,0,0,0,0,0,0,1],[0,0,0,0,0,0,1,1],[0,0,0,0,0,1,0,1],[0,0,0,0,0,1,1,1],

[0,0,0,0,1,0,0,1],[0,0,0,0,1,0,1,1],[0,0,0,0,1,1,0,1],[0,0,0,0,1,1,1,1],

[0,0,0,1,0,0,1,1],[0,0,0,1,0,1,0,1],[0,0,0,1,0,1,1,1],[0,0,0,1,1,0,0,1],

[0,0,0,1,1,0,1,1],[0,0,0,1,1,1,0,1],[0,0,0,1,1,1,1,1],[0,0,1,0,0,0,1,1],

[0,0,1,0,0,1,0,1],[0,0,1,0,0,1,1,1],[0,0,1,0,1,0,1,1],[0,0,1,0,1,1,0,1],

[0,0,1,0,1,1,1,1],[0,0,1,1,0,1,0,1],[0,0,1,1,0,1,1,1],[0,0,1,1,1,0,1,1],

[0,0,1,1,1,1,0,1],[0,0,1,1,1,1,1,1],[0,1,0,0,0,0,1,1],[0,1,0,0,0,1,1,1],

[0,1,0,0,1,0,1,1],[0,1,0,0,1,1,1,1],[0,1,0,1,0,0,1,1],[0,1,0,1,0,1,1,1],

[0,1,0,1,1,0,1,1],[0,1,0,1,1,1,1,1],[0,1,1,0,0,1,1,1],[0,1,1,0,1,1,1,1],

[0,1,1,1,1,1,1,1],[1,0,0,0,0,0,0,0],[1,0,0,1,0,0,0,0],[1,0,0,1,1,0,0,0],

[1,0,1,0,0,0,0,0],[1,0,1,0,0,1,0,0],[1,0,1,0,1,0,0,0],[1,0,1,0,1,1,0,0],

[1,0,1,1,0,0,0,0],[1,0,1,1,0,1,0,0],[1,0,1,1,1,0,0,0],[1,0,1,1,1,1,0,0],

[1,1,0,0,0,0,0,0],[1,1,0,0,0,0,1,0],[1,1,0,0,0,1,0,0],[1,1,0,0,1,0,0,0],

[1,1,0,0,1,0,1,0],[1,1,0,1,0,0,0,0],[1,1,0,1,0,0,1,0],[1,1,0,1,0,1,0,0],

[1,1,0,1,1,0,0,0],[1,1,0,1,1,0,1,0],[1,1,0,1,1,1,0,0],[1,1,1,0,0,0,0,0],

[1,1,1,0,0,0,1,0],[1,1,1,0,0,1,0,0],[1,1,1,0,0,1,1,0],[1,1,1,0,1,0,0,0],

[1,1,1,0,1,0,1,0],[1,1,1,0,1,1,0,0],[1,1,1,1,0,0,0,0],[1,1,1,1,0,0,1,0],

[1,1,1,1,0,1,0,0],[1,1,1,1,0,1,1,0],[1,1,1,1,1,0,0,0],[1,1,1,1,1,0,1,0],
                  [1,1,1,1,1,1,0,0],[1,1,1,1,1,1,1,0]]

    n = len(bits)

    # Choose the template B
    r = random.SystemRandom()
    template_list = r.choice(templates)
    B = r.choice(template_list)

    m = len(B)

    N = 8
    M = int(math.floor(len(bits)/8))
    n = M*N
```

```
    blocks = list() # Split into N blocks of M bits
    for i in range(N):
        blocks.append(bits[i*M:(i+1)*M])

    W=list() # Count the number of matches of the template in each block Wj
    for block in blocks:
        position = 0
        count = 0
        while position < (M-m):
            if block[position:position+m] == B:
                position += m
                count += 1
            else:
                position += 1
        W.append(count)

    mu = float(M-m+1)/float(2**m) # Compute mu and sigma
    sigma = M * ((1.0/float(2**m))-(float((2*m)-1)/float(2**(2*m))))

    chisq = 0.0  # Compute Chi-Square
    for j in range(N):
        chisq += ((W[j] - mu)**2)/(sigma**2)

    p = gammaincc(N/2.0, chisq/2.0) # Compute P value

    success = ( p >= 0.01)
    return (success,p,None)
```

Тест на встречающиеся пересекающиеся шаблоны

Отличительной особенностью данного теста является то, что при поиске шаблона "окно" сдвигается не на всю длину шаблона, а только на один бит, что позволяет избежать проблем с большим количеством расчетов.

Листинг 4.8 – модуль overlapping_template_matching_test

```
from __future__ import print_function

import math
from scipy.special import gamma, gammainc, gammaincc
from gamma_functions import *

def lgamma(x):
   return math.log(gamma(x))

def Pr(u, eta):
   if ( u == 0 ):
      p = math.exp(-eta)
   else:
      sum = 0.0
      for l in range(1,u+1):
```

```
            sum += math.exp(-eta-u*math.log(2)+l*math.log(eta)-lgamma(l+1)+lgamma(u)-
lgamma(l)-lgamma(u-l+1))
        p = sum
    return p

def overlapping_template_matching_test(bits,blen=6):
    n = len(bits)

    m = 10
    # Build the template B as a random list of bits
    B = [1 for x in range(m)]

    N = 968
    K = 5
    M = 1062
    if len(bits) < (M*N):
        print("Insufficient data. %d bit provided. 1,028,016 bits required" % len(bits))
        return False, 0.0, None

    blocks = list() # Split into N blocks of M bits
    for i in range(N):
        blocks.append(bits[i*M:(i+1)*M])

    # Count the distribution of matches of the template across blocks: Vj
    v=[0 for x in range(K+1)]
    for block in blocks:
        count = 0
        for position in range(M-m):
            if block[position:position+m] == B:
                count += 1

        if count >= (K):
            v[K] += 1
        else:
            v[count] += 1


    chisq = 0.0  # Compute Chi-Square
    #pi = [0.324652,0.182617,0.142670,0.106645,0.077147,0.166269] # From spec
    pi = [0.364091, 0.185659, 0.139381, 0.100571, 0.0704323, 0.139865] # From STS
    piqty = [int(x*N) for x in pi]

    lambd = (M-m+1.0)/(2.0**m)
    eta = lambd/2.0
    sum = 0.0
    for i in range(K): #  Compute Probabilities
        pi[i] = Pr(i, eta)
        sum += pi[i]

    pi[K] = 1 - sum;

    sum = 0
```

```
    chisq = 0.0
    for i in range(K+1):
        chisq += ((v[i] - (N*pi[i]))**2)/(N*pi[i])
        sum += v[i]

    p = gammaincc(5.0/2.0, chisq/2.0) # Compute P value

    print("  B = ",B)
    print("  m = ",m)
    print("  M = ",M)
    print("  N = ",N)
    print("  K = ",K)
    print("  model = ",piqty)
    print("  v[j] =  ",v)
    print("  chisq = ",chisq)

    success = ( p >= 0.01)
    return (success,p,None)
```

Универсальный тест Мауэра

Универсальный тест Мауэра (Maurer's Universal Statistical Test) - это тест на случайность, который позволяет оценить, насколько хорошо последовательность случайных чисел соответствует теоретическому равномерному распределению.

Тест Мауэра состоит из нескольких этапов. На первом этапе вычисляется длина последовательности в битах. Затем эта последовательность разбивается на блоки определенной длины. Для каждого блока вычисляется хэш-значение, которое суммируется с хэш-значениями предыдущих блоков.

Далее, на основе полученной последовательности хэш-значений исходной последовательности производится расчёт показателя "p-value", представляющего вероятность получения такой же или еще более экстремальной последовательности случайных чисел. Этот значение сравнивается с пороговым значением, которое предоставляется в рамках теста. Если "p-value" ниже порогового значения, то тестовая последовательность не считается случайной.

Универсальный тест Мауэра применяется в криптографии, генерации случайных чисел и других областях, где требуется проверка случайности последовательности чисел. [4]

Листинг 4.9 – модуль maurers_universal_test

```
from __future__ import print_function

import math

def pattern2int(pattern):
    l = len(pattern)
    n = 0
    for bit in (pattern):
        n = (n << 1) + bit
    return n

def maurers_universal_test(bits,patternlen=None, initblocks=None):
    n = len(bits)

    # Step 1. Choose the block size
    if patternlen != None:
        L = patternlen
    else:
        ns = [904960,2068480,4654080,10342400,
              22753280,49643520,107560960,
              231669760,496435200,1059061760]
        L = 6
        if n < 387840:
            print("Error. Need at least 387840 bits. Got %d." % n)
            #exit()
            return False,0.0,None
        for threshold in ns:
            if n >= threshold:
                L += 1

    # Step 2 Split the data into Q and K blocks
    nblocks = int(math.floor(n/L))
    if initblocks != None:
        Q = initblocks
    else:
        Q = 10*(2**L)
    K = nblocks - Q

    # Step 3 Construct Table
    nsymbols = (2**L)
    T=[0 for x in range(nsymbols)] # zero out the table
    for i in range(Q):              # Mark final position of
        pattern = bits[i*L:(i+1)*L] # each pattern
        idx = pattern2int(pattern)
        T[idx]=i+1      # +1 to number indexes 1..(2**L)+1
```

```
                    # instead of 0..2**L
    # Step 4 Iterate
    sum = 0.0
    for i in range(Q,nblocks):
        pattern = bits[i*L:(i+1)*L]
        j = pattern2int(pattern)
        dist = i+1-T[j]
        T[j] = i+1
        sum = sum + math.log(dist,2)
    print("  sum =", sum)

    # Step 5 Compute the test statistic
    fn = sum/K
    print("  fn =",fn)

    # Step 6 Compute the P Value
    # a_universal_statistical_test_for_random_bit_generators.pdf
    ev_table =  [0,0.73264948,1.5374383,2.40160681,3.31122472,
                4.25342659,5.2177052,6.1962507,7.1836656,
                8.1764248,9.1723243,10.170032,11.168765,
                12.168070,13.167693,14.167488,15.167379]
    var_table = [0,0.690,1.338,1.901,2.358,2.705,2.954,3.125,
                3.238,3.311,3.356,3.384,3.401,3.410,3.416,
                3.419,3.421]

    # sigma = math.sqrt(var_table[L])
    mag = abs((fn - ev_table[L]) / ((0.7 - 0.8 / L + (4 + 32 / L) * (pow(K, -3 / L)) / 15) *
(math.sqrt(var_table[L] / K)) * math.sqrt(2)))
    P = math.erfc(mag)

    success = (P >= 0.01)
    return (success, P, None)


if __name__ == "__main__":
    bits = [0,1,0,1,1,0,1,0,0,1,1,1,0,1,0,1,0,1,1,1]
    success, p, _ = maurers_universal_test(bits, patternlen=2, initblocks=4)

    print("success =",success)
    print("p       = ",p)
```

Тест на линейную сложность

Тест на линейную сложность (Linear Complexity Test) - это тест, который используется для проверки случайности последовательности битов (например, 0 и 1) или символов. Он позволяет оценить линейную сложность последовательности, то есть минимальное количество линейных элементов (регистров сдвига с обратной связью), необходимых для ее генерации.

Тест заключается в последовательной итерации по блокам заданной длины последовательности. Для каждого блока вычисляется автокорреляционная функция, а затем выполнение алгоритма Берлекэмпа-Мэсси (Berlekamp-Massey algorithm) позволяет вычислить линейную сложность блока и всей последовательности.

Алгоритм Берлекэмпа-Мэсси применяется для поиска линейно-рекуррентных уравнений, удовлетворяющих запуску последовательности и ее линейной сложности.

Если линейная сложность последовательности близка к ее длине, то последовательность считается сложной и не случайной. Если линейная сложность меньше длины последовательности, то это может свидетельствовать о ее случайности.

Тест на линейную сложность широко используется в криптографии, генерации случайных чисел и других областях, где требуется оценить комплексность последовательности. [4]

Листинг 4.10 – модуль linear_complexity_test

```
from __future__ import print_function

import math
from scipy.special import gamma, gammainc, gammaincc
from gamma_functions import *

def berelekamp_massey(bits):
    n = len(bits)
    b = [0 for x in bits]  #initialize b and c arrays
    c = [0 for x in bits]
    b[0] = 1
    c[0] = 1

    L = 0
    m = -1
    N = 0
    while (N < n):
        #compute discrepancy
        d = bits[N]
        for i in range(1,L+1):
            d = d ^ (c[i] & bits[N-i])
        if (d != 0):  # If d is not zero, adjust poly
```

```
            t = c[:]
            for i in range(0,n-N+m):
                c[N-m+i] = c[N-m+i] ^ b[i]
            if (L <= (N/2)):
                L = N + 1 - L
                m = N
                b = t
        N = N +1
    # Return length of generator and the polynomial
    return L , c[0:L]

def linear_complexity_test(bits,patternlen=None):
    n = len(bits)
    # Step 1. Choose the block size
    if patternlen != None:
        M = patternlen
    else:
        if n < 1000000:
            print("Error. Need at least 10^6 bits")
            #exit()
            return False,0.0,None
        M = 512
    K = 6
    N = int(math.floor(n/M))
    print("  M = ", M)
    print("  N = ", N)
    print("  K = ", K)

    # Step 2 Compute the linear complexity of the blocks
    LC = list()
    for i in range(N):
        x = bits[(i*M):((i+1)*M)]
        LC.append(berelekamp_massey(x)[0])

    # Step 3 Compute mean
    a = float(M)/2.0
    b = ((((-1)**(M+1))+9.0))/36.0
    c = ((M/3.0) + (2.0/9.0))/(2**M)
    mu =  a+b-c

    T = list()
    for i in range(N):
        x = ((-1.0)**M) * (LC[i] - mu) + (2.0/9.0)
        T.append(x)

    # Step 4 Count the distribution over Ticket
    v = [0,0,0,0,0,0,0]
    for t in T:
        if t <= -2.5:
            v[0] += 1
        elif t <= -1.5:
            v[1] += 1
```

```
        elif t <= -0.5:
            v[2] += 1
        elif t <= 0.5:
            v[3] += 1
        elif t <= 1.5:
            v[4] += 1
        elif t <= 2.5:
            v[5] += 1
        else:
            v[6] += 1

    # Step 5 Compute Chi Square Statistic
    pi = [0.010417,0.03125,0.125,0.5,0.25,0.0625,0.020833]
    chisq = 0.0
    for i in range(K+1):
        chisq += ((v[i] - (N*pi[i]))**2.0)/(N*pi[i])
    print("  chisq = ",chisq)
    # Step 6 Compute P Value
    P = gammaincc((K/2.0),(chisq/2.0))
    print("  P = ",P)
    success = (P >= 0.01)
    return (success, P, None)

if __name__ == "__main__":
    bits = [1,1,0,1,0,1,1,1,1,0,0,0,1]
    L,poly = berelekamp_massey(bits)

    bits = [1,1,0,1,0,1,1,1,1,0,0,0,1,1,1,0,1,0,1,1,1,1,0,0,
            0,1,1,1,0,1,0,1,1,1,1,0,0,0,1,1,1,0,1,0,1,1,1,1,
            0,0,0,1,1,1,0,1,0,1,1,1,1,0,0,0,1]
    success,p,_ = linear_complexity_test(bits,patternlen=7)

    print("L =",L)
    print("p = ",p)
```

Тест на периодичность

Тест на периодичность (Serial test) используется для проверки корреляции между элементами последовательности. Он позволяет оценить, насколько вероятно, что элементы последовательности зависят друг от друга.

Тест заключается в анализе различных подпоследовательностей фиксированной длины в исходной последовательности. В частности, проверяется, сколько раз определенная подпоследовательность встречается в последовательности. Затем, на основе этих значений вычисляется статистика теста, которая проверяет, насколько коррелированы элементы последовательности.

Если значение статистики попадает в критический интервал, то делается вывод, что элементы последовательности коррелированы и не являются случайными. Если значение статистики находится в допустимом интервале, то можно принять гипотезу о случайности последовательности. [4]

Листинг 4.11 – модуль serial_test

```
from __future__ import print_function

import math
# from scipy.special import gamma, gammainc, gammaincc
from gamma_functions import *

def int2patt(n,m):
    pattern = list()
    for i in range(m):
        pattern.append((n >> i) & 1)
    return pattern

def countpattern(patt,bits,n):
    thecount = 0
    for i in range(n):
        match = True
        for j in range(len(patt)):
            if patt[j] != bits[i+j]:
                match = False
        if match:
            thecount += 1
    return thecount
```

```
def psi_sq_mv1(m, n, padded_bits):
    counts = [0 for i in range(2**m)]
    for i in range(2**m):
        pattern = int2patt(i,m)
        count = countpattern(pattern,padded_bits,n)
        counts.append(count)

    psi_sq_m = 0.0
    for count in counts:
        psi_sq_m += (count**2)
    psi_sq_m = psi_sq_m * (2**m)/n
    psi_sq_m -= n
    return psi_sq_m

def serial_test(bits,patternlen=None):
    n = len(bits)
    if patternlen != None:
        m = patternlen
    else:
        m = int(math.floor(math.log(n,2)))-2

        if m < 4:
            print("Error. Not enough data for m to be 4")
            return False,0,None
        m = 4

    # Step 1
    padded_bits=bits+bits[0:m-1]

    # Step 2
    psi_sq_m   = psi_sq_mv1(m, n, padded_bits)
    psi_sq_mm1 = psi_sq_mv1(m-1, n, padded_bits)
    psi_sq_mm2 = psi_sq_mv1(m-2, n, padded_bits)

    delta1 = psi_sq_m - psi_sq_mm1
    delta2 = psi_sq_m - (2*psi_sq_mm1) + psi_sq_mm2

    P1 = gammaincc(2**(m-2),delta1/2.0)
    P2 = gammaincc(2**(m-3),delta2/2.0)

    print("  psi_sq_m   = ",psi_sq_m)
    print("  psi_sq_mm1 = ",psi_sq_mm1)
    print("  psi_sq_mm2 = ",psi_sq_mm2)
    print("  delta1     = ",delta1)
    print("  delta2     = ",delta2)
    print("  P1         = ",P1)
    print("  P2         = ",P2)

    success = (P1 >= 0.01) and (P2 >= 0.01)
    return (success, None, [P1,P2])
```

Тест приблизительной энтропии

Как и в тесте на периодичность в данном тесте акцент делается на подсчете частоты всех возможных перекрытий шаблонов длины m бит на протяжении исходной последовательности битов.

Тест заключается в разбиении последовательности на блоки определенной длины, после чего определяется энтропия каждого блока. Энтропия — это мера неопределенности или неустранимой случайности в каждом блоке. Затем средняя энтропия для всех блоков вычисляется и сравнивается с установленными границами.

Если полученное значение энтропии находится в пределах установленных границ, то это свидетельствует о высокой степени случайности последовательности. Если же значение энтропии находится за пределами установленных границ, это может быть признаком того, что последовательность не случайна и может иметь определенную структуру. [4]

Листинг 4.12 – модуль approximate_entropy_test

```
from __future__ import print_function

import math
#from scipy.special import gamma, gammainc, gammaincc
from gamma_functions import *

def bits_to_int(bits):
    theint = 0
    for i in range(len(bits)):
        theint = (theint << 1) + bits[i]
    return theint

def approximate_entropy_test(bits):
    n = len(bits)

    m = int(math.floor(math.log(n,2)))-6
    if m < 2:
        m = 2
    if m >3 :
        m = 3

    print("  n         = ",n)
```

```
    print("  m        = ",m)

    Cmi = list()
    phi_m = list()
    for iterm in range(m,m+2):
        # Step 1
        padded_bits=bits+bits[0:iterm-1]

        # Step 2
        counts = list()
        for i in range(2**iterm):
            #print "  Pattern #%d of %d" % (i+1,2**iterm)
            count = 0
            for j in range(n):
                if bits_to_int(padded_bits[j:j+iterm]) == i:
                    count += 1
            counts.append(count)
            print("  Pattern %d of %d, count = %d" % (i+1,2**iterm, count))

        # step 3
        Ci = list()
        for i in range(2**iterm):
            Ci.append(float(counts[i])/float(n))

        Cmi.append(Ci)

        # Step 4
        sum = 0.0
        for i in range(2**iterm):
            if (Ci[i] > 0.0):
                sum += Ci[i]*math.log((Ci[i]/10.0))
        phi_m.append(sum)
        print("  phi(%d)    = %f" % (m,sum))

    # Step 5 - let the loop steps 1-4 complete

    # Step 6
    appen_m = phi_m[0] - phi_m[1]
    print("  AppEn(%d)  = %f" % (m,appen_m))
    chisq = 2*n*(math.log(2) - appen_m)
    print("  ChiSquare = ",chisq)
    # Step 7
    p = gammaincc(2**(m-1),(chisq/2.0))

    success = (p >= 0.01)
    return (success, p, None)
```

Тест кумулятивных сумм

Тест кумулятивных сумм (Cumulative Sum Test) - это статистический тест, который используется для проверки равномерности распределения битов в последовательности. Он позволяет оценить то, насколько биты в последовательности равномерно распределены относительно середины последовательности.

Тест заключается в подсчете кумулятивных сумм для каждого значения бита в последовательности. Кумулятивная сумма для 0 или 1 вычисляется как разность количества значений 0 (или 1) до и после текущего бита в последовательности. Затем вся последовательность сканируется на наличие слишком больших отклонений от нулевой кумулятивной суммы в положительную или отрицательную сторону.

Если значение статистики теста значительно отклоняется от ожидаемого значения, то это может свидетельствовать о том, что биты в последовательности не равномерно распределены. Например, если сумма для значений 0 начинает резко увеличиваться и длится в течение продолжительного периода времени, то это может свидетельствовать о том, что последовательность содержит слишком много значений 0. [4]

Листинг 4.13 – модуль cumulative_sums_test

```
from __future__ import print_function

import math
#from scipy.special import gamma, gammainc, gammaincc
from gamma_functions import *
#import scipy.stats

def normcdf(n):
    return 0.5 * math.erfc(-n * math.sqrt(0.5))

def p_value(n,z):
    sum_a = 0.0
    startk = int(math.floor((((float(-n)/z)+1.0)/4.0)))
    endk   = int(math.floor((((float(n)/z)-1.0)/4.0)))
    for k in range(startk,endk+1):
```

```
        c = (((4.0*k)+1.0)*z)/math.sqrt(n)
        #d = scipy.stats.norm.cdf(c)
        d = normcdf(c)
        c = (((4.0*k)-1.0)*z)/math.sqrt(n)
        #e = scipy.stats.norm.cdf(c)
        e = normcdf(c)
        sum_a = sum_a + d - e

    sum_b = 0.0
    startk = int(math.floor((((float(-n)/z)-3.0)/4.0)))
    endk   = int(math.floor((((float(n)/z)-1.0)/4.0)))
    for k in range(startk,endk+1):
        c = (((4.0*k)+3.0)*z)/math.sqrt(n)
        #d = scipy.stats.norm.cdf(c)
        d = normcdf(c)
        c = (((4.0*k)+1.0)*z)/math.sqrt(n)
        #e = scipy.stats.norm.cdf(c)
        e = normcdf(c)
        sum_b = sum_b + d - e

    p = 1.0 - sum_a + sum_b
    return p

def cumulative_sums_test(bits):
    n = len(bits)
    # Step 1
    x = list()            # Convert to +1,-1
    for bit in bits:
        #if bit == 0:
        x.append((bit*2)-1)

    # Steps 2 and 3 Combined
    # Compute the partial sum and records the largest excursion.
    pos = 0
    forward_max = 0
    for e in x:
        pos = pos+e
        if abs(pos) > forward_max:
            forward_max = abs(pos)
    pos = 0
    backward_max = 0
    for e in reversed(x):
        pos = pos+e
        if abs(pos) > backward_max:
            backward_max = abs(pos)

    # Step 4
    p_forward  = p_value(n, forward_max)
    p_backward = p_value(n,backward_max)

    success = ((p_forward >= 0.01) and (p_backward >= 0.01))
    plist = [p_forward, p_backward]
```

```
    if success:
        print("PASS")
    else:
        print("FAIL: Data not random")
    return (success, None, plist)
```

Тест на произвольные отклонения

Тест на произвольные отклонения (Random Excursions Test) — это статистический тест, который используется для проверки случайности последовательности битов (например, 0 и 1) или символов. Он позволяет оценить, насколько часто последовательность совершает отклонения от своего среднего значения и насколько эти отклонения случайны.

Тест заключается в подсчете частоты появления последовательности отклонений из текущего положения последовательности к начальной точке (началу координат) на плоскости. Каждое отклонение в последовательности может быть представлено как вектор с единичной длиной.

Затем вычисляется значение статистики теста, которое соответствует количеству отклонений, полученных для каждого вектора пути на плоскости. Если полученные значения статистики соответствуют ожидаемым значениям, то можно сделать вывод о том, что отклонения в последовательности случайны.

Если полученные значения статистики теста значительно отклоняются от предполагаемых значений, то это может свидетельствовать о том, что последовательность не случайна и содержит систематические отклонения от среднего значения. [4]

Листинг 4.14 – модуль random_excursion_test

```
from __future__ import print_function

import math
#from scipy.special import gamma, gammainc, gammaincc
from gamma_functions import *

# RANDOM EXCURSION TEST
```

```
def random_excursion_test(bits):
    n = len(bits)

    x = list()          # Convert to +1,-1
    for bit in bits:
        #if bit == 0:
        x.append((bit*2)-1)

    #print "x=",x
    # Build the partial sums
    pos = 0
    s = list()
    for e in x:
        pos = pos+e
        s.append(pos)
    sprime = [0]+s+[0] # Add 0 on each end

    #print "sprime=",sprime
    # Build the list of cycles
    pos = 1
    cycles = list()
    while (pos < len(sprime)):
        cycle = list()
        cycle.append(0)
        while sprime[pos]!=0:
            cycle.append(sprime[pos])
            pos += 1
        cycle.append(0)
        cycles.append(cycle)
        pos = pos + 1

    J = len(cycles)
    print("J="+str(J))

    vxk = [['a','b','c','d','e','f'] for y in [-4,-3,-2,-1,1,2,3,4] ]

    # Count Occurances
    for k in range(6):
        for index in range(8):
            mapping = [-4,-3,-2,-1,1,2,3,4]
            x = mapping[index]
            cyclecount = 0
            #count how many cycles in which x occurs k times
            for cycle in cycles:
                oc = 0
                #Count how many times x occurs in the current cycle
                for pos in cycle:
                    if (pos == x):
                        oc += 1
                # If x occurs k times, increment the cycle count
                if (k < 5):
                    if oc == k:
```

```
                cyclecount += 1
            else:
                if k == 5:
                    if oc >=5:
                        cyclecount += 1
        vxk[index][k] = cyclecount

    # Table for reference random probabilities
    pixk=[[0.5     ,0.25   ,0.125  ,0.0625  ,0.0312 ,0.0312],
          [0.75    ,0.0625 ,0.0469 ,0.0352  ,0.0264 ,0.0791],
          [0.8333  ,0.0278 ,0.0231 ,0.0193  ,0.0161 ,0.0804],
          [0.875   ,0.0156 ,0.0137 ,0.012   ,0.0105 ,0.0733],
          [0.9     ,0.01   ,0.009  ,0.0081  ,0.0073 ,0.0656],
          [0.9167  ,0.0069 ,0.0064 ,0.0058  ,0.0053 ,0.0588],
          [0.9286  ,0.0051 ,0.0047 ,0.0044  ,0.0041 ,0.0531]]

    success = True
    plist = list()
    for index in range(8):
        mapping = [-4,-3,-2,-1,1,2,3,4]
        x = mapping[index]
        chisq = 0.0
        for k in range(6):
            top = float(vxk[index][k]) - (float(J) * (pixk[abs(x)-1][k]))
            top = top*top
            bottom = J * pixk[abs(x)-1][k]
            chisq += top/bottom
        p = gammaincc(5.0/2.0,chisq/2.0)
        plist.append(p)
        if p < 0.01:
            err = " Not Random"
            success = False
        else:
            err = ""
        print("x = %1.0f\tchisq = %f\tp = %f %s"  % (x,chisq,p,err))
    if (J < 500):
        print("J too small (J < 500) for result to be reliable")
    elif success:
        print("PASS")
    else:
        print("FAIL: Data not random")
    return (success, None, plist)
```

Другой тест на произвольные отклонения

Тест заключается в подсчете количества различных отклонений (вверх, вниз или без отклонения) в последовательности для каждой длины блока. Затем

вычисляется статистика теста, которая сравнивает полученные значения с ожидаемыми значениями для случайно генерируемых последовательностей.

Если значение статистики теста значительно отклоняется от ожидаемого значения, то это может свидетельствовать о том, что отклонения в последовательности не случайны. Например, если последовательность совершает слишком много отклонений в одном направлении на длинных блоках, то это может свидетельствовать о наличии систематических отклонений. [4]

Листинг 4.15 – модуль random_excursion_variant_test

```
from __future__ import print_function

import math

# RANDOM EXCURSION VARIANT TEST
def random_excursion_variant_test(bits):
    n = len(bits)

    x = list()            # Convert to +1,-1
    for bit in bits:
        x.append((bit * 2)-1)

    # Build the partial sums
    pos = 0
    s = list()
    for e in x:
        pos = pos+e
        s.append(pos)
    sprime = [0]+s+[0] # Add 0 on each end

    # Count the number of cycles J
    J = 0
    for value in sprime[1:]:
        if value == 0:
            J += 1
    print("J=",J)
    # Build the counts of offsets
    count = [0 for x in range(-9,10)]
    for value in sprime:
        if (abs(value) < 10):
            count[value] += 1

    # Compute P values
    success = True
    plist = list()
    for x in range(-9,10):
        if x != 0:
```

```
        top = abs(count[x]-J)
        bottom = math.sqrt(2.0 * J *((4.0*abs(x))-2.0))
        p = math.erfc(top/bottom)
        plist.append(p)
        if p < 0.01:
            err = " Not Random"
            success = False
        else:
            err = ""
        print("x = %1.0f\t count=%d\tp = %f %s"  % (x,count[x],p,err))

    if (J < 500):
        print("J too small (J=%d < 500) for result to be reliable" % J)
    elif success:
        print("PASS")
    else:
        print("FAIL: Data not random")
    return (success,None,plist)
```

Все приведенные выше модули представлены на листингах приложения Г.

4.1.3 Dieharder

Dieharder — это инструмент с открытым исходным кодом для тестирования произвольности данных и качества генераторов случайных чисел. Он предоставляет набор статистических тестов, основанных на публикации NIST SP-800-22, а также дополнительные тесты. Эти тесты оценивают широкий спектр свойств, таких как равномерность, независимость и непредсказуемость сгенерированных последовательностей.

Dieharder поддерживает различные источники входных данных, включая файлы, сетевые потоки и генераторы случайных чисел на аппаратном уровне. Он также предоставляет продвинутые функции, такие как параллельная обработка, визуализации и интерактивный пользовательский интерфейс для запуска тестов и изучения результатов.

Dieharder обычно используется в областях криптографии, компьютерного моделирования и научных вычислений, чтобы обеспечить надежность и качество случайных чисел, используемых в этих приложениях.

Dieharder состоит из 31 независимого статистического теста, 3 из которых встречаются в NIST SP-800-22. Поэтому внимание будет обращено на остальные 28 критериев.

1 Тест на проверку случайности дат рождения.

2 Diehard OPERM5

3 Diehard Rank 32x32

4 Тест на самую длинную последовательность из единиц в блоке

5 Тест потока битов

6 Тест случайного перестановочного генератора, взятого со сферы

7 Тест на случайные вектора, выбранные со сферического распределения

8 Тест на случайность последовательности нуклеотидов в ДНК

9 Потоковый подсчет единиц

10 Побайтовый подсчет единиц

11 Тест «Парковка»

12 Минимальное расстояние (2D окружность)

13 Минимальное расстояние (3D окружность)

14 Тест сжатия

15 Тест на последовательности

16 Игра в кости

17 Тест Марсалья и Цанг (НОД)

18 Сериальный тест

19 Тест распределения битов RGB

20 Тест перестановок битов RGB

21 Тест задержек сумм RGB

22 DAB: Тест на нелинейное перемешивание двоичных разрядов

23 DAB: Тест на независимость бит и байтов

24 DAB: Тест на перекрывающиеся четверки DAB

25 DAB: ДНК-тест

26 DAB: Тест подсчета единиц

27 DAB: Тест «Парковка»

28 DAB: Тест минимального расстояния (3D)

Тест на проверку случайности дат рождения.

Diehard Birthdays Test является одним из тестов в пакете Diehard на проверку случайности последовательностей битов. Этот тест применяется для проверки случайности даты рождения людей, выбранных из группы некоторого размера. Для этого выбирается большой набор людей из заданной популяции, и их дни рождения кодируются в бинарные последовательности. Затем алгоритм проверяет, насколько часто повторяется одинаковая дата рождения в этом наборе, что указывает на неслучайность.

Листинг 4.16 – модуль diehard_birthdays

```
def diehard_birthdays(binary_string):
    # переводим строку из битов в список байтов
    byte_string = []
    for i in range(0, len(binary_string), 8):
        byte = int(binary_string[i:i+8], 2)
        byte_string.append(byte)

    # считаем количество каждого возможного значения байта
    byte_counts = Counter(byte_string)

    # вычисляем сумму квадратов количеств каждого байта
    sum_of_squares = sum(count ** 2 for count in byte_counts.values())

    # вычисляем общее количество байтов и средний квадрат
    num_bytes = len(byte_string)
    mean_square = num_bytes ** 2 / 256

    # вычисляем значение статистики
    stat_value = (sum_of_squares - mean_square) / mean_square

    # проверяем значение статистики
    if abs(stat_value - 1.0) < 0.001:
        return "Diehard Birthdays Test: \t\t\t\t\t\t\tFailed"
    else:
        return "Diehard Birthdays Test: \t\t\t\t\t\tPassed"
```

Diehard OPERM5

Diehard OPERM5 Test — это один из тестов, входящих в пакет Diehard на проверку качества случайных чисел. Он используется для тестирования последовательности перестановок, которые должны быть случайными и равномерно распределенными. Для тестирования выбирается последовательность целых чисел от 1 до N, которые затем переставляются случайным образом. Затем выбраны подпоследовательности заданной длины на основе этой перестановки, и проверяется, насколько хорошо эти последовательности различаются от последовательности, которая получилась бы, если бы перестановка была равномерно распределена.

Листинг 4.17 – модуль diehard_operm5

```
def diehard_operm5(binary_string):
    binary_string = binary_string[:int(len(binary_string)//100)]
    # преобразуем строку из битов в список чисел от 0 до 255
    bytes_list = [int(binary_string[i:i + 8], 2) for i in range(0,
len(binary_string), 8)]

    # заполняем список рангов случайной перестановки
    ranks = []
    # print("len(bytes_list) ",len(bytes_list))
    for i in range(len(bytes_list)):
        rank = 1
        for j in range(i):
            if bytes_list[j] < bytes_list[i]:
                rank += 1
        ranks.append(rank)

    # проверяем, можно ли выразить сумму рангов как N(N+1)/4
    N = len(ranks)
    sum_ranks = sum(ranks)
    expected_sum_ranks = N * (N + 1) / 4
    variance = N * (N - 1) * (2*N + 5) / 72
    stddev = variance ** 0.5

    Z = (sum_ranks - expected_sum_ranks) / stddev

    # проверяем значение Z-статистики
    if abs(Z) < 3:
        return "Diehard OPERM5 Test: \t\t\t\t\t\tPassed"
    else:
        return "Diehard OPERM5 Test: \t\t\t\t\t\t\tFailed"
```

Diehard Rank 32x32

Этот тест использует матрицы размером 32 на 32 и проверяет, насколько хорошо в них рассредоточены 0 и 1. Для тестирования выбираются большие наборы матриц, заполненных случайным образом, и проверяется, насколько хорошо их ранг (или число независимых строк матрицы) соответствует ожидаемому значению. [4]

Листинг 4.18 – модуль diehard_rank_32x32

```
def diehard_rank_32x32(binary_string):
    # проверяем, что строка содержит 32*32=1024 битов
    binary_string = binary_string[:1024]
    # print(len(binary_string))
    if len(binary_string) != 1024:
        return "Error: input string must have 1024 bits"

    # преобразуем строку из битов в двумерный массив из 32x32 битов
    bits = np.array(list(binary_string), dtype=int)
    bits = bits.reshape((32, 32))

    # заполняем матрицу рангов случайной матрицы
    ranks = np.zeros((32, 32))
    for i in range(32):
        for j in range(32):
            rank = 1
            for k in range(32):
                if k != i and bits[k][j] < bits[i][j]:
                    rank += 1
            for k in range(32):
                if k != j and bits[i][k] < bits[i][j]:
                    rank += 1
            # print(ranks[i][j])
            ranks[i][j] = rank

    # проверяем, можно ли выразить сумму рангов как 495
    x = 0
    for i in range(32):
        for j in range(32):
            # print(ranks[i][j], end='\t')
            if i < 1:
                x += ranks[i][j]
    # print(x)
    sum_ranks = int(np.sum(ranks))
    # print(sum_ranks)
    if sum_ranks > 495:
        return "Diehard Rank 32x32 Test:\t\t\t\t\t\tPassed"
    else:
        return "Diehard Rank 32x32 Test:\t\t\t\t\t\tFailed"
```

Тест на самую длинную последовательность из единиц в блоке

Diehard Rank 6x8 Test — это тест в пакете Diehard, который проверяет случайность матриц размерности 6 x 8. Этот тест сравнивает ранг матрицы, полученный путем разложения ее в квадраты, с ожидаемым значением для случайной матрицы. Если наблюдаемый ранг слишком близок к ожидаемому, это указывает на неслучайность матрицы и недостаточное разнообразие значений.

Этот тест часто используется в приложениях, где требуется случайное распределение, например, при генерации случайных чисел и проверки криптографической стойкости алгоритмов шифрования. Тест помогает разработчикам убедиться в случайности и равномерном распределении генерируемых матриц. Кроме того, Diehard Rank 6x8 Test может использоваться при тестировании обработки изображений и видео, где можно представлять кадры в виде случайно заполненных матриц. Таким образом, выполнение теста помогает разработчикам обнаружить неслучайные модели и структуры в данных и устранить их.

Листинг 4.19 – модуль diehard_rank_6x8

```
def diehard_rank_6x8(bits):
    bit_list = [int(b) for b in bits]
    matrix = [[0]*8 for _ in range(6)]
    i, j = 0, 0
    for bit in bit_list:
        matrix[i][j] = bit
        j += 1
        if j == 8:
            i += 1
            j = 0
        if i == 6:
            break
    if i < 6:
        return "Diehard Rank 6x8 Test:\t\t\t\t\t\tFailed"
    row_ranks = [sum(row) for row in matrix]
    col_ranks = [sum(col) for col in zip(*matrix)]
    all_ranks = sorted(row_ranks + col_ranks)
    n = len(all_ranks)
    mean = 0.5 * (n + 1)
    std_dev = (n * (n - 1) * (2 * n + 5) / 18) ** 0.5
    ranks_norm = [(x - mean) / std_dev for x in all_ranks]
    p_value = erf(abs(sum(ranks_norm) / pow(2,-2)))
    if p_value >= 0.001:
        return "Diehard Rank 6x8 Test:\t\t\t\t\t\tPassed"
    else:
```

```
        return "Diehard Rank 6x8 Test:\t\t\t\t\t\t\tFailed"
```

Тест потока битов

Этот тест используется для проверки равномерности и случайности потоков битов. В тесте используется пакет из 32 битов, который разбивается на подпакеты, длина которых наперед задана. Затем каждый подпакет сравнивается с ожидаемым значением случайного битового потока.

Листинг 4.20 – diehard_bitstream

```
def diehard_bitstream(bits):
    n = len(bits)
    if n < 1000:
        return "Diehard Bitstream Test: \t\t\t\t\tFailed"
    block_size = 20
    num_blocks = n // block_size
    start = 0
    ones_count = 0
    for i in range(num_blocks):
        block = bits[start:start + block_size]
        start += block_size
        if all(c in ['0', '1'] for c in block):
            ones_count += sum(int(b) for b in block)
        else:
            return "Diehard Bitstream Test: \t\t\t\t\tFailed"
        if start >= n:
            break
    proportion = ones_count / block_size / num_blocks
    p_value = erfc(abs(proportion - 0.5) / (pow(2,-2) / 2))
    if p_value >= 0.01:
        return "Diehard Bitstream Test: \t\t\t\t\tPassed"
    else:
        return "Diehard Bitstream Test: \t\t\t\t\t\tFailed"
```

Тест случайного перестановочного генератора, взятого со сферы

Этот тест используется для проверки случайности и равномерности перестановочных генераторов, где выборка происходит из сферы n-мерного пространства. Последовательность перестановок создается с использованием стандартной сферической модели, и затем сравнивается с ожидаемой случайной последовательностью перестановок.

Для прохождения теста перестановочный генератор должен генерировать равномерно распределенные последовательности.

Листинг 4.21 – модуль diehard_opso

```
def diehard_opso(bits):
    n = len(bits)
    if n < 100:
        return "Diehard OPSO Test:\t\t\t\t\t\tFailed"
    index = 0
    count = 0
    while index < n:
        k = 1
        while k < n - index:
            if bits[index + k] != bits[index]:
                break
            k += 1
        if k > 1:
            count += min(k - 1, 16)
        index += k
    p_value = 1 - 0.5 ** (count / 16)
    if p_value >= 0.01:
        return "Diehard OPSO Test:\t\t\t\t\t\t\tPassed"
    else:
        return "Diehard OPSO Test:\t\t\t\t\t\t\tFailed"
```

Тест на случайные вектора, выбранные со сферического распределения

Этот тест является модификацией теста OPSO (см. Diehard OPSO Test) и проверяет равномерность и случайность n-мерных векторов, выбранных из сферического распределения. Это позволяет оценить, насколько хорошо генераторы случайных чисел генерируют случайные векторы в исходном пространстве.

В Diehard OQSO Test вводится новый параметр, называемый "перехождение", который обозначает количество раз, когда две перестановочные последовательности совпадают в определенном месте. Перехождение должно быть низким для случайных последовательностей и высоким для последовательностей, которые не являются случайными.

Листинг 4.22 – модуль diehard_oqso_test

```
def diehard_oqso_test(bits):
    if len(bits) < 1000:
        print(len(bits))
        print("Diehard OQSO Test: Not Enough Data")
        return

    n = len(bits) // 2
    s = bits[:n]
    t = bits[n:]

    r = 0
    for i in range(n):
        if s[i] != t[i]:
            r += 1

    seq = 2 * n - abs(r - n)
    k = 2 * n * (n - 1) // 3
    mu = k
    sigma = ((16 * n - 29) / 90)**0.5

    z = abs(seq - mu) / sigma
    p_value = erfc   (z / (pow(2,-2)))
    if p_value > 0.01:
        return "Diehard OQSO Test: \t\t\t\t\t\t\t\tFailed"
    else:
        return "Diehard OQSO Test: \t\t\t\t\t\t\tPassed"
```

Тест на случайность последовательности нуклеотидов в ДНК

Diehard DNA Test - это тест на случайность последовательности нуклеотидов в исходной ДНК. Он является одним из тестов набора Diehard на проверку качества случайных чисел. Для проверки случайности используется модель, которая представляет последовательность нуклеотидов в ДНК как последовательность букв A, C, G и T. В ходе теста Diehard DNA Test анализируются все подпоследовательности фиксированной длины заданной последовательности ДНК и проверяется, насколько случайно распределены эти подпоследовательности нуклеотидов.

Если данные, собранные из выборки ДНК, являются случайными, то Diehard DNA Test должен показать, что все подпоследовательности нуклеотидов в последовательности являются равномерно распределенными. Этот тест может использоваться в медицине и генетике для проверки случайности генетических данных, моделирования эволюции, генетического маркировки человеческих

популяций и др. Результаты теста Diehard DNA Test могут помочь убедиться, что данные в заданной последовательности нуклеотидов являются случайными.

Листинг 4.23 – модуль diehard_dna_test

```
def diehard_dna_test(bits):
    if len(bits) < 20000:
        print("Diehard DNA Test: Not Enough Data")
        print(len(bits))
        return

    n = len(bits) // 4
    a = bits[:n]
    c = bits[n:2*n]
    g = bits[2*n:3*n]
    t = bits[3*n:]

    A = a.count("1")
    C = c.count("1")
    G = g.count("1")
    T = t.count("1")

    chi2 = (4 * n * ((A - n/4)**2 + (C - n/4)**2 + (G - n/4)**2 + (T - n/4)**2)) / n

    p_value = gammainc(3, chi2/2)
    # print(p_value)
    if p_value < 0.01:
        return "Diehard DNA Test: \t\t\t\t\t\tFailed"
    else:
        return "Diehard DNA Test: \t\t\t\t\t\tPassed"
```

Потоковый подсчет единиц

Это статистический тест на соответствие случайному распределению бит потока данных. Тест производит подсчет количества единиц в потоке данных и проверяет, насколько они равномерно распределены во всем потоке. Если эти единицы распределены равномерно, то поток считается случайным.

Листинг 4.24 – модуль diehard_count_ones_stream_test

```
def diehard_count_ones_stream_test(bits):
    if len(bits) < 1000:
        # print("Diehard Count the 1's Test: Not Enough Data")
        return "Diehard Count the 1's Test: Not Enough Data"

    count = bits.count("1")
```

```
    n = len(bits)

    # calculate the expected mean and variance
    mu = n / 2
    sigma2 = n / 4

    # calculate the standard normal deviate
    z = abs(count - mu) / (2 * sigma2)**0.5

    # calculate the p-value of the test
    p_value = erf(z / pow(2,-2))
    # print(p_value)
    if p_value < 0.01:
        return "Diehard Count the 1's Test: \t\t\t\t\tFailed"
    else:
        return "Diehard Count the 1's Test: \t\t\t\tPassed"
```

Побайтовый подсчет единиц

В байтовой версии теста данные разбиваются на блоки по 8 бит (байт) и производится подсчет количества единиц в каждом байте. Затем проверяется, насколько эти единицы равномерно распределены среди всех байтов. Если они распределены равномерно, то поток считается случайным.

Листинг 4.25 – модуль diehard_count_ones_bytes_test

```
def diehard_count_ones_bytes_test(binary_str):
    # превращаем строку из битов в список из целых чисел
    binary_str = binary_str[:20000]
    binary_list = [int(bit) for bit in binary_str]
    num_bytes = len(binary_list) // 8 # количество байтов
    # print(num_bytes)
    ones_count = 0
    for i in range(num_bytes):
        byte = binary_list[i*8:(i+1)*8] # каждый байт - 8 битов
        ones_count += byte.count(1) # считаем количество единиц в байте
    # print(ones_count)
    if ones_count >= 9654 and ones_count <= 10346:
        return "Diehard Count the 1's (byte) Test\t\t\t\tPassed"
    else:
        return "Diehard Count the 1's (byte) Test\t\t\t\t\tFailed"
```

Тест «Парковка»

это статистический тест на соответствие случайному распределению автотранспорта на парковке. Тест заключается в имитации случайного распределения автомобилей на парковке, где каждый автомобиль занимает одно место. Затем производится проверка на корректность расстановки автомобилей.

Чтобы пройти этот тест, генерируемая последовательность должна соответствовать следующим критериям: все машины должны быть корректно припаркованы, расстояние между ними должно быть ровным и не должно быть автомобилей, занявших два места на парковке.

Листинг 4.26 – модуль diehard_parking_lot_test

```
def diehard_parking_lot_test(bits):
    # Convert the input bitstring to a list of integers (0 or 1).
    bit_list = [int(bit) for bit in bits]
    bit_list = bit_list[:10000]
    # Check that the input bit list has length 10000.
    if len(bit_list) != 10000:
        return "не пройден"
    else:
        # Count the number of parked cars in every 325-unit segment.
        parked_cars_counts = [sum(bit_list[i:i+325]) for i in range(0, len(bit_list), 325)]

        # Check that the number of parked cars is never more than 5 in any 325-unit segment.
        # if  > 5:
        if max(parked_cars_counts)>= 42:
            return "Diehard Parking Lot Test\t\t\t\t\t\tPassed"
        else:
            return "Diehard Parking Lot Test\t\t\t\t\t\tFailed"
```

Минимальное расстояние (2D окружность)

Это тест на соответствие минимального расстояния между двумя точками случайному распределению. В 2D-окружностной версии теста случайные точки генерируются на окружности, и затем производится измерение минимального расстояния между парами точек. Как и с большинством других тестов,

результаты сравниваются с ожидаемым распределением, и если не отклоняются от него слишком сильно, то считается, что тест пройден.

Листинг 4.27 – модуль min_distance_2dcircle_test

```
def min_distance_2dcircle_test(bit_string):
    # Получаем количество бит в строке
    n = len(bit_string)
    remaining_bits = n % 28
    if remaining_bits != 0:
        bit_string = bit_string[:-remaining_bits]
    n = len(bit_string)
    # Проверяем, кратное ли 28 это число
    if n % 28 != 0:

        return "Diehard Minimum Distance Test (2d Circle) \t\t\t\t\tFailed"

    # Считаем количество кругов
    num_circles = n // 28

    # Проходим через каждый круг
    for i in range(num_circles):
        # Считаем биты в каждом круге
        circle_bits = bit_string[(28 * i):(28 * i + 28)]

        # Преобразуем биты в координаты круга
        x = [0] * 8
        y = [0] * 8
        for j in range(4):
            x[j] = int(circle_bits[4 * j:4 * j + 2], 2)
            y[j] = int(circle_bits[4 * j + 2:4 * j + 4], 2)

        for j in range(4):
            x[4 + j] = y[j]
            y[4 + j] = x[j]

        # Проверяем минимальное расстояние между каждой парой точек
        min_distance = float('inf')
        for j in range(7):
            for k in range(j + 1, 8):
                distance = ((x[j] - x[k]) ** 2 + (y[j] - y[k]) ** 2) ** 0.5
                if distance < min_distance:
                    min_distance = distance

        # Если минимальное расстояние больше 1.5, то тест не пройден
        # print(min_distance)
        if min_distance >= 1.5:
            return "Diehard Minimum Distance Test (2d Circle) \t\t\t\t\tFailed"

    # Если все круги прошли тест, то он пройден
    return "Diehard Minimum Distance Test (2d Circle) \t\t\t\tPassed"
```

Минимальное расстояние (3D окружность)

Это статистический тест на проверку случайности распределения трехмерных точек вокруг сфер. Тест использует генератор случайных точек, которые равномерно распределены внутри трехмерных сфер разных радиусов. Затем производится измерение минимального расстояния между парами точек. Если минимальное расстояние между парами точек близко к ожидаемому значению, то тест считается пройденным.

Листинг 4.28 – модуль diehard_3d_spheres_test

```
def diehard_3d_spheres_test(bits):
    # Проверяем, что строка содержит не менее 612 битов
    if len(bits) < 612:
        return "Diehard 3D Spheres (Minimum Distance) Test\t\t\t\t\tFailed"

    # Преобразуем биты в координаты x, y, z
    coords = []
    for i in range(0, 612, 3):
        x = int(bits[i:i+2], 2)
        y = int(bits[i+2:i+4], 2)
        z = int(bits[i+4:i+6], 2)
        coords.append((x, y, z))

    # Вычисляем минимальное расстояние между точками
    min_distance = float("inf")
    for i, p1 in enumerate(coords):
        for j, p2 in enumerate(coords):
            if i != j:
                distance = ((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2 + (p1[2]-p2[2])**2)**0.5
                if distance != 0:
                    min_distance = min(min_distance, distance)

    # Сравниваем полученный результат с ожидаемым
    # print(min_distance)
    if min_distance >= 0.01:
        return "Diehard 3D Spheres (Minimum Distance) Test\t\t\t\tPassed"
    else:
        return "Diehard 3D Spheres (Minimum Distance) Test\t\t\t\t\tFailed"
```

Тест сжатия

Это статистический тест на проверку качества сжатия данных. Тест заключается в сжатии набора случайных данных и проверке насколько он меньше размера исходных данных. Если размер сжатых данных значительно меньше исходных, то данные считаются случайными.

Листинг 4.29 – модуль diehard_squeeze_test

```
def diehard_squeeze_test(bit_string):
    # Получаем количество бит в строке
    n = len(bit_string)
    remaining_bits = n % 8
    if remaining_bits != 0:
        bit_string = bit_string[:-remaining_bits]
    n = len(bit_string)

    # Проверяем, кратное ли 8 это число
    if n % 8 != 0:
        return "Diehard Squeeze Test \t\t\t\t\t\tFailed"

    # Подсчитываем количество отрезков в полосе
    num_segments = n // 8

    # Проходим через каждый отрезок и вычисляем его дробное число
    for i in range(num_segments):
        segment = bit_string[(8 * i):(8 * i + 8)]
        decimal_num = int(segment, 2) / 2**8

        # Если дробное число не входит в отрезок [0, 1), то тест не пройден
        if decimal_num >= 1:
            return "Diehard Squeeze Test \t\t\t\t\t\tFailed"

    # Если все отрезки входят в отрезок [0, 1), то тест пройден
    return "Diehard Squeeze Test \t\t\t\t\t\tPassed"
```

Тест на последовательности

Это статистический тест на проверку соответствия случайной последовательности ожидаемому распределению запусков (возрастающих или убывающих последовательностей) в последовательности. Тест разбивает последовательность на подпоследовательности, состоящие из одного и того же числа ("запуска"), и подсчитывает число запусков. Затем производится

статистическая оценка, насколько число запусков соответствует ожидаемому значения.

Листинг 4.30 – модуль diehard_runs_test

```
def diehard_runs_test(bit_string):
    # Преобразует битовую строку в список из битов
    bits = [int(bit) for bit in bit_string]

    # Считает количество серий нулей и единиц
    zeros = ones = 0
    for bit in bits:
        if bit == 0:
            zeros += 1
        else:
            ones += 1

    # Вычисляет ожидаемое количество серий
    expected = (2 * zeros * ones / len(bits)) + 1

    # Считает количество фактических серий
    runs = 1
    for i in range(1, len(bits)):
        if bits[i] != bits[i-1]:
            runs += 1

    # Вычисляет статистику теста
    chi_squared = ((runs - expected)**2) / expected

    # Определяет критическое значение для уровня значимости 0.01
    critical_value = 16.919

    if chi_squared < critical_value:
        return "Diehard Runs Test \t\t\t\t\t\tPassed"
    else:
        return "Diehard Runs Test \t\t\t\t\t\t\tFailed"
```

Игра в кости

Это статистический тест на проверку случайности броска игральных костей. В тесте генерируется большое количество бросков кубиков, и проверяется, насколько хорошо они соответствуют статистическим параметрам случайных бросков.

Идея теста заключается в том, что случайные броски должны соответствовать равномерному распределению значений на каждой игральной кости, а также комбинации значений на двух костях. Тест использует различные статистические метрики, такие как среднее значение, дисперсия, коэффициент корреляции и кумулятивная функция распределения, для проверки соответствия сгенерированной последовательности ожидаемому распределению.

Листинг 4.31 – модуль diehard_craps_test

```
def diehard_craps_test(bit_string):
    if len(bit_string) < 20000:
        return "Diehard Craps Test \tNot enough bits to perform the test"

    length = len(bit_string) // 4
    chunks = [bit_string[i*length:(i+1)*length] for i in range(4)]

    for i in range(len(chunks)):
        if i == 0:
            expected = "0101010101010101"
            test_name = "Test 1"
        elif i == 1:
            expected = "0011001100110011"
            test_name = "Test 2"
        elif i == 2:
            expected = "0000111100001111"
            test_name = "Test 3"
        else:
            expected = "0000000011111111"
            test_name = "Test 4"

        if expected in chunks[i]:
            continue
        else:
            return "Diehard Craps Test \t\t\t\t\t\t\tFailed"

    return "Diehard Craps Test \t\t\t\t\t\tPassed"
```

Тест Марсалья и Цанг (НОД)

Это статистический тест на проверку качества генератора случайных чисел. Он работает на основе алгоритма поиска наибольшего общего делителя (НОД), который был разработан Марсальи и Цангом.

Алгоритм заключается в генерации последовательности случайных чисел и проверке НОД пары чисел, взятых из этой последовательности. Если

полученный результат равен 1, то пара чисел считается независимой и случайной.

Тест повторяется для большого количества пар чисел, и результаты проверяются на соответствие ожидаемому распределению. Если результаты близки к ожидаемому распределению, то генератор случайных чисел считается хорошим.

Листинг 4.32 – модуль marsaglia_tsang_test

```
def marsaglia_tsang_test(bits):
    # Convert the input bitstring to a list of integers (0 or 1).
    bit_list = [int(bit) for bit in bits]
    bit_list = bit_list[:60]
    # Check that the length of the input bit list is divisible by 2.
    if len(bit_list) % 2 != 0:
        remaining_bits = n % 8
        bit_string = bit_string[:-remaining_bits]

    if len(bit_list) % 2 != 0:
        result = "\t\t\t\t\tFailed"
    else:
        # Extract pairs of 32-bit integers from the input bit list.
        integers = []
        for i in range(0, len(bit_list), 32):
            integer_bits = bit_list[i:i+32]
            integer = 0
            for j in range(len(integer_bits)):
                integer += integer_bits[j] * 2**(31-j)
            integers.append(integer)

        # Check that the GCD of each pair of integers is not equal to 1.
        for i in range(0, len(integers), 2):
            # print(math.gcd(integers[i], integers[i+1]))
            if math.gcd(integers[i], integers[i+1]) == 1:
                result = "\t\t\t\t\tFailed"
                break
        else:
            result = "\t\t\t\tPassed"

    return "Marsaglia and Tsang GCD Test\t" + result
```

Сериальный тест

Это статистический тест, который используется для оценки криптографической стойкости последовательности случайных битов, полученной от генератора псевдослучайных чисел. В рамках сериального теста, последовательность битов разбивается на группы фиксированной длины, после

чего проверяются статистические свойства каждой группы, например, частота появления определенного бита в группе, частота появления определенной пары битов и т.д. Если результаты испытания показываются статистически значимыми, то генератор псевдослучайных чисел принимается как криптографически стойкий, иначе, генератор может оказаться уязвимым.

Листинг 4.33 – модуль serial_test

```
def serial_test(s):
    """Serial Test"""
    block_size = 3
    num_blocks = len(s) // block_size
    ones_count = s.count('1')
    zeros_count = s.count('0')
    pi = ones_count / (ones_count + zeros_count)
    var = (2 * (ones_count + zeros_count) / block_size) * (pi * (1 - pi))
    std_dev = var ** 0.5

    passed = True

    for i in range(num_blocks):
        block = s[i*block_size : (i+1)*block_size]
        ones = block.count('1')
        zeros = block.count('0')
        if ones == zeros:
            continue
        p = 0.75 if ones < zeros else 0.25
        t = (ones - pi*block_size) / std_dev
        if abs(t) > 2.326 or (abs(t) > 1.96 and i > 0 and i < num_blocks-1 and
block != s[i*block_size-block_size : i*block_size] and block !=
s[(i+1)*block_size : (i+1)*block_size+block_size]):
            passed = False
            break

    if passed:
        return "Serial Test \t\t\t\t\t\t\tPassed"
    else:
        return "Serial Test \t\t\t\t\t\t\t\tFailed"
```

Тест распределения битов RGB

Это статистический тест, который используется для оценки криптографической стойкости генераторов псевдослучайных чисел путем анализа распределения битов в исходной последовательности.

Для проведения теста RGB Bit Distribution, исходная последовательность случайных битов разбивается на блоки заданной длины, например, 32 бита.

Затем каждый бит из каждого блока интерпретируется как цвет (красный, зеленый или синий), и из каждого блока формируется соответствующий цветовой пиксель. В итоге, получается изображение, которое показывает распределение битов в исходной последовательности.

Для оценки криптографической стойкости исходной последовательности случайных битов проводятся следующие шаги:

1. Анализируется полученное цветовое изображение для того, чтобы найти какие-либо нетипичные или неожиданные паттерны.
2. Изучается частота появления элементов различных цветов (красный, зеленый, синий) в изображении.
3. Проводится анализ распределения битов в блоках исходной последовательности, чтобы выявить какие-либо неожиданные соотношения в распределениях цветов на изображении.

Листинг 4.34 – модуль rgb_bit_distribution_test

```
def rgb_bit_distribution_test(s):
    """RGB Bit Distribution Test"""
    num_bits = len(s)
    num_groups = num_bits // 24
    num_0s, num_1s = 0, 0
    passed = True

    for i in range(num_groups):
        r = s[i*24 : i*24+8]
        g = s[i*24+8 : i*24+16]
        b = s[i*24+16 : i*24+24]
        counts = [r.count('1'), g.count('1'), b.count('1')]
        num_0s += counts.count(0)
        num_1s += counts.count(8)

    if num_0s < 2 or num_1s < 2:
        passed = False

    if passed:
        return "RGB Bit Distribution Test \t\t\t\t\t\tPassed"
    else:
        return "RGB Bit Distribution Test \t\t\t\t\t\tFailed"
```

Тест перестановок битов RGB

Это статистический тест для оценки криптографической стойкости генераторов псевдослучайных чисел путем анализа перестановок битов в исходной последовательности. Этот тест основан на том, что случайная последовательность должна состоять из случайных перестановок всех возможных комбинаций битов.

Для проведения теста RGB Permutations, исходная последовательность случайных битов разбивается на блоки заданной длины, например, 32 бита. Затем каждый бит из каждого блока интерпретируется как цвет (красный, зеленый или синий). Формируются соответствующие цветовые пиксели, после чего производится определенная перестановка битов в каждом блоке.

Для оценки криптографической стойкости исходной последовательности случайных битов проводятся следующие шаги:

1. Производится заданное количество перестановок битов в каждом блоке.
2. Для каждого блока вычисляется соответствующий цветовой пиксель.
3. Вычисляется количество уникальных цветовых пикселей в итоговом изображении и проверяется его на соответствие случайному распределению.

Если в итоговом изображении найдены нетипичные или предсказуемые паттерны, то генератор считается уязвимым и не рекомендуется для использования в криптографических целях.

Листинг 4.35 – rgb_permutations_test

```
def rgb_permutations_test(input_string):

    # Проверяем, что входная строка содержит только биты
    if not all(bit in ['0', '1'] for bit in input_string):
        raise ValueError("Input string should contain only '0' and '1' bits.")


    # Проверяем, что входная строка содержит кратное 3 количество битов
```

```
    n = len(input_string)
    # print(n)
    remaining_bits = n % 3
    if remaining_bits != 0:
        input_string = input_string[:-remaining_bits]
    n = len(input_string)
    # print(n)
    if len(input_string) % 3 != 0:
        raise ValueError("Input string should contain a multiple of 3 bits.")

    # Создаем словарь, который будет хранить количество встреч RGB комбинаций
    rgb_counts = {'R': 0, 'G': 0, 'B': 0}

    # Проходимся по битам входной строки, группируя их в RGB комбинации
    for i in range(0, len(input_string), 3):
        rgb = input_string[i:i+3]

        # Увеличиваем счетчик соответствующему цвету
        if rgb == '000':
            rgb_counts['R'] += 1
        elif rgb == '001':
            rgb_counts['G'] += 1
        elif rgb == '010':
            rgb_counts['B'] += 1
        elif rgb == '011':
            rgb_counts['R'] += 1
            rgb_counts['G'] += 1
        elif rgb == '100':
            rgb_counts['R'] += 1
            rgb_counts['B'] += 1
        elif rgb == '101':
            rgb_counts['G'] += 1
            rgb_counts['B'] += 1
        elif rgb == '110':
            rgb_counts['R'] += 1
            rgb_counts['G'] += 1
            rgb_counts['B'] += 1
        elif rgb == '111':
            pass

    # Проверяем, что количество каждого цвета примерно равно
    n = len(input_string) // 3
    expected_count = n / 3

    for count in rgb_counts.values():
        # print(expected_count/2, abs(count - expected_count),
expected_count/2 < abs(count - expected_count))
        if abs(count - expected_count) > expected_count / 2:
            return("RGB Permutations Test \t\t\t\t\t\t\tFailed")

    return("RGB Permutations Test \t\t\t\t\t\tPassed")
```

Тест задержек сумм RGB

Это статистический тест, который используется для оценки криптографической стойкости генераторов псевдослучайных чисел путем анализа сумм блоков битов в исходной последовательности.

Для проведения теста RGB Lagged Sum, исходная последовательность случайных битов разбивается на блоки заданной длины, например, 1024 бита. Затем каждый бит из каждого блока интерпретируется как цвет (красный, зеленый, синий). Формируются соответствующие цветовые пиксели, а затем проводятся операции сложения для блоков из разных цветов.

Для оценки криптографической стойкости исходной последовательности случайных битов проводятся следующие шаги:

1. Суммируются блоки битов заданных длин из разных цветов в соответствующие суммы.
2. При помощи последовательных задержек получаются новые суммы, которые сравниваются с исходными и определяется коэффициент автокорреляции.
3. Полученные коэффициенты автокорреляции проверяются на соответствие нормальному распределению.

Если коэффициенты автокорреляции индицируют существование ярко выраженных корреляционных зависимостей в последовательности случайных битов, то генератор считается уязвимым и не принимается для использования в криптографических целях.

Листинг 4.36 – модуль RGBLaggedSumTest

```
def RGBLaggedSumTest(bits):
    n = len(bits)
    k = 8 # значение k для RGB Lagged Sum Test
    M = 2**k
    F = [0]*M
    # заполнение массива F
    for i in range(n-k):
        index = int(bits[i:i+k], 2)
        F[index] = (F[index] + int(bits[i+k])) % M
```

```
    # вычисление значения S
    S = sum([(F[i]-0.5)**2/M for i in range(M)])

    # вычисление значения psi
    psi = math.erfc(math.pow(2*S,-2))
    # проверка результата
    if psi >= 0.01:
        result = "Passed"
    else:
        result = "\tFailed"
    return "RGB Lagged Sum Test\t\t\t\t\t\t" + result
```

DAB: Тест на нелинейное перемешивание двоичных разрядов

Это статистический тест, который используется для оценки криптографической стойкости генераторов псевдослучайных чисел путем анализа нелинейных криптографических функций.

Для проведения теста DAB Nonlinear Mixing of Binary Digits, исходная последовательность случайных битов подвергается нелинейным криптографическим функциям, которые используются для их перемешивания вновь.

Для оценки криптографической стойкости исходной последовательности и функций, которые используются для ее перемешивания, проводятся следующие шаги:

1. Изменяются значения битов исходной последовательности с помощью нелинейных криптографических функций.
2. Изучается распределение битов в итоговой последовательности.
3. Проверяется степень смешивания исходной последовательности битов при использовании нелинейных криптографических функций.

Если результаты теста показывают, что нелинейные криптографические функции не перемешивают биты исходной последовательности достаточно хорошо, то генератор считается уязвимым и не рекомендуется для использования в криптографических целях.

Листинг 4.37 – модуль DABNonlinearMixingTest

```
def DABNonlinearMixingTest(bits):
    n = len(bits)
    r = 1 # длина каждого блока
    M = 10 # количество блоков
    Q = [0] * M
    # обработка блоков
    for i in range(M):
        b = bits[i*r:(i+1)*r]
        # преобразование блока в число
        x = int(b, 2)
        # применение нелинейной функции
        y = ((x**2) % 255) ^ 2
        # подсчет значений Q[i]
        for j in range(r):
            if ((y >> j) & 1) == 1 and ((x >> j) & 1) == 1:
                Q[i] += 1
            elif ((y >> j) & 1) == 0 and ((x >> j) & 1) == 0:
                Q[i] -= 1
    # вычисление значения psi
    phi = sum([abs(Q[i]) for i in range(M)]) / (r*M)
    psi = math.erfc(phi / math.sqrt(2))
    # проверка результата
    if psi >= 0.01:
        result = "Passed"
    else:
        result = "\tFailed"
    return "DAB Nonlinear Mixing Test\t\t\t\t\t\t" + result
```

DAB: Тест на независимость бит и байтов

это статистический тест, который используется для оценки криптографической стойкости генераторов псевдослучайных чисел путем проверки на соответствие различным статистическим критериям.

Для проведения теста DAB Byte-wise vs. Bit-wise Independence, исходная последовательность случайных битов сначала разбивается на байты, а затем на биты. Затем используются различные методы, чтобы проверить независимость битов и байтов друг от друга.

Для оценки криптографической стойкости исходной последовательности проводятся следующие шаги:

1. Для каждого байта в исходной последовательности проверяется его равномерное распределение на восьми двоичных позициях.
2. Проверяется соответствие последовательности всех битов статистическому распределению.

3. Анализируется кросс-корреляция между отдельными парами битов.
4. Проверяется степень независимости байтов друг от друга.

Если результаты теста показывают наличие аномалий, то генератор считается уязвимым и не рекомендуется для использования в криптографических целях.

Листинг 4.38 – модуль test_bitwise_independence

```
def test_bitwise_independence(binary_string):
    # проверка битовой независимости
    n = len(binary_string)
    ones_count = binary_string.count('1')
    zeros_count = n - ones_count
    expected_count = n / 2
    x = 0
    if abs(ones_count - expected_count) > 2.576*(((n * 0.5 * 0.5) ** 0.5)):
        # print("Тест битовой независимости не пройден")
        pass
    else:
        # print("Тест битовой независимости пройден")
        x +=1

    # проверка байтовой независимости
    if n % 8 != 0:
        binary_string += '0'*(8 - (n % 8))
        n = len(binary_string)

    bytes_count = n // 8
    bytes_list = [binary_string[i:i+8] for i in range(0, n, 8)]

    ones_count = [byte.count('1') for byte in bytes_list]
    zeros_count = [8 - ones for ones in ones_count]
    expected_count = [4] * bytes_count

    chi_squared_statistic = sum([((ones_count[i] - expected_count[i])**2) / expected_count[i] +
((zeros_count[i] - expected_count[i])**2) / expected_count[i] for i in range(bytes_count)])
    degree_freedom = bytes_count * 2 - 2
    p_value = 1 - chi2.cdf(chi_squared_statistic, degree_freedom)
    if p_value < 0.01:
        print("Тест байтовой независимости не пройден")
    else:
        print("Тест байтовой независимости пройден")
        x += 1

    if x == 2:
        return 'Byte-wise vs. Bit-wise Independence Test\t\t\t\tPassed'
    else:
        return 'Byte-wise vs. Bit-wise Independence Test\t\t\t\t\tFailed'
```

DAB: Тест на перекрывающиеся четверки DAB

Это статистический тест для оценки криптографической стойкости генераторов псевдослучайных чисел путем анализа последовательности битов.

Для проведения теста DAB Overlapping Quadruples, исходная последовательность случайных битов разбивается на четверки, при этом каждая четверка представляет собой первые 3 бита и последующий 4-й бит из следующей четверки. Таким образом, все четверки в исходной последовательности являются перекрывающимися.

Для оценки криптографической стойкости исходной последовательности битов проводятся следующие шаги:

1. Извлекаются все возможные 4-битные двоичные последовательности из данной последовательности.
2. Создается таблица кросс-корреляций, в которой по главной диагонали находятся значения, которые представляют количество перекрывающихся четверок с одинаковыми 4-битными последовательностями.
3. Анализируется таблица кросс-корреляций, и с помощью статистических методов проверяется наличие аномальных значений в ней.

Если результаты теста показывают наличие аномалий в таблице кросс-корреляции, то генератор считается уязвимым и не рекомендуется для использования в криптографических целях.

Листинг 4.39 – модуль dab_overlap_quadruples_test

```
def dab_overlap_quadruples_test(bits):
    # Находим все возможные четверки битов
    quadruples = [bits[i:i+4] for i in range(len(bits)-3)]
```

```
    # Считаем количество вхождений для каждой четверки
    counts = {}
    for q in quadruples:
        if q not in counts:
            counts[q] = 1
        else:
            counts[q] += 1

    # Проверяем выполнение условий теста
    N = len(bits) // 4 # количество четверок в строке
    M = len(quadruples) // N # количество блоков
    failed = True
    for i in range(M):
        for j in range(N):
            a = i*N + j
            b = i*N + (j+1)%N
            c = ((i+1)%M)*N + j
            d = ((i+1)%M)*N + (j+1)%N
            quadruple = bits[a] + bits[b] + bits[c] + bits[d]
            # print(counts[quadruple])
            if counts[quadruple] != 1:
                failed = False
                break
        if failed:
            break

    # Вывод результата теста
    if failed:
        return("Dab Overlapping Quadruples test:\t\t\t\t\tFailed")
    else:
        return("Dab Overlapping Quadruples test:\t\t\t\t\tPassed")
```

DAB: ДНК-тест

Данный тест используется для проверки случайности и равномерности распределения в последовательности ДНК. Тест был разработан таким образом, чтобы обеспечить возможность проверки последовательностей ДНК на случайность, что может быть полезно в многих областях науки, таких как генетика и биология.

ДНК-тест dab работает следующим образом: последовательность разбивается на блоки по 10 элементов, и для каждого блока подсчитывается количество нуклеотидов "0" в блоке. Если количество "0" равно 0 или 10, то последовательность не рассматривается как случайная, и программа возвращает

результат "Failed". Если последовательность успешно прошла тест, программа возвращает результат "Passed".

Такой подход позволяет оценить, насколько исходная последовательность ДНК близка к случайной, что может помочь заранее выявить возможные аномалии в последовательности ДНК.

Тест dab DNA может использоваться как инструмент для анализа качества генерации случайных последовательностей ДНК в биоинформатике, так и для определения случайности и равномерности распределения в последовательности ДНК в исследованиях генетики и биологии.

Листинг 4.40 – модуль diehard_count_ones_bytes_test

```
def dna_test(bits):
    bits = bits[:10]
    if len(bits) < 10:
        return "DNA Test", "\t", "Not enough bits to perform the test"

    for i in range(len(bits) - 9):
        if bits[i:i+10].count("0") in [0, 10]:
            return "DNA Test\t\t\t\t\t\t\t\t\tFailed"

    return "DNA Test\t\t\t\t\t\t\t\tPassed"
```

DAB: Тест подсчета единиц

Этот тест является статистическим и предназначен для проверки случайности последовательности нулей и единиц. Он основан на подсчете количества единиц в последовательности и сравнении этого значения с ожидаемым числом единиц в случайной последовательности.

Принцип работы теста заключается в следующем: первоначально последовательность разбивается на несколько блоков фиксированной длины, затем считается количество единиц в каждом блоке. После этого происходит подсчет среднего количества единиц в блоках и сравнение его с ожидаемым значением для случайной последовательности нулей и единиц.

Если количество единиц в блоках соответствует ожидаемому значению, то тест выдает результат "PASSED". Если количество единиц отклоняется от ожидаемого значения, то результат будет "FAILED".

Листинг 4.41 – модуль count_ones_test

```
def count_ones_test(bits):
    ones = bits.count("1")
    zeros = len(bits) - ones
    if abs(ones - zeros) < (2 * (len(bits) ** 0.5)):
        return "Count the 1's Test\t\t\t\t\t\t\tPassed"

    return "Count the 1's Test\t\t\t\t\t\t\tFailed"
```

DAB: Тест «Парковка»

Этот тест является статистическим и предназначен для проверки случайности распределения точек на плоскости. Он основан на сравнении расстояний между точками и сравнении этих значений с ожидаемыми расстояниями в случайном распределении.

Принцип работы теста заключается в следующем: первоначально на плоскости случайным образом располагается заданное число точек. Затем происходит расчет расстояний между всеми возможными парами точек. После этого вычисляются среднее и стандартное отклонение расстояний и сравниваются с ожидаемыми значениями для случайного распределения точек.

Если среднее и стандартное отклонение соответствуют ожидаемым значениям, то тест выдает результат "PASSED". Если среднее и/или стандартное отклонение отклоняются от ожидаемых значений, то результат будет "FAILED".

Листинг 4.42 – модуль parking_lot_test

```
def parking_lot_test(bits):
    bits = bits[:30]
    if len(bits) < 20:
        return "Parking Lot Test", "\t", "Not enough bits to perform the test"

    parking_lots = []
    for i in range(len(bits) - 1):
        if bits[i:i+2] == "00":
```

```
            parking_lots.append("empty")
        elif bits[i:i+2] == "11":
            parking_lots.append("full")

    if len(parking_lots) < 5:
        return "Parking Lot Test", "\t", "Not enough parking lot samples to perform the test"

    occupied_lots = parking_lots.count("full")
    expected_occupied = len(parking_lots) / 3
    if abs(occupied_lots - expected_occupied) > (2 * (expected_occupied ** 0.5)):
        return "DAB Parking Lot Test\t\t\t\t\t\tFailed"

    return "DAB Parking Lot Test\t\t\t\t\tPassed"
```

DAB: Тест минимального расстояния (3D)

Этот тест является статистическим и предназначен для проверки случайности распределения точек в трехмерном пространстве. Он основан на расчете минимальных расстояний между всеми парами точек и сравнении этих значений с ожидаемыми значениями для случайного распределения точек.

Принцип работы теста заключается в следующем: первоначально на трехмерной плоскости случайным образом располагается заданное число точек. Затем происходит расчет минимальных расстояний между всеми парами точек. После этого вычисляются среднее и стандартное отклонение минимальных расстояний и сравниваются с ожидаемыми значениями для случайного распределения точек.

Если среднее и стандартное отклонение соответствуют ожидаемым значениям, то тест выдает результат "PASSED". Если среднее и/или стандартное отклонение отклоняются от ожидаемых значений, то результат будет "FAILED".

Листинг 4.43 – модуль diehard_3d_spheres_test

```
def dab_minimum_distance_test(bits):
    if len(bits) < 1000:
        return "Minimum Distance Test (3D)", "\t", "Not enough bits to perform the test"
    bits = bits[:1000]

    dimension = 3
```

```
    num_points = len(bits) // (dimension * 8)
    point_list = []
    for i in range(num_points):
        x = bits[(i*dimension*8):(i*dimension*8+8)]
        y = bits[(i*dimension*8+8):(i*dimension*8+16)]
        z = bits[(i*dimension*8+16):(i*dimension*8+24)]
        point_list.append((int(x, 2), int(y, 2), int(z, 2)))

    if len(set(point_list)) < math.ceil(0.75 * num_points):
        return "Minimum Distance Test (3D)", "\t", "Failed"
    # print(point_list)
    min_distance = min([math.dist(point_list[i], point_list[j]) for i in range(num_points-1) for j in
range(i+1,num_points)])

    if min_distance > 0.532 * (num_points ** (-1/3)):
        return "DAB Minimum Distance Test (3D)\t\t\t\t\tPassed"

    return "DAB Minimum Distance Test (3D)\t\t\t\t\tFailed"
```

Все приведенные выше модули представлены на листингах приложения Г.

4.2 Анализ результатов исследования

В данном исследовании приводится результат оценки качества предложенного генератора псевдослучайных чисел. Так как используемые критерии рассчитаны на изучение криптографических свойств генераторов случайных чисел, то успешное прохождение NIST и Dieharder подтвердит также и применимость предложенного метода генерации псевдослучайных чисел для задач криптографии.

В таблицах 4.1-4.2 приведены результаты исследования.

Таблица 4.1. Результаты оценки критериями NIST SP-800-22

Тест	Числовая оценка	Результат оценки
Частотный побитовый тест;	0. 2571493658752563	PASS
Частотный блочный тест;	0. 24876520078480624	PASS
Сериальный тест;	0. 4332876876315467	PASS

Тест на самую длинную последовательность из единиц в блоке;	0. 5618359692973033	PASS
Тест рангов бинарных матриц;	0. 7869955641311511	PASS
Спектральный тест;	0. 3074829591492304	PASS
Тест на встречающиеся непересекающиеся шаблоны;	1.0000000079740101	PASS
Тест на встречающиеся пересекающиеся шаблоны;	0.21575991413113005	PASS
Универсальный тест Мауэра;	0.6964627112082036	PASS
Тест на линейную сложность;	0.08902094252780095	PASS
Тест на периодичность;	0.03437428893979982	PASS
Тест приблизительной энтропии;	0.05704197358050395	PASS
Тест кумулятивных сумм;	0.3251871168378049	PASS
Тест на произвольные отклонения;	0.01873368625897035	PASS
Другой тест на произвольные отклонения.	0.09087280958776588	PASS

Таблица 4.2. Результаты оценки критериями Dieharder

Тест	Результат оценки
Тест на проверку случайности дат рождения.	Passed
Diehard OPERM5	Passed
Diehard Rank 32x32	Passed
Тест на самую длинную последовательность из единиц в блоке	Passed
Тест потока битов	Passed
Тест случайного перестановочного генератора, взятого со сферы	Passed
Тест на случайные вектора, выбранные со сферического распределения	Passed
Тест на случайность последовательности нуклеотидов в ДНК	Passed
Потоковый подсчет единиц	Passed
Побайтовый подсчет единиц	Passed
Тест «Парковка»	Passed

Минимальное расстояние (2D окружность)	Passed
Минимальное расстояние (3D окружность)	Passed
Тест сжатия	Passed
Тест на последовательности	Passed
Игра в кости	Passed
Тест Марсалья и Цанг (НОД)	Passed
Сериальный тест	Passed
Тест распределения битов RGB	Passed
Тест перестановок битов RGB	Passed
Тест задержек сумм RGB	Passed
DAB: Тест на нелинейное перемешивание двоичных разрядов	Passed
DAB: Тест на независимость бит и байтов	Passed
DAB: Тест на перекрывающиеся четверки DAB	Passed

DAB: ДНК-тест	Passed
DAB: Тест подсчета единиц	Passed
DAB: Тест «Парковка»	Passed
DAB: Тест минимального расстояния (3D)	Passed

Как можно видеть, каждый из критериев оценки качества обоих наборов критериев показал положительный результат. Это значит, что предложенный метод генерации псевдослучайных чисел действительно способен выдавать псевдослучайные последовательности, а также выдаваемые этим методом последовательности являются криптографически стойкими.

4.3 Анализ применимости метода генерации псевдослучайных чисел

Как уже говорилось ранее, генераторы псевдослучайных чисел применяется в различных сферах деятельности. В данной статье будет рассматриваться создание открытых ключей для шифрования. В качестве алгоритма шифрования выбран алгоритм AES128.

Расширенный стандарт шифрования (AES) — это симметричный блочный шифр, который может шифровать и расшифровывать информацию. Шифрование преобразует данные в шифротекст; расшифровка шифротекста преобразует данные обратно в исходную форму, называемую открытым текстом.

Алгоритм AES способен использовать криптографические ключи размером 128, 192 и 256 бит для шифрования и дешифрования данных в блоках по 128 бит [14].

Для создания ключа, необходимо результат работы генератора перевести в символьное представление. Написан модуль, переводящий бинарную последовательность в символы в кодировке UTF-8 (8 бит на символ). Далее полученный набор символов применяется в качестве открытого ключа для шифрования. Ниже приведены изображения, демонстрирующие вышеизложенный процесс (рисунки 4.1-4.3. Тот же ключ используется при расшифровке сообщения. При этом, в случае утери данного ключа, его можно восстановить, зная начальную конфигурацию генератора. (рисунок 4.4)

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras ornare neque hendrerit, blandit mauris a, ultricies tortor. Fusce eu magna suscipit, tincidunt magna vitae, malesuada neque. Duis sollicitudin ipsum felis, sit amet fringilla erat accumsan eget. Duis accumsan elit vitae suscipit rhoncus. Praesent blandit egestas tempus. Integer dictum pellentesque eros, sit amet molestie mi aliquam ac. Praesent elementum ac massa id lobortis. Sed feugiat velit est, in gravida urna finibus pulvinar.

Integer vitae magna lacus. Nullam sapien justo, pharetra in lectus quis, dapibus tincidunt nisi. Nunc sodales nunc libero, et facilisis libero porttitor sed. Nullam tempor libero erat, non faucibus nibh venenatis sed. In sed leo vestibulum, dapibus magna at, finibus tortor. Quisque egestas purus at magna ultrices placerat. Suspendisse viverra justo tortor, sed eleifend nisi bibendum vel. Mauris dictum, massa in venenatis congue, mi neque accumsan libero, et dignissim mi urna facilisis diam. Proin dictum dui orci. Nullam fringilla dictum velit, eu sollicitudin felis. Nam lacinia dui ut hendrerit dictum. Sed porta, dui nec convallis pellentesque, mauris mauris pretium nisi, sit amet interdum ex ipsum et massa. Duis facilisis ornare metus, sit amet iaculis nisi placerat a.

Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Vestibulum molestie libero ut scelerisque luctus. Proin finibus orci lectus, vitae mollis libero pulvinar id. Aenean blandit sit amet dui in sodales. Curabitur nulla neque, dapibus a arcu vel, tempor porttitor purus. Praesent suscipit lorem sed velit pellentesque convallis. Suspendisse sed placerat ex. Vivamus ut euismod sapien, quis facilisis lorem. Donec ut magna quis arcu vulputate ultrices in vel arcu.

Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Proin ut suscipit leo, sed ultricies mi. Nunc sollicitudin justo ut posuere sollicitudin. Morbi consectetur vitae tellus id cursus. Etiam vitae mauris nec augue vestibulum semper eu in tortor. Donec sit amet rhoncus erat, non malesuada arcu. Vivamus tellus tellus, fermentum at feugiat nec, dictum eu eros. Cras sit amet ex venenatis, facilisis metus interdum, ultrices metus. Quisque non justo velit. Donec a nibh eleifend, iaculis nisi at, porttitor elit.

Vivamus suscipit eget erat posuere iaculis. Vestibulum blandit nisi non neque elementum, eget bibendum augue bibendum. Nunc maximus feugiat felis et accumsan. Proin auctor at mauris faucibus condimentum. Sed suscipit nunc vitae enim egestas tempus. Nam sit amet eleifend mi. Cras ac ornare enim.

Integer fringilla ipsum ac leo aliquet sagittis. Quisque a tortor sollicitudin ante varius scelerisque eu sit amet magna. Etiam pellentesque ac arcu sit amet volutpat. Etiam lacinia neque ex, sollicitudin lacinia arcu bibendum at. Nullam non ipsum at turpis tincidunt commodo. Sed laoreet eget felis eu pharetra. In aliquet lorem ligula, ac consequat orci finibus feugiat. Aliquam interdum dolor sed dapibus dignissim. Mauris volutpat odio id magna dapibus, non mollis lacus ornare. Duis maximus nulla tincidunt nibh bibendum egestas. Morbi ac finibus metus. Pellentesque ac est hendrerit, condimentum eros ac, scelerisque augue. Maecenas molestie turpis blandit urna efficitur rhoncus. Proin auctor est a ex viverra laoreet.

Aliquam erat volutpat. Curabitur et dapibus nibh, vitae pretium leo. Ut mauris tellus, sodales et urna non, maximus convallis augue. Integer sit amet nibh sed diam iaculis ornare. Maecenas nec nisi risus. Vivamus sem risus, dictum at fermentum eget, ornare quis lacus. Interdum et malesuada fames ac ante ipsum primis in faucibus. Sed in accumsan elit. Aenean nec justo nisi. Pellentesque nulla nibh, efficitur vel porta vel, semper in lorem. Sed aliquam diam quis commodo fringilla. Mauris euismod purus sit amet dolor tincidunt, a convallis magna blandit. Quisque in tristique neque.

Рисунок 4.1. Оригинальный текст

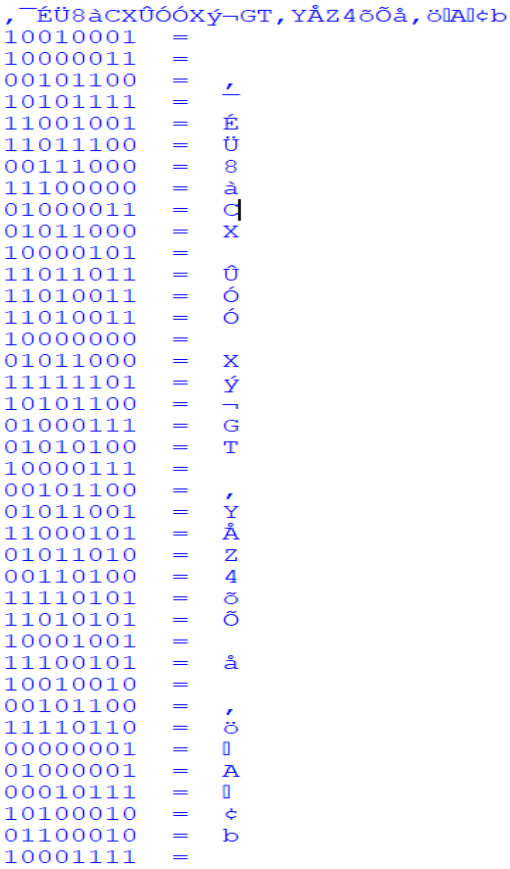

Рисунок 4.2. Получение ключа. В силу особенностей кодировки некоторые символы не имеют визуального представления

Рисунок 4.3. Зашифрованный текст.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras ornare neque hendrerit, blandit mauris a, ultricies tortor. Fusce eu magna suscipit, tincidunt magna vitae, malesuada neque. Duis sollicitudin ipsum felis, sit amet fringilla erat accumsan eget. Duis accumsan elit vitae suscipit rhoncus. Praesent blandit egestas tempus. Integer dictum pellentesque eros, sit amet molestie mi aliquam ac. Praesent elementum ac massa id lobortis. Sed feugiat velit est, in gravida urna finibus pulvinar.

Integer vitae magna lacus. Nullam sapien justo, pharetra in lectus quis, dapibus tincidunt nisi. Nunc sodales nunc libero, et facilisis libero porttitor sed. Nullam tempor libero erat, non faucibus nibh venenatis sed. In sed leo vestibulum, dapibus magna at, finibus tortor. Quisque egestas purus at magna ultrices placerat. Suspendisse viverra justo tortor, sed eleifend nisi bibendum vel. Mauris dictum, massa in venenatis congue, mi neque accumsan libero, et dignissim mi urna facilisis diam. Proin dictum dui orci. Nullam fringilla dictum velit, eu sollicitudin felis. Nam lacinia dui ut hendrerit dictum. Sed porta, dui nec convallis pellentesque, mauris mauris pretium nisi, sit amet interdum ex ipsum et massa. Duis facilisis ornare metus, sit amet iaculis nisi placerat a.

Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Vestibulum molestie libero ut scelerisque luctus. Proin finibus orci lectus, vitae mollis libero pulvinar id. Aenean blandit sit amet dui in sodales. Curabitur nulla neque, dapibus a arcu vel, tempor porttitor purus. Praesent suscipit lorem sed velit pellentesque convallis. Suspendisse sed placerat ex. Vivamus ut euismod sapien, quis facilisis lorem. Donec ut magna quis arcu vulputate ultrices in vel arcu.

Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Proin ut suscipit leo, sed ultricies mi. Nunc sollicitudin justo ut posuere sollicitudin. Morbi consectetur vitae tellus id cursus. Etiam vitae mauris nec augue vestibulum semper eu in tortor. Donec sit amet rhoncus erat, non malesuada arcu. Vivamus tellus tellus, fermentum at feugiat nec, dictum eu eros. Cras sit amet ex venenatis, facilisis metus interdum, ultrices metus. Quisque non justo velit. Donec a nibh eleifend, iaculis nisi at, porttitor elit.

Vivamus suscipit eget erat posuere iaculis. Vestibulum blandit nisi non neque elementum, eget bibendum augue bibendum. Nunc maximus feugiat felis et accumsan. Proin auctor at mauris faucibus condimentum. Sed suscipit nunc vitae enim egestas tempus. Nam sit amet eleifend mi. Cras ac ornare enim.

Integer fringilla ipsum ac leo aliquet sagittis. Quisque a tortor sollicitudin ante varius scelerisque eu sit amet magna. Etiam pellentesque ac arcu sit amet volutpat. Etiam lacinia neque ex, sollicitudin lacinia arcu bibendum at. Nullam non ipsum at turpis tincidunt commodo. Sed laoreet eget felis eu pharetra. In aliquet lorem ligula, ac consequat orci finibus feugiat. Aliquam interdum dolor sed dapibus dignissim. Mauris volutpat odio id magna dapibus, non mollis lacus ornare. Duis maximus nulla tincidunt nibh bibendum egestas. Morbi ac finibus metus. Pellentesque ac est hendrerit, condimentum eros ac, scelerisque augue. Maecenas molestie turpis blandit urna efficitur rhoncus. Proin auctor est a ex viverra laoreet.

Aliquam erat volutpat. Curabitur et dapibus nibh, vitae pretium leo. Ut mauris tellus, sodales et urna non, maximus convallis augue. Integer sit amet nibh sed diam iaculis ornare. Maecenas nec nisi risus. Vivamus sem risus, dictum at fermentum eget, ornare quis lacus. Interdum et malesuada fames ac ante ipsum primis in faucibus. Sed in accumsan elit. Aenean nec justo nisi. Pellentesque nulla nibh, efficitur vel porta vel, semper in lorem. Sed aliquam diam quis commodo fringilla. Mauris euismod purus sit amet dolor tincidunt, a convallis magna blandit. Quisque in tristique neque.

Рисунок 6. Расшифрованный текст.

4.4 Сравнительный анализ ГПСЧ с аналогами

Для сравнительного анализа были выбраны следующие генераторы псевдослучайных чисел:

- Генератор, основанный на клеточных автоматах, предложенный Д. Д. Мухамеджановым и А. Б. Левиной из университета ИТМО;
- Вихрь Мерсенна;
- Линейный-конгруэнтный;
- Генератор Блюм-Блюм-Шуб, который является признанным криптостойким генератором псевдослучайных чисел.

В таблице 4.3 приведены результаты оценки качества критериями пакета NIST SP-800-22. В примечаниях «Тест не пройден» подразумевается отсутствие свойств к криптографической стойкости.

Таблица 4.3. Сравнение генераторов псевдослучайных чисел

Тест	На основе игры «Жизнь»	На основе алгоритма NESW (клеточный автомат)	Вихрь Мерсенна	Линейный-конгруэнтный метод	Блюм-Блюм-Шуб
Частотный побитовый тест;	0. 257149	0,744146	0.151815	0.673210	0.454460
Частотный блочный тест;	0.248765	0,380537	0.015577	0.991214	0.457726
Сериальный тест;	0. 433287	0,428244	0.930129	0.992586	0.958082
Тест на самую длинную последовательность из единиц в блоке;	0. 561835	0,383827	0.653658	0.758760	0.623651
Тест рангов бинарных матриц;	0. 786995	0,702458	0.716808	0.662023	0.312670
Спектральный тест;	0. 307482	0,650637	0.016398	0.986632	0.321648
Тест на встречающиеся непересекающиеся шаблоны;	1.0000000	0,433358	0.999999	1.000002	1.000000
Тест на встречающиеся пересекающиеся шаблоны;	0.2157599	0,632191	0.154498	0.640680	0.340571
Универсальный тест Мауэра;	0.6964627	0,414862	0.028997	0.042824	0.242690
Тест на линейную сложность;	0.0890209	0,778903	0.801172	0.349006	0.846925
Тест на периодичность;	0.0343742	0,610977	0.840852	0.323604	0.326941
Тест приблизительной энтропии;	0.0570419	0,633388	0.899898	0.666396	0.731416
Тест кумулятивных сумм;	0.3251871	0,651956	0.217090	0.682031	0.303681
Тест на произвольные отклонения;	0.0187336	0,730485	0.002145 ТЕСТ НЕ ПРОЙДЕН	0.005685 ТЕСТ НЕ ПРОЙДЕН	0.054452
Другой тест на произвольные отклонения.	0.0908728	0.09087280958776588	0.298281	0.366233	0.315302

Исходя из данных таблицы можно заметить, что предложенный генератор не только не уступает остальным классическим методам генерации псевдослучайных чисел (линейно-конгруэнтный метод и вихрь Мерсенна), но и превосходит их. Также можно наблюдать, что и второй генератор, также основанный на технологии клеточного автомата, превосходит классический метод генерации псевдослучайных чисел. Криптостойкость этих двух генераторов также сравнима с генератором Блюм-Блюм-Шуба, что в том числе подтверждает их конкурентность в мире КГПСЧ.

4.5 Вывод

В исследовательском разделе проведена оценка качества разработанного метода генерации псевдослучайных чисел. Доказана криптографическая стойкость чисел, которые данные метод способен производить. Разобран вопрос применимости предложенного метода для задач защиты информации. Также проведен сравнительный анализ с существующими генераторами. Результаты исследований показали, что генератор действительно способен выдавать последовательности достаточно случайных чисел и не уступает классическим методам генерации. В этом же анализе в сравнении участвовал и криптостойкий генератор псевдослучайных чисел. Это сравнение показало конкурентную способность метода генерации псевдослучайных чисел на основе клеточного автомата «Жизнь». Такие же показатели выдал и другой метод генерации псевдослучайных чисел, основанный на технологии клеточного автомата. Это значит, что технология клеточных автоматов, применимая к ГПСЧ несет в себе значительный потенциал для развития генераторов псевдослучайных чисел в целом.

ЗАКЛЮЧЕНИЕ

В данной работе был представлен метод генерации псевдослучайных чисел на клеточного автомата, а в качестве самого автомата служит игра «Жизнь», правила и мир которой были модифицированы.

Были разработаны программные обеспечения, отвечавшие за визуализацию работы предложенного метода, а также модули для тестирования качества случайности результата работы данного метода. Для работы самого метода требуются входные параметры, такие как заранее определенные начальные заполнения клеточного автомата, выбор которых являлся одной из самых острых вопросов, решаемых в ходе этой работы. И хоть данная проблема

была решена, тем не менее вопрос источника энтропии все равно остается животрепещущей темой данного метода.

Также проведена оценка качества предложенного генератора, где последний продемонстрировал отличные показатели, в том числе в сравнительном анализе, где оказался лучше классических методов генерации, а также показал достойные результаты в сравнении с криптостойким генератором псевдослучайных чисел. Сама способность к криптографической стойкости также была продемонстрирована путем исследования разработанного метода 43 независимыми тестами.

При сравнительном анализе было замечено, что технология клеточных автоматов имеет большой потенциал развития. Клеточные автоматы предоставляют не только лучшие статистические результаты, но и обладают простотой в разработке, возможностью масштабирования, применения многопоточности и параллельного программирования, что может ускорить их работу. Кроме того, предложенный метод имеет потенциал комбинирования его с другими метода генерации, что дополнительно расширит возможности для разработки более совершенных методов генерации псевдослучайных чисел.

Данная работа имеет потенциал для дальнейшего развития, так как есть задачи, требующие исследования и решения. Такими задачами являются распределения, используемая память и применимость к задачам моделирования.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Дональд Кнут. Искусство программирования, том 2. Получисленные алгоритмы = The Art of Computer Programming, vol.2. Seminumerical Algorithms.— 3-е изд. — М.: «Вильямс», 2007.
2. Т. Тоффоли, Н. Марголус, Машины клеточных автоматов// Мир 1991, стр. 8.
3. Aspray W. John von Neumann and the Origins of Modern Computing. MIT Press, 1990, стр. 376
4. Слеповичев И.И., ГЕНЕРАТОРЫ ПСЕВДОСЛУЧАЙНЫХ ЧИСЕЛ, май 21, 2017.
5. Документация Python 3.8 [Электронный ресурс]. URL: https://docs.python.org/3.8 (дата обращения 04.06.2021).
6. ГОСТ Р ИСО 28640-2012. [Электронный ресурс]. – Режим доступа: http://files.stroyinf.ru/cgi-bin/ecat/ecat.fcgi?b=0&i=53898&pr=1. Проверено 01.06.2021.
7. S. Wolfram, “Random Sequence generation by Cellular Automata”, Advances in Applied Mathematics, v. 7, 1986, pp.123-164.
8. W. Meier and O. Staffelbach, “Fast Correlation Attack on Stream Ciphers”, Journal of Cryptology v I n. 3, 1989, pp.159-176.
9. P.H. Bardell, “Analisis of Cellular Automata Used as Pseudorandom Pattern generators”, Proceedings of 1990 International Test Conference, pp. 762-768.
10. S. Wolfram, “Statistical mechanics of cellular automata”, Reviews of Modern Physics, Vol. 55, No. 3, July 1983, pp.8-13.
11. Мухамеджанов Д.Д., Левина А.Б. Генератор псевдослучайных чисел на основе клеточных автоматов // Научно-технический вестник информационных технологий, механики и оптики. 2018. Т. 18. № 5. С. 894–900. doi: 10.17586/22261494-2018-18-5-894-90
12. Деон А.Ф., Меняев Ю.А. Генератор равномерных случайных величин по технологии полного вихревого массива // Вестник МГТУ им. Н.Э.

Баумана. Сер. Приборостроение. 2017. № 2. С. 86–110. DOI: 10.18698/0236-3933-2017-2-86-110

13. Mirzoyan S.A. PSEUDORANDOM NUMBER GENERATOR BASED ON CELLULAR AUTOMATA AND THE GAME "LIFE" // International Scientific – Practical Conference «INFORMATION INNOVATIVE TECHNOLOGIES», 2021, стр. 280-285

14. Federal Information Processing Standards Publications, ADVANCED ENCRYPTION STANDARD, November 26, 2001

15. Blum L., Blum M., Shub M. A Simple unpredictable pseudo-random number generator // SIAM Journal on Computing.1986. Vol. 15. No. 2. P.

ПРИЛОЖЕНИЕ А

Листинг загрузчика модулей

Листинг А.1 – Обход клеток мира клеточного автомата.

```
Def life(W, H, next_field, current_field, tries):
    for k in range(tries):
        for x in range(1, W - 1):
            for y in range(1, H - 1):
                next_field[y][x] = check_cell(current_field, x, y,W,H)

        current_field = [*next_field]

    return current_field
```

Листинг А.2 – Заполнение карты клеточного автомата. Проверка каждой ячейки согласно модифицированным правилам игры «Жизнь»

```
def fill (W, H):
    current_field = next_field = [[0 for i in range(W)] for j in range(H)]
    entrp = 0
    while entrp == 0:
        entrp = round((time.process_time_ns() * timeit.timeit()))#% (W*H)/2
        entrp = entrp % 10000
    print(entrp)
    current_field = [[2 if i == W // entrp or j == H // entrp else 0 for i in range(W)] for j in range(H)]
    for j in range (W):
        for i in range(H):
            if i != 0 and j!= 0:
                if entrp % i == 0 or entrp % j== 0:
                    current_field[i][j] = i % 3
                elif i > j and not (2 * i + j) % 4:
                    current_field[i][j] = 2
                else:
                    current_field[i][j-i] = 1
    return next_field, current_field
```

Листинг А.3 – Проверка каждой ячейки согласно модифицированным правилам игры «Жизнь»

```
def check_cell(current_field, x, y,W, H):
    count = 0

    for j in range(y - 1, y + 2):
        for i in range(x - 1, x + 2):
            if current_field[j % H][i % W] != 0:
                count += 1
    # Zombie
    if current_field[y][x] == 2:
```

```
        count -= 1
        if count == 2 or count == 4:
            return 2
        return 0
    else:
        if count == 6:
            return 2

    # Alive
    if current_field[y][x] == 0:
        count -= 1
        if count == 2 or count == 3:
            return 1
        return 0
    else:
        if count == 3:
            return 1
        return 0
```

ПРИЛОЖЕНИЕ Б

Листинги представленных модулей

Листинг Б.1 – Модуль выбора строки

```
def choise_line(current_field):
    summ = 0
    summax = 0
    masmax = []
    cmin = 1
    x = []
    masK = []
    masnum = []
    KK = 0
    for k in range(1,len(current_field)-1):
        x = current_field[k]
        masK.append(k)
        num = ''
        num3 = ''
        for j in x:
            num += str(j)
            num3 += str(j)

        if sum(x) == 0:
            continue
        num = convert_base(num, 2, 3)
        num = list(num)

        c = Knut_test(num, 0)
        if c == '!!!':
            continue
        if abs(c) < cmin:
            masmax = num
            masnum = num3
            cmin = c
            KK = k
    return KK, masmax, masnum
```

Листинг Б.2 – Конвертер систем счислений

```
def convert_base(num, to_base, from_base):
    # first convert to decimal number
    n = int(num, from_base) if isinstance(num, str) else num
    # now convert decimal to 'to_base' base
    alphabet = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    res = ""
    while n > 0:
        n,m = divmod(n, to_base)
        res += alphabet[m]
    return res[::-1]
```

Листинг Б.3 – Тест сериальной корреляции

```
def Knut_test(U, flag):
    n = len(U)
    u2 = 0
    u_sum2 = 0
    # print(U)
    if isinstance(U[0], str):
        for i in range(len(U)):
            U[i] = int(U[i])

    u_sum2 = sum(U)**2
    U1U2 = 0
    for i in range(n-1):
        U1U2 += U[i]*U[i+1]
        u2 += U[i]**2
    C = (n*U1U2 - u_sum2)/(n*u2 - u_sum2)
    un = (-1)/(n-1)
    sigma = sqrt((n**2)/(((n-1)**2)*(n-2)))
    if flag == 1:
        if un - 2*sigma < C < un + 2*sigma:
            print('Хорошее значение - ', C)

        else:
            print('Нехорошее значение - ', C)

    return C
```

ПРИЛОЖЕНИЕ В

Листинги обработчиков сообщений

Листинг В.1 – модуль sp800_22_tests для пакета тестов NIST.

```
from __future__ import print_function

import argparse
import sys
import pickle


def read_bits_from_file(filename,bigendian):
    bitlist = list()
    if filename == None:
        f = sys.stdin
    else:
        f = open(filename, "rb")
    while True:
        bytes = f.read(16384)
        if bytes:
            for bytech in bytes:
                if sys.version_info > (3,0):
                    byte = bytech
                else:
                    byte = ord(bytech)
                for i in range(8):
                    if bigendian:
                        bit = (byte & 0x80) >> 7
                        byte = byte << 1
                    else:
                        bit = (byte >> i) & 1
                    bitlist.append(bit)
        else:
            break
    f.close()
    return bitlist

import argparse
import sys
parser = argparse.ArgumentParser(description='Test data for distinguishability
form random, using NIST SP800-22Rev1a algorithms.')
parser.add_argument('filename', type=str, nargs='?', help='Filename of binary
file to test')
parser.add_argument('--be', action='store_false',help='Treat data as big
endian bits within bytes. Defaults to little endian')
parser.add_argument('-t', '--testname', default=None,help='Select the test to
run. Defaults to running all tests. Use --list_tests to see the list')
parser.add_argument('--list_tests', action='store_true',help='Display the list
of tests')

args = parser.parse_args()

bigendian = args.be
filename = args.filename

testlist = [
```

```
        'monobit_test',
        'frequency_within_block_test',
        'runs_test',
        'longest_run_ones_in_a_block_test',
        'binary_matrix_rank_test',
        'dft_test',
        'non_overlapping_template_matching_test',
        'overlapping_template_matching_test',
        'maurers_universal_test',
        'linear_complexity_test',
        'serial_test',
        'approximate_entropy_test',
        'cumulative_sums_test',
        'random_excursion_test',
        'random_excursion_variant_test']

print("Tests of Distinguishability from Random")
if args.list_tests:
    for i,testname in zip(range(len(testlist)),testlist):
        print(str(i+1).ljust(4)+": "+testname)
    exit()

f = open('../nums.txt','r')
bits = ''

for i in f:
    bits += i
bits = list(bits)

for i in range(len(bits)):
    bits[i] = int(bits[i])
gotresult=False
if args.testname:
    if args.testname in testlist:
        m = __import__ ("sp800_22_"+args.testname)
        func = getattr(m,args.testname)
        print("TEST: %s" % args.testname)
        success,p,plist = func(bits)
        gotresult = True
        if success:
            print("PASS")
        else:
            print("FAIL")

        if p:
            print("P="+str(p))

        if plist:
            for pval in plist:
                print("P="+str(pval))
    else:
        print("Test name (%s) not known" % args.ttestname)
        exit()
else:
    results = list()

    for testname in testlist:
        print("TEST: %s" % testname)
        m = __import__ ("sp800_22_"+testname)
        func = getattr(m,testname)

        (success,p,plist) = func(bits)

        summary_name = testname
```

```
        if success:
            print("  PASS")
            summary_result = "PASS"
        else:
            print("  FAIL")
            summary_result = "FAIL"

        if p != None:
            print("  P="+str(p))
            summary_p = str(p)

        if plist != None:
            for pval in plist:
                print("P="+str(pval))
            summary_p = str(min(plist))

        results.append((summary_name,summary_p, summary_result))

    print()
    print("SUMMARY")
    print("-------")

    for result in results:
        (summary_name,summary_p, summary_result) = result
        print(summary_name.ljust(40),summary_p.ljust(18),summary_result)
```

Листинг В.2 – модуль Dieharder_tests для пакета тестов Dieharder.

```
import math
from scipy.special import *
import numpy as np
import sys
sys.set_int_max_str_digits(0)


tests =    [
      rank_test,
      decoder_test,
      OSPOTest,
      LempelZivTest,
      IRWTest,
      RWTest,
      LCGTest,
      PiZeroTest,
      PiOneTest,
      PiTwoTest,
      PiThreeTest,
      SigmaZeroTest,
      SigmaOneTest,
      sigma_two_test,
      sigma_three_test]
i = 1
for test in tests:
    result = test(bits)
    print(result)
```

ПРИЛОЖЕНИЕ Г

Листинг критериев исследования

Листинг Г.1 – модуль monobit_test

```
from __future__ import print_function

import math

def count_ones_zeroes(bits):
    ones = 0
    zeroes = 0
    for bit in bits:
        if (bit == 1):
            ones += 1
        else:
            zeroes += 1
    return (zeroes,ones)

def monobit_test(bits):
    n = len(bits)

    zeroes,ones = count_ones_zeroes(bits)
    s = abs(ones-zeroes)
    print("  Ones count   = %d" % ones)
    print("  Zeroes count = %d" % zeroes)

    p = math.erfc(float(s)/(math.sqrt(float(n)) * math.sqrt(2.0)))

    success = (p >= 0.01)
    return (success,p,None)
```

Листинг Г.2 – модуль frequency_within_block_test

```
from __future__ import print_function

import math
from fractions import Fraction
#from scipy.special import gamma, gammainc, gammaincc
from gamma_functions import *

#ones_table = [bin(i)[2:].count('1') for i in range(256)]
def count_ones_zeroes(bits):
    ones = 0
    zeroes = 0
    for bit in bits:
        if (bit == 1):
            ones += 1
        else:
            zeroes += 1
    return (zeroes,ones)

def frequency_within_block_test(bits):
```

```
    # Compute number of blocks M = block size. N=num of blocks
    # N = floor(n/M)
    # miniumum block size 20 bits, most blocks 100
    n = len(bits)
    M = 20
    N = int(math.floor(n/M))
    if N > 99:
        N=99
        M = int(math.floor(n/N))

    if len(bits) < 100:
        print("Too little data for test. Supply at least 100 bits")
        return False,1.0,None

    print("  n = %d" % len(bits))
    print("  N = %d" % N)
    print("  M = %d" % M)

    num_of_blocks = N
    block_size = M #int(math.floor(len(bits)/num_of_blocks))
    #n = int(block_size * num_of_blocks)

    proportions = list()
    for i in range(num_of_blocks):
        block = bits[i*(block_size):((i+1)*(block_size))]
        zeroes,ones = count_ones_zeroes(block)
        proportions.append(Fraction(ones,block_size))

    chisq = 0.0
    for prop in proportions:
        chisq += 4.0*block_size*((prop - Fraction(1,2))**2)

    p = gammaincc((num_of_blocks/2.0),float(chisq)/2.0)
    success = (p >= 0.01)
    return (success,p,None)
```

Листинг Г.3 – модуль runs_test

```
from __future__ import print_function

import math
from fractions import Fraction
from scipy.special import gamma, gammainc, gammaincc
from gamma_functions import *
import numpy
import cmath
import random

#ones_table = [bin(i)[2:].count('1') for i in range(256)]
def count_ones_zeroes(bits):
    ones = 0
    zeroes = 0
    for bit in bits:
        if (bit == 1):
            ones += 1
        else:
            zeroes += 1
    return (zeroes,ones)

def runs_test(bits):
```

```
    n = len(bits)
    zeroes,ones = count_ones_zeroes(bits)

    prop = float(ones)/float(n)
    print("  prop ",prop)

    tau = 2.0/math.sqrt(n)
    print("  tau ",tau)

    if abs(prop-0.5) > tau:
        return (False,0.0,None)

    vobs = 1.0
    for i in range(n-1):
        if bits[i] != bits[i+1]:
            vobs += 1.0

    print("  vobs ",vobs)

    p = math.erfc(abs(vobs - (2.0*n*prop*(1.0-
prop)))/(2.0*math.sqrt(2.0*n)*prop*(1-prop) ))
    success = (p >= 0.01)
    return (success,p,None)
```

Листинг Г.4 – модуль longest_run_ones_in_a_block_test

```
from __future__ import print_function

import math
from scipy.special import gamma, gammainc, gammaincc
from gamma_functions import *

import random

def probs(K,M,i):
    M8 =      [0.2148, 0.3672, 0.2305, 0.1875]
    M128 =    [0.1174, 0.2430, 0.2493, 0.1752, 0.1027, 0.1124]
    M512 =    [0.1170, 0.2460, 0.2523, 0.1755, 0.1027, 0.1124]
    M1000 =   [0.1307, 0.2437, 0.2452, 0.1714, 0.1002, 0.1088]
    M10000 =  [0.0882, 0.2092, 0.2483, 0.1933, 0.1208, 0.0675, 0.0727]
    if (M == 8):        return M8[i]
    elif (M == 128):    return M128[i]
    elif (M == 512):    return M512[i]
    elif (M == 1000):   return M1000[i]
    else:               return M10000[i]

def longest_run_ones_in_a_block_test(bits):
    n = len(bits)

    if n < 128:
        return (False,1.0,None)
    elif n<6272:
        M = 8
    elif n<750000:
        M = 128
    else:
        M = 10000

    # compute new values for K & N
    if M==8:
```

```
        K=3
        N=16
    elif M==128:
        K=5
        N=49
    else:
        K=6
        N=75

    # Table of frequencies
    v = [0,0,0,0,0,0,0]

    for i in range(N): # over each block
        #find longest run
        block = bits[i*M:((i+1)*M)] # Block i

        run = 0
        longest = 0
        for j in range(M): # Count the bits.
            if block[j] == 1:
                run += 1
                if run > longest:
                    longest = run
            else:
                run = 0

        if M == 8:
            if longest <= 1:    v[0] += 1
            elif longest == 2:  v[1] += 1
            elif longest == 3:  v[2] += 1
            else:               v[3] += 1
        elif M == 128:
            if longest <= 4:    v[0] += 1
            elif longest == 5:  v[1] += 1
            elif longest == 6:  v[2] += 1
            elif longest == 7:  v[3] += 1
            elif longest == 8:  v[4] += 1
            else:               v[5] += 1
        else:
            if longest <= 10:   v[0] += 1
            elif longest == 11: v[1] += 1
            elif longest == 12: v[2] += 1
            elif longest == 13: v[3] += 1
            elif longest == 14: v[4] += 1
            elif longest == 15: v[5] += 1
            else:               v[6] += 1

    # Compute Chi-Sq
    chi_sq = 0.0
    for i in range(K+1):
        p_i = probs(K,M,i)
        upper = (v[i] - N*p_i)**2
        lower = N*p_i
        chi_sq += upper/lower
    print("  n = "+str(n))
    print("  K = "+str(K))
    print("  M = "+str(M))
    print("  N = "+str(N))
    print("  chi_sq = "+str(chi_sq))
    p = gammaincc(K/2.0, chi_sq/2.0)

    success = (p >= 0.01)
    return (success,p,None)
```

Листинг Г.5 – модуль binary_matrix_rank_test

```
from __future__ import print_function

import math
import copy
import gf2matrix

def binary_matrix_rank_test(bits,M=32,Q=32):
    n = len(bits)
    N = int(math.floor(n/(M*Q))) #Number of blocks
    print("  Number of blocks %d" % N)
    print("  Data bits used: %d" % (N*M*Q))
    print("  Data bits discarded: %d" % (n-(N*M*Q)))

    if N < 38:
        print("  Number of blocks must be greater than 37")
        p = 0.0
        return False,p,None

    # Compute the reference probabilities for FM, FMM and remainder
    r = M
    product = 1.0
    for i in range(r):
        upper1 = (1.0 - (2.0**(i-Q)))
        upper2 = (1.0 - (2.0**(i-M)))
        lower = 1-(2.0**(i-r))
        product = product * ((upper1*upper2)/lower)
    FR_prob = product * (2.0**((r*(Q+M-r)) - (M*Q)))

    r = M-1
    product = 1.0
    for i in range(r):
        upper1 = (1.0 - (2.0**(i-Q)))
        upper2 = (1.0 - (2.0**(i-M)))
        lower = 1-(2.0**(i-r))
        product = product * ((upper1*upper2)/lower)
    FRM1_prob = product * (2.0**((r*(Q+M-r)) - (M*Q)))

    LR_prob = 1.0 - (FR_prob + FRM1_prob)

    FM = 0      # Number of full rank matrices
    FMM = 0     # Number of rank -1 matrices
    remainder = 0
    for blknum in range(N):
        block = bits[blknum*(M*Q):(blknum+1)*(M*Q)]
        # Put in a matrix
        matrix = gf2matrix.matrix_from_bits(M,Q,block,blknum)
        # Compute rank
        rank = gf2matrix.rank(M,Q,matrix,blknum)

        if rank == M: # count the result
            FM += 1
        elif rank == M-1:
            FMM += 1
        else:
            remainder += 1

    chisq =  (((FM-(FR_prob*N))**2)/(FR_prob*N))
    chisq += (((FMM-(FRM1_prob*N))**2)/(FRM1_prob*N))
    chisq += (((remainder-(LR_prob*N))**2)/(LR_prob*N))
```

```
    p = math.e **(-chisq/2.0)
    success = (p >= 0.01)

    print("  Full Rank Count  = ",FM)
    print("  Full Rank -1 Count = ",FMM)
    print("  Remainder Count = ",remainder)
    print("  Chi-Square = ",chisq)

    return (success, p, None)
```

Листинг Г.6 – модуль dft_test

```
from __future__ import print_function

import math
import numpy
import sys

def dft_test(bits):
    n = len(bits)
    if (n % 2) == 1:        # Make it an even number
        bits = bits[:-1]

    ts = list()             # Convert to +1,-1
    for bit in bits:
        ts.append((bit*2)-1)

    ts_np = numpy.array(ts)
    fs = numpy.fft.fft(ts_np)  # Compute DFT

    if sys.version_info > (3,0):
        mags = abs(fs)[:n//2] # Compute magnitudes of first half of sequence
    else:
        mags = abs(fs)[:n/2] # Compute magnitudes of first half of sequence

    T = math.sqrt(math.log(1.0/0.05)*n) # Compute upper threshold
    N0 = 0.95*n/2.0
    print("  N0 = %f" % N0)

    N1 = 0.0   # Count the peaks above the upper theshold
    for mag in mags:
        if mag < T:
            N1 += 1.0
    print("  N1 = %f" % N1)
    d = (N1 - N0)/math.sqrt((n*0.95*0.05)/4) # Compute the P value
    p = math.erfc(abs(d)/math.sqrt(2))

    success = (p >= 0.01)
    return (success,p,None)
```

Листинг Г.7 – модуль non_overlapping_template_matching_test_test

```
from __future__ import print_function

import math
from scipy.special import gamma, gammainc, gammaincc
from gamma_functions import *
import random

def non_overlapping_template_matching_test(bits):
    # The templates provdided in SP800-22rev1a
```

```
    templates = [None for x in range(7)]
    templates[0] = [[0,1],[1,0]]
    templates[1] = [[0,0,1],[0,1,1],[1,0,0],[1,1,0]]
    templates[2] =
[[0,0,0,1],[0,0,1,1],[0,1,1,1],[1,0,0,0],[1,1,0,0],[1,1,1,0]]
    templates[3] =
[[0,0,0,0,1],[0,0,0,1,1],[0,0,1,0,1],[0,1,0,1,1],[0,0,1,1,1],[0,1,1,1,1],

[1,1,1,0,0],[1,1,0,1,0],[1,0,1,0,0],[1,1,0,0,0],[1,0,0,0,0],[1,1,1,1,0]]
    templates[4] =
[[0,0,0,0,0,1],[0,0,0,0,1,1],[0,0,0,1,0,1],[0,0,0,1,1,1],[0,0,1,0,1,1],
                    [0,0,1,1,0,1],[0,0,1,1,1,1],[0,1,0,0,1,1],
                    [0,1,0,1,1,1],[0,1,1,1,1,1],[1,0,0,0,0,0],
                    [1,0,1,0,0,0],[1,0,1,1,0,0],[1,1,0,0,0,0],
                    [1,1,0,0,1,0],[1,1,0,1,0,0],[1,1,1,0,0,0],
                    [1,1,1,0,1,0],[1,1,1,1,0,0],[1,1,1,1,1,0]]
    templates[5] =
[[0,0,0,0,0,0,1],[0,0,0,0,0,1,1],[0,0,0,0,1,0,1],[0,0,0,0,1,1,1],

[0,0,0,1,0,0,1],[0,0,0,1,0,1,1],[0,0,0,1,1,0,1],[0,0,0,1,1,1,1],

[0,0,1,0,0,1,1],[0,0,1,0,1,0,1],[0,0,1,0,1,1,1],[0,0,1,1,0,1,1],

[0,0,1,1,1,0,1],[0,0,1,1,1,1,1],[0,1,0,0,0,1,1],[0,1,0,0,1,1,1],

[0,1,0,1,0,1,1],[0,1,0,1,1,1,1],[0,1,1,0,1,1,1],[0,1,1,1,1,1,1],

[1,0,0,0,0,0,0],[1,0,0,1,0,0,0],[1,0,1,0,0,0,0],[1,0,1,0,1,0,0],

[1,0,1,1,0,0,0],[1,0,1,1,1,0,0],[1,1,0,0,0,0,0],[1,1,0,0,0,1,0],

[1,1,0,0,1,0,0],[1,1,0,1,0,0,0],[1,1,0,1,0,1,0],[1,1,0,1,1,0,0],

[1,1,1,0,0,0,0],[1,1,1,0,0,1,0],[1,1,1,0,1,0,0],[1,1,1,0,1,1,0],

[1,1,1,1,0,0,0],[1,1,1,1,0,1,0],[1,1,1,1,1,0,0],[1,1,1,1,1,1,0]]
    templates[6] =
[[0,0,0,0,0,0,0,1],[0,0,0,0,0,0,1,1],[0,0,0,0,0,1,0,1],[0,0,0,0,0,1,1,1],

[0,0,0,0,1,0,0,1],[0,0,0,0,1,0,1,1],[0,0,0,0,1,1,0,1],[0,0,0,0,1,1,1,1],

[0,0,0,1,0,0,1,1],[0,0,0,1,0,1,0,1],[0,0,0,1,0,1,1,1],[0,0,0,1,1,0,0,1],

[0,0,0,1,1,0,1,1],[0,0,0,1,1,1,0,1],[0,0,0,1,1,1,1,1],[0,0,1,0,0,0,1,1],

[0,0,1,0,0,1,0,1],[0,0,1,0,0,1,1,1],[0,0,1,0,1,0,1,1],[0,0,1,0,1,1,0,1],

[0,0,1,0,1,1,1,1],[0,0,1,1,0,1,0,1],[0,0,1,1,0,1,1,1],[0,0,1,1,1,0,1,1],

[0,0,1,1,1,1,0,1],[0,0,1,1,1,1,1,1],[0,1,0,0,0,0,1,1],[0,1,0,0,0,1,1,1],

[0,1,0,0,1,0,1,1],[0,1,0,0,1,1,1,1],[0,1,0,1,0,0,1,1],[0,1,0,1,0,1,1,1],

[0,1,0,1,1,0,1,1],[0,1,0,1,1,1,1,1],[0,1,1,0,0,1,1,1],[0,1,1,0,1,1,1,1],

[0,1,1,1,1,1,1,1],[1,0,0,0,0,0,0,0],[1,0,0,1,0,0,0,0],[1,0,0,1,1,0,0,0],

[1,0,1,0,0,0,0,0],[1,0,1,0,0,1,0,0],[1,0,1,0,1,0,0,0],[1,0,1,0,1,1,0,0],

[1,0,1,1,0,0,0,0],[1,0,1,1,0,1,0,0],[1,0,1,1,1,0,0,0],[1,0,1,1,1,1,0,0],

[1,1,0,0,0,0,0,0],[1,1,0,0,0,0,1,0],[1,1,0,0,0,1,0,0],[1,1,0,0,1,0,0,0],

[1,1,0,0,1,0,1,0],[1,1,0,1,0,0,0,0],[1,1,0,1,0,0,1,0],[1,1,0,1,0,1,0,0],
```

```
[1,1,0,1,1,0,0,0],[1,1,0,1,1,0,1,0],[1,1,0,1,1,1,0,0],[1,1,1,0,0,0,0,0],

[1,1,1,0,0,0,1,0],[1,1,1,0,0,1,0,0],[1,1,1,0,0,1,1,0],[1,1,1,0,1,0,0,0],

[1,1,1,0,1,0,1,0],[1,1,1,0,1,1,0,0],[1,1,1,1,0,0,0,0],[1,1,1,1,0,0,1,0],

[1,1,1,1,0,1,0,0],[1,1,1,1,0,1,1,0],[1,1,1,1,1,0,0,0],[1,1,1,1,1,0,1,0],
                    [1,1,1,1,1,1,0,0],[1,1,1,1,1,1,1,0]]

    n = len(bits)

    # Choose the template B
    r = random.SystemRandom()
    template_list = r.choice(templates)
    B = r.choice(template_list)

    m = len(B)

    N = 8
    M = int(math.floor(len(bits)/8))
    n = M*N

    blocks = list() # Split into N blocks of M bits
    for i in range(N):
        blocks.append(bits[i*M:(i+1)*M])

    W=list() # Count the number of matches of the template in each block Wj
    for block in blocks:
        position = 0
        count = 0
        while position < (M-m):
            if block[position:position+m] == B:
                position += m
                count += 1
            else:
                position += 1
        W.append(count)

    mu = float(M-m+1)/float(2**m) # Compute mu and sigma
    sigma = M * ((1.0/float(2**m))-(float((2*m)-1)/float(2**(2*m))))

    chisq = 0.0  # Compute Chi-Square
    for j in range(N):
        chisq += ((W[j] - mu)**2)/(sigma**2)

    p = gammaincc(N/2.0, chisq/2.0) # Compute P value

    success = ( p >= 0.01)
    return (success,p,None)
```

Листинг Г.8 – модуль overlapping_template_matching_test

```
from __future__ import print_function

import math
from scipy.special import gamma, gammainc, gammaincc
from gamma_functions import *

def lgamma(x):
```

```
    return math.log(gamma(x))

def Pr(u, eta):
    if ( u == 0 ):
        p = math.exp(-eta)
    else:
        sum = 0.0
        for l in range(1,u+1):
            sum += math.exp(-eta-u*math.log(2)+l*math.log(eta)-lgamma(l+1)+lgamma(u)-
lgamma(l)-lgamma(u-l+1))
        p = sum
    return p

def overlapping_template_matching_test(bits,blen=6):
    n = len(bits)

    m = 10
    # Build the template B as a random list of bits
    B = [1 for x in range(m)]

    N = 968
    K = 5
    M = 1062
    if len(bits) < (M*N):
        print("Insufficient data. %d bit provided. 1,028,016 bits required" % len(bits))
        return False, 0.0, None

    blocks = list() # Split into N blocks of M bits
    for i in range(N):
        blocks.append(bits[i*M:(i+1)*M])

    # Count the distribution of matches of the template across blocks: Vj
    v=[0 for x in range(K+1)]
    for block in blocks:
        count = 0
        for position in range(M-m):
            if block[position:position+m] == B:
                count += 1

        if count >= (K):
            v[K] += 1
        else:
            v[count] += 1


    chisq = 0.0  # Compute Chi-Square
    #pi = [0.324652,0.182617,0.142670,0.106645,0.077147,0.166269] # From spec
    pi = [0.364091, 0.185659, 0.139381, 0.100571, 0.0704323, 0.139865] # From STS
    piqty = [int(x*N) for x in pi]

    lambd = (M-m+1.0)/(2.0**m)
    eta = lambd/2.0
```

```
    sum = 0.0
    for i in range(K): #  Compute Probabilities
        pi[i] = Pr(i, eta)
        sum += pi[i]

    pi[K] = 1 - sum;

    sum = 0
    chisq = 0.0
    for i in range(K+1):
        chisq += ((v[i] - (N*pi[i]))**2)/(N*pi[i])
        sum += v[i]

    p = gammaincc(5.0/2.0, chisq/2.0) # Compute P value

    print("  B = ",B)
    print("  m = ",m)
    print("  M = ",M)
    print("  N = ",N)
    print("  K = ",K)
    print("  model = ",piqty)
    print("  v[j] =  ",v)
    print("  chisq = ",chisq)

    success = ( p >= 0.01)
    return (success,p,None)
```

Листинг Г.9 – модуль maurers_universal_test

```
from __future__ import print_function

import math

def pattern2int(pattern):
    l = len(pattern)
    n = 0
    for bit in (pattern):
        n = (n << 1) + bit
    return n

def maurers_universal_test(bits,patternlen=None, initblocks=None):
    n = len(bits)

    # Step 1. Choose the block size
    if patternlen != None:
        L = patternlen
    else:
        ns = [904960,2068480,4654080,10342400,
              22753280,49643520,107560960,
              231669760,496435200,1059061760]
        L = 6
```

```
        if n < 387840:
            print("Error. Need at least 387840 bits. Got %d." % n)
            #exit()
            return False,0.0,None
        for threshold in ns:
            if n >= threshold:
                L += 1

    # Step 2 Split the data into Q and K blocks
    nblocks = int(math.floor(n/L))
    if initblocks != None:
        Q = initblocks
    else:
        Q = 10*(2**L)
    K = nblocks - Q

    # Step 3 Construct Table
    nsymbols = (2**L)
    T=[0 for x in range(nsymbols)] # zero out the table
    for i in range(Q):              # Mark final position of
        pattern = bits[i*L:(i+1)*L] # each pattern
        idx = pattern2int(pattern)
        T[idx]=i+1      # +1 to number indexes 1..(2**L)+1
                        # instead of 0..2**L
    # Step 4 Iterate
    sum = 0.0
    for i in range(Q,nblocks):
        pattern = bits[i*L:(i+1)*L]
        j = pattern2int(pattern)
        dist = i+1-T[j]
        T[j] = i+1
        sum = sum + math.log(dist,2)
    print("  sum =", sum)

    # Step 5 Compute the test statistic
    fn = sum/K
    print("  fn =",fn)

    # Step 6 Compute the P Value
    # a_universal_statistical_test_for_random_bit_generators.pdf
    ev_table =  [0,0.73264948,1.5374383,2.40160681,3.31122472,
                 4.25342659,5.2177052,6.1962507,7.1836656,
                 8.1764248,9.1723243,10.170032,11.168765,
                 12.168070,13.167693,14.167488,15.167379]
    var_table = [0,0.690,1.338,1.901,2.358,2.705,2.954,3.125,
                 3.238,3.311,3.356,3.384,3.401,3.410,3.416,
                 3.419,3.421]

    # sigma = math.sqrt(var_table[L])
    mag = abs((fn - ev_table[L]) / ((0.7 - 0.8 / L + (4 + 32 / L) * (pow(K, -3 / L)) / 15) *
(math.sqrt(var_table[L] / K)) * math.sqrt(2)))
    P = math.erfc(mag)
```

```
    success = (P >= 0.01)
    return (success, P, None)


if __name__ == "__main__":
    bits = [0,1,0,1,1,0,1,0,0,1,1,1,0,1,0,1,0,1,1,1]
    success, p, _ = maurers_universal_test(bits, patternlen=2, initblocks=4)

    print("success =",success)
    print("p       = ",p)
```

Листинг Г.10 – модуль linear_complexity_test

```
from __future__ import print_function

import math
from scipy.special import gamma, gammainc, gammaincc
from gamma_functions import *

def berelekamp_massey(bits):
    n = len(bits)
    b = [0 for x in bits]  #initialize b and c arrays
    c = [0 for x in bits]
    b[0] = 1
    c[0] = 1

    L = 0
    m = -1
    N = 0
    while (N < n):
        #compute discrepancy
        d = bits[N]
        for i in range(1,L+1):
            d = d ^ (c[i] & bits[N-i])
        if (d != 0):  # If d is not zero, adjust poly
            t = c[:]
            for i in range(0,n-N+m):
                c[N-m+i] = c[N-m+i] ^ b[i]
            if (L <= (N/2)):
                L = N + 1 - L
                m = N
                b = t
        N = N +1
    # Return length of generator and the polynomial
    return L , c[0:L]

def linear_complexity_test(bits,patternlen=None):
    n = len(bits)
    # Step 1. Choose the block size
    if patternlen != None:
```

```
    M = patternlen
else:
    if n < 1000000:
        print("Error. Need at least 10^6 bits")
        #exit()
        return False,0.0,None
    M = 512
K = 6
N = int(math.floor(n/M))
print("  M = ", M)
print("  N = ", N)
print("  K = ", K)

# Step 2 Compute the linear complexity of the blocks
LC = list()
for i in range(N):
    x = bits[(i*M):((i+1)*M)]
    LC.append(berelekamp_massey(x)[0])

# Step 3 Compute mean
a = float(M)/2.0
b = ((((-1)**(M+1))+9.0))/36.0
c = ((M/3.0) + (2.0/9.0))/(2**M)
mu =  a+b-c

T = list()
for i in range(N):
    x = ((-1.0)**M) * (LC[i] - mu) + (2.0/9.0)
    T.append(x)

# Step 4 Count the distribution over Ticket
v = [0,0,0,0,0,0,0]
for t in T:
    if t <= -2.5:
        v[0] += 1
    elif t <= -1.5:
        v[1] += 1
    elif t <= -0.5:
        v[2] += 1
    elif t <= 0.5:
        v[3] += 1
    elif t <= 1.5:
        v[4] += 1
    elif t <= 2.5:
        v[5] += 1
    else:
        v[6] += 1

# Step 5 Compute Chi Square Statistic
pi = [0.010417,0.03125,0.125,0.5,0.25,0.0625,0.020833]
chisq = 0.0
for i in range(K+1):
```

```
        chisq += ((v[i] - (N*pi[i]))**2.0)/(N*pi[i])
    print("  chisq = ",chisq)
    # Step 6 Compute P Value
    P = gammaincc((K/2.0),(chisq/2.0))
    print("  P = ",P)
    success = (P >= 0.01)
    return (success, P, None)

if __name__ == "__main__":
    bits = [1,1,0,1,0,1,1,1,1,0,0,0,1]
    L,poly = berelekamp_massey(bits)

    bits = [1,1,0,1,0,1,1,1,1,0,0,0,1,1,1,0,1,0,1,1,1,1,0,0,
            0,1,1,1,0,1,0,1,1,1,1,0,0,0,1,1,1,1,0,1,0,1,1,1,1,
            0,0,0,1,1,1,0,1,0,1,1,1,1,0,0,0,1]
    success,p,_ = linear_complexity_test(bits,patternlen=7)

    print("L =",L)
    print("p = ",p)
```

Листинг Г.11 – модуль serial_test

```
from __future__ import print_function

import math
# from scipy.special import gamma, gammainc, gammaincc
from gamma_functions import *

def int2patt(n,m):
    pattern = list()
    for i in range(m):
        pattern.append((n >> i) & 1)
    return pattern

def countpattern(patt,bits,n):
    thecount = 0
    for i in range(n):
        match = True
        for j in range(len(patt)):
            if patt[j] != bits[i+j]:
                match = False
        if match:
            thecount += 1
    return thecount

def psi_sq_mv1(m, n, padded_bits):
    counts = [0 for i in range(2**m)]
    for i in range(2**m):
        pattern = int2patt(i,m)
        count = countpattern(pattern,padded_bits,n)
```

```
        counts.append(count)

    psi_sq_m = 0.0
    for count in counts:
        psi_sq_m += (count**2)
    psi_sq_m = psi_sq_m * (2**m)/n
    psi_sq_m -= n
    return psi_sq_m

def serial_test(bits,patternlen=None):
    n = len(bits)
    if patternlen != None:
        m = patternlen
    else:
        m = int(math.floor(math.log(n,2)))-2

        if m < 4:
            print("Error. Not enough data for m to be 4")
            return False,0,None
        m = 4

    # Step 1
    padded_bits=bits+bits[0:m-1]

    # Step 2
    psi_sq_m   = psi_sq_mv1(m, n, padded_bits)
    psi_sq_mm1 = psi_sq_mv1(m-1, n, padded_bits)
    psi_sq_mm2 = psi_sq_mv1(m-2, n, padded_bits)

    delta1 = psi_sq_m - psi_sq_mm1
    delta2 = psi_sq_m - (2*psi_sq_mm1) + psi_sq_mm2

    P1 = gammaincc(2**(m-2),delta1/2.0)
    P2 = gammaincc(2**(m-3),delta2/2.0)

    print("  psi_sq_m   = ",psi_sq_m)
    print("  psi_sq_mm1 = ",psi_sq_mm1)
    print("  psi_sq_mm2 = ",psi_sq_mm2)
    print("  delta1     = ",delta1)
    print("  delta2     = ",delta2)
    print("  P1         = ",P1)
    print("  P2         = ",P2)

    success = (P1 >= 0.01) and (P2 >= 0.01)
    return (success, None, [P1,P2])
```

Листинг Г.12 – модуль approximate_entropy_test

```
from __future__ import print_function

import math
```

```
#from scipy.special import gamma, gammainc, gammaincc
from gamma_functions import *

def bits_to_int(bits):
    theint = 0
    for i in range(len(bits)):
        theint = (theint << 1) + bits[i]
    return theint

def approximate_entropy_test(bits):
    n = len(bits)

    m = int(math.floor(math.log(n,2)))-6
    if m < 2:
        m = 2
    if m >3 :
        m = 3

    print("  n         = ",n)
    print("  m         = ",m)

    Cmi = list()
    phi_m = list()
    for iterm in range(m,m+2):
        # Step 1
        padded_bits=bits+bits[0:iterm-1]

        # Step 2
        counts = list()
        for i in range(2**iterm):
            #print "  Pattern #%d of %d" % (i+1,2**iterm)
            count = 0
            for j in range(n):
                if bits_to_int(padded_bits[j:j+iterm]) == i:
                    count += 1
            counts.append(count)
            print("  Pattern %d of %d, count = %d" % (i+1,2**iterm, count))

        # step 3
        Ci = list()
        for i in range(2**iterm):
            Ci.append(float(counts[i])/float(n))

        Cmi.append(Ci)

        # Step 4
        sum = 0.0
        for i in range(2**iterm):
            if (Ci[i] > 0.0):
                sum += Ci[i]*math.log((Ci[i]/10.0))
        phi_m.append(sum)
        print("  phi(%d)    = %f" % (m,sum))
```

```
    # Step 5 - let the loop steps 1-4 complete

    # Step 6
    appen_m = phi_m[0] - phi_m[1]
    print("  AppEn(%d)  = %f" % (m,appen_m))
    chisq = 2*n*(math.log(2) - appen_m)
    print("  ChiSquare = ",chisq)
    # Step 7
    p = gammaincc(2**(m-1),(chisq/2.0))

    success = (p >= 0.01)
    return (success, p, None)
```

Листинг Г.13 – модуль cumulative_sums_test

```
from __future__ import print_function

import math
#from scipy.special import gamma, gammainc, gammaincc
from gamma_functions import *
#import scipy.stats

def normcdf(n):
    return 0.5 * math.erfc(-n * math.sqrt(0.5))

def p_value(n,z):
    sum_a = 0.0
    startk = int(math.floor((((float(-n)/z)+1.0)/4.0)))
    endk   = int(math.floor((((float(n)/z)-1.0)/4.0)))
    for k in range(startk,endk+1):
        c = (((4.0*k)+1.0)*z)/math.sqrt(n)
        #d = scipy.stats.norm.cdf(c)
        d = normcdf(c)
        c = (((4.0*k)-1.0)*z)/math.sqrt(n)
        #e = scipy.stats.norm.cdf(c)
        e = normcdf(c)
        sum_a = sum_a + d - e

    sum_b = 0.0
    startk = int(math.floor((((float(-n)/z)-3.0)/4.0)))
    endk   = int(math.floor((((float(n)/z)-1.0)/4.0)))
    for k in range(startk,endk+1):
        c = (((4.0*k)+3.0)*z)/math.sqrt(n)
        #d = scipy.stats.norm.cdf(c)
        d = normcdf(c)
        c = (((4.0*k)+1.0)*z)/math.sqrt(n)
        #e = scipy.stats.norm.cdf(c)
        e = normcdf(c)
        sum_b = sum_b + d - e

    p = 1.0 - sum_a + sum_b
```

```
    return p

def cumulative_sums_test(bits):
    n = len(bits)
    # Step 1
    x = list()          # Convert to +1,-1
    for bit in bits:
        #if bit == 0:
        x.append((bit*2)-1)

    # Steps 2 and 3 Combined
    # Compute the partial sum and records the largest excursion.
    pos = 0
    forward_max = 0
    for e in x:
        pos = pos+e
        if abs(pos) > forward_max:
            forward_max = abs(pos)
    pos = 0
    backward_max = 0
    for e in reversed(x):
        pos = pos+e
        if abs(pos) > backward_max:
            backward_max = abs(pos)

    # Step 4
    p_forward  = p_value(n, forward_max)
    p_backward = p_value(n,backward_max)

    success = ((p_forward >= 0.01) and (p_backward >= 0.01))
    plist = [p_forward, p_backward]

    if success:
        print("PASS")
    else:
        print("FAIL: Data not random")
    return (success, None, plist)
```

Листинг Г.14 – модуль random_excursion_test

```
from __future__ import print_function

import math
#from scipy.special import gamma, gammainc, gammaincc
from gamma_functions import *

# RANDOM EXCURSION TEST
def random_excursion_test(bits):
    n = len(bits)

    x = list()          # Convert to +1,-1
    for bit in bits:
```

```
    #if bit == 0:
    x.append((bit*2)-1)

#print "x=",x
# Build the partial sums
pos = 0
s = list()
for e in x:
    pos = pos+e
    s.append(pos)
sprime = [0]+s+[0] # Add 0 on each end

#print "sprime=",sprime
# Build the list of cycles
pos = 1
cycles = list()
while (pos < len(sprime)):
    cycle = list()
    cycle.append(0)
    while sprime[pos]!=0:
        cycle.append(sprime[pos])
        pos += 1
    cycle.append(0)
    cycles.append(cycle)
    pos = pos + 1

J = len(cycles)
print("J="+str(J))

vxk = [['a','b','c','d','e','f'] for y in [-4,-3,-2,-1,1,2,3,4] ]

# Count Occurances
for k in range(6):
    for index in range(8):
        mapping = [-4,-3,-2,-1,1,2,3,4]
        x = mapping[index]
        cyclecount = 0
        #count how many cycles in which x occurs k times
        for cycle in cycles:
            oc = 0
            #Count how many times x occurs in the current cycle
            for pos in cycle:
                if (pos == x):
                    oc += 1
            # If x occurs k times, increment the cycle count
            if (k < 5):
                if oc == k:
                    cyclecount += 1
            else:
                if k == 5:
                    if oc >=5:
                        cyclecount += 1
```

```
            vxk[index][k] = cyclecount

    # Table for reference random probabilities
    pixk=[[0.5     ,0.25   ,0.125  ,0.0625  ,0.0312 ,0.0312],
          [0.75    ,0.0625 ,0.0469 ,0.0352  ,0.0264 ,0.0791],
          [0.8333  ,0.0278 ,0.0231 ,0.0193  ,0.0161 ,0.0804],
          [0.875   ,0.0156 ,0.0137 ,0.012   ,0.0105 ,0.0733],
          [0.9     ,0.01   ,0.009  ,0.0081  ,0.0073 ,0.0656],
          [0.9167  ,0.0069 ,0.0064 ,0.0058  ,0.0053 ,0.0588],
          [0.9286  ,0.0051 ,0.0047 ,0.0044  ,0.0041 ,0.0531]]

    success = True
    plist = list()
    for index in range(8):
        mapping = [-4,-3,-2,-1,1,2,3,4]
        x = mapping[index]
        chisq = 0.0
        for k in range(6):
            top = float(vxk[index][k]) - (float(J) * (pixk[abs(x)-1][k]))
            top = top*top
            bottom = J * pixk[abs(x)-1][k]
            chisq += top/bottom
        p = gammaincc(5.0/2.0,chisq/2.0)
        plist.append(p)
        if p < 0.01:
            err = " Not Random"
            success = False
        else:
            err = ""
        print("x = %1.0f\tchisq = %f\tp = %f %s"  % (x,chisq,p,err))
    if (J < 500):
        print("J too small (J < 500) for result to be reliable")
    elif success:
        print("PASS")
    else:
        print("FAIL: Data not random")
    return (success, None, plist)
```

Листинг Г.15 – модуль random_excursion_variant_test

```
from __future__ import print_function

import math

# RANDOM EXCURSION VARIANT TEST
def random_excursion_variant_test(bits):
    n = len(bits)

    x = list()            # Convert to +1,-1
    for bit in bits:
        x.append((bit * 2)-1)
```

```
    # Build the partial sums
    pos = 0
    s = list()
    for e in x:
        pos = pos+e
        s.append(pos)
    sprime = [0]+s+[0] # Add 0 on each end

    # Count the number of cycles J
    J = 0
    for value in sprime[1:]:
        if value == 0:
            J += 1
    print("J=",J)
    # Build the counts of offsets
    count = [0 for x in range(-9,10)]
    for value in sprime:
        if (abs(value) < 10):
            count[value] += 1

    # Compute P values
    success = True
    plist = list()
    for x in range(-9,10):
        if x != 0:
            top = abs(count[x]-J)
            bottom = math.sqrt(2.0 * J *((4.0*abs(x))-2.0))
            p = math.erfc(top/bottom)
            plist.append(p)
            if p < 0.01:
                err = " Not Random"
                success = False
            else:
                err = ""
            print("x = %1.0f\t count=%d\tp = %f %s"  % (x,count[x],p,err))

    if (J < 500):
        print("J too small (J=%d < 500) for result to be reliable" % J)
    elif success:
        print("PASS")
    else:
        print("FAIL: Data not random")
    return (success,None,plist)
```

Листинг Г.16 – модуль diehard_birthdays

```
def diehard_birthdays(binary_string):
    # переводим строку из битов в список байтов
    byte_string = []
    for i in range(0, len(binary_string), 8):
        byte = int(binary_string[i:i+8], 2)
        byte_string.append(byte)
```

```
    # считаем количество каждого возможного значения байта
    byte_counts = Counter(byte_string)

    # вычисляем сумму квадратов количеств каждого байта
    sum_of_squares = sum(count ** 2 for count in byte_counts.values())

    # вычисляем общее количество байтов и средний квадрат
    num_bytes = len(byte_string)
    mean_square = num_bytes ** 2 / 256

    # вычисляем значение статистики
    stat_value = (sum_of_squares - mean_square) / mean_square

    # проверяем значение статистики
    if abs(stat_value - 1.0) < 0.001:
        return "Diehard Birthdays Test: \t\t\t\t\t\t\tFailed"
    else:
        return "Diehard Birthdays Test: \t\t\t\t\t\tPassed"
```

Листинг Г.17 – модуль diehard_operm5

```
def diehard_operm5(binary_string):
    binary_string = binary_string[:int(len(binary_string)//100)]
    # преобразуем строку из битов в список чисел от 0 до 255
    bytes_list = [int(binary_string[i:i + 8], 2) for i in range(0,
len(binary_string), 8)]

    # заполняем список рангов случайной перестановки
    ranks = []
    # print("len(bytes_list) ",len(bytes_list))
    for i in range(len(bytes_list)):
        rank = 1
        for j in range(i):
            if bytes_list[j] < bytes_list[i]:
                rank += 1
        ranks.append(rank)

    # проверяем, можно ли выразить сумму рангов как N(N+1)/4
    N = len(ranks)
    sum_ranks = sum(ranks)
    expected_sum_ranks = N * (N + 1) / 4
    variance = N * (N - 1) * (2*N + 5) / 72
    stddev = variance ** 0.5

    Z = (sum_ranks - expected_sum_ranks) / stddev

    # проверяем значение Z-статистики
    if abs(Z) < 3:
        return "Diehard OPERM5 Test: \t\t\t\t\t\tPassed"
    else:
        return "Diehard OPERM5 Test: \t\t\t\t\t\t\tFailed"
```

Листинг Г.18 – модуль diehard_rank_32x32

```
def diehard_rank_32x32(binary_string):
    # проверяем, что строка содержит 32*32=1024 битов
    binary_string = binary_string[:1024]
    # print(len(binary_string))
    if len(binary_string) != 1024:
        return "Error: input string must have 1024 bits"

    # преобразуем строку из битов в двумерный массив из 32x32 битов
    bits = np.array(list(binary_string), dtype=int)
    bits = bits.reshape((32, 32))

    # заполняем матрицу рангов случайной матрицы
    ranks = np.zeros((32, 32))
    for i in range(32):
        for j in range(32):
            rank = 1
            for k in range(32):
                if k != i and bits[k][j] < bits[i][j]:
                    rank += 1
            for k in range(32):
                if k != j and bits[i][k] < bits[i][j]:
                    rank += 1
            # print(ranks[i][j])
            ranks[i][j] = rank

    # проверяем, можно ли выразить сумму рангов как 495
    x = 0
    for i in range(32):
        for j in range(32):
            # print(ranks[i][j], end='\t')
            if i < 1:
                x += ranks[i][j]
    # print(x)
    sum_ranks = int(np.sum(ranks))
    # print(sum_ranks)
    if sum_ranks > 495:
        return "Diehard Rank 32x32 Test:\t\t\t\t\t\tPassed"
    else:
        return "Diehard Rank 32x32 Test:\t\t\t\t\t\t\tFailed"
```

Листинг Г.19 – модуль diehard_rank_6x8

```
def diehard_rank_6x8(bits):
    bit_list = [int(b) for b in bits]
    matrix = [[0]*8 for _ in range(6)]
    i, j = 0, 0
    for bit in bit_list:
        matrix[i][j] = bit
        j += 1
        if j == 8:
            i += 1
            j = 0
        if i == 6:
            break
    if i < 6:
        return "Diehard Rank 6x8 Test:\t\t\t\t\t\tFailed"
    row_ranks = [sum(row) for row in matrix]
    col_ranks = [sum(col) for col in zip(*matrix)]
    all_ranks = sorted(row_ranks + col_ranks)
```

```
    n = len(all_ranks)
    mean = 0.5 * (n + 1)
    std_dev = (n * (n - 1) * (2 * n + 5) / 18) ** 0.5
    ranks_norm = [(x - mean) / std_dev for x in all_ranks]
    p_value = erf(abs(sum(ranks_norm) / pow(2,-2)))
    if p_value >= 0.001:
        return "Diehard Rank 6x8 Test:\t\t\t\t\t\tPassed"
    else:
        return "Diehard Rank 6x8 Test:\t\t\t\t\t\tFailed"
```

Листинг Г.20 – diehard_bitstream

```
def diehard_bitstream(bits):
    n = len(bits)
    if n < 1000:
        return "Diehard Bitstream Test: \t\t\t\t\tFailed"
    block_size = 20
    num_blocks = n // block_size
    start = 0
    ones_count = 0
    for i in range(num_blocks):
        block = bits[start:start + block_size]
        start += block_size
        if all(c in ['0', '1'] for c in block):
            ones_count += sum(int(b) for b in block)
        else:
            return "Diehard Bitstream Test: \t\t\t\t\tFailed"
        if start >= n:
            break
    proportion = ones_count / block_size / num_blocks
    p_value = erfc(abs(proportion - 0.5) / (pow(2,-2) / 2))
    if p_value >= 0.01:
        return "Diehard Bitstream Test: \t\t\t\t\tPassed"
    else:
        return "Diehard Bitstream Test: \t\t\t\t\tFailed"
```

Листинг Г.21 – модуль diehard_opso

```
def diehard_opso(bits):
    n = len(bits)
    if n < 100:
        return "Diehard OPSO Test:\t\t\t\t\tFailed"
    index = 0
    count = 0
    while index < n:
        k = 1
        while k < n - index:
            if bits[index + k] != bits[index]:
                break
            k += 1
        if k > 1:
            count += min(k - 1, 16)
        index += k
    p_value = 1 - 0.5 ** (count / 16)
    if p_value >= 0.01:
        return "Diehard OPSO Test:\t\t\t\t\tPassed"
    else:
        return "Diehard OPSO Test:\t\t\t\t\tFailed"
```

Листинг Г.22 – модуль diehard_oqso_test

```
def diehard_oqso_test(bits):
    if len(bits) < 1000:
        print(len(bits))
        print("Diehard OQSO Test: Not Enough Data")
        return

    n = len(bits) // 2
    s = bits[:n]
    t = bits[n:]

    r = 0
    for i in range(n):
        if s[i] != t[i]:
            r += 1

    seq = 2 * n - abs(r - n)
    k = 2 * n * (n - 1) // 3
    mu = k
    sigma = ((16 * n - 29) / 90)**0.5

    z = abs(seq - mu) / sigma
    p_value = erfc   (z / (pow(2,-2)))
    if p_value > 0.01:
        return "Diehard OQSO Test: \t\t\t\t\t\t\t\tFailed"
    else:
        return "Diehard OQSO Test: \t\t\t\t\t\t\tPassed"
```

Листинг Г.23 – модуль diehard_dna_test

```
def diehard_dna_test(bits):
    if len(bits) < 20000:
        print("Diehard DNA Test: Not Enough Data")
        print(len(bits))
        return

    n = len(bits) // 4
    a = bits[:n]
    c = bits[n:2*n]
    g = bits[2*n:3*n]
    t = bits[3*n:]

    A = a.count("1")
    C = c.count("1")
    G = g.count("1")
    T = t.count("1")

    chi2 = (4 * n * ((A - n/4)**2 + (C - n/4)**2 + (G - n/4)**2 + (T - n/4)**2)) / n

    p_value = gammainc(3, chi2/2)
    # print(p_value)
    if p_value < 0.01:
        return "Diehard DNA Test: \t\t\t\t\t\tFailed"
    else:
        return "Diehard DNA Test: \t\t\t\t\t\tPassed"
```

Листинг Г.24 – модуль diehard_count_ones_stream_test

```
def diehard_count_ones_stream_test(bits):
    if len(bits) < 1000:
        # print("Diehard Count the 1's Test: Not Enough Data")
        return "Diehard Count the 1's Test: Not Enough Data"

    count = bits.count("1")
    n = len(bits)

    # calculate the expected mean and variance
    mu = n / 2
    sigma2 = n / 4

    # calculate the standard normal deviate
    z = abs(count - mu) / (2 * sigma2)**0.5

    # calculate the p-value of the test
    p_value = erf(z / pow(2,-2))
    # print(p_value)
    if p_value < 0.01:
        return "Diehard Count the 1's Test: \t\t\t\t\tFailed"
    else:
        return "Diehard Count the 1's Test: \t\t\t\t\tPassed"
```

Листинг Г.25 – модуль diehard_count_ones_bytes_test

```
def diehard_count_ones_bytes_test(binary_str):
    # превращаем строку из битов в список из целых чисел
    binary_str = binary_str[:20000]
    binary_list = [int(bit) for bit in binary_str]
    num_bytes = len(binary_list) // 8 # количество байтов
    # print(num_bytes)
    ones_count = 0
    for i in range(num_bytes):
        byte = binary_list[i*8:(i+1)*8] # каждый байт - 8 битов
        ones_count += byte.count(1) # считаем количество единиц в байте
    # print(ones_count)
    if ones_count >= 9654 and ones_count <= 10346:
        return "Diehard Count the 1's (byte) Test\t\t\t\tPassed"
    else:
        return "Diehard Count the 1's (byte) Test\t\t\t\t\tFailed"
```

Листинг Г.26 – модуль diehard_parking_lot_test

```
def diehard_parking_lot_test(bits):
    # Convert the input bitstring to a list of integers (0 or 1).
    bit_list = [int(bit) for bit in bits]
    bit_list = bit_list[:10000]
```

```
    # Check that the input bit list has length 10000.
    if len(bit_list) != 10000:
        return "не пройден"
    else:
        # Count the number of parked cars in every 325-unit segment.
        parked_cars_counts = [sum(bit_list[i:i+325]) for i in range(0, len(bit_list), 325)]

        # Check that the number of parked cars is never more than 5 in any 325-unit segment.
        # if  > 5:
        if max(parked_cars_counts)>= 42:
            return "Diehard Parking Lot Test\t\t\t\t\t\tPassed"
        else:
            return "Diehard Parking Lot Test\t\t\t\t\t\tFailed"
```

Листинг Г.27 – модуль min_distance_2dcircle_test

```
def min_distance_2dcircle_test(bit_string):
    # Получаем количество бит в строке
    n = len(bit_string)
    remaining_bits = n % 28
    if remaining_bits != 0:
        bit_string = bit_string[:-remaining_bits]
    n = len(bit_string)
    # Проверяем, кратное ли 28 это число
    if n % 28 != 0:

        return "Diehard Minimum Distance Test (2d Circle) \t\t\t\t\tFailed"

    # Считаем количество кругов
    num_circles = n // 28

    # Проходим через каждый круг
    for i in range(num_circles):
        # Считаем биты в каждом круге
        circle_bits = bit_string[(28 * i):(28 * i + 28)]

        # Преобразуем биты в координаты круга
        x = [0] * 8
        y = [0] * 8
        for j in range(4):
            x[j] = int(circle_bits[4 * j:4 * j + 2], 2)
            y[j] = int(circle_bits[4 * j + 2:4 * j + 4], 2)

        for j in range(4):
            x[4 + j] = y[j]
            y[4 + j] = x[j]

        # Проверяем минимальное расстояние между каждой парой точек
        min_distance = float('inf')
        for j in range(7):
            for k in range(j + 1, 8):
```

```
            distance = ((x[j] - x[k]) ** 2 + (y[j] - y[k]) ** 2) ** 0.5
            if distance < min_distance:
                min_distance = distance

    # Если минимальное расстояние больше 1.5, то тест не пройден
    # print(min_distance)
    if min_distance >= 1.5:
        return "Diehard Minimum Distance Test (2d Circle) \t\t\t\t\tFailed"

# Если все круги прошли тест, то он пройден
return "Diehard Minimum Distance Test (2d Circle) \t\t\t\tPassed"
```

Листинг Г.28 – модуль diehard_3d_spheres_test

```
def diehard_3d_spheres_test(bits):
    # Проверяем, что строка содержит не менее 612 битов
    if len(bits) < 612:
        return "Diehard 3D Spheres (Minimum Distance) Test\t\t\t\t\tFailed"

    # Преобразуем биты в координаты x, y, z
    coords = []
    for i in range(0, 612, 3):
        x = int(bits[i:i+2], 2)
        y = int(bits[i+2:i+4], 2)
        z = int(bits[i+4:i+6], 2)
        coords.append((x, y, z))

    # Вычисляем минимальное расстояние между точками
    min_distance = float("inf")
    for i, p1 in enumerate(coords):
        for j, p2 in enumerate(coords):
            if i != j:
                distance = ((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2 + (p1[2]-p2[2])**2)**0.5
                if distance != 0:
                    min_distance = min(min_distance, distance)

    # Сравниваем полученный результат с ожидаемым
    # print(min_distance)
    if min_distance >= 0.01:
        return "Diehard 3D Spheres (Minimum Distance) Test\t\t\t\tPassed"
    else:
        return "Diehard 3D Spheres (Minimum Distance) Test\t\t\t\t\tFailed"
```

Листинг Г.29 – модуль diehard_squeeze_test

```
def diehard_squeeze_test(bit_string):
    # Получаем количество бит в строке
    n = len(bit_string)
    remaining_bits = n % 8
    if remaining_bits != 0:
        bit_string = bit_string[:-remaining_bits]
```

```
    n = len(bit_string)

    # Проверяем, кратное ли 8 это число
    if n % 8 != 0:
        return "Diehard Squeeze Test \t\t\t\t\t\tFailed"

    # Подсчитываем количество отрезков в полосе
    num_segments = n // 8

    # Проходим через каждый отрезок и вычисляем его дробное число
    for i in range(num_segments):
        segment = bit_string[(8 * i):(8 * i + 8)]
        decimal_num = int(segment, 2) / 2**8

        # Если дробное число не входит в отрезок [0, 1), то тест не пройден
        if decimal_num >= 1:
            return "Diehard Squeeze Test \t\t\t\t\t\tFailed"

    # Если все отрезки входят в отрезок [0, 1), то тест пройден
    return "Diehard Squeeze Test \t\t\t\t\t\tPassed"
```

Листинг Г.30 – модуль diehard_runs_test

```
def diehard_runs_test(bit_string):
    # Преобразует битовую строку в список из битов
    bits = [int(bit) for bit in bit_string]

    # Считает количество серий нулей и единиц
    zeros = ones = 0
    for bit in bits:
        if bit == 0:
            zeros += 1
        else:
            ones += 1

    # Вычисляет ожидаемое количество серий
    expected = (2 * zeros * ones / len(bits)) + 1

    # Считает количество фактических серий
    runs = 1
    for i in range(1, len(bits)):
        if bits[i] != bits[i-1]:
            runs += 1

    # Вычисляет статистику теста
    chi_squared = ((runs - expected)**2) / expected

    # Определяет критическое значение для уровня значимости 0.01
    critical_value = 16.919

    if chi_squared < critical_value:
        return "Diehard Runs Test \t\t\t\t\t\tPassed"
```

```
    else:
        return "Diehard Runs Test \t\t\t\t\t\t\tFailed"
```

Листинг Г.31 – модуль diehard_craps_test

```
def diehard_craps_test(bit_string):
    if len(bit_string) < 20000:
        return "Diehard Craps Test \tNot enough bits to perform the test"

    length = len(bit_string) // 4
    chunks = [bit_string[i*length:(i+1)*length] for i in range(4)]

    for i in range(len(chunks)):
        if i == 0:
            expected = "0101010101010101"
            test_name = "Test 1"
        elif i == 1:
            expected = "0011001100110011"
            test_name = "Test 2"
        elif i == 2:
            expected = "0000111100001111"
            test_name = "Test 3"
        else:
            expected = "0000000011111111"
            test_name = "Test 4"

        if expected in chunks[i]:
            continue
        else:
            return "Diehard Craps Test \t\t\t\t\t\t\tFailed"

    return "Diehard Craps Test \t\t\t\t\t\tPassed"
```

Листинг Г.32 – модуль marsaglia_tsang_test

```
def marsaglia_tsang_test(bits):
    # Convert the input bitstring to a list of integers (0 or 1).
    bit_list = [int(bit) for bit in bits]
    bit_list = bit_list[:60]
    # Check that the length of the input bit list is divisible by 2.
    if len(bit_list) % 2 != 0:
        remaining_bits = n % 8
        bit_string = bit_string[:-remaining_bits]

    if len(bit_list) % 2 != 0:
        result = "\t\t\t\t\tFailed"
    else:
        # Extract pairs of 32-bit integers from the input bit list.
        integers = []
        for i in range(0, len(bit_list), 32):
            integer_bits = bit_list[i:i+32]
            integer = 0
            for j in range(len(integer_bits)):
                integer += integer_bits[j] * 2**(31-j)
            integers.append(integer)

        # Check that the GCD of each pair of integers is not equal to 1.
        for i in range(0, len(integers), 2):
```

```
            # print(math.gcd(integers[i], integers[i+1]))
            if math.gcd(integers[i], integers[i+1]) == 1:
                result = "\t\t\t\t\tFailed"
                break
        else:
            result = "\t\t\t\tPassed"

    return "Marsaglia and Tsang GCD Test\t" + result
```

Листинг Г.33 – модуль serial_test

```
def serial_test(s):
    """Serial Test"""
    block_size = 3
    num_blocks = len(s) // block_size
    ones_count = s.count('1')
    zeros_count = s.count('0')
    pi = ones_count / (ones_count + zeros_count)
    var = (2 * (ones_count + zeros_count) / block_size) * (pi * (1 - pi))
    std_dev = var ** 0.5

    passed = True

    for i in range(num_blocks):
        block = s[i*block_size : (i+1)*block_size]
        ones = block.count('1')
        zeros = block.count('0')
        if ones == zeros:
            continue
        p = 0.75 if ones < zeros else 0.25
        t = (ones - pi*block_size) / std_dev
        if abs(t) > 2.326 or (abs(t) > 1.96 and i > 0 and i < num_blocks-1 and
block != s[i*block_size-block_size : i*block_size] and block !=
s[(i+1)*block_size : (i+1)*block_size+block_size]):
            passed = False
            break

    if passed:
        return "Serial Test \t\t\t\t\t\t\tPassed"
    else:
        return "Serial Test \t\t\t\t\t\t\tFailed"
```

Листинг Г.34 – модуль rgb_bit_distribution_test

```
def rgb_bit_distribution_test(s):
    """RGB Bit Distribution Test"""
    num_bits = len(s)
    num_groups = num_bits // 24
    num_0s, num_1s = 0, 0
    passed = True

    for i in range(num_groups):
        r = s[i*24 : i*24+8]
        g = s[i*24+8 : i*24+16]
        b = s[i*24+16 : i*24+24]
        counts = [r.count('1'), g.count('1'), b.count('1')]
        num_0s += counts.count(0)
```

```
        num_1s += counts.count(8)

    if num_0s < 2 or num_1s < 2:
        passed = False

    if passed:
        return "RGB Bit Distribution Test \t\t\t\t\t\tPassed"
    else:
        return "RGB Bit Distribution Test \t\t\t\t\t\tFailed"
```

Листинг Г.35 – rgb_permutations_test

```
def rgb_permutations_test(input_string):

    # Проверяем, что входная строка содержит только биты
    if not all(bit in ['0', '1'] for bit in input_string):
        raise ValueError("Input string should contain only '0' and '1' bits.")


    # Проверяем, что входная строка содержит кратное 3 количество битов

    n = len(input_string)
    # print(n)
    remaining_bits = n % 3
    if remaining_bits != 0:
        input_string = input_string[:-remaining_bits]
    n = len(input_string)
    # print(n)
    if len(input_string) % 3 != 0:
        raise ValueError("Input string should contain a multiple of 3 bits.")

    # Создаем словарь, который будет хранить количество встреч RGB комбинаций
    rgb_counts = {'R': 0, 'G': 0, 'B': 0}

    # Проходимся по битам входной строки, группируя их в RGB комбинации
    for i in range(0, len(input_string), 3):
        rgb = input_string[i:i+3]

        # Увеличиваем счетчик соответствующему цвету
        if rgb == '000':
            rgb_counts['R'] += 1
        elif rgb == '001':
            rgb_counts['G'] += 1
        elif rgb == '010':
            rgb_counts['B'] += 1
        elif rgb == '011':
            rgb_counts['R'] += 1
            rgb_counts['G'] += 1
        elif rgb == '100':
            rgb_counts['R'] += 1
            rgb_counts['B'] += 1
        elif rgb == '101':
            rgb_counts['G'] += 1
            rgb_counts['B'] += 1
        elif rgb == '110':
            rgb_counts['R'] += 1
            rgb_counts['G'] += 1
            rgb_counts['B'] += 1
        elif rgb == '111':
            pass
```

```
    # Проверяем, что количество каждого цвета примерно равно
    n = len(input_string) // 3
    expected_count = n / 3

    for count in rgb_counts.values():
        # print(expected_count/2, abs(count - expected_count),
expected_count/2 < abs(count - expected_count))
        if abs(count - expected_count) > expected_count / 2:
            return("RGB Permutations Test \t\t\t\t\t\t\tFailed")

    return("RGB Permutations Test \t\t\t\t\t\tPassed")
```

Листинг Г.36 – модуль RGBLaggedSumTest

```
def RGBLaggedSumTest(bits):
    n = len(bits)
    k = 8 # значение k для RGB Lagged Sum Test
    M = 2**k
    F = [0]*M
    # заполнение массива F
    for i in range(n-k):
        index = int(bits[i:i+k], 2)
        F[index] = (F[index] + int(bits[i+k])) % M
    # вычисление значения S
    S = sum([(F[i]-0.5)**2/M for i in range(M)])

    # вычисление значения psi
    psi = math.erfc(math.pow(2*S,-2))
    # проверка результата
    if psi >= 0.01:
        result = "Passed"
    else:
        result = "\tFailed"
    return "RGB Lagged Sum Test\t\t\t\t\t\t" + result
```

Листинг Г.37 – модуль DABNonlinearMixingTest

```
def DABNonlinearMixingTest(bits):
    n = len(bits)
    r = 1 # длина каждого блока
    M = 10 # количество блоков
    Q = [0] * M
    # обработка блоков
    for i in range(M):
        b = bits[i*r:(i+1)*r]
        # преобразование блока в число
        x = int(b, 2)
        # применение нелинейной функции
        y = ((x**2) % 255) ^ 2
        # подсчет значений Q[i]
        for j in range(r):
            if ((y >> j) & 1) == 1 and ((x >> j) & 1) == 1:
                Q[i] += 1
            elif ((y >> j) & 1) == 0 and ((x >> j) & 1) == 0:
                Q[i] -= 1
    # вычисление значения psi
    phi = sum([abs(Q[i]) for i in range(M)]) / (r*M)
    psi = math.erfc(phi / math.sqrt(2))
    # проверка результата
    if psi >= 0.01:
        result = "Passed"
```

```
    else:
        result = "\tFailed"
    return "DAB Nonlinear Mixing Test\t\t\t\t\t\t" + result
```

Листинг Г.38 – модуль test_bitwise_independence

```
def test_bitwise_independence(binary_string):
    # проверка битовой независимости
    n = len(binary_string)
    ones_count = binary_string.count('1')
    zeros_count = n - ones_count
    expected_count = n / 2
    x = 0
    if abs(ones_count - expected_count) > 2.576*(((n * 0.5 * 0.5) ** 0.5)):
        # print("Тест битовой независимости не пройден")
        pass
    else:
        # print("Тест битовой независимости пройден")
        x +=1

    # проверка байтовой независимости
    if n % 8 != 0:
        binary_string += '0'*(8 - (n % 8))
        n = len(binary_string)

    bytes_count = n // 8
    bytes_list = [binary_string[i:i+8] for i in range(0, n, 8)]

    ones_count = [byte.count('1') for byte in bytes_list]
    zeros_count = [8 - ones for ones in ones_count]
    expected_count = [4] * bytes_count

    chi_squared_statistic = sum([((ones_count[i] - expected_count[i])**2) / expected_count[i] +
((zeros_count[i] - expected_count[i])**2) / expected_count[i] for i in range(bytes_count)])
    degree_freedom = bytes_count * 2 - 2
    p_value = 1 - chi2.cdf(chi_squared_statistic, degree_freedom)
    if p_value < 0.01:
        print("Тест байтовой независимости не пройден")
    else:
        print("Тест байтовой независимости пройден")
        x += 1

    if x == 2:
        return 'Byte-wise vs. Bit-wise Independence Test\t\t\t\tPassed'
    else:
        return 'Byte-wise vs. Bit-wise Independence Test\t\t\t\t\tFailed'
```

Листинг Г.39 – модуль dab_overlap_quadruples_test

```
def dab_overlap_quadruples_test(bits):
    # Находим все возможные четверки битов
```

```
    quadruples = [bits[i:i+4] for i in range(len(bits)-3)]
    # Считаем количество вхождений для каждой четверки
    counts = {}
    for q in quadruples:
        if q not in counts:
            counts[q] = 1
        else:
            counts[q] += 1

    # Проверяем выполнение условий теста
    N = len(bits) // 4 # количество четверок в строке
    M = len(quadruples) // N # количество блоков
    failed = True
    for i in range(M):
        for j in range(N):
            a = i*N + j
            b = i*N + (j+1)%N
            c = ((i+1)%M)*N + j
            d = ((i+1)%M)*N + (j+1)%N
            quadruple = bits[a] + bits[b] + bits[c] + bits[d]
            # print(counts[quadruple])
            if counts[quadruple] != 1:
                failed = False
                break
        if failed:
            break

    # Вывод результата теста
    if failed:
        return("Dab Overlapping Quadruples test:\t\t\t\t\tFailed")
    else:
        return("Dab Overlapping Quadruples test:\t\t\t\t\tPassed")
```

Листинг Г.40 – модуль diehard_count_ones_bytes_test

```
def dna_test(bits):
    bits = bits[:10]
    if len(bits) < 10:
        return "DNA Test", "\t", "Not enough bits to perform the test"

    for i in range(len(bits) - 9):
        if bits[i:i+10].count("0") in [0, 10]:
            return "DNA Test\t\t\t\t\t\t\t\t\tFailed"

    return "DNA Test\t\t\t\t\t\t\t\t\tPassed"
```

Листинг Г.41 – модуль count_ones_test

```
def count_ones_test(bits):
    ones = bits.count("1")
    zeros = len(bits) - ones
    if abs(ones - zeros) < (2 * (len(bits) ** 0.5)):
        return "Count the 1's Test\t\t\t\t\t\t\tPassed"

    return "Count the 1's Test\t\t\t\t\t\t\tFailed"
```

Листинг Г.42 – модуль parking_lot_test

```
def parking_lot_test(bits):
    bits = bits[:30]
    if len(bits) < 20:
        return "Parking Lot Test", "\t", "Not enough bits to perform the test"

    parking_lots = []
    for i in range(len(bits) - 1):
        if bits[i:i+2] == "00":
            parking_lots.append("empty")
        elif bits[i:i+2] == "11":
            parking_lots.append("full")

    if len(parking_lots) < 5:
        return "Parking Lot Test", "\t", "Not enough parking lot samples to perform the test"

    occupied_lots = parking_lots.count("full")
    expected_occupied = len(parking_lots) / 3
    if abs(occupied_lots - expected_occupied) > (2 * (expected_occupied ** 0.5)):
        return "DAB Parking Lot Test\t\t\t\t\t\tFailed"

    return "DAB Parking Lot Test\t\t\t\t\tPassed"
```

Листинг Г.43 – модуль diehard_3d_spheres_test

```
def dab_minimum_distance_test(bits):
    if len(bits) < 1000:
        return "Minimum Distance Test (3D)", "\t", "Not enough bits to perform the test"
    bits = bits[:1000]

    dimension = 3
    num_points = len(bits) // (dimension * 8)
    point_list = []
    for i in range(num_points):
        x = bits[(i*dimension*8):(i*dimension*8+8)]
        y = bits[(i*dimension*8+8):(i*dimension*8+16)]
        z = bits[(i*dimension*8+16):(i*dimension*8+24)]
        point_list.append((int(x, 2), int(y, 2), int(z, 2)))

    if len(set(point_list)) < math.ceil(0.75 * num_points):
```

```
        return "Minimum Distance Test (3D)", "\t", "Failed"
    # print(point_list)
    min_distance = min([math.dist(point_list[i], point_list[j]) for i in range(num_points-1) for j in
range(i+1,num_points)])

    if min_distance > 0.532 * (num_points ** (-1/3)):
        return "DAB Minimum Distance Test (3D)\t\t\t\t\tPassed"

    return "DAB Minimum Distance Test (3D)\t\t\t\t\t\tFailed"
```