МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт кибернетики (ИК)
Кафедра высшей математики (ВМ)

РАБОТА ДОПУЩЕНА К ЗАЩИТЕ

Заведующий
кафедрой _____ Ю.И.Худак

«____» _____ 2020 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

по направлению подготовки бакалавров
01.03.02 «Прикладная математика и информатика»

На тему: Свойства (t, s)-последовательностей и их верификация

Обучающийся: _____ Елисеев А.А.
подпись

Шифр 16К0166

Группа КМБО–03–16

Руководитель работы _____ канд. техн. наук, доцент, доцент Парфенов Д.В.
подпись

Москва 2020 г.

МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт кибернетики
Кафедра высшей математики

СОГЛАСОВАНО

Заведующий
кафедрой _____ Ю.И. Худак

«_____» _____ 2020 г.

УТВЕРЖДАЮ

Директор
института _____ М.П. Романов

«_____» _____ 2020 г.

ЗАДАНИЕ
на выполнение выпускной квалификационной работы бакалавра

Обучающийся: Елисеев Андрей Александрович
Шифр: 16К0166
Направление подготовки 01.03.02 «Прикладная математика и информатика»
Группа КМБО–03–16

1. Тема выпускной квалификационной работы: Свойства (t,s)-последовательностей и их верификация.

2. Цель и задачи выпускной квалификационной работы

Цель работы: разработать методы верификации свойств цифровых (t,s)-последовательностей, позволяющих оценивать равномерность заполнения их элементами области $\mathcal{I}^s$.

Задачи работы:

1. Разработать и математически обосновать универсальный алгоритм верификации параметра t цифровых (t,s)-последовательностей, не зависящий от их основания;
2. Разработать и математически обосновать универсальный алгоритм верификации многомерных корреляционных свойств цифровых (t,s)-последовательностей, не зависящий от их основания, путём применения к ним метода главных компонент;
3. Протестировать разработанные алгоритмы на цифровых (t,s)-последовательностях Нидеррайтера с основанием 2.

3. Этапы выполнения выпускной квалификационной работы бакалавра

№ этапа	Содержание этапа ВКР	Результат выполнения этапа ВКР	Срок выполнения
1	Анализ первоисточников	Собрана и проанализирована литература по (t,s)-последовательностям	25.02.2020
2	Разработка алгоритма верификации t	Разработан и протестирован алгоритм верификации t	10.03.2020

3	Разработка алгоритма верификации главных осей	Разработан и протестирован алгоритм верификации главных осей	14.04.2020
4	Подготовка презентации	Презентация	05.05.2020
5	Оформление работы	Оформленный текст ВКР	27.05.2020

4. Перечень разрабатываемых документов и графических материалов:

Иллюстрация нескольких первых точек различных (t, s)-последовательностей;

Иллюстрация ухудшения качества однородности (t, s)-последовательностей с ростом параметра t;

Иллюстрации корреляционных несовершенств точек (t, s)-последовательностей;

Таблица с результатами тестирования алгоритма верификации параметра t на цифровых (t, s)-последовательностях Нидеррайтера с основанием 2;

Таблица с результатами тестирования алгоритма верификации главных осей на цифровых (t, s)-последовательностях Нидеррайтера с основанием 2;

Графики изменения отклонения доли дисперсии, приходящейся на первую главную ось, от значения s^{-1} в зависимости от периода цифровых (t, s)-последовательностей Нидеррайтера с основанием 2 при фиксированном s и варьирующемся t;

Графики изменения отклонения доли дисперсии, приходящейся на первую главную ось, от значения s^{-1} в зависимости от периода цифровых (t, s)-последовательностей Нидеррайтера с основанием 2 при фиксированном t и варьирующемся s;

Листинг с исходным кодом реализации разработанного алгоритма верификации параметра t на языке программирования C++;

Листинг с исходным кодом реализации разработанного алгоритма верификации главных осей на языке программирования C++.

5. Руководитель и консультанты выпускной квалификационной работы

Функциональные обязанности	**Должность в Университете**	**Ф.И.О.**	**Подпись**
Руководитель выпускной работы	доцент	Парфенов Д.В.	

Задание выдал

Руководитель ВКР _____ Д.В. Парфенов

«_____»_____2020 г.

Задание принял к исполнению

Обучающийся _____А.А. Елисеев

«_____»_____2020 г.

Аннотация

В настоящей выпускной квалификационной работе рассмотрены свойства цифровых (t, s)-последовательностей, позволяющие оценивать равномерность заполнения многомерного единичного куба их элементами. Предложен алгоритм верификации параметра t, демонстрирующий высокие показатели эффективности по времени. Предложен способ оценки равномерности с помощью верификации главных осей цифровых (t, s)-последовательностей. Предложен алгоритм верификации главных осей для цифровых (t, s)-последовательностей, демонстрирующий высокие показатели эффективности по памяти. Для построения всех предложенных алгоритмов были впервые получены и доказаны необходимые теоретические результаты.

Оглавление

Определения и обозначения

В настоящей работе используются следующие определения и обозначения:

- $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$ – множество *натуральных чисел с нулём*;
- $[a..b]$, где $a \in \mathbb{Z}$, $b \in \mathbb{Z}$ и $a \leq b$, – множество всех целых чисел c таких, что $a \leq c \leq b$;
- $\mathcal{I} = [0, 1)$ – единичный полуинтервал с открытой правой границей;
- $V_s(\mathcal{A})$ – s-мерная *мера Лебега* множества $\mathcal{A}$, она же в дальнейшем *объём*;
- $(n)_{b,k}$, где $n \in \mathbb{N}_0$, $b \in \mathbb{N} \setminus \{1\}$ и $k \in \mathbb{N}_0$, – k-ый коэффициент разложения $n = \sum_{i=0}^{+\infty} (n)_{b,i} \cdot b^i$ такого, что $\forall i \in \mathbb{N}_0\ (n)_{b,i} \in [0..b-1]$;
- $\oplus$ – оператор *исключающей дизъюнкции* такой, что для $a \in \mathbb{N}_0$, $b \in \mathbb{N}_0$ и $c \in \mathbb{N}_0$ верно, что $c = a \oplus b$, если $\forall k \in \mathbb{N}_0\ (c)_{2,k} = (a)_{2,k} \oplus (b)_{2,k}$;
- $|\mathcal{A}|$ – *мощность* множества $\mathcal{A}$;
- $\binom{n}{k}$ – число *сочетаний* из n по k;
- *последовательностью* $\left\{x^{(n)}\right\}$ элементов множества $\mathcal{A}$ называется отображение $n \longmapsto x^{(n)}$ из $\mathbb{N}_0$ в $\mathcal{A}$;
- *разбиением* множества $\mathcal{A}$ называется множество множеств $\{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n\}$ такое, что $\forall i \in [1..n]$ и $\forall j \in [1..n] \setminus \{i\}$ верно то, что $\mathcal{A}_i \subset \mathcal{A}$, $\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$, а $\bigcup_{i=1}^{n} \mathcal{A}_i = \mathcal{A}$.

Введение

В современном мире численные методы являются одним из самых востребованных разделов прикладной математики. Многие вопросы, связанные с точностью расчётов, устойчивостью решений, пределами применимости тех или иных способов до сих пор открыты, а сложность поиска ответов на них может многократно увеличиваться с ростом размерности задач. В решении таких популярных на сегодняшний день проблем как многомерное численное интегрирование или многомерная численная оптимизация часто используется алгоритм квази-Монте-Карло, эффективность которого полностью обуславливается свойствами специальным образом построенных многомерных точек – (t, s)-последовательностей. Наиболее важным из их свойств является равномерность заполнения некоторого многомерного объёма.

Существует большое число исследований, посвящённых данному вопросу [1–4], однако значительная их доля обладает больше теоретическим значением, чем практическим. Среди таковых можно выделить, например, разнообразные асимптотические оценки на бесконечности или результаты для одномерных случаев. Помимо этого, необходимо учитывать то, что современная техника имеет серьёзные сложности при конструировании бесконечных последовательностей, ввиду чего на практике обыкновенно применяются так называемые «цифровые» (t, s)-последовательности, обладающие периодичностью.

Таким образом, актуальность настоящей работы заключается в том, что она предлагает практико-ориентированные способы оценки равномерности заполнения области $\mathcal{I}^s$ точками цифровых (t, s)-последовательностей.

Объект исследования: цифровые (t, s)-последовательности.

Предмет исследования: свойства цифровых (t, s)-последовательностей.

Цель: разработать методы верификации свойств цифровых (t, s)-последовательностей, позволяющих оценивать равномерность заполнения их элементами области $\mathcal{I}^s$.

Задачи:

1. Разработать и математически обосновать универсальный алгоритм верификации параметра t цифровых (t, s)-последовательностей, не зависящий от их основания;
2. Разработать и математически обосновать универсальный алгоритм верификации многомерных корреляционных свойств цифровых (t, s)-последовательностей, не зависящий от их основания, путём применения к ним метода главных компонент;
3. Протестировать разработанные алгоритмы на цифровых (t, s)-последовательностях Нидеррайтера с основанием 2.

1. Обзор источников

1.1. Общие сведения о (t, s)-последовательностях

Со второй половины XX века математиками по всему миру проводилась работа по усовершенствованию метода Монте-Карло (статистических испытаний), что в итоге привело к введению в общую практику его модификации – методов квази-Монте-Карло. Основное преимущество нового подхода заключалось в том, что он, в отличие от своего предшественника, опирался не на случайные или псевдослучайные величины, а на так называемые квазислучайные величины, сохранявшие важные стохастические свойства, но в то же время обладавшие дополнительными полезными характеристиками, позволявшими добиться существенного увеличения скорости сходимости [5].

На сегодняшний день алгоритмы квази-Монте-Карло являются доминирующим инструментом решения целого ряда численных задач: интегрирования функций многих переменных, оптимизации функций многих переменных, нахождения многомерных объёмов и многих других, благодаря чему он находит применение в разных областях деятельности человека, таких, например, как вычислительная физика, экономика, или компьютерная графика [2]. В зависимости от конкретной задачи метод может иметь характерные для неё особенности, однако его фундаментом всегда выступает конечное множество точек, определённым образом заполняющих некоторую область $\mathcal{B}$, которую зачастую принимают многомерным единичным кубом $\mathcal{I}^s$, где $s \in \mathbb{N}$ [6].

Идея о квазислучайности демонстрировала свою эффективность и активно развивалась на протяжении нескольких десятилетий, одним из основных результатов чего стало появление в 1987 году в математической номенклатуре новых сущностей – (t, s)-последовательностей и неразрывно связанных с ними (t, m, s)-сетей [7].

Определение 1.1.1 [8]

Пусть $b \in \mathbb{N} \setminus \{1\}$, $s \in \mathbb{N}$, а также $\mathcal{D} = (d_1, d_2, \dots, d_s) \in \mathbb{N}_0^s$, $\mathcal{A} = (a_1, a_2, \dots, a_s) \in \mathbb{N}_0^s$ и $a_i < b^{d_i}$ для всех $i \in [1..s]$. Тогда *элементарным интервалом с основанием* b называется множество

$$\mathcal{E} = E(b, \mathcal{A}, \mathcal{D}) = \bigtimes_{i=1}^{s} \left[\frac{a_i}{b^{d_i}}, \frac{a_i + 1}{b^{d_i}}\right) \subset \mathcal{I}^s.$$

Утверждение 1.1.1 [9]

Множество всех элементарных интервалов с равными параметрами b и $\mathcal{D}$ образуют разбиение $\mathcal{I}^s$.

Определение 1.1.2 [8]

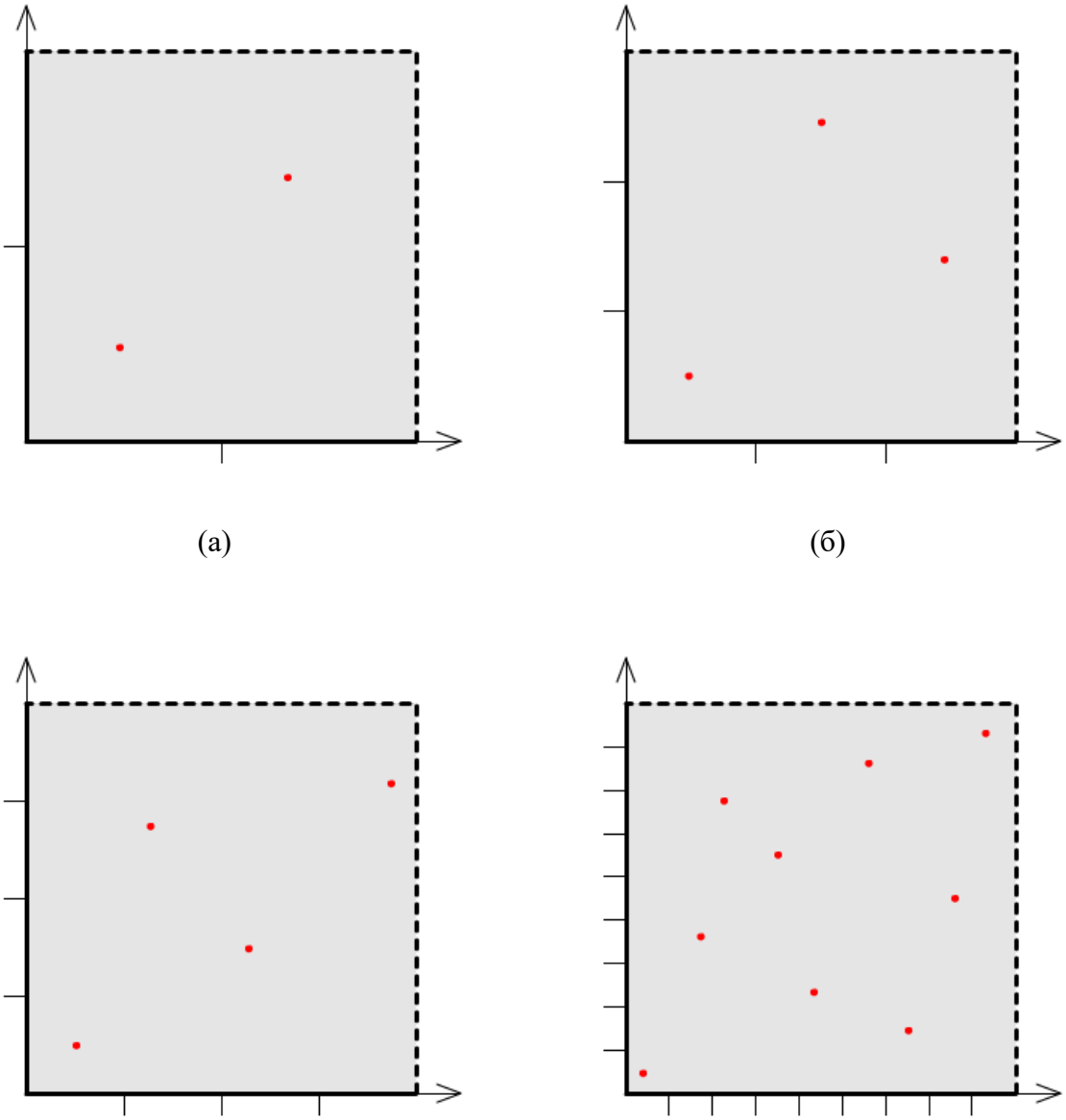
Пусть $t \in \mathbb{N}_0$, $m \in \mathbb{N}_0$, $s \in \mathbb{N}$, $b \in \mathbb{N} \setminus \{1\}$, причём $t \leq m$. Тогда множество точек $\mathcal{P} \subset \mathcal{I}^s$ является (t, m, s)-*сетью с основанием* b, если в любом элементарном интервале $\mathcal{E}$ с s-мерной мерой Лебега $V_s(\mathcal{E}) = b^{t-m}$ содержится ровно b^t точек из $\mathcal{P}$.

Утверждение 1.1.2 [10]

Каждая (t, m, s)-сеть с основанием b состоит из b^m точек.

Важнейшим свойством (t, m, s)-сетей, выделяющим их на фоне всех остальных квазислучайных множеств, является равномерность заполнения элементами сети всего объёма куба $\mathcal{I}^s$, или, иначе говоря, однородность данных множеств, что отражено в определении 1.1.2 равным числом точек внутри каждого элементарного интервала. Качество однородности разных (t, m, s)-сетей может отличаться, однако доказано, что любая (t, m, s)-сеть более однородна, чем набор псевдослучайных точек [11].

Для наглядности ниже, на рисунке 1.1.1, представлено несколько примеров различных (t, m, s)-сетей.



(а) (б)

(в) (г)

Рисунок 1.1.1. Примеры $(0, m, 2)$-сетей (красные точки) внутри единичного квадрата $\mathcal{I}^2$:
(а) $m = 1, b = 2,$ (б) $m = 1, b = 3,$
(в) $m = 2, b = 2,$ (г) $m = 2, b = 3.$

(t, s)-последовательности, в свою очередь, были введены как вспомогательная конструкция, используемая для генерации (t, m, s)-сетей.

Определение 1.1.3 [8]

Пусть $t \in \mathbb{N}_0$, $s \in \mathbb{N}$, $b \in \mathbb{N} \setminus \{1\}$. Тогда последовательность $\{x^{(n)}\}$ точек $\mathcal{I}^s$ называется (t, s)-*последовательностью с основанием* b, если для любых $k \in \mathbb{N}_0$, $m \in \mathbb{N}_0$, где $t \leq m$, множество $\{x^{(kb^m)}, x^{(kb^m+1)}, \ldots, x^{(kb^m+b^m-1)}\}$ является (t, m, s)-сетью с основанием b.

На практике для построения (t, m, s)-сетей часто используют так называемые «цифровые» (t, s)-последовательности [12], отличительным свойством которых является периодичность с периодом $b^{\widehat{m}}$ (т.е. $x^{(n)} = x^{(n+b^{\widehat{m}})}$). Каждая i-ая координата каждой n-й точки таких последовательностей определяется формулой:

$$x_i^{(n)} = \frac{\alpha_{i1}(n)}{b} + \frac{\alpha_{i2}(n)}{b^2} + \cdots + \frac{\alpha_{i\widehat{m}}(n)}{b^{\widehat{m}}}, \qquad (1.1.1)$$

где функции $\alpha_{ij}(n)$ возвращают целые числа из $[0..b-1]$ и определённым образом выражаются через s матриц, именуемых генерирующими. Разные учёные, такие как Соболь, Фор, Нидеррайтер и другие, в разное время предлагали свои алгоритмы как явного, так и неявного построения генерирующих матриц и, как следствие, свои формулы для $\alpha_{ij}(n)$ [13–16].

С практической точки зрения отдельную важность представляют (t, s)-последовательности с основанием 2, что связано, в частности, с особенностями двоичной арифметики, используемой в компьютерах [17].

1.2. Актуальные вопросы и исследования

Ввиду своего относительно недавнего возникновения, (t, s)-последовательности продолжают быть предметом активного изучения среди математиков. В многочисленных трудах, посвящённых им, исследователями рассматривается, в первую очередь, вопрос равномерности заполнения

единичного куба $\mathcal{I}^s$ их элементами, так как обнаружение наиболее однородного множества точек приведёт к наибольшей точности методов квази-Монте-Карло [18]. Необходимость сравнения разных (t, s)-последовательностей друг с другом по данному критерию порождает потребность в изучении различных их свойств, позволяющих тем или иным образом оценивать их однородность [1–4, 19, 20].

Одно из наиболее распространённых таковых свойств называется в литературе «нормами неравномерности» («discrepancies»). Они представляют собой количественные оценки однородности, которые обыкновенно определяются таким образом, чтобы меньшие числовые значения соответствовали менее дефектным множествам.

Классическими нормами неравномерности считаются D и D^*, введённые Гаральдом Нидеррайтером. Точное определение D для множества точек $\mathcal{P}$ задаётся выражением

$$D(\mathcal{P}) = \sup_{\mathcal{W}} \left| \frac{|\mathcal{W} \cap \mathcal{P}|}{|\mathcal{P}|} - V_s(\mathcal{W}) \right|, \qquad (1.2.1)$$

где $\mathcal{W}$ – это s-мерный интервал вида $\overset{s}{\underset{i=1}{\times}}[w_{i1}, w_{i2})$ при условии того, что для любого допустимого i верно $0 \leq w_{i1} < w_{i2} < 1$ [21].

Нетрудно видеть, что в формуле (1.2.1) уменьшаемое под модулем представляет собой получаемое с помощью множества $\mathcal{P}$ приближённое значение объёма $\mathcal{W}$. Исходя из этого, можно заключить, что величина D – это точная верхняя грань абсолютной ошибки, допускаемая при аппроксимации объёмов всех возможных интервалов $\mathcal{W}$. Данное свойство, пожалуй, наиболее качественно характеризует однородность, так как содержит в себе информацию о том, насколько велик может быть незаполненный точками участок пространства.

Помимо этого, совместно с D зачастую используется другая норма неравномерности – D^*, которая определяется как

$$D^*(\mathcal{P}) = \sup_{\mathcal{W}^*} \left| \frac{|\mathcal{W}^* \cap \mathcal{P}|}{|\mathcal{P}|} - V_s(\mathcal{W}^*) \right|, \qquad (1.2.2)$$

где $\mathcal{W}^*$ – это s-мерный интервал вида $\overset{s}{\underset{i=1}{\times}}[0, w_i)$ при условии того, что для любого допустимого i верно $0 < w_i < 1$ [21]. Легко заметить, что единственное отличие формулы (1.2.2) от (1.2.1) заключается в структуре рассматриваемых многомерных интервалов: интервалы $\mathcal{W}^*$ образуют подмножество среди интервалов $\mathcal{W}$. Согласно Нидеррайтеру [8], величины D и D^* связаны друг с другом следующим образом:

$$D^*(\mathcal{P}) \leq D(\mathcal{P}) \leq 2^s D^*(\mathcal{P}),$$

что даёт возможность оценивать значение D с помощью D^*.

Нормы D и D^* широко используются в теоретических работах, направленных на выявление асимптотических характеристик точек (t, s)-последовательностей. Так, исходя из имеющихся исследований, для (t, m, s)-сетей $\mathcal{P}$, получаемых с помощью цифровых последовательностей Нидеррайтера, верно выражение

$$D^*(\mathcal{P}) = \mathcal{O}\left(\frac{(\ln|\mathcal{P}|)^s}{|\mathcal{P}|}\right) \text{ [22].} \qquad (1.2.3)$$

На данный момент показатель (1.2.3) является наилучшим среди всех известных (t, s)-последовательностей [23].

Наравне с D и D^* большой популярностью также пользуется норма L_2. Она, в отличие от предыдущих, предполагает исследование того, насколько сильно эмпирическое распределение точек последовательности отличается от теоретического равномерного [8]. Данный показатель определяется с помощью формулы

$$L_2(\mathcal{P}) = \sqrt{\int_{I^s} \left| \frac{|\mathcal{W}^* \cap \mathcal{P}|}{|\mathcal{P}|} - \lambda_s(\mathcal{W}^*) \right|^2 \mathrm{d}w}, \qquad (1.2.4)$$

где $\mathcal{W}^*$ – это s-мерный интервал вида $\overset{s}{\underset{i=1}{\times}}[0, w_i)$ при условии того, что для любого допустимого i верно $0 < w_i < 1$, а $w = (w_1, w_2, \dots, w_s)$ [12]. Критцингер и Пилихсхэммер в некоторых своих работах рассматривают и более общую норму неравномерности – L_p:

$$L_p(\mathcal{P}) = \left(\int_{I^s} \left| \frac{|\mathcal{W}^* \cap \mathcal{P}|}{|\mathcal{P}|} - \lambda_s(\mathcal{W}^*) \right|^p \mathrm{d}w \right)^{\frac{1}{p}}, \tag{1.2.5}$$

где $p \in \mathbb{N}$ [24–27].

Отличительной особенностью норм L_p выступает гораздо меньшая чувствительность к локальным дефектам $\mathcal{P}$, что может послужить причиной недостаточно качественного отражения желаемых характеристик. Так, например, как указывает в своих работах Йиржи Матушек [28], L_2 является совершенно недостоверным показателем, если число точек в $\mathcal{P}$ мало относительно размерности s (например, если $|\mathcal{P}| < 10^4$ при $s = 30$). Вместе с тем, несмотря на все свои недостатки, данная норма крайне популярна в прикладных задачах при оценке однородности, так как для многих квазислучайных последовательностей были рассчитаны её значения или описаны алгоритмы, позволяющие за разумное время её найти [29, 30].

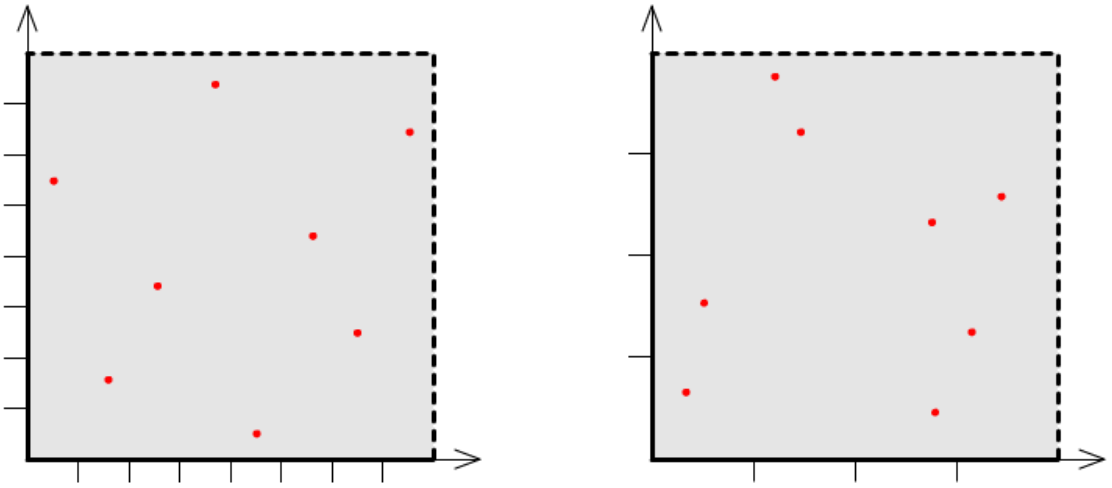
К сожалению, то же самое нельзя сказать про наиболее корректные нормы D и D^*. На данный момент известны их верхние оценки для отдельных частных случаев [22], а алгоритм расчёта значений существует только при $s = 1$ – он был рассмотрен в докторской диссертации Барта Вандевустайне [31]. Непреодолимая вычислительная сложность нахождения D и D^* в пространствах большей размерности, подробно описанная в работе Доэрр, Гневуча и Вальстрёма [32], обуславливает их низкую практическую значимость, и, как следствие, потребность в поиске альтернативных свойств, способных характеризовать однородность (t, s)-последовательностей.

Одно из таких свойств рассматривается в работах Нидеррайтера [8], Джо и Куо [33]. Согласно авторам, параметр t, включённый в обозначение

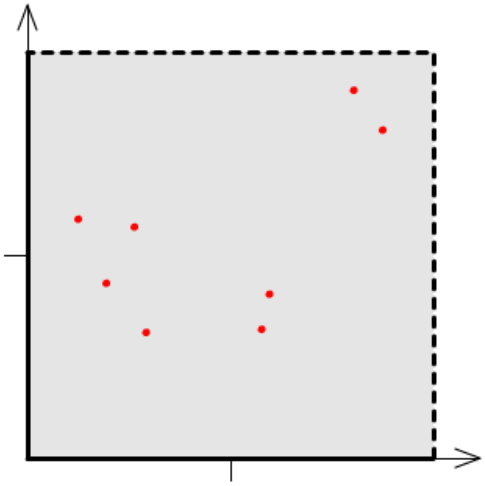
(t, m, s)-сети, может служить хорошим показателем того, насколько равномерно точки заполняют область $\mathcal{I}^s$. Обоснование данного тезиса строится на определении 1.1.2 (стр. 9), из которого следует, что при фиксированном значении m (или, альтернативно, при фиксированном числе элементов сети) с ростом t всё большему числу точек дозволяется присутствовать в элементарных интервалах всё большего объёма, следовательно, бóльшие значения t соответствуют менее равномерно распределённым сетям, что наглядно продемонстрировано далее, на рисунке 1.2.1.

Важным преимуществом использования величины t в качестве характеристики равномерности является то, что её точные минимальные значения рассчитаны в зависимости от параметра s для многих (t, s)-последовательностей [16, 33, 34]. Вместе с тем, существуют крайне быстрые алгоритмы расчёта t для цифровых (t, s)-последовательностей, что ещё больше повышает прикладную значимость как самих цифровых последовательностей, так и применения параметра t для оценки их однородности [35, 36].

Однако главной проблемой таких алгоритмов является то, что их метод работы предполагает априорное знание всех генерирующих матриц последовательности в явном виде, что, к сожалению, далеко не всегда возможно. Для таких ситуаций Бурахановой и Золкиным [37] в 2018 году был предложен алгоритм, верифицирующий значение t для цифровых (t, s)-последовательностей с периодом $b^{\widehat{m}}$ путём верификации параметра t порождённой ей $(t, \widehat{m}, s)$-сети. Их метод производит проверку определения 1.1.2 (стр. 9) и, следовательно, никаким образом не зависит от генерирующих матриц.



(а) (б)



(в)

Рисунок 1.2.1. Демонстрация ухудшения однородности $(t, 3,2)$-сети с основанием 2 с ростом значения параметра t:
(а) $t = 0$, (б) $t = 1$, (в) $t = 2$.

Подробная схема работы алгоритма Бурахановой и Золкина, построенная на основе авторского исходного кода на языке C++, приведена ниже, в алгоритме 1.2.1.

Алгоритм 1.2.1 [37]

Входные данные:

1. $F(n)$ – функция, возвращающая n-ую точку цифровой (t, s)-последовательности $\{x^{(n)}\}$;
2. s – размерность пространства;
3. b – основание $\{x^{(n)}\}$;
4. $\widehat{m}$ – показатель периода $\{x^{(n)}\}$;
5. $\hat{t}$ – проверяемое значение параметра t.

Последовательность действий:

1. для всех $\mathcal{D} \in \{\mathcal{D} \in \mathbb{N}_0^s | \sum_{i=1}^{s} d_i = \widehat{m} - \hat{t}\}$;
   1.1. для всех $\mathcal{A} \in \{\mathcal{A} \in \mathbb{N}_0^s | \forall i \in [1..s]\ a_i < b^{d_i}\}$;
      1.1.1. $c \leftarrow 0$;
      1.1.2. для всех $n \in [0..b^{\widehat{m}} - 1]$;
         1.1.2.1. $x \leftarrow F(n)$;
         1.1.2.2. если для всех допустимых i верно $\frac{a_i}{b^{d_i}} < x_i \leq \leq \frac{a_i+1}{b^{d_i}}$, то $c \leftarrow c + 1$;
      1.1.3. если $c \neq b^{\hat{t}}$, то $\{x^{(n)}\}$ не является $(\hat{t}, s)$-последовательностью, алгоритм досрочно завершён;
2. $\{x^{(n)}\}$ является $(\hat{t}, s)$-последовательностью.

Преимущество представленного подхода, несомненно, заключается также и в однозначном установлении (t, m, s)-сети, что делает тест Бурахановой и Золкина полезным и с позиции проверки корректности работы самого генератора точек. Тем не менее, дальнейший анализ показывает, что представленная авторами программа имеет очень высокую вычислительную сложность. Так, например, можно заметить, что в ней присутствует полный перебор всех элементарных интервалов, для каждого из которых раз за разом

генерируется один и тот же участок последовательности, что с ростом значений $\hat{m}$ и s приводит к крайне большим затратам по времени.

Помимо норм неравномерности и параметра t существует ещё одно свойство, которое упоминалось в работах Джо и Куо [33], а также в ранних трудах Морокова и Кафлиша [38] и которое может отражать однородность точек, а именно попарная корреляция их компонент. Так, например, Джо и Куо показывают, что несмотря на формальное следование определению 1.1.2 (стр. 9), точки последовательностей могут быть сильно коррелированны по парам отдельных координат, что показано на рисунке 1.2.2.

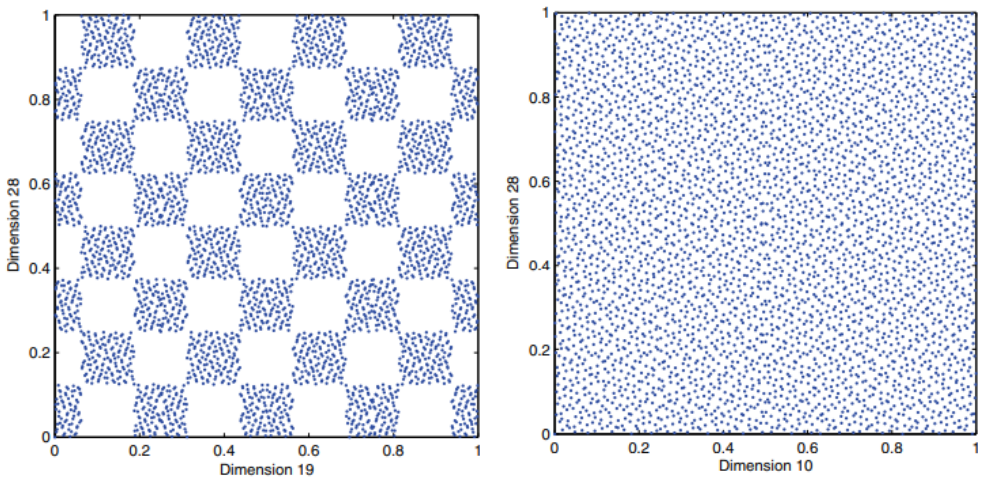


Рисунок 1.2.2. Построенный Джо и Куо пример $(t, 12, s)$-сети с явно коррелированными координатами $x_{19}^{(n)}$ и $x_{28}^{(n)}$ и с отсутствием очевидной корреляции между $x_{10}^{(n)}$ и $x_{28}^{(n)}$ [33].

Результат с иллюстрации выше был получен учёными с помощью цифровой (t, s)-последовательности Соболя, элементы которой выстраиваются в отдельных проекциях в заметный невооружённым глазом шаблон, что не соответствует представлениям об идеально равномерно заполняющем пространство множестве точек. В настоящее время многими учёными продолжается изучение попарных корреляций координат других частных случаев цифровых (t, s)-последовательностей, изобретаются способы их ликвидации или уменьшения [39, 40]. Тем не менее,

небезосновательно считать, что корреляционные свойства могут содержать и более общие сведения об однородности.

1.3. Выводы

1. Цифровые (t, s)-последовательности являются доминирующим инструментом решения прикладных задач квази-Монте-Карло.
2. Научное сообщество демонстрирует высокий интерес к исследованию равномерности заполнения куба $\mathcal{I}^s$ точками (t, s)-последовательностей. Вместе с этим, имеющиеся знания о самом популярном способе определения однородности – мерах неравномерности – либо имеют исключительно теоретическое значение (предельные оценки), либо дают приближённые ответы на вопрос (верхние границы диапазонов значений), либо решают задачу в частных случаях. Данные обстоятельства позволяют сделать заключение о целесообразности применения на практике алгоритмов, верифицирующих альтернативные свойства цифровых последовательностей, которые тем или иным образом помогут оценить их однородность.
3. Упомянутые работы Нидеррайтера, Джо и Куо говорят о возможности использования для данных целей параметра t. Результаты, достигнутые Бурахановой и Золкиным, позволяют предположить наличие модификаций, благодаря которым алгоритм верификации параметра t может стать гораздо более оптимальным по временным затратам.
4. Многочисленные работы, исследующие попарные корреляции координат точек, возможно развить и рассмотреть вопрос несколько шире путём изучения и выявления корреляций во всём s-мерном пространстве.

2. Верификация параметра t

2.1. Алгоритм

В настоящей работе предлагается новый алгоритм верификации параметра t цифровой (t, s)-последовательности с основанием b и периодом $b^{\widehat{m}}$ в условиях неизвестности её генерирующих матриц путём верификации параметра t порождённой ей $(t, \widehat{m}, s)$-сети.

Рассматриваемая задача формулируется следующим образом. Пусть известна цифровая (t, s)-последовательность $\{x^{(n)}\}$ с основанием b и периодом $b^{\widehat{m}}$, а также число $\hat{t} \in \mathbb{N}_0$ такое, что $\hat{t} \leq \widehat{m}$. Необходимо выяснить, является ли множество $\mathcal{P} = \left\{x^{(0)},\ x^{(1)},\ \ldots,\ x^{(b^{\widehat{m}}-1)}\right\}$ $(\hat{t}, \widehat{m}, s)$-сетью с основанием b.

Из определения 1.1.2 (стр. 9) известно, что каждый элементарный интервал $\mathcal{E}$ с мерой Лебега $V_s(\mathcal{E}) = b^{\hat{t}-\widehat{m}}$ должен содержать в себе $b^{\hat{t}}$ точек $\mathcal{P}$, из чего непосредственно следует перечень задач, которые необходимо выполнить алгоритму:

1. Посчитать число точек внутри каждого допустимого элементарного интервала;
2. Проверить для каждого допустимого элементарного интервала равенство числа точек, попавших внутрь него, с контрольным значением $b^{\hat{t}}$.

Метод, предложенный Бурахановой и Золкиным, является буквальной реализацией данной схемы, осуществляющей последовательный перебор всех элементарных интервалов и всех $b^{\widehat{m}}$ точек для каждого из них. В данной работе предлагается альтернативный алгоритм решения поставленной задачи, для построения математического фундамента которого были сформулированы и доказаны нижеописанные теоретические выводы.

Утверждение 2.1.1

Пусть $b \in \mathbb{N} \setminus \{1\}$, $\mathcal{D} = (d_1, d_2, \dots, d_s) \in \mathbb{N}_0^s$ и $x \in \mathcal{I}^s$. Тогда $\mathcal{A} = A(x, b, \mathcal{D}) = (a_1, a_2, \dots, a_s) \in \mathbb{N}_0^s$, где $a_i = \lfloor x_i \cdot b^{d_i} \rfloor$ для всех $i \in [1..s]$, является таким множеством, что $x \in E(b, \mathcal{A}, \mathcal{D})$.

Доказательство 2.1.1

Очевидно, что $x \in E(b, \mathcal{A}, \mathcal{D}) \Leftrightarrow x_i \in \left[\frac{a_i}{b^{d_i}}, \frac{a_i+1}{b^{d_i}}\right)$ при всех $i \in [1..s]$, или, иначе, $\frac{a_i}{b^{d_i}} \le x_i < \frac{a_i+1}{b^{d_i}}$ при всех $i \in [1..s]$. Покажем, что $\frac{\lfloor x_i \cdot b^{d_i} \rfloor}{b^{d_i}} \le x_i < \frac{\lfloor x_i \cdot b^{d_i} \rfloor + 1}{b^{d_i}}$.

Так как $b^{d_i} > 0$, из того, что $\frac{x_i \cdot b^{d_i}}{b^{d_i}} = x_i$ и $\lfloor x_i \cdot b^{d_i} \rfloor \le x_i \cdot b^{d_i}$, немедленно следует то, что $\frac{\lfloor x_i \cdot b^{d_i} \rfloor}{b^{d_i}} \le x_i$. Используя аналогичные рассуждения, можно показать, что $x_i < \frac{x_i \cdot b^{d_i}+1}{b^{d_i}} \le \frac{\lfloor x_i \cdot b^{d_i}+1 \rfloor}{b^{d_i}} = \frac{\lfloor x_i \cdot b^{d_i} \rfloor + 1}{b^{d_i}}$. Таким образом, утверждение доказано.

Утверждение 2.1.2

В $\mathcal{I}^s$ число всех элементарных интервалов с s-мерной мерой Лебега b^{t-m} равно

$$C(t, m, s, b) = \binom{m - t + s - 1}{s - 1} \cdot b^{m-t}. \qquad (2.1.1)$$

Доказательство 2.1.2

Из определения 1.1.1 (стр. 9) видно, что для s-мерного элементарного интервала $\mathcal{E} = E(b, \mathcal{A}, \mathcal{D})$ верно, что $V_s(\mathcal{E}) = \prod_{i=1}^{s} b^{-d_i} = b^{-d_1 - d_2 - \cdots - d_s} = b^{-\sum_{i=1}^{s} d_i}$. Отсюда, $V_s(\mathcal{E}) = b^{t-m} \Leftrightarrow \sum_{i=1}^{s} d_i = m - t$. Рассмотрим некоторое $\mathcal{D}$ такое, что $d_1 + d_2 + \cdots + d_s = m - t$. Найдём число всех элементарных интервалов с данными b и $\mathcal{D}$, или, эквивалентно, число всех допустимых $\mathcal{A}$. Из определения 1.1.1 (стр. 9) известно, что $a_i < b^{d_i}$ для всех i, следовательно, число допустимых a_i

равно b^{d_i}. Отсюда, число всех допустимых $\mathcal{A}$ равно $b^{d_1} \cdot b^{d_2} \cdot \ldots \cdot b^{d_s} = b^{d_1+d_2+\cdots+d_s} = b^{m-t}$.

Теперь найдём число множеств $\mathcal{D}$ таких, что $d_1 + d_2 + \cdots + d_s = m - t$. Для решения этой подзадачи можно воспользоваться комбинаторным методом перегородок [41], суть которого заключается в следующем. Пусть имеется $N + M$ «фишек» $\mathcal{P} = \{p_1, p_2, \ldots, p_{N+M}\}$. Выбирая среди них любые M «фишек» $\mathcal{B} = \{p_{r_1}, p_{r_2}, \ldots, p_{r_M}\}$, где при $x < y$ верно $r_x < r_y$, и назначая их «перегородками», представляется возможным построить $\{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_{M+1}\}$ – разбиение множества $\mathcal{P} \setminus \mathcal{B}$, где $\mathcal{S}_1 = \{p_1, p_2, \ldots, p_{r_1-1}\}$, $\mathcal{S}_2 = \{p_{r_1+1}, p_{r_1+2}, \ldots, p_{r_2-1}\}$, …, $\mathcal{S}_{M+1} = \{p_{r_{M-1}+1}, p_{r_{M-1}+2}, \ldots, p_{r_M-1}\}$. Сумма мощностей всех множеств такого разбиения $|\mathcal{S}_1| + |\mathcal{S}_2| + \cdots + |\mathcal{S}_{M+1}| = |\mathcal{P} \setminus \mathcal{B}| = N$, следовательно, перебрав все возможные способы выбрать из $N + M$ «фишек» M «перегородок», получается количество способов представить число N в виде суммы $M + 1$ неотрицательного числа: $N_1 + N_2 + \cdots + N_{M+1} = N$, где $N_k = |\mathcal{S}_k|$ для всех $k \in [1..M + 1]$. Число способов выбрать из $N + M$ «фишек» M «перегородок» определяется количеством сочетаний

$$\binom{N+M}{M}.$$

В рассматриваемом случае $N = m - t$, $M = s - 1$.

Таким образом, перемножив число допустимых множеств $\mathcal{D}$ с числом допустимых множеств $\mathcal{A}$, получаем формулу (2.1.1). Утверждение доказано.

Из утверждения 1.1.1 (стр. 9) следует, что построенное в утверждении 2.1.1 множество $\mathcal{A}$ задаёт единственный элементарный интервал при данных b и $\mathcal{D}$, которому принадлежит данная x. Также важно отметить, что в случае, когда рассматриваемая точка является членом $x^{(n)}$ некоторой цифровой

(t, s)-последовательности с основанием b и периодом $b^{\hat{m}}$, приведённую формулу для a_i можно модифицировать так, чтобы все вычисления стали целочисленными. Для этого достаточно заметить, что из формулы (1.1.1) (стр. 11) следует то, что $X^{(n)} = b^{\hat{m}} \cdot x^{(n)} \in \mathbb{N}_0^s$. Отсюда получается, что $a_i = \left\lfloor x_i^{(n)} \cdot b^{d_i} \right\rfloor = \left\lfloor b^{\hat{m}} \cdot x_i^{(n)} \cdot b^{d_i} \cdot b^{-\hat{m}} \right\rfloor = \left\lfloor X_i^{(n)} \cdot b^{-(\hat{m}-d_i)} \right\rfloor$. Зная, что $d_1 + d_2 + \cdots + d_s = \hat{m} - \hat{t}$, можно заключить, что каждая $d_i \leq \hat{m}$, следовательно, $b^{\hat{m}-d_i} \in \mathbb{N}$ и выражение для a_i представляется в виде $a_i = X_i^{(n)} \operatorname{div} b^{\hat{m}-d_i}$. Использование данной формулы в случае цифровых последовательностей предпочтительнее, так как целочисленные вычисления полностью лишают программу ошибок округления, возникающих при работе с дробными числами.

Что касается подсчёта числа точек внутри каждого элементарного интервала, то для этой задачи, в соответствии с утверждением 2.1.2, достаточно использовать множество переменных-счётчиков $\mathcal{C} = \{c_0, c_1, \dots, c_{C(\hat{t},\hat{m},s,b)-1}\}$, где каждому элементарному интервалу $\mathcal{E} = E(b, \mathcal{A}, \mathcal{D})$ такому, что $V_s(\mathcal{E}) = b^{\hat{t}-\hat{m}}$, соответствует переменная-счётчик с индексом $\mu(b, \mathcal{A}, \mathcal{D})$, где μ – это заранее заданная биекция, сопоставляющая каждому допустимому интервалу целое число от 0 до $C(\hat{t}, \hat{m}, s, b) - 1$.

Таким образом, алгоритм можно представить в следующей форме.

Алгоритм 2.1.1

Входные данные:

1. $F(n)$ – функция, возвращающая n-ую точку цифровой (t, s)-последовательности $\{x^{(n)}\}$;
2. s – размерность пространства;
3. b – основание $\{x^{(n)}\}$;
4. $\hat{m}$ – показатель периода $\{x^{(n)}\}$;
5. $\hat{t}$ – проверяемое значение параметра t.

Последовательность действий:

1. для всех $n \in [0..b^{\hat{m}} - 1]$;
   1.1. $x \leftarrow F(n)$;
   1.2. для всех $\mathcal{D} \in \{\mathcal{D} \in \mathbb{N}_0^s | \sum_{i=1}^{s} d_i = \hat{m} - \hat{t}\}$;
      1.2.1. $\mathcal{A} \leftarrow A(x, b, \mathcal{D})$;
      1.2.2. $c_{\mu(b,\mathcal{A},\mathcal{D})} \leftarrow c_{\mu(b,\mathcal{A},\mathcal{D})} + 1$;
2. если $\forall k \in [0..C(\hat{t}, \hat{m}, s, b) - 1]$ верно то, что $c_k = b^{\hat{t}}$, то $\{x^{(n)}\}$ является $(\hat{t}, s)$-последовательностью, иначе – не является.

Нетрудно видеть, что данный алгоритм также возможно использовать для однозначного установления (t, m, s)-сети.

2.2. Тестирование

В качестве объекта тестирования выступал генератор цифровых последовательностей Нидеррайтера с основанием 2, находящийся в свободном доступе [42]. Реализация алгоритма 2.1.1 на языке программирования C++ приведена в приложении А. Программа включает в себя конструкции явной оптимизации количества потребляемых ресурсов памяти, в частности, побитовую упаковку счётчиков $\mathcal{C}$, что позволяет максимально расширить пределы применимости кода. Для получения всех возможных множеств $\mathcal{D}$ была реализована функция, генерирующая каждое последующее $\mathcal{D}$ на основе известного предыдущего. Каждому $\mathcal{D}$ сопоставляется порядковый индекс $\nu(\mathcal{D})$. Функция μ, в свою очередь, реализована следующим образом:

$$\mu(b, \mathcal{A}, \mathcal{D}) = \nu(\mathcal{D}) \cdot b^{\hat{m}-\hat{t}} + \sum_{i=1}^{s} a_i \cdot b^{\sum_{j=i+1}^{s} d_j},$$

где в данном случае $b = 2$.

Ниже, в таблице 2.2.1, приведены результаты тестирования алгоритма 2.1.1 и алгоритма, предложенного Бурахановой и Золкиным. В таблице использованы следующие символьные обозначения:

- $T_{2.1.1}$ – время работы алгоритма 2.1.1 в секундах;
- $T_{1.2.1}$ – время работы алгоритма Бурахановой и Золкина в секундах.

Контрольные величины $\hat{t}$ взяты из работы Нидеррайтера [16]. Тестирование осуществлялось на процессоре Intel® Core™ i7-8700 @ 3.20 GHz, значение «> 86400» означает, что за 24 часа алгоритм не завершил свою работу.

Таблица 2.2.1. Результаты тестирования

s	$\hat{t}$	$\hat{m}$	$T_{2.1.1}$	$T_{1.2.1}$
2	0	10	0.015	5.753
		15	0.039	2807.748
		20	0.891	> 86400
5	5	10	0.009	3.114
		15	2.392	6456.463
		20	549.663	> 86400
7	11	15	0.537	25.160
		20	418.502	> 86400
9	18	20	4.383	45.501
		25	19620.925	> 86400
10	22	25	699.654	> 86400

Как видно, алгоритм 2.1.1 во всех тестовых ситуациях многократно превосходит аналог по времени работы. Наименьшая разница во времени установлена в девятимерном случае (приблизительно в 10 раз быстрее), что возможно объяснить близкими значениями $\hat{t}$ и $\hat{m}$, благодаря чему число элементарных интервалов становится относительно мало и их полный перебор происходит сравнительно быстро.

Дополнительное профилирование программы показало, что наибольшую долю времени работы занимала генерация очередной точки последовательности, производившаяся с помощью внутренних средств генератора и не зависящая от реализации самого алгоритма.

2.3. Выводы

1. Представленный в работе алгоритм 2.1.1, опирающийся на сформулированные и доказанные утверждения, демонстрирует меньшую эффективность по памяти, чем известный аналог [37], однако, вместе с этим, несопоставимо большую эффективность по времени (в 10 и более раз быстрее). Необходимость постоянного хранения информации о каждом элементарном интервале компенсируется значительным превосходством в скорости производимых расчётов.
2. Задача верификации параметра t в условии неизвестности генерирующих матриц в результате проделанной работы серьёзно упростилась, однако всё ещё остаётся вычислительно сложной. Объёмы времени и ресурсов, затрачиваемых на её решение, демонстрируют быстрый рост с увеличением размерности, параметра $\hat{m}$ и разницы параметров $\hat{m}$ и $\hat{t}$.

3. Верификация корреляционных свойств

3.1. Алгоритм

В настоящей работе также рассматривается вопрос верификации корреляционных свойств точек цифровой (t, s)-последовательности с основанием b и периодом $b^{\widehat{m}}$. С этой целью предлагается использовать метод главных компонент (principal component analysis).

Метод главных компонент даёт возможность найти для некоторого s-мерного множества точек так называемую систему главных координат (principal coordinate system) – ортонормированную систему координат $(p_1, p_2, \dots, p_s)$, в которой дисперсия точек вдоль p_x больше, чем дисперсия точек вдоль p_y, если $x < y$, а попарные корреляции разных компонент сведены к нулю. С математической точки зрения, метод главных компонент для центрированного набора точек $\mathcal{X} = \{x_1, x_2, \dots, x_N\} \subset \mathbb{R}^s$ представляет собой спектральное разложение его ковариационной матрицы $\mathfrak{C} = \mathfrak{X}^T\mathfrak{X}$, где $\mathfrak{X} \in \mathbb{R}^{N \times s}$ и каждая i-ая строка $\mathfrak{X}$ представляет собой компоненты точки x_i в некоторой исходной системе координат. Полученные собственные векторы w_i описывают систему главных координат, а соответствующие им собственные значения λ_i – доли дисперсии, приходящиеся на них [43].

Отсюда, рассматриваемая задача формулируется следующим образом. Пусть известна цифровая (t, s)-последовательность $\{x^{(n)}\}$ с основанием b и периодом $b^{\widehat{m}}$. Необходимо для построенной с её помощью $(t, \widehat{m}, s)$-сети найти систему главных координат и долю общей дисперсии, приходящуюся на каждую из них.

На данный момент существуют различные подходы к решению таких задач. Классические алгоритмы метода главных компонент, дающие наиболее точные ответы, предполагают хранение в памяти всего массива исследуемых точек, что в данном случае с ростом s и $\widehat{m}$ может стать

ощутимой проблемой. Для её решения обычно предлагается использовать так называемый инкрементный алгоритм (incremental principal component analysis algorithm), который производит исследование генеральной совокупности по частям и не требует постоянного хранения её в памяти. С другой стороны, данный подход приводит к получению в качестве ответа приближённых значений, погрешность которых обуславливается мощностью генеральной совокупности и параметром, обычно именуемым «forgetting factor» или «forgetting horizon». Его значение устанавливает то, насколько сильное влияние на текущий результат оказывают точки, учтённые алгоритмом до этого, что позволяет освобождать память, храня в ней только относительно недавно рассмотренные элементы [44].

В данной работе предлагается модификация этого алгоритма, которая не предполагает хранения в памяти какого-либо массива точек и даёт корректные ответы для цифровых (t, s)-последовательностей, что достигается с помощью нижеследующего априорного анализа.

Утверждение 3.1.1 [45]

(t, s)-последовательности определяют равномерно распределённые в $\mathcal{I}^s$ точки.

Для равномерного распределения известно математическое ожидание. В случае распределения по $\mathcal{I}^s$ оно выражается как вектор $e = (1/2\ \ 1/2\ \cdots\ 1/2)^T \in \mathcal{I}^s$, из чего следует, что сама (t, s)-последовательность не является центрированным набором точек, однако благодаря утверждению 3.1.1 центрирование $(t, \widehat{m}, s)$-сети возможно произвести без явного вычисления эмпирического среднего, путём вычитания заранее известного математического ожидания e из каждой точки. Опираясь на это, представляется возможным построить с помощью однопроходного инкрементного метода точную ковариационную матрицу, а не её приближение.

Утверждение 3.1.2

Пусть $\{x^{(n)}\}$ – (t,s)-последовательность, $\mathfrak{C}_n = \mathfrak{X}_n^T \mathfrak{X}_n$ – ковариационная матрица для первых n её точек. Тогда для $n+1$ первых точек верно:

$$\mathfrak{C}_{n+1} = \mathfrak{C}_n + \left(x^{(n+1)} - e\right)\left(x^{(n+1)} - e\right)^T.$$

Доказательство 3.1.2

$$\mathfrak{C}_{n+1} = \mathfrak{X}_{n+1}^T \mathfrak{X}_{n+1} =$$

$$= \begin{pmatrix} & \begin{vmatrix} x_1^{(n+1)} - 1/2 \\ x_2^{(n+1)} - 1/2 \\ x_3^{(n+1)} - 1/2 \\ \vdots \\ x_s^{(n+1)} - 1/2 \end{vmatrix} \end{pmatrix} \begin{pmatrix} \mathfrak{X}_n \\ \hline x_1^{(n+1)} - \frac{1}{2} \quad x_2^{(n+1)} - \frac{1}{2} \quad \dots \quad x_s^{(n+1)} - \frac{1}{2} \end{pmatrix} =$$

$$= \begin{pmatrix} \{\mathfrak{C}_n\}_{1,1} + \left(x_1^{(n+1)} - \frac{1}{2}\right)\left(x_1^{(n+1)} - \frac{1}{2}\right) & \cdots & \{\mathfrak{C}_n\}_{1,s} + \left(x_1^{(n+1)} - \frac{1}{2}\right)\left(x_s^{(n+1)} - \frac{1}{2}\right) \\ \vdots & \ddots & \vdots \\ \{\mathfrak{C}_n\}_{s,1} + \left(x_s^{(n+1)} - \frac{1}{2}\right)\left(x_1^{(n+1)} - \frac{1}{2}\right) & \cdots & \{\mathfrak{C}_n\}_{s,s} + \left(x_s^{(n+1)} - \frac{1}{2}\right)\left(x_s^{(n+1)} - \frac{1}{2}\right) \end{pmatrix} =$$

$= \mathfrak{C}_n + \left(x^{(n+1)} - e\right)\left(x^{(n+1)} - e\right)^T$. Что и требовалось доказать.

Рассчитав с помощью утверждения 3.1.2 ковариационную матрицу $\mathfrak{C} = \mathfrak{C}_{b^{\widehat{m}}}$ для первых $b^{\widehat{m}}$ точек данной (t,s)-последовательности, с целью получения окончательного ответа достаточно найти её спектральное разложение, применив соответствующий метод.

Таким образом, модификацию инкрементного алгоритма можно представить в следующей форме.

Алгоритм 3.1.1

Входные данные:

1. $F(n)$ – функция, возвращающая n-ую точку цифровой (t,s)-последовательности $\{x^{(n)}\}$;
2. s – размерность пространства;
3. b – основание $\{x^{(n)}\}$;

4. $\widehat{m}$ – показатель периода $\{x^{(n)}\}$.

Последовательность действий:

1. для всех $n \in [0..b^{\widehat{m}} - 1]$;
    1.1. $x \leftarrow F(n) - e$;
    1.2. $\mathfrak{C} \leftarrow \mathfrak{C} + xx^T$;
2. найти спектральное разложение $\mathfrak{C}$, в котором собственные векторы будут определять искомую систему координат, а соответствующие им собственные значения – приходящиеся на них доли дисперсии.

При использовании представленного метода логично ожидать незначительные расхождения с классическими алгоритмами, обусловленные тем, что классические алгоритмы разработаны для общего случая, в которых никакой информации о распределении точек не известно. Применение математического ожидания вместо эмпирического среднего при центрировании данных позволяет сделать результаты вычислений сопоставимыми друг с другом, так как начало главных координат во всех случаях будет располагаться в одном и том же месте – в точке $O + e$, где O – начало исходной системы координат.

3.2. Тестирование

В качестве объекта тестирования выступал генератор цифровых последовательностей Нидеррайтера с основанием 2, находящийся в свободном доступе [42]. Реализация алгоритма 3.1.1 на языке программирования C++ приведена в приложении Б. Методом расчёта спектрального разложения выступил метод Якоби, реализация которого была взята из открытого источника [46].

В рамках тестирования произведена проверка того, насколько сильно отличаются друг от друга выходные данные, полученные алгоритмом 3.1.1 и

классическим алгоритмом анализа главных компонент. Реализация классического алгоритма была взята из открытого источника [47].

Ниже, в таблице 3.2.1, представлены результаты тестирования. В таблице использованы следующие символьные обозначения:

- $p_{3.1.1}$ – главные оси, найденные алгоритмом 3.1.1;
- $Var_{3.1.1}$ – доли дисперсии, найденные алгоритмом 3.1.1;
- Δ_p – евклидово расстояние между главными осями, найденными с помощью алгоритма 3.1.1 и с помощью классического алгоритма;
- Δ_{Var} – абсолютная разница между долями дисперсии, найденными с помощью алгоритма 3.1.1 и с помощью классического алгоритма;
- $\times$ – ресурсов памяти оказалось недостаточно для запуска классического алгоритма.

Тестирование осуществлялось на процессоре Intel® Core™ i7-8700 @ 3.20 GHz.

Таблица 3.2.1. Результаты тестирования

$s = 2$	$\widehat{m} = 10$		$\widehat{m} = 15$		$\widehat{m} = 20$	
	Ось 1	Ось 2	Ось 1	Ось 2	Ось 1	Ось 2
$p_{3.1.1}$	0.707107	0.707107	0.707107	0.707107	0.707107	0.707107
	0.707107	-0.707107	0.707107	-0.707107	0.707107	-0.707107
$Var_{3.1.1}$	0.500026	0.499974	0.500023	0.499977	0.5	0.5
Δ_p	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$
Δ_{Var}	0.000001	0.000001	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$

$s = 5$	$\widehat{m} = 15$					$\widehat{m} = 20$				
	Ось 1	Ось 2	Ось 3	Ось 4	Ось 5	Ось 1	Ось 2	Ось 3	Ось 4	Ось 5
$p_{3.1.1}$	0.316452	0.000305	-0.246480	0.900747	-0.166617	0.480782	-0.070865	-0.530937	-0.192232	0.667068
	0.592010	0.006688	0.657009	0.057469	0.463160	0.583792	-0.008051	0.000714	-0.564314	-0.583670
	0.741177	0.002843	-0.420609	-0.430490	-0.297347	0.620096	0.011067	0.135216	0.763633	-0.118070
	0.003422	-0.708509	-0.403580	-0.004318	0.578885	-0.125340	0.695944	-0.625953	0.157936	-0.288429
	-0.005298	0.705664	-0.409632	-0.003534	0.578099	0.166752	0.714460	0.554982	-0.191097	0.342371
$Var_{3.1.1}$	0.200028	0.200025	0.20002	0.200001	0.199926	0.2	0.2	0.2	0.2	0.2
Δ_p	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$
Δ_{Var}	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$	$< 10^{-6}$

Продолжение таблицы 3.2.1

$s = 10$	$\widehat{m} = 25$									
	Ось 1	Ось 2	Ось 3	Ось 4	Ось 5	Ось 6	Ось 7	Ось 8	Ось 9	Ось 10
$p_{3.1.1}$	0.090278	-0.18758	-0.000404	-0.000198	0.000482	0.001702	-0.034426	-0.567687	0.741021	-0.28999
	0.075511	-0.226593	-0.001403	-0.000513	0.000535	0.002464	0.027146	0.679194	0.594645	0.356793
	-0.181228	0.704059	0.208305	0.020144	0.180989	-0.14839	-0.16006	-0.204679	0.232576	0.501278
	0.07337	0.072508	0.456796	-0.20854	0.713507	-0.038999	0.127237	0.2104	0.016959	-0.407247
	0.021577	0.233133	0.62347	-0.135632	-0.648678	0.171558	0.074519	0.137801	0.071314	-0.241307
	-0.013913	0.334188	-0.346997	-0.101079	0.106789	0.845152	0.035516	0.080512	0.101457	-0.117381
	0.035506	0.463254	-0.47729	-0.083493	-0.150431	-0.477842	0.136324	0.25392	0.141801	-0.441847
	0.703759	0.093453	-0.000304	-0.004719	-0.0004	-0.000838	-0.697187	0.060954	-0.065055	-0.044157
	0.033053	0.110556	0.104959	0.959398	0.05774	0.066337	0.058794	0.099394	0.03171	-0.182064
	0.670547	0.104494	-0.000315	-0.007304	-0.000652	-0.001417	0.664337	-0.163476	-0.046711	0.262948
$Var_{3.1.1}$	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
Δ_p	×	×	×	×	×	×	×	×	×	×
Δ_{Var}	×	×	×	×	×	×	×	×	×	×

Как показывают результаты экспериментального тестирования, алгоритм 3.1.1 по всем рассмотренным показателям демонстрирует незначительные отклонения от классического алгоритма, которые крайне быстро уменьшаются с увеличением числа точек в рассматриваемой сети. Также благодаря своей инкрементности предложенный алгоритм успешно справляется с многомерными задачами, предполагающими рассмотрение большого количества элементов.

Более детальные результаты для цифровых (t, s)-последовательностей Нидеррайтера с основанием 2 показаны на рисунках 3.2.1 и 3.2.2.

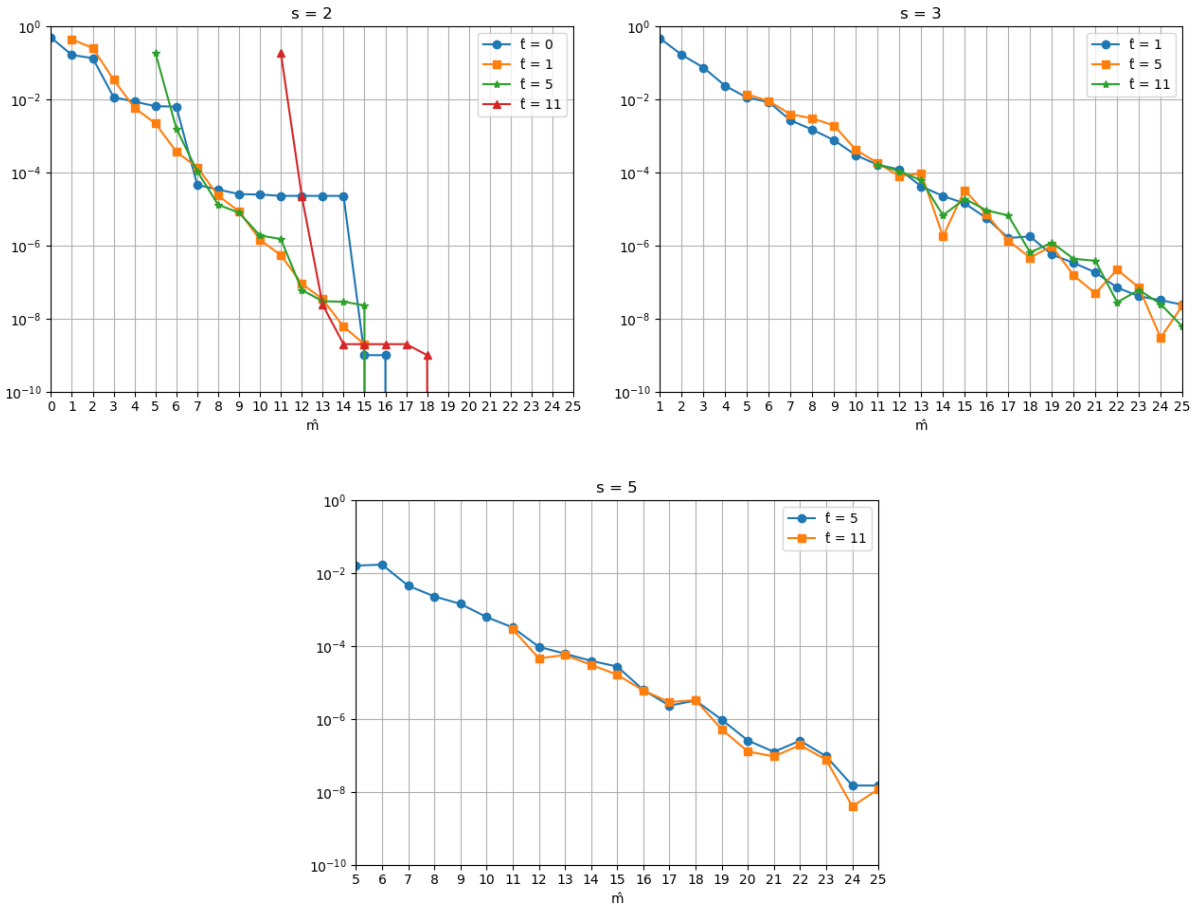


Рисунок 3.2.1. Значение модуля отклонения доли дисперсии, приходящейся на первую главную ось, от s^{-1} в зависимости от параметра $\hat{m}$ при фиксированном s и варьирующемся $\hat{t}$.

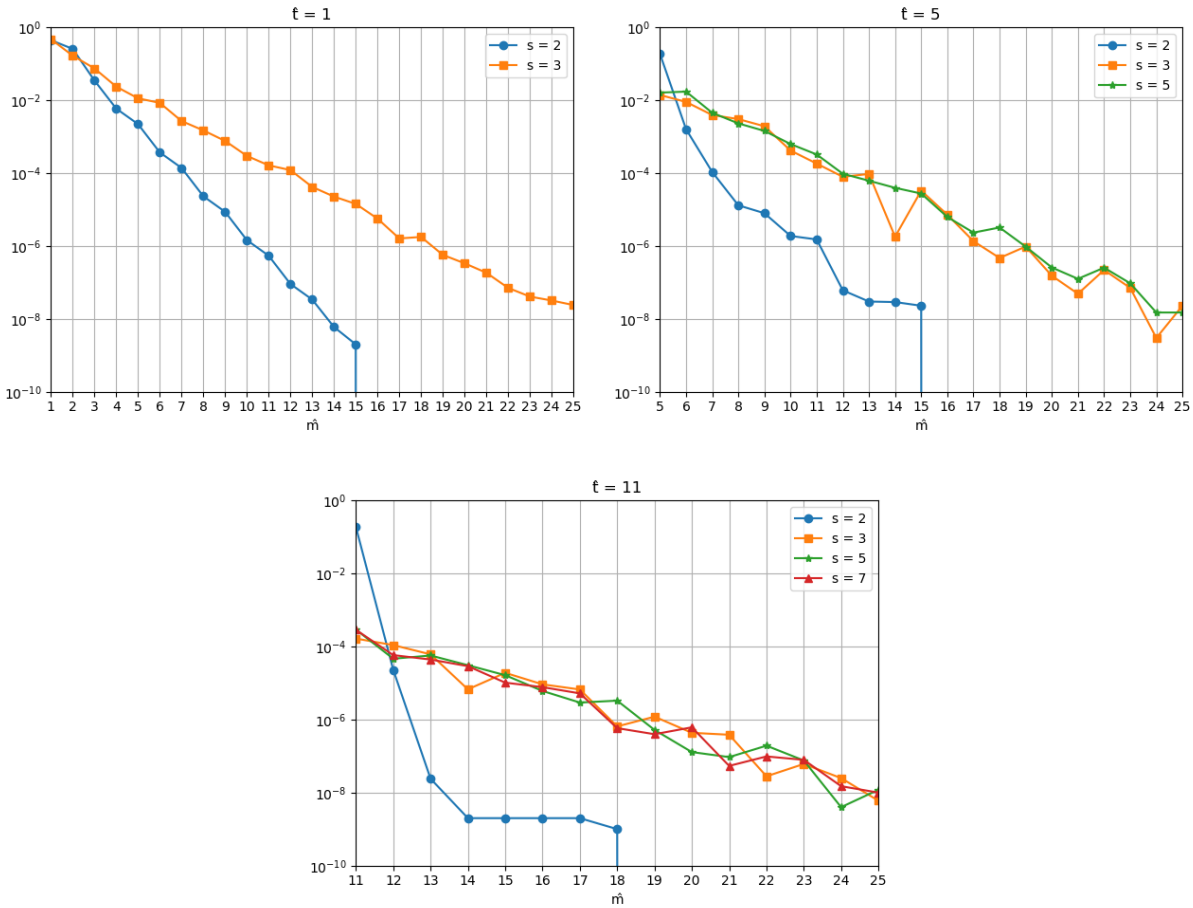


Рисунок 3.2.2. Значение модуля отклонения доли дисперсии, приходящейся на первую главную ось, от s^{-1} в зависимости от параметра $\hat{m}$ при фиксированном $\hat{t}$ и варьирующемся s.

На рисунках отчётливо видно экспоненциальное убывание модуля отклонения доли дисперсии, приходящейся на первую главную ось, от s^{-1} с ростом периода цифровой (t, s)-последовательности. Важно отметить, что характер данного убывания демонстрирует низкую зависимость от значений $\hat{t}$ и s: так, например, при $\hat{m} = 11$ почти во всех случаях невязка равна $\approx 10^{-4}$, кроме случая $s = 2$, который, будучи примечательным исключением из общей тенденции, демонстрирует различные значения для разных $\hat{t}$. Также, зная то, что доля дисперсии, приходящаяся на любую иную главную ось, не превосходит долю дисперсии, приходящуюся на первую ось, возможно обобщить данную закономерность на все оси главной системы координат.

3.3. Выводы

1. Представленная модификация инкрементного алгоритма метода главных компонент, основанная на сформулированных и доказанных в работе утверждениях, показала высокую точность вычислений (расхождение с классическим алгоритмом [47] во всех тестовых случаях не более, чем 10^{-6}) одновременно с низкими требованиями к ресурсам памяти.
2. Результаты верификации главных осей цифровых (t, s)-последовательностей Нидеррайтера с основанием 2 дают возможность предположить существование общего для них значения $\widehat{m}_0(\varepsilon)$, где $\varepsilon > 0$, такого, что для любого $\widehat{m} > \widehat{m}_0(\varepsilon)$ доля дисперсии, приходящаяся на каждую главную ось, отстоит от значения s^{-1} не более, чем на ε. Истинность данного тезиса позволит утверждать, что любому допускаемому отклонению доли дисперсии, приходящейся на каждую главную ось, от величины s^{-1} можно однозначно сопоставить минимальное $\widehat{m}$, при котором оно будет достигаться, причём независимо от $\hat{t}$ и s.

Заключение

Таким образом, в настоящей работе приведены постановки задач верификации параметра t и верификации главных осей цифровых (t, s)-последовательностей, сформулированы и доказаны теоретические утверждения, на основе которых были построены алгоритмы решения данных задач.

Как итог, объёмы времени, затрачиваемого на работу предложенного метода верификации параметра t, ниже, чем у аналога [37], минимум в 10 раз. Результаты описанного алгоритма верификации главных осей согласуются с результатами классического алгоритма [47] (расхождение во всех тестовых случаях не более, чем 10^{-6}), однако, вместе с тем, последний требует крупные объёмы памяти для своей работы, что становится препятствием для его использования при больших значениях $\widehat{m}$ и s.

Применение метода главных компонент к цифровым (t, s)-последовательностям Нидеррайтера с основанием 2 показало независимость многомерных корреляционных свойств их точек от параметров последовательностей. Данное поведение позволяет предположить существование общего для них значения $\widehat{m}_0(\varepsilon)$, где $\varepsilon > 0$, такого, что для любого $\widehat{m} > \widehat{m}_0(\varepsilon)$ доля дисперсии, приходящаяся на каждую главную ось, отстоит от значения s^{-1} не более, чем на ε.

Также важно отметить то, что представленные методы в силу их построения возможно использовать не только для цифровых (t, s)-последовательностей, но и для произвольных (t, m, s)-сетей, независимо от того, каким образом и с применением каких средств они были построены, что говорит об универсальности алгоритмов и их высокой практической значимости, обуславливаемой возможностью верифицировать свойства произвольной конечной сети.

Список использованных источников

1. Faure H., Kritzer P. New star discrepancy bounds for (t, m, s)-nets and (t, s)-sequences // Monatshefte für Mathematik. — 2013. — Vol. 172, No. 1. — P. 55-75.
2. Tezuka S. On the discrepancy of generalized Niederreiter sequences // Journal of Complexity. — 2013. — Vol. 29, No. 3-4. — P. 240-247.
3. Drmota M., Hofer R., Larcher G. On the discrepancy of Halton–Kronecker sequences // Number Theory–Diophantine Problems, Uniform Distribution and Applications. — Cham: Springer, 2017. — p. 219-226.
4. Faure H., Lemieux C. A review of discrepancy bounds for (t, s)- and (t, e, s)-sequences with numerical comparisons // Mathematics and Computers in Simulation. — 2017. — Vol. 135. — P. 63-71.
5. Pagés G. Numerical Probability. — Cham: Springer, 2018. — 579 p.
6. Lemieux C. Monte Carlo and Quasi-Monte Carlo Sampling. — NY: Springer, 2009. — 373 p.
7. Niederreiter H. Point sets and sequences with small discrepancy // Monatshefte für Mathematik. — 1987. — Vol. 104, No. 4. — P. 273-337.
8. Niederreiter H. Random Number Generation and Quasi-Monte Carlo Methods. — Philadelphia: SIAM, 1992. — 241 p.
9. Grünschloß L., Raab M., Keller A. Enumerating quasi-monte carlo point sequences in elementary intervals // Monte Carlo and Quasi-Monte Carlo Methods 2010. — 2012. — Vol. 23. — P. 399-408.
10. Niederreiter H., Pirsic G. Duality for digital nets and its applications // ACTA ARITHMETICA-WARSZAWA-. — 2001. — Vol. 97, No. 2. — P. 173-182.
11. Fedorov D.V. Correlated Gaussians and low-discrepancy sequences // Few-Body Systems. — 2019. — Vol. 60, No. 3. — P. 55.1-55.4.
12. Dick J., Pillichshammer F. Digital Nets and Sequences: Discrepancy Theory and Quasi–Monte Carlo Integration. — NY: Cambridge University Press, 2010. — 618 p.

13. Sobol I.M. Distribution of points in a cube and approximate evaluation of integrals // U.S.S.R Comput. Maths. Math. Phys. — 1967. — Vol. 7. — P. 784-802.
14. Faure H. Good permutations for extreme discrepancy // Journal of Number Theory. — 1992. — Vol. 42. — P. 45-56.
15. Faure H. Discrepance de suites associees a un systeme de numeration (en dimension s) // Acta Arithmetica. — 1982. — Vol. 42. — P. 337-351.
16. Niederreiter H. Low-Discrepancy and Low-Dispersion Sequences // Journal of Number Theory. — 1988. — Vol. 30. — P. 51-70.
17. Kritzinger R., Pillichshammer F. Digital nets in dimension two with the optimal order of L_p discrepancy // Journal de Théorie des Nombres de Bordeaux. — 2019. — Vol. 39. — P. 179-204.
18. Schlier C. Error trends in quasi-Monte Carlo integration // Computer physics communications. — 2004. — Vol. 159, No. 2. — P. 93-105.
19. Larcher G., Puchhammer F. An improved bound for the star discrepancy of sequences in the unit interval // Uniform distribution theory. — 2016. — Vol. 11, No. 1. — P. 1-14.
20. Pillichshammer F. Discrepancy of Digital Sequences: New Results on a Classical QMC Topic // Springer Proceedings in Mathematics & Statistics. — 2020. — Vol. 324. — P. 81-103.
21. De Marchi S., Elefante G. Quasi-Monte Carlo integration on manifolds with mapped low-discrepancy points and greedy minimal Riesz s-energy points // Applied Numerical Mathematics. — 2018. — Vol. 127. — P. 110-124.
22. Gille-Genest A. Low Discrepancy Sequences. URL: https://www.rocq.inria.fr/mathfi/Premia/free-version/doc/premia-doc/pdf_html/mc_quasi_doc/index.html (дата обращения: 27.04.2020).
23. Dalal I.L., Stefan D., Harwayne-Gidansky J. Low Discrepancy Sequences for Monte Carlo Simulations on Reconfigurable Platforms // Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors. 2-4 July 2008, Leuven, Belgium, p. 108-113.

24. Kritzinger R. An exact formula for the L_2 discrepancy of the symmetrized Hammersley point set // Mathematics and Computers in Simulation. — 2018. — Vol. 143. — P. 3-13.
25. Pillichshammer F. L_p discrepancy of generalized two-dimensional Hammersley point sets // Monatshefte für Mathematik. — 2009. — Vol. 158, No. 1. — P. 31-61.
26. Kritzinger R. L_p- and $S_{p,q}^r B$-discrepancy of the symmetrized van der Corput sequence and modified Hammersley point sets in arbitrary bases // Journal of Complexity. — 2016. — Vol. 33. — P. 145-168.
27. Pillichshammer F. On the discrepancy of (0, 1)-sequences // Journal of Number Theory. — 2004. — Vol. 104, No. 2. — P. 301-314.
28. Matoušek J. On the L_2-Discrepancy for Anchored Boxes // Journal of Complexity. — 1998. — Vol. 14, No. 4. — P. 527-556.
29. Kritzinger R. An exact formula for the L_2 discrepancy of the symmetrized Hammersley point set // Mathematics and Computers in Simulation. — 2018. — Vol. 143. — P. 3-13.
30. Kritzinger R., Kritzinger L. M. L_2 discrepancy of symmetrized generalized Hammersley point sets in base b // Journal of Number Theory. — 2016. — Vol. 166. — P. 250-275.
31. Vandewoestyne B. A Quasi Monte Carlo (QMC) library written in Fortran 95 as part of my PhD. URL: https://github.com/BartVandewoestyne/qmcpack (дата обращения: 23.04.2020).
32. Doerr C., Gnewuch M., Wahlström M. Calculation of Discrepancy Measures and Applications // A Panorama of Discrepancy Theory. — 2014. — Vol. 2107. — P. 621-678.
33. Joe S., Kuo F.Y. Constructing Sobol' sequences with better two-dimensional projections // SIAM Journal on Scientific Computing. — 2008. — Vol. 30, No. 5. — P. 2635-2654.

34. Niederreiter H., Xing C.P. Low-discrepancy sequences obtained from algebraic function fields over finite fields // Acta Arithmetica. — 1995. — Vol. 72. — P. 281-298.
35. Dick J., Matsumoto M. On the fast computation of the weight enumerator polynomial and the t value of digital nets over finite Abelian groups // SIAM Journal on Discrete Mathematics. — 2013. — Vol. 27. — P. 1335-1359.
36. Marion P., Godin M., L'Ecuyer P. An algorithm to compute the t-value of a digital net and of its projections. URL: https://arxiv.org/pdf/1910.02277.pdf (дата обращения: 15.05.2020).
37. Бураханова А., Золкин А. TMSParametersTest. URL: https://github.com/dmkz/minimization/blob/master/test-tms/TMSParametersTest.hpp (дата обращения: 28.04.2020).
38. Morokoff W.J., Caflisch R.E. Quasi-random sequences and their discrepancies // SIAM Journal on Scientific Computing. — 1994. — Vol. 15. — P. 1251-1279.
39. Decorrelation of low discrepancy sequences for progressive rendering // US Patent #10074212. 2016 / Waechter C., Binder N.
40. Mohammed N.A. Comparing Halton and Sobol Sequences in Integral Evaluation // ZANCO Journal of Pure and Applied Sciences. — Vol. 31, No. 1. — P. 32-39.
41. Feller W. An Introduction to Probability Theory and Its Applications. — NY: Wiley, 1950. — 461 p.
42. (t, s)-sequences generator / Генератор (t, s)-последовательностей. URL: https://github.com/jointpoints/tms-nets (дата обращения: 17.05.2020).
43. Wold S., Esbensen K., Geladi P. Principal Component Analysis // Chemometrics and Intelligent Laboratory Systems. — 1987. — Vol. 2. — P. 37-52.
44. Fujiwara T., Chou J.K., Shilpika S., et al. An Incremental Dimensionality Reduction Method for Visualizing Streaming Multidimensional Data // IEEE

Transactions on Visualization and Computer Graphics. — 2020. — Vol. 26, No. 1. — P. 418-428.

45. Tuffin B. On the use of low discrepancy sequences in Monte Carlo methods // Monte Carlo Methods and Applications. — 1996. — Vol. 2, No. 4. — P. 295-320.

46. a short, simple public-domain header-only C++ library for calculating eigenvalues and eigenvectors of real symmetric matrices. URL: https://github.com/jewettaij/jacobi_pd (дата обращения: 17.05.2020).

47. Principal Component Analysis. URL: https://github.com/scikit-learn/scikit-learn/blob/483cd3eaa/sklearn/decomposition/_pca.py (дата обращения: 17.05.2020).

Реализация алгоритма 2.1.1

Файл `util/common.hpp` (определяет макросы для вывода информации на разных уровнях вербозности и коды возврата):

```
/*!
 * \file common.hpp
 *
 * \author
 *   Andrew Yeliseyev (Russian Technological University, KMBO-03-16, Russia,
2020)
 */
#ifndef _COMMON_HPP_
#define _COMMON_HPP_



#include <cstdint>
#include <cstdio>
#include <functional>
#include <vector>
#include <time.h>



#ifndef TSTESTS_VERBOSITY_LEVEL
#  define TSTESTS_VERBOSITY_LEVEL          0
#endif // TSTESTS_VERBOSITY_LEVEL

#define LOGBLOCK(core)                        do {time(&_time); _ltime =
*localtime(&_time); core} while(0);
#define LOGTIMESTAMP                          _ltime.tm_mday, _ltime.tm_mon+1,
_ltime.tm_year+1900, _ltime.tm_hour, _ltime.tm_min, _ltime.tm_sec

#if TSTESTS_VERBOSITY_LEVEL > 0
#  define TSTESTS_TEST_FUNCTION_BEGIN(name, file)  char const *const
_test_name = #name; time_t _time; struct tm _ltime; clock_t _start_clock =
clock(); clock_t _finish_clock; FILE *_log = file;
#  define TSTESTS_TEST_FUNCTION_END               _finish_clock = clock();
fprintf(_log, "Test completed in %Lf seconds.\n\n", (long
double)(_finish_clock - _start_clock) / CLOCKS_PER_SEC);
#else
#  define TSTESTS_TEST_FUNCTION_BEGIN(name, file)  (void) 0;
#  define TSTESTS_TEST_FUNCTION_END               (void) 0;
#endif // TSTESTS_VERBOSITY_LEVEL 0

#if TSTESTS_VERBOSITY_LEVEL == 1
#  define LOG(message)                            "%s [%02d.%02d.%d
%02d:%02d:%02d] %s\n", _test_name, LOGTIMESTAMP, message
#  define PUSHLOG_1(message)                      LOGBLOCK(fprintf(_log,
LOG(message));)
#  define LOGF(format, ...)                       "%s [%02d.%02d.%d
%02d:%02d:%02d] " format "\n", _test_name, LOGTIMESTAMP, __VA_ARGS__
#  define PUSHLOGF_1(format, ...)                 LOGBLOCK(fprintf(_log,
LOGF(format, __VA_ARGS__));)
#else
#  define PUSHLOG_1(message)                      (void) 0;
#  define PUSHLOGF_1(format, ...)                 (void) 0;
#endif // TSTESTS_VERBOSITY_LEVEL 1
```

```
#if TSTESTS_VERBOSITY_LEVEL == 2
# define LOG(message)                                "%s [%02d.%02d.%d
%02d:%02d:%02d] %s\n", _test_name, LOGTIMESTAMP, message
# define PUSHLOG_2(message)                          LOGBLOCK(fprintf(_log,
LOG(message));)
# define LOGF(format, ...)                           "%s [%02d.%02d.%d
%02d:%02d:%02d] " format "\n", _test_name, LOGTIMESTAMP, __VA_ARGS__
# define PUSHLOGF_2(format, ...)                     LOGBLOCK(fprintf(_log,
LOGF(format, __VA_ARGS__));)
#else
# define PUSHLOG_2(message)                          (void) 0;
# define PUSHLOGF_2(format, ...)                     (void) 0;
#endif // TSTESTS_VERBOSITY_LEVEL 2

#if TSTESTS_VERBOSITY_LEVEL >= 3
# define LOG(delim, message)                         "%s%s%s\n", _test_name,
delim, message
# define LOGCONT(message)                            "\t%s\n", message
# define PUSHLOG_3(message)                          LOGBLOCK(fprintf(_log,
LOG("\n\t", message));)
# define APPENDLOG_3(message)                        LOGBLOCK(fprintf(_log,
LOGCONT(message));)
# define LOGF(delim, format, ...)                    "%s%s" format "\n",
_test_name, delim, __VA_ARGS__
# define LOGFCONT(format, ...)                       "\t" format "\n",
__VA_ARGS__
# define PUSHLOGF_3(format, ...)                     LOGBLOCK(fprintf(_log,
LOGF("\n\t", format, __VA_ARGS__));)
# define APPENDLOGF_3(format, ...)                   LOGBLOCK(fprintf(_log,
LOGFCONT(format, __VA_ARGS__));)
#else
# define PUSHLOG_3(message)                          (void) 0;
# define APPENDLOG_3(message)                        (void) 0;
# define PUSHLOGF_3(format, ...)                     (void) 0;
# define APPENDLOGF_3(format, ...)                   (void) 0;
#endif // TSTESTS_VERBOSITY_LEVEL 3

#if TSTESTS_VERBOSITY_LEVEL >= 4
# define PUSHLOG_4(message)                          LOGBLOCK(fprintf(_log,
LOGF("", " [%02d.%02d.%d %02d:%02d:%02d] %s", LOGTIMESTAMP, message));)
# define PUSHLOGF_4(format, ...)                     LOGBLOCK(fprintf(_log,
LOGF("", " [%02d.%02d.%d %02d:%02d:%02d] " format, LOGTIMESTAMP,
__VA_ARGS__));)
#else
# define PUSHLOG_4(message)                          (void) 0;
# define PUSHLOGF_4(format, ...)                     (void) 0;
#endif // TSTESTS_VERBOSITY_LEVEL 4



#define TSTESTS_COORDINATE_TYPE     long double
#define TSTESTS_DIGITAL_TYPE        uint64_t

#define TSTESTS_LOG_IN_CONSOLE      stdout



typedef struct TsTestsInfo
{
  uint8_t     t;
  uint8_t     m;
  uint8_t     s;
  uint8_t     bitwidth;
```

```
  std::function<std::vector<TSTESTS_COORDINATE_TYPE>(uint64_t const)>
next_point_getter;
  FILE       *log_file;
}
TsTestsInfo;



typedef enum TsTestsReturnCode
{
  TSTESTS_RETURNCODE_SUCCESS              =  0,
  TSTESTS_RETURNCODE_FAIL_GENERAL         = -1,
  TSTESTS_RETURNCODE_FAIL_INPUT           = -2,
  TSTESTS_RETURNCODE_FAIL_MEMORY          = -3
}
TsTestsReturnCode;

#define TSTESTS_PRINT_RETURNCODE(code)      do {
\
                                                char const *const ret[] =
{"TSTESTS_RETURNCODE_FAIL_MEMORY",            \

"TSTESTS_RETURNCODE_FAIL_INPUT",               \

"TSTESTS_RETURNCODE_FAIL_GENERAL",             \

"TSTESTS_RETURNCODE_SUCCESS"};                 \
                                                   printf("%s\n", ret[(code) +
3]);                                          \
                                              } while(0);



#endif // _COMMON_HPP_
```

Файл `util/bit_counters.hpp` (объявляет функционал битовых счётчиков):

```
/*!
 * \file bit_counters.hpp
 *
 * \author
 *   Andrew Yeliseyev (Russian Technological University, KMBO-03-16, Russia,
2019-2020)
 */
#ifndef BIT_COUNTERS_H
#define BIT_COUNTERS_H



#include "common.hpp"
#include <cstdint>



typedef struct BitCounters
{
  uint8_t         *counters;
  uint64_t         amount_of_counters;
  uint64_t         counter_size;
}
BitCounters;
```

```
TsTestsReturnCode   init_bit_counters   (BitCounters       *target, uint64_t
const num_of_counters, uint64_t const size_of_counter);
void                set_bit            (BitCounters       *target, uint64_t
const bit_i, uint8_t const value);
uint8_t  const      get_bit            (BitCounters const *target, uint64_t
const bit_i);
void                increment_counter  (BitCounters       *target, uint64_t
const i);
uint64_t const      get_counter        (BitCounters const *target, uint64_t
const i);
uint8_t  const      verify_counter     (BitCounters const *target, uint64_t
const i, uint64_t const desired_value);
uint8_t  const      verify_counters    (BitCounters const *target,
uint64_t const desired_value);
void                destroy_bit_counters(BitCounters      *target);


#endif // BIT_COUNTERS_H
```

Файл `util/bit_counters.cpp` (определяет функционал битовых счётчиков):

```
/*!
 * \file bit_counters.cpp
 *
 * \author
 *   Andrew Yeliseyev (Russian Technological University, KMBO-03-16, Russia,
2019-2020)
 */
#include "bit_counters.hpp"
#include <cstdlib>




/*
 * Allocate memory for counters
 */
TsTestsReturnCode init_bit_counters(BitCounters *target, uint64_t const
num_of_counters, uint64_t const size_of_counter)
{
  /*
   * We define the size of the array of bit counters by the following scheme.
   * <1>. (num_of_counters * size_of_counter) accounts for the total amount
of bits that's needed.
   * <2>. To identify the number of needed bytes we divide this number by 8:
<1> >> 3.
   * <3>. If 8 doesn't divide <1>, then we need to add an extra byte since
not all of the bits managed
   *      to fit into <2> bytes.
   */
  uint64_t container_size = ((num_of_counters * size_of_counter) >> 3) +
((num_of_counters * size_of_counter) % 8 != 0);

  target->amount_of_counters = num_of_counters;
  target->counter_size = size_of_counter;
  target->counters = (uint8_t *) malloc(container_size * sizeof(uint8_t));
  if (target->counters == NULL)
    return TSTESTS_RETURNCODE_FAIL_MEMORY;
```

```
  for (uint64_t i = 0; i < container_size; ++i)
    target->counters[i] = 0;

  return TSTESTS_RETURNCODE_SUCCESS;
}

/*
 * Set bit_i-th bit in target to value
 */
void set_bit(BitCounters *target, uint64_t const bit_i, uint8_t const value)
{
  if (value)
    target->counters[bit_i / 8] |=   1U << (bit_i % 8);
  else
    target->counters[bit_i / 8] &= ~(1U << (bit_i % 8));

  return;
}

/*
 * Get bit_i-th bit in target
 */
uint8_t const get_bit(BitCounters const *target, uint64_t const bit_i)
{
  return (target->counters[bit_i / 8] & (1U << (bit_i % 8))) >> (bit_i % 8);
}

/*
 * Increment i-th counter
 */
void increment_counter(BitCounters *target, uint64_t const i)
{
  uint64_t bit_index = i * target->counter_size;

  while (1)
  {
    if (bit_index >= (i + 1) * target->counter_size)
      break;
    if (get_bit(target, bit_index) == 0)
    {
      set_bit(target, bit_index, 1);
      for (uint64_t bit_i = i * target->counter_size; bit_i < bit_index;
++bit_i)
        set_bit(target, bit_i, 0);
      break;
    }
    else
      ++bit_index;
  }

  return;
}

/*
 * Get numeric value of an i-th counter
 */
uint64_t const get_counter(BitCounters const *target, uint64_t const i)
{
  uint64_t result = 0;
  for (uint64_t j = 0; j < target->counter_size; ++j)
  {
```

```
    result |= ((uint64_t) get_bit(target, i * target->counter_size + j)) <<
j;
  }

  return result;
}

/*
 * Check if the i-th counter equals to desired_value
 */
uint8_t const verify_counter(BitCounters const *target, uint64_t const i,
uint64_t const desired_value)
{
  return get_counter(target, i) == desired_value;
}

/*
 * Check if all counters equal to desired_value
 */
uint8_t const verify_counters(BitCounters const *target, uint64_t const
desired_value)
{
  for (uint64_t i = 0; i < target->amount_of_counters; ++i)
    if (!verify_counter(target, i, desired_value))
      return 0;

  return 1;
}

/*
 * Destroy unneeded bit counters
 */
void destroy_bit_counters(BitCounters *target)
{
  if (target != NULL)
    free(target->counters);

  return;
}
```

Файл `tstest_definition.hpp` (содержит реализацию алгоритма 2.1.1):

```
/*!
 * \file tstest_definition.hpp
 *
 * \author
 *   Andrew Yeliseyev (Russian Technological University, KMBO-03-16, Russia,
2019-2020)
 */
#ifndef _TSTEST_DEFINITION_HPP_
#define _TSTEST_DEFINITION_HPP_




#include "util/common.hpp"
#include "util/bit_counters.hpp"
#include <cstring>
```

```
/*
 * Generate consecutive set of d's that are used for construction of
elementary intervals
 */
bool const get_next_d_set(uint8_t *d_set, uint8_t const dim, uint8_t const
d_sum)
{
  uint8_t    *borders     = new uint8_t[dim + 1];
  bool        success     = false;

  borders[0] = 0;
  for (uint8_t dim_i = 1; dim_i < dim + 1; ++dim_i)
    borders[dim_i] = borders[dim_i - 1] + d_set[dim_i - 1] + 1;
  for (uint8_t dim_i = dim - 1; dim_i > 0; --dim_i)
  {
    if (borders[dim_i] <= d_sum + dim_i - 1)
    {
      ++borders[dim_i];
      for (uint8_t dim_j = dim_i + 1; dim_j < dim; ++dim_j)
        borders[dim_j] = borders[dim_j - 1] + 1;
      success = true;
      goto total_break;
    }
  }

  total_break:

  if (success)
  {
    for (uint8_t dim_i = 0; dim_i < dim; ++dim_i)
      d_set[dim_i] = borders[dim_i + 1] - borders[dim_i] - 1;
  }

  delete [] borders;

  return success;
}
```

```
/**
 *  \brief
 *  This test validates the definition of (t,s)-sequence for the generated
 *  set of points.
 *
 *  Validation of definition is performed by calculation of points within
 *  each elementary interval. Counters of points occupy the least possible
 *  amount of memory due to bitwise packaging.
 *
 *  \param[in]  test_info   A valid pointer to \c TsTestsInfo.
 *
 *  \return
 *  \c TSTESTS_RETURNCODE_SUCCESS in case of successful definition assertion.
 *  \c TSTESTS_RETURNCODE_FAIL_GENERAL in case when generated points fail to
 *  meet the requirements of definition by not demonstrating the needed
amount
 *  of points within at least one elementary interval.
 *  \c TSTESTS_RETURNCODE_FAIL_INPUT in case of invalidity of \c test_info
 *  pointer or in case when generated points fail to meet the requirements
```

```
 *  of definition by improperly set \c t and \c m parameters.
 *  \c TSTESTS_RETURNCODE_FAIL_MEMORY in case of dynamic memory allocation
 *  fail.
 */
TsTestsReturnCode const tstest_definition(TsTestsInfo *const test_info)
{
  TSTESTS_TEST_FUNCTION_BEGIN(TSTEST_DEFINITION, test_info->log_file)

  PUSHLOG_4("Test started.")

  TsTestsReturnCode   answer      = TSTESTS_RETURNCODE_SUCCESS;
  BitCounters        *counters    = NULL;

  uint8_t      t              = 0;
  uint8_t      m              = 0;
  uint8_t      s              = 0;
  uint64_t     points_count   = 0;
  uint8_t     *d              = NULL;
  uint64_t    *a              = NULL;
  uint8_t      d_sum          = 0;
  uint64_t     numerator      = 1;
  uint64_t     denominator    = 1;

  if (test_info == NULL)
  {
    answer = TSTESTS_RETURNCODE_FAIL_INPUT;
    goto instant_death;
  }

  /*
   * Specify net's parameters
   */
  t            = test_info->t;
  m            = test_info->m;
  s            = test_info->s;
  points_count = 1ULL << m;
  d            = new uint8_t[s];
  a            = new uint64_t[s];
  d_sum        = m - t;

  /*
   * Check relation between (t) and (m)
   */
  if (t > m)
  {
    answer = TSTESTS_RETURNCODE_FAIL_INPUT;
    goto instant_death;
  }

  // All the following is pointless if we already have
TSTESTS_RETURNCODE_FAIL_INPUT
  /*
   * Setup counters
   * Their amount is calculated as
   *   / m - t + s - 1 \    m - t
   *   |               | * 2
   *   \     s - 1     /
   */
  for (uint8_t i = 1; i <= s - 1; ++i)
  {
    numerator   *= m - t + s - i;
    denominator *= i;
  }
```

```
  counters = new BitCounters;
  answer = init_bit_counters(/*target          = */counters,
                             /*num_of_counters = */(numerator / denominator)
* (1ULL << d_sum),
                             /*size_of_counter = */d_sum == 0 ? t + 1 : t +
2);
  if (answer != TSTESTS_RETURNCODE_SUCCESS)
    goto instant_death;

  /*
   * Iterate over points
   */
  PUSHLOG_4("Checking points...")
  for (uint64_t point_i = 0; point_i < points_count; ++point_i)
  {
    // After the following (point) is expected to be (s)-dimensional
#   ifdef TSTESTS_OPTIMISE_FOR_DIGITAL_NETS
    std::vector<TSTESTS_COORDINATE_TYPE>    point_tmp   = test_info-
>next_point_getter(point_i);
    std::vector<TSTESTS_DIGITAL_TYPE>       point(s, 0);
    std::transform(point_tmp.begin(), point_tmp.end(), point.begin(),
[test_info](TSTESTS_COORDINATE_TYPE c){return (TSTESTS_DIGITAL_TYPE)(c *
(1ULL << test_info->bitwidth));});
#   else
    std::vector<TSTESTS_COORDINATE_TYPE> point = test_info-
>next_point_getter(point_i);
#   endif // TSTESTS_OPTIMISE_FOR_DIGITAL_NETS

    memset(d, 0, s * sizeof(uint8_t));
    d[s - 1] = d_sum;
    memset(a, 0, s * sizeof(uint64_t));

    /*
     * Find all elementary intervals in which (point) falls
     */
    uint64_t    d_sets_counter  = 0;
    // When t == m
    if (d_sum == 0)
      increment_counter(counters, d_sets_counter++);
    // When t < m
    // (when t == m, this loop is skipped right after the first line)
    while (true)
    {
      for (uint8_t dim_i = 0; dim_i < s; ++dim_i)
#       ifdef TSTESTS_OPTIMISE_FOR_DIGITAL_NETS
        a[dim_i] = point[dim_i] >> (test_info->bitwidth - d[dim_i]);
#       else
        a[dim_i] = (uint64_t) std::floor(point[dim_i] * (1 << d[dim_i]));
#       endif // TSTESTS_OPTIMISE_FOR_DIGITAL_NETS

      // Find out index of corresponding bit counter
      uint64_t    counter_index   = (1ULL << d_sum) * d_sets_counter;
      uint8_t     exponent        = 0;
      for (int16_t i = s - 1; i >= 0; --i)
      {
        counter_index += a[i] * (1ULL << exponent);
        exponent += d[i];
      }

      increment_counter(counters, counter_index);

      ++d_sets_counter;
```

```
        if (!get_next_d_set(d, s, d_sum)) break;
      }

      if (point_i + 1 == points_count >> 2)
        PUSHLOG_4("25% of points checked.")
      if (point_i + 1 == points_count >> 1)
        PUSHLOG_4("50% of points checked.")
      if (point_i + 1 == 3 * (points_count >> 2))
        PUSHLOG_4("75% of points checked.")
    }

    PUSHLOG_4("100% of points checked.")

    if (!verify_counters(counters, 1ULL << t))
      answer = TSTESTS_RETURNCODE_FAIL_GENERAL;

    instant_death:

    PUSHLOG_4("Test finished.")

    /*
     * Print the result
     */
    switch (answer)
    {
      case TSTESTS_RETURNCODE_SUCCESS:
        {
          PUSHLOG_1    ("+")
          PUSHLOGF_2   ("+ (%u, %u, %u)", +t, +m, +s)
          PUSHLOG_3    ("Answer: POSITIVE.")
          APPENDLOG_3  ("Verified parameters of (t, m, s)-net:")
          APPENDLOGF_3("t = %u\tm = %u\ts = %u", +t, +m, +s)
          break;
        }
      case TSTESTS_RETURNCODE_FAIL_GENERAL:
        {
          PUSHLOG_1    ("-")
          PUSHLOGF_2   ("- (%u, %u, %u)", +t, +m, +s)
          PUSHLOG_3    ("Answer: NEGATIVE.")
          APPENDLOG_3  ("Unverified parameters of (t, m, s)-net:")
          APPENDLOGF_3("t = %u\tm = %u\ts = %u", +t, +m, +s)
          APPENDLOG_3  ("Test failed at the elementary intervals with
parameters:")
#         if TSTESTS_VERBOSITY_LEVEL >= 3
          uint64_t    d_sets_counter              = 0;
          uint64_t    failed_intervals_counter    = 0;
          for (uint8_t i = 0; i < s - 1; ++i)
            d[i] = 0;
          d[s - 1] = d_sum;
          // When t == m
          if (d_sum == 0)
          {
            fprintf(test_info->log_file, "\t1)\ta: ");
            for (uint8_t j = 0; j < s; ++j)
              fprintf(test_info->log_file, "0\t");
            fprintf(test_info->log_file, "\n\t\td: ");
            for (uint8_t j = 0; j < s; ++j)
              fprintf(test_info->log_file, "0\t");
            fprintf(test_info->log_file, "\n");
            APPENDLOGF_3("\tExpected amount of points inside: %llu.", 1ULL <<
t)
            APPENDLOGF_3("\tActual   amount of points inside: %llu.",
get_counter(counters, 0))
```

```
          }
          // When t < m
          // (when t == m, this loop is skipped right after the first line)
          while (1)
          {
            for (uint64_t i = 0; i < (1ULL << d_sum); ++i)
            {
              uint64_t    i_copy      = i;
              uint8_t     exponent    = m - t;
              for (uint8_t j = 0; j < s; ++j)
              {
                exponent -= d[j];
                a[j] = i_copy / (1ULL << exponent);
                i_copy %= 1ULL << exponent;
              }

              if (!verify_counter(counters, (1ULL << d_sum) * d_sets_counter
+ i, 1ULL << t))
              {
                fprintf(test_info->log_file, "\t%llu)\ta: ",
++failed_intervals_counter);
                for (uint8_t j = 0; j < s; ++j)
                  fprintf(test_info->log_file, "%llu\t", a[j]);
                fprintf(test_info->log_file, "\n\t\td: ");
                for (uint8_t j = 0; j < s; ++j)
                  fprintf(test_info->log_file, "%u\t", d[j]);
                fprintf(test_info->log_file, "\n");
                APPENDLOGF_3("\tExpected amount of points inside: %llu.",
1ULL << t)
                APPENDLOGF_3("\tActual   amount of points inside: %llu.",
get_counter(counters, (1ULL << d_sum) * d_sets_counter + i))
              }
            }

            ++d_sets_counter;

            if (!get_next_d_set(d, s, d_sum)) break;
          }
#         endif // TSTESTS_VERBOSITY_LEVEL
          break;
        }
      case TSTESTS_RETURNCODE_FAIL_INPUT:
        {
          if (test_info != NULL)
          {
            PUSHLOG_1    ("-")
            PUSHLOGF_2   ("- (%u, %u, %u)", +t, +m, +s)
            PUSHLOG_3    ("Answer: NEGATIVE.")
            APPENDLOG_3  ("Unacceptable parameters of (t, m, s)-net:")
            APPENDLOGF_3("t = %u\tm = %u\ts = %u", +t, +m, +s)
            APPENDLOG_3  ("t must be less than or equal to m.")
          }
          else
          {
            PUSHLOG_1    ("- [!]")
            PUSHLOG_2    ("- [!]")
            PUSHLOG_3    ("Answer: NEGATIVE.")
            APPENDLOG_3  ("Invalid test info.")
          }
          break;
        }
      case TSTESTS_RETURNCODE_FAIL_MEMORY:
        {
```

```
            PUSHLOG_1    ("- [!]")
            PUSHLOGF_2   ("- [!]", +t, +m, +s)
            PUSHLOG_3    ("Answer: NEGATIVE.")
            APPENDLOG_3  ("Operating system rejected memory allocation calls.")
            APPENDLOG_3  ("Try reducing values of m or s.")
            break;
        }
    }

    delete [] d;
    delete [] a;
    destroy_bit_counters(counters);
    delete counters;

    TSTESTS_TEST_FUNCTION_END

    return answer;
}



#endif // _TSTEST_DEFINITION_HPP_
```

Реализация алгоритма 3.1.1

Файл `util/common.hpp` (определяет макросы для вывода информации на разных уровнях вербозности и коды возврата):

```
/*!
 * \file common.hpp
 *
 * \author
 *   Andrew Yeliseyev (Russian Technological University, KMBO-03-16, Russia,
2020)
 */
#ifndef _COMMON_HPP_
#define _COMMON_HPP_



#include <cstdint>
#include <cstdio>
#include <functional>
#include <vector>
#include <time.h>



#ifndef TSTESTS_VERBOSITY_LEVEL
#  define TSTESTS_VERBOSITY_LEVEL          0
#endif // TSTESTS_VERBOSITY_LEVEL

#define LOGBLOCK(core)                        do {time(&_time); _ltime =
*localtime(&_time); core} while(0);
#define LOGTIMESTAMP                          _ltime.tm_mday, _ltime.tm_mon+1,
_ltime.tm_year+1900, _ltime.tm_hour, _ltime.tm_min, _ltime.tm_sec

#if TSTESTS_VERBOSITY_LEVEL > 0
#  define TSTESTS_TEST_FUNCTION_BEGIN(name, file)  char const *const
_test_name = #name; time_t _time; struct tm _ltime; clock_t _start_clock =
clock(); clock_t _finish_clock; FILE *_log = file;
#  define TSTESTS_TEST_FUNCTION_END                _finish_clock = clock();
fprintf(_log, "Test completed in %Lf seconds.\n\n", (long
double)(_finish_clock - _start_clock) / CLOCKS_PER_SEC);
#else
#  define TSTESTS_TEST_FUNCTION_BEGIN(name, file)  (void) 0;
#  define TSTESTS_TEST_FUNCTION_END                (void) 0;
#endif // TSTESTS_VERBOSITY_LEVEL 0

#if TSTESTS_VERBOSITY_LEVEL == 1
#  define LOG(message)                             "%s [%02d.%02d.%d
%02d:%02d:%02d] %s\n", _test_name, LOGTIMESTAMP, message
#  define PUSHLOG_1(message)                       LOGBLOCK(fprintf(_log,
LOG(message));)
#  define LOGF(format, ...)                        "%s [%02d.%02d.%d
%02d:%02d:%02d] " format "\n", _test_name, LOGTIMESTAMP, __VA_ARGS__
#  define PUSHLOGF_1(format, ...)                  LOGBLOCK(fprintf(_log,
LOGF(format, __VA_ARGS__));)
#else
#  define PUSHLOG_1(message)                       (void) 0;
#  define PUSHLOGF_1(format, ...)                  (void) 0;
#endif // TSTESTS_VERBOSITY_LEVEL 1
```

```
#if TSTESTS_VERBOSITY_LEVEL == 2
# define LOG(message)                                "%s [%02d.%02d.%d
%02d:%02d:%02d] %s\n", _test_name, LOGTIMESTAMP, message
# define PUSHLOG_2(message)                          LOGBLOCK(fprintf(_log,
LOG(message));)
# define LOGF(format, ...)                           "%s [%02d.%02d.%d
%02d:%02d:%02d] " format "\n", _test_name, LOGTIMESTAMP, __VA_ARGS__
# define PUSHLOGF_2(format, ...)                     LOGBLOCK(fprintf(_log,
LOGF(format, __VA_ARGS__));)
#else
# define PUSHLOG_2(message)                          (void) 0;
# define PUSHLOGF_2(format, ...)                     (void) 0;
#endif // TSTESTS_VERBOSITY_LEVEL 2

#if TSTESTS_VERBOSITY_LEVEL >= 3
# define LOG(delim, message)                         "%s%s%s\n", _test_name,
delim, message
# define LOGCONT(message)                            "\t%s\n", message
# define PUSHLOG_3(message)                          LOGBLOCK(fprintf(_log,
LOG("\n\t", message));)
# define APPENDLOG_3(message)                        LOGBLOCK(fprintf(_log,
LOGCONT(message));)
# define LOGF(delim, format, ...)                    "%s%s" format "\n",
_test_name, delim, __VA_ARGS__
# define LOGFCONT(format, ...)                       "\t" format "\n",
__VA_ARGS__
# define PUSHLOGF_3(format, ...)                     LOGBLOCK(fprintf(_log,
LOGF("\n\t", format, __VA_ARGS__));)
# define APPENDLOGF_3(format, ...)                   LOGBLOCK(fprintf(_log,
LOGFCONT(format, __VA_ARGS__));)
#else
# define PUSHLOG_3(message)                          (void) 0;
# define APPENDLOG_3(message)                        (void) 0;
# define PUSHLOGF_3(format, ...)                     (void) 0;
# define APPENDLOGF_3(format, ...)                   (void) 0;
#endif // TSTESTS_VERBOSITY_LEVEL 3

#if TSTESTS_VERBOSITY_LEVEL >= 4
# define PUSHLOG_4(message)                          LOGBLOCK(fprintf(_log,
LOGF("", " [%02d.%02d.%d %02d:%02d:%02d] %s", LOGTIMESTAMP, message));)
# define PUSHLOGF_4(format, ...)                     LOGBLOCK(fprintf(_log,
LOGF("", " [%02d.%02d.%d %02d:%02d:%02d] " format, LOGTIMESTAMP,
__VA_ARGS__));)
#else
# define PUSHLOG_4(message)                          (void) 0;
# define PUSHLOGF_4(format, ...)                     (void) 0;
#endif // TSTESTS_VERBOSITY_LEVEL 4



#define TSTESTS_COORDINATE_TYPE     long double
#define TSTESTS_DIGITAL_TYPE        uint64_t

#define TSTESTS_LOG_IN_CONSOLE      stdout



typedef struct TsTestsInfo
{
  uint8_t     t;
  uint8_t     m;
  uint8_t     s;
  uint8_t     bitwidth;
```

```
  std::function<std::vector<TSTESTS_COORDINATE_TYPE>(uint64_t const)>
next_point_getter;
  FILE       *log_file;
}
TsTestsInfo;



typedef enum TsTestsReturnCode
{
  TSTESTS_RETURNCODE_SUCCESS              =  0,
  TSTESTS_RETURNCODE_FAIL_GENERAL         = -1,
  TSTESTS_RETURNCODE_FAIL_INPUT           = -2,
  TSTESTS_RETURNCODE_FAIL_MEMORY          = -3
}
TsTestsReturnCode;

#define TSTESTS_PRINT_RETURNCODE(code)      do {
\
                                                char const *const ret[] =
{"TSTESTS_RETURNCODE_FAIL_MEMORY",            \

"TSTESTS_RETURNCODE_FAIL_INPUT",              \

"TSTESTS_RETURNCODE_FAIL_GENERAL",            \

"TSTESTS_RETURNCODE_SUCCESS"};                \
                                                printf("%s\n", ret[(code) +
3]);                                          \
                                              } while(0);



#endif // _COMMON_HPP_
```

Файл `util/incremental_pca.hpp` (объявляет функционал метода главных компонент):

```
/*!
 * \file incremental_pca.hpp
 *
 * \author
 *   Andrew Yeliseyev (Russian Technological University, KMBO-03-16, Russia,
2020)
 */
#ifndef _INCREMENTAL_PCA_HPP_
#define _INCREMENTAL_PCA_HPP_



#include "common.hpp"



typedef std::vector<TSTESTS_COORDINATE_TYPE>    PCAVector;
typedef std::vector<PCAVector>                      PCAMatrix;

typedef struct IncrementalPCAInfo
{
  uint64_t                        points_count;
  PCAMatrix                       scaled_covariance;
```

```
  PCAMatrix                       eigenvectors;
  PCAVector                       eigenvalues;
  bool                            eigen_up_to_date;
}
IncrementalPCAInfo;


TsTestsReturnCode init_incremental_pca              (IncrementalPCAInfo
*const pca_info, uint8_t const dim);
void              partial_fit                       (IncrementalPCAInfo
*const pca_info, PCAMatrix &new_points);
PCAMatrix         get_principal_axes                (IncrementalPCAInfo
*const pca_info);
PCAVector         get_components_variance           (IncrementalPCAInfo
*const pca_info);
PCAVector         get_components_normalised_variance(IncrementalPCAInfo
*const pca_info);


#endif // _INCREMENTAL_PCA_HPP_
```

Файл `util/incremental_pca.cpp` (определяет функционал метода главных компонент):

```
/*!
 * \file incremental_pca.hpp
 *
 * \author
 *   Andrew Yeliseyev (Russian Technological University, KMBO-03-16, Russia,
2020)
 */
#include "incremental_pca.hpp"
#include "jacobi/jacobi.hpp" // [46]
#include <algorithm>



TsTestsReturnCode init_incremental_pca(IncrementalPCAInfo *const pca_info,
uint8_t const dim)
{
  try
  {
    pca_info->points_count = 0;
    pca_info->scaled_covariance.resize(dim);
    pca_info->eigenvectors.resize(dim);
    pca_info->eigenvalues.resize(dim, 0.0);
    for (uint8_t i = 0; i < dim; ++i)
    {
      pca_info->scaled_covariance[i].resize(dim, 0.0);
      pca_info->eigenvectors[i].resize(dim, 0.0);
    }
    pca_info->eigen_up_to_date = true;
  }
  catch(...)
  {
    return TSTESTS_RETURNCODE_FAIL_MEMORY;
  }
```

```
  return TSTESTS_RETURNCODE_SUCCESS;
}



void partial_fit(IncrementalPCAInfo *const pca_info, PCAMatrix &new_points)
{
  /*
   * Iterate over all rows/points in (new_points)
   */
  for (uint64_t point_i = 0; point_i < new_points.size(); ++point_i)
    for (uint8_t dim_i = 0; dim_i < pca_info->eigenvalues.size(); ++dim_i)
      /*
       * Update covariance matrix
       */
      for (uint8_t dim_j = dim_i; dim_j < pca_info->eigenvalues.size();
++dim_j)
        pca_info->scaled_covariance[dim_i][dim_j] = pca_info-
>scaled_covariance[dim_j][dim_i] += (new_points[point_i][dim_i] - .5) *
(new_points[point_i][dim_j] - .5);

  pca_info->points_count += new_points.size();
  pca_info->eigen_up_to_date = false;

  return;
}



void update_eigen(IncrementalPCAInfo *const pca_info)
{
  PCAMatrix covariance(pca_info->scaled_covariance);
  jacobi_public_domain::Jacobi<TSTESTS_COORDINATE_TYPE, PCAVector&,
PCAMatrix&> eigen_calc(pca_info->eigenvalues.size());

  eigen_calc.Diagonalize(covariance, pca_info->eigenvalues, pca_info-
>eigenvectors);
  pca_info->eigen_up_to_date = true;
}

PCAMatrix get_principal_axes(IncrementalPCAInfo *const pca_info)
{
  if (!pca_info->eigen_up_to_date)
    update_eigen(pca_info);

  return pca_info->eigenvectors;
}

PCAVector get_components_variance(IncrementalPCAInfo *const pca_info)
{
  if (!pca_info->eigen_up_to_date)
    update_eigen(pca_info);

  return pca_info->eigenvalues;
}

PCAVector get_components_normalised_variance(IncrementalPCAInfo *const
pca_info)
```

```
{
  PCAVector variance(get_components_variance(pca_info));
  TSTESTS_COORDINATE_TYPE sum = 0.0;

  std::for_each(variance.begin(), variance.end(),
[&sum](TSTESTS_COORDINATE_TYPE elem){sum += elem;});
  std::transform(variance.begin(), variance.end(), variance.begin(),
[&sum](TSTESTS_COORDINATE_TYPE elem){return elem / sum;});

  return variance;
}
```

Файл `tstest_principals.hpp` (содержит реализацию алгоритма 3.1.1):

```
/*!
 * \file tstest_principals.hpp
 *
 * \author
 *   Andrew Yeliseyev (Russian Technological University, KMBO-03-16, Russia,
2020)
 */
#ifndef _TSTEST_PRINCIPALS_HPP_
#define _TSTEST_PRINCIPALS_HPP_




#include "util/common.hpp"
#include "util/incremental_pca.hpp"




/**
 *  \brief
 *  This test performs the incremental principal component analysis for
 *  the set of generated points.
 *
 *  This test can be used to find the axes in multidimensional space
 *  along which the coordinates of points are variated the most. This is
 *  useful for recognising linear patterns in the mutual disposition of
 *  points.
 *
 *  \warning
 *  The IPCA method used in this function is a modification made for
achieving
 *  the most accurate results and optimal performance in **case of (t,m,s)-
nets**!
 *  This function will not work properly for any general dataset.
 *
 *  \param[in]  test_info   A valid pointer to \c TsTestsInfo.
 *
 *  \return
 *  \c TSTESTS_RETURNCODE_SUCCESS in case of successful completion of
calculations.
 *  \c TSTESTS_RETURNCODE_FAIL_INPUT in case of invalidity of \c test_info
 *  pointer.
 *  \c TSTESTS_RETURNCODE_FAIL_MEMORY in case of dynamic memory allocation
 *  fail.
 */
```

```
TsTestsReturnCode const tstest_principals(TsTestsInfo *const test_info)
{
  TSTESTS_TEST_FUNCTION_BEGIN(TSTEST_PRINCIPALS, test_info->log_file)

  PUSHLOG_4("Test started.")

  TsTestsReturnCode   answer  = TSTESTS_RETURNCODE_SUCCESS;
  uint8_t             s       = 0;
  uint64_t            amount  = 0;

  PCAMatrix           points_matrix;
  IncrementalPCAInfo  pca_info;

  if (test_info == NULL)
  {
    answer = TSTESTS_RETURNCODE_FAIL_INPUT;
    goto instant_death;
  }

  s      = test_info->s;
  amount = 1ULL << test_info->m;

  /*
   * Prepare incremental principal component analyser
   */
  points_matrix = PCAMatrix(s, PCAVector(s, 0.0));
  init_incremental_pca(&pca_info, s);

  /*
   * If (amount) is less than dimensionality (s), test is inapplicable
   */
  if (amount < s)
  {
    answer = TSTESTS_RETURNCODE_FAIL_INPUT;
    goto instant_death;
  }

  PUSHLOG_4("Processing points...")
  for (uint64_t point_i = 0; point_i < amount; ++point_i)
  {
    // After the following line (point) is expected to be (s)-dimensional
    std::vector<TSTESTS_COORDINATE_TYPE> point = test_info-
>next_point_getter(point_i);
    for (uint8_t dim_i = 0; dim_i < s; ++dim_i)
    {
      points_matrix[point_i % s][dim_i] = point[dim_i];
    }

    if (point_i % s == (uint8_t)(s - 1))
      partial_fit(&pca_info, points_matrix);

    if (point_i + 1 == amount >> 2)
      PUSHLOG_4("25% of points processed.")
    if (point_i + 1 == amount >> 1)
      PUSHLOG_4("50% of points processed.")
    if (point_i + 1 == 3 * (amount >> 2))
      PUSHLOG_4("75% of points processed.")
  }

  PUSHLOG_4("100% of points checked.")

  instant_death:
```

```
  PUSHLOG_4("Test finished.")

  switch (answer)
  {
    case TSTESTS_RETURNCODE_SUCCESS:
      {
        PCAMatrix axes;
        PCAVector variance;
        if (s > 1)
        {
          axes      = get_principal_axes(&pca_info);
          variance  = get_components_normalised_variance(&pca_info);
        }
        else
        {
          axes      = {{1.0}};
          variance  = {1.0};
        }
        PUSHLOG_1  ("+")
        PUSHLOG_2  ("+")
        PUSHLOG_3  ("Answer: POSITIVE.")
        APPENDLOG_3("The following principal axes have been detected:")
#       if TSTESTS_VERBOSITY_LEVEL >= 3
        for (uint8_t axis_i = 0; axis_i < s; ++axis_i)
        {
          fprintf(test_info->log_file, "\t%u)\t", axis_i + 1);
          for (uint8_t dim_i = 0; dim_i < s; ++dim_i)
            fprintf(test_info->log_file, "%Lf\t", axes[axis_i][dim_i]);
          fprintf(test_info->log_file, "\n\t\tExplained variance ratio :
%Lf.\n", variance[axis_i]);
        }

        uint8_t   main_axes = 0;
        for (TSTESTS_COORDINATE_TYPE accumulated_sum = 0.0; accumulated_sum
<= 0.8; ++main_axes)
          accumulated_sum += variance[main_axes];
#       endif // TSTESTS_VERBOSITY_LEVEL
        if (variance.front() <= 0.8)
          APPENDLOGF_3("80%% of the variance is explained by the first %u
axes.", main_axes)
        else
          APPENDLOG_3 ("80% of the variance is explained by the first axis
alone.")
      }
    case TSTESTS_RETURNCODE_FAIL_GENERAL:
      {
        break;
      }
    case TSTESTS_RETURNCODE_FAIL_INPUT:
      {
        PUSHLOG_1  ("- [!]")
        PUSHLOG_2  ("- [!]")
        PUSHLOG_3  ("Answer: NEGATIVE.")
        APPENDLOG_3("Invalid test info.")
        break;
      }
    case TSTESTS_RETURNCODE_FAIL_MEMORY:
      {
        PUSHLOG_1  ("- [!]")
        PUSHLOG_2  ("- [!]")
        PUSHLOG_3  ("Answer: NEGATIVE.")
        APPENDLOG_3("Operating system rejected memory allocation calls.")
        APPENDLOG_3("Try reducing values of m or s.")
```

```
            break;
        }
    }

    TSTESTS_TEST_FUNCTION_END

    return answer;
}



#endif // _TSTEST_PRINCIPALS_HPP_
```