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

ФАКУЛЬТЕТ ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

КАФЕДРА ПРИКЛАДНАЯ МАТЕМАТИКА И ИНФОРМАТИКА

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОЙ РАБОТЕ НА ТЕМУ:

Разработка библиотеки численных методов на языке программирования **Kotlin**

Студент ИУ9-71Б _____ М. Вареница
(Группа) (Подпись, дата) (И. О. Фамилия)

Руководитель курсовой работы _____ Д. П. Посевин
(Подпись, дата) (И. О. Фамилия)

2021 г.

СОДЕРЖАНИЕ

ПОСТАНОВКА ЗАДАЧИ

Целью курсовой работы является разработка библиотеки численных методов для языка программирования Kotlin.

Для реализации поставленной цели сформулированы следующие задачи:

1) проанализировать существующие библиотеки численных методов для языка программирования Kotlin и выявить степень актуальности разработки новой библиотеки;
2) выбрать необходимые технологии для стека разработки на Kotlin, определить архитектурные подходы и реализовать библиотеку численных методов;
3) провести тестирование библиотеки: юнит-тестирование и тестирование производительности;
4) рассмотреть способы автоматизации процесса формирования документации к библиотеке;
5) для подтверждения возможности масштабирования, исследовать возможность импорта разработанной библиотеки в виде дополнительного модуля в некоторую существующую математическую библиотеку для Kotlin, с открытым исходным кодом.

1. Поиск и анализ существующих библиотек численных методов для языка программирования Kotlin. Необходимость реализации новой библиотеки

Для того, чтобы выявить актуальность разработки библиотеки численных методов на языке Kotlin, необходимо найти и проанализировать уже существующие решения. В процессе анализа существующих решений следует обращать внимание на количество и назначение уже реализованных численных методов. Важно, чтобы библиотека предоставляла как можно большое количество реализованных методов, чтобы их можно было использовать для различных задач.

Численные методы могут использоваться:

- в промышленном программировании, для реализации какого-то специализированного алгоритма или задачи, например, требующей решения какого-то дифференциального уравнения или интеграла;
- в научных или исследовательских целях**.** Для экономии времени нет необходимости реализовывать какой-либо метод самостоятельно, можно воспользоваться готовыми реализациями библиотеки;
- в учебных целях для сравнения собственных реализаций численных методов с реализациями, предоставляемыми библиотекой численных методов (полезно для таких университетских курсов, как «Численные и вычислительные методы линейной алгебры», «Вычислительная математика», «Матричный анализ», «Вычислительная физика»).

Выполнив поиск существующих библиотек с помощью поискового запроса «kotlin math lib» в поисковой системе Google, получены результаты, представленные на рисунке 1.1 и рисунке 1.2.

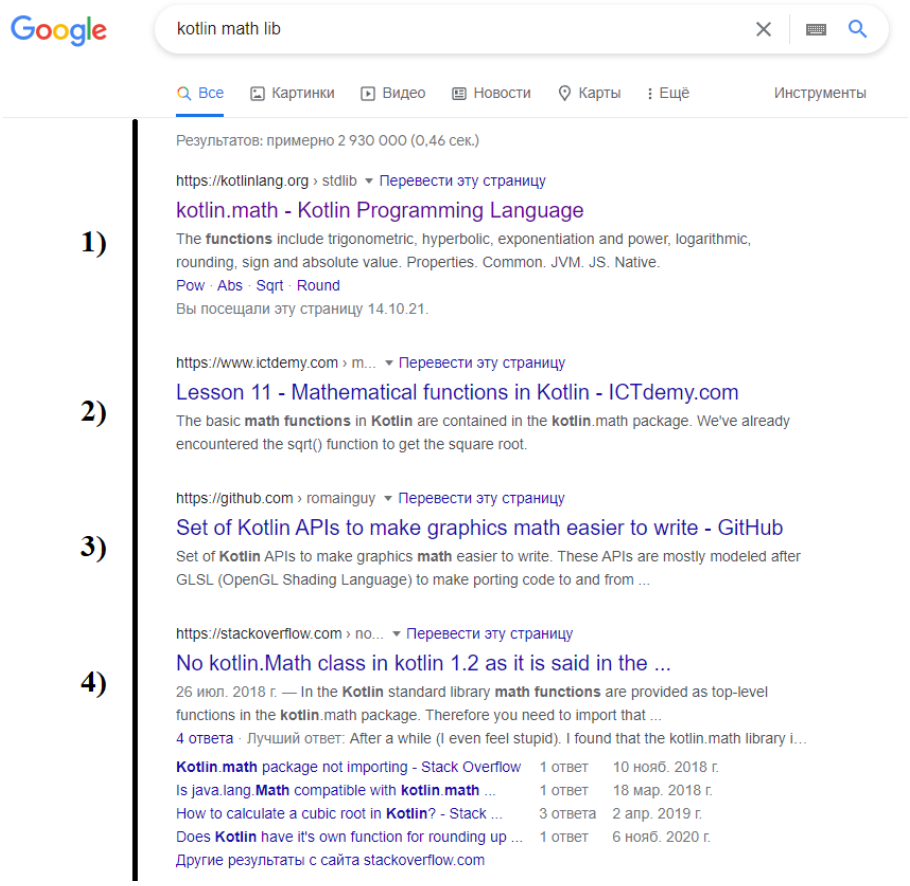


Рисунок 1.1 – Поисковой запрос «kotlin math lib» в поисковой системе Google

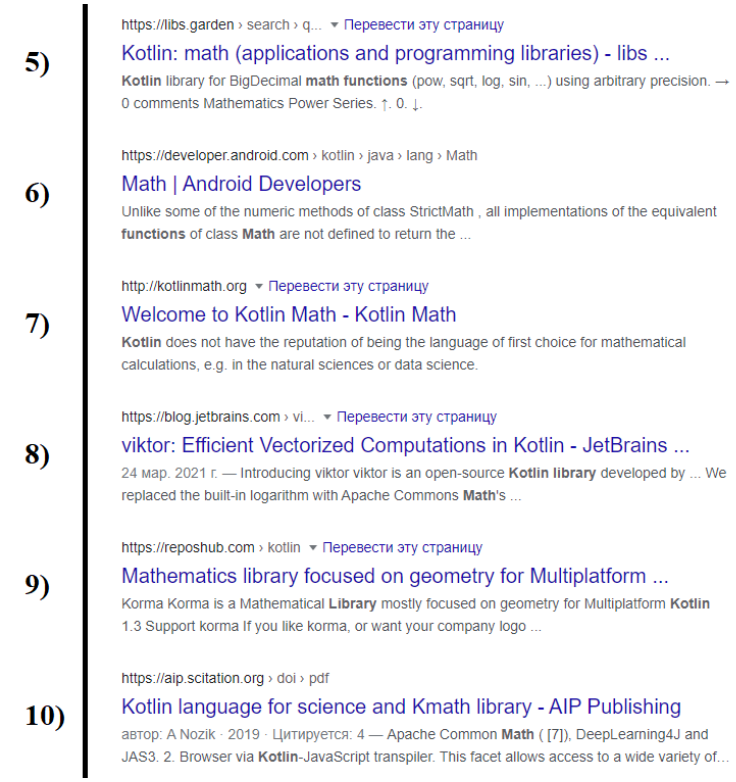


Рисунок 1.2 – Поисковой запрос «kotlin math lib» в поисковой системе Google

Анализ результатов поискового запроса, приведённого на рисунке 1.1 и рисунке 1.2 (порядковый номер элементов перечисления соответствует отметкам, приведенным на рисунках 1.1 и 1.2):

1) на данной странице содержится описание и API стандартной библиотеки math языка Kotlin. Данная библиотека включает в себя различные математические функции: тригонометрические, гиперболические, возведения в степень, логарифмические, округление, модуля, корня, знака и другие. Никаких функций численных методов данная библиотека не предоставляет;
2) на данной странице содержится информация для работы со стандартной библиотекой math;
3) данная страница представляет собой репозиторий библиотеки kotlin-math на GitHub [1]. Согласно описанию из файла README для данного репозитория, эта библиотека реализовывалась для GLSL (OpenGL Shading Language) и предназначена для упрощения переноса кода в шейдеры и обратно. Данная библиотека предоставляет векторные, матричные типы и некоторые операции для работы с ними. Никаких функций численных методов рассматриваемая библиотека не предоставляет;
4) данная страница не удовлетворяет теме поискового запроса;
5) данная ссылка ведет на сервис «libs.garden», который предоставляет информацию о существующих библиотеках для различных языков программирования, согласно поисковому запросу. Для поискового запроса «math» для языка Kotlin представлен ряд результатов (библиотек), описание которых приводится ниже:

    5.1) библиотека MathView, предназначена для адаптивного отображения математических формул в приложениях, никаких функций численных методов данная библиотека не предоставляет;

    5.2) библиотека kotlin-big-math предоставляет математические функции для работы с числами с плавающей точкой, с произвольной точностью (BigDecimal): pow, sqrt, log, sin и другие, никаких функций численных методов данная библиотека не предоставляет;

    5.3) библиотека AndroidMath, позволяет адаптивно визуализировать математические уравнения LaTeX в Android приложениях, никаких функций численных методов данная библиотека не предоставляет;

    5.4) библиотека kotlin-math: описание данной библиотеки приводилось в пункте 3;

5.5) библиотека ExprK, предоставляет «вычислители» для математических выражений, никаких функций численных методов данная библиотека не предоставляет;

5.6) библиотека korma, предоставляет классы для точек, матриц, векторов, углов, прямоугольников, позволяет отображать кривые Безье, создана для решения задач компьютерной графики, позволяет построить пространственную сетку объекта, никаких функций численных методов данная библиотека не предоставляет;

5.7) библиотека glm, предназначена для работы с OpenGL Shading Language (GLSL) в Kotlin и для решения задач компьютерной графики. Никаких функций численных методов данная библиотека не предоставляет;

5.8) библиотека kmath, позволяет работать с комплексными числами, кватернионами, предоставляет реализации для алгебраических структур, таких как кольца, поля. Предоставляет API для работы с матрицами (обращение, LU разложение и т.д), предоставляет язык выражений (expressions), позволяющий создавать формулы и подставлять в них различные структуры, никаких функций численных методов данная библиотека не предоставляет;

5.9) библиотека kotlin-statistics, предназначена для решения задач математической статистики, никаких функций численных методов данная библиотека не предоставляет.

6) на данной странице располагается описание библиотеки math, рассмотренной в пункте 1;
7) данная ссылка ведет на сайт библиотеки Kotlin Math, которая предоставляет операции для работы с комплексными числами, никаких функций численных методов данная библиотека не предоставляет;
8) данная страница не удовлетворяет теме поискового запроса;
9) на данной странице располагается описание библиотеки korma, описанной в пункте 5.6;
10) данная страница не удовлетворяет теме поискового запроса.

Выполнив поиск существующих библиотек с помощью поискового запроса «kotlin numeric methods lib» в поисковой системе Google, возвращена лишь единственная ссылка, удовлетворяющая теме поискового запроса. Данная ссылка ведет на библиотеку koma. Аналогично библиотеке korma, рассмотренной в пункте 5.6 вышестоящего списка, данная библиотека никаких функций численных методов не предоставляет.

Дополнительно была осуществлена попытка поиска библиотек численных методов для Kotlin в сервисе GitHub: все результаты вели на

репозитории уже описанных выше библиотек или на репозитории проектов, не предоставляющих существенных результатов.

Кроме того, сервис «libs.garden», рассмотренный в пункте 5, вышестоящего списка, автоматически выполняет поиск соответствующих библиотек по различным репозиториям в сети интернет. С его помощью не удалось обнаружить библиотеки численных методов на Kotlin.

Таким образом, не удалось найти специализированной библиотеки для численных методов на Kotlin. Не исключено, что существуют небольшие, частичные реализации, возможно коммерческие, без открытого исходного кода, однако их обнаружить не удалось.

Отсутствие многих нестандартных и узконаправленных библиотек для Kotlin обусловлено тем, что Kotlin является относительно новым языком программирования, который активно начал распространяться в течении последних 8 лет и для него многие решения еще не реализованы.

Одним из возможных решений отсутствия библиотеки численных методов на Kotlin является использование библиотек, написанных на других языках программирования, например, на Python или Java. Но данный подход имеет ряд недостатков:

- программисты реализующие прикладное программное обеспечение могут не знать другие языки программирования, а значит, могут возникнуть различные трудности в их использовании;
- такое решение приводит к тому, что при реализации проекта придется использовать другие языки программирования и их технологии, что не является хорошим стилем разработки, так как чем больше в проекте использовано различных языков и технологий, тем сложнее его поддерживать, придется искать более квалифицированные кадры и переплачивать за время работы программистов, чтобы они осваивали новые языки и технологии.

Например, использование обертки математической библиотеки numpy [2], которая изначально реализована для Python, для Kotlin выглядит как временное решение. Кроме того, анализируя документацию к библиотеке numpy было обнаружено, что эта библиотека разрабатывалась для решения большого множества математических задач, из различных сфер математики и Data Science, и она не позиционирует себя как библиотека, предоставляющая реализации алгоритмов численных методов. Аналогичные выводы можно сделать и для других похожих библиотек.

Многие инженерные задачи, а также задачи моделирования требуют реализации алгоритмов численных методов, так, например, в numpy реализован метод Гаусса для решения системы алгебраических уравнений, но поскольку разработчики numpy не стремились позиционировать свою библиотеку, как инструмент решения численных методов, то они реализовали лишь самые необходимые и производительные, что является ее недостатком. Например, в numpy не обнаружены реализации методов Якоби [9.40] и Зейделя

[9.41], хотя разработчикам прикладного программного обеспечения могут потребоваться именно эти реализации для решения каких-либо исследовательских задач. Кроме того, библиотека numpy не предоставляет реализации многих методов интегрирования и использует лишь метод трапеций, например пакет numpy.trapz.

Для устранения недостатков приведенных выше сообщество Python программистов реализовало новую библиотеку scipy, которая основана на numpy и содержит многие реализации численных методов, в дополнение к библиотеке numpy. Но в отличии от numpy, для Kotlin не удалось найти обертки для scipy.

Таким образом, если разработчику прикладного программного обеспечения потребуется реализация некоторых численных методов, то покрытия numpy может не хватить (примеры отсутствующих в numpy методов приведены выше), а большинство алгоритмов придется реализовывать самостоятельно, затрачивая дополнительное время.

Таким образом, задача разработки новой библиотеки численных методов с открытым исходным кодом для Kotlin актуальна.

2. Выбор технологий

Для разработки библиотеки численных методов для Kotlin были выбраны необходимые технологии.

2.1. Система контролей версий git и сервис GitHub

Поскольку библиотека подразумевает использование сообществом программистов, то ее необходимо опубликовать и создать соответствующий репозиторий в некоторой системе, для предоставления исходного кода, документации и возможности предоставить обратную связь по работе функционала библиотеки. Для данного курсового проекта была использована система контроля версий git с использованием сервиса GitHub. В сервисе GitHub был создан репозиторий библиотеки.

2.2. Язык программирования

В качестве языка программирования для реализации библиотеки использовался язык Kotlin версии 1.5. Реализация библиотеки поддерживает платформу JVM начиная с версии 1.8.

2.3. Среда разработки

Работа над проектом велась в среде IntelliJ IDEA Ultimate версии [3], поддерживающей ряд фреймворков и инструментов для разработки корпоративных приложений, а также имеющей все необходимые инструменты для компиляции, отладки и тестирования библиотеки.

2.4. Система сборки Gradle

Для разработки серьезных проектов важно правильно выбрать систему сборки, которая поможет автоматизировать подключение к проекту и скачивание зависимостей, компиляцию, тестирование, а также предоставит ряд других полезных инструментов для разработки и запуска проекта. Для проектов, реализуемых на языках JVM самыми популярными системами сборок являются:

– Ant [4];
– Maven [5];
– Gradle [6].

Система сборки Ant является устаревшим вариантом, поскольку она основана на похожих принципах конфигурирования и сборки проекта инструмента Make (для C и C++ проектов). В качестве альтернативы Ant были разработаны системы сборок Maven и Gradle, в которых конфигурирование проекта проще и автоматизировано, а размер кода конфигурационных файлов меньше в десятки раз.

Исторически сложилось, что для проектов на Java и Scala в основном используется Maven, а для проектов на Kotlin чаще используется Gradle, популярность которого связана с его использованием в разработке для Android. Система сборки Gradle глобально не отличается от Maven, важным отличием является то, что Gradle поддерживает некоторые возможности системы сборки Ant, а конфигурационные файлы Gradle описываются на более легком языке, похожем на JSON, тогда как в Maven используется язык разметки XML.

В силу того, что большинство проектов на Kotlin реализовывается на основе системы сборки Gradle и учитывая ее большой рост популярности, была выбрана система сборки Gradle.

Таким образом, проект обернут в систему сборки Gradle версии 6.3, предназначенной для сборки проекта, автоматизации тестирования и управления зависимостями.

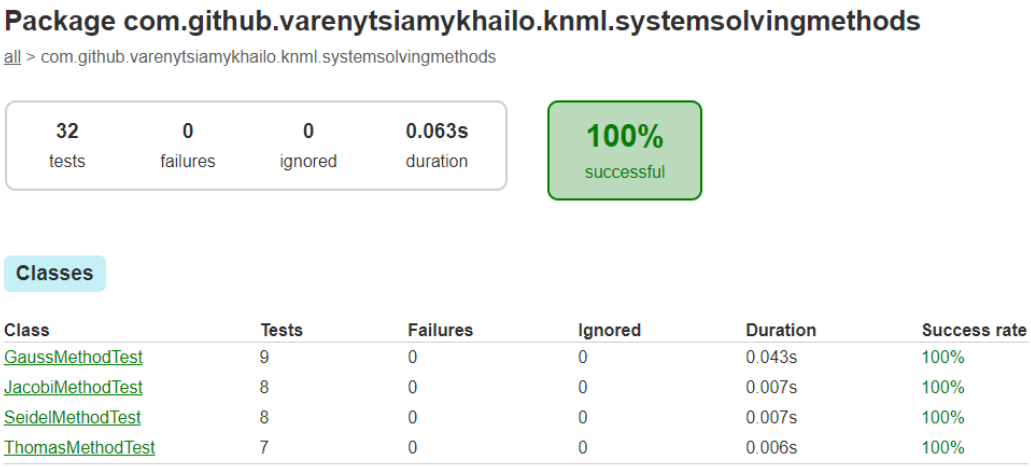
2.5. Фреймворк юнит-тестирования JUnit

Хорошей практикой разработки приложений, систем и библиотек является разработка через тестирование. Данный подход можно реализовать через юнит-тестирование, с помощью фреймворка JUnit [7], разработанного для JVM языков.

Реализация юнит-тестов к сущностям и функциям проекта позволит сократить количество ошибок в логике работы кода, которые могут появляться вследствие модернизации и изменения программного кода. Кроме того, для того, чтобы с более успешной вероятностью интегрировать разработанную библиотеку в другую, ее разработчики могут обратить внимание на наличие юнит-тестов в проекте, чтобы получить больше гарантий в правильности работы реализованного функционала.

Таким образом, разработка библиотеки велась параллельно с реализацией юнит-тестов с помощью фреймворка JUnit версии 5.8.1.

Система сборки Gradle предоставляет графический интерфейс в виде html страницы, на котором можно посмотреть отчет о выполнении юнит-тестов в проекте (см. рисунок 2.5.1).



Package com.github.varenytsiamykhailo.knml.systemsolvingmethods

all > com.github.varenytsiamykhailo.knml.systemsolvingmethods

32 tests	0 failures	0 ignored	0.063s duration	100% successful

Classes

Class	Tests	Failures	Ignored	Duration	Success rate
GaussMethodTest	9	0	0	0.043s	100%
JacobiMethodTest	8	0	0	0.007s	100%
SeidelMethodTest	8	0	0	0.007s	100%
ThomasMethodTest	7	0	0	0.006s	100%

Рисунок 2.5.1 – Демонстрация выполненных юнит-тестов

2.6. Инструмент генерации документации Dokka и язык KDoc

Для разработки документации библиотеки существует несколько решений:

- самостоятельно писать документацию в какой-то сторонней системе или реализовать собственную среду, например, на базе web-сайта;
- использовать автоматические инструменты для генерации документации.

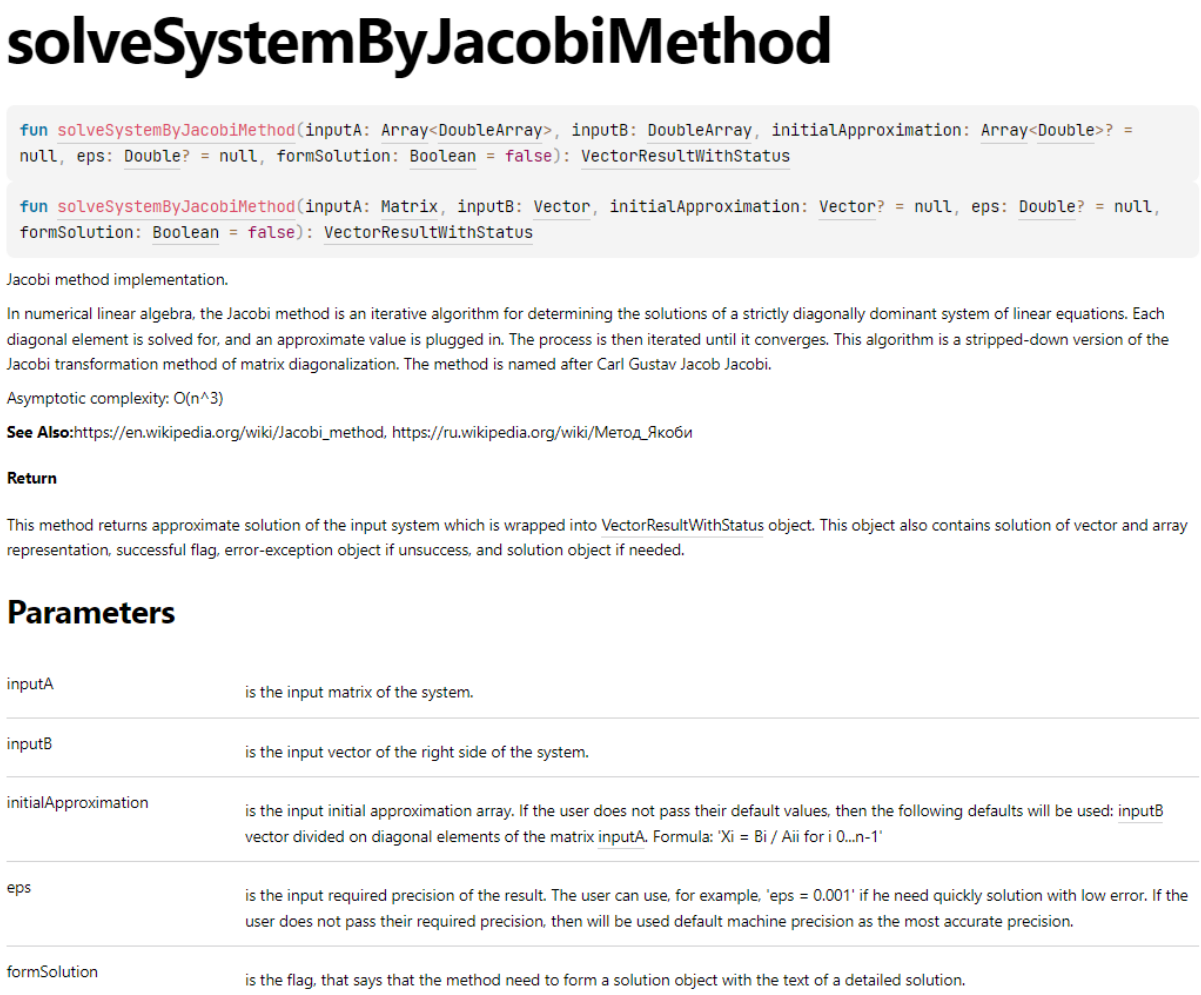
Одним из известных способов для генерации документации к программам, написанным на Java является инструмент JavaDoc, который работает следующим образом. Документация к сущностям и методам описывается в виде JavaDoc комментариев к программному коду, следуя определенным правилам синтаксиса, а соответствующий JavaDoc инструмент извлекает информацию из комментариев и автоматически генерирует документацию в виде html страниц, с использованием CSS разметки и JavaScript. Таким образом отпадает острая необходимость разрабатывать сайт с документацией и заполнять его вручную.

В Kotlin отсутствует поддержка инструмента JavaDoc и вместо него предоставляется аналогичный инструмент-утилита Dokka [8], с языком написания комментариев KDoc. По сути, KDoc комбинирует JavaDoc синтаксис для тегов, расширенных для поддержки специфичных Kotlin конструкций, и Markdown для встроенной разметки.

Dokka поставляет плагин для Gradle, с помощью которого можно интегрировать создание документации в процесс сборки с помощью Gradle. Данный плагин позволяет генерировать документацию в нескольких

форматах: в kotlin-html формате, в JavaDoc формате, а также в стиле GitHub документации.

Пример сгенерированного документа в формате kotlin-html представлен на рисунке 2.6.1.



solveSystemByJacobiMethod

```
fun solveSystemByJacobiMethod(inputA: Array<DoubleArray>, inputB: DoubleArray, initialApproximation: Array<Double>? =
null, eps: Double? = null, formSolution: Boolean = false): VectorResultWithStatus
```

```
fun solveSystemByJacobiMethod(inputA: Matrix, inputB: Vector, initialApproximation: Vector? = null, eps: Double? = null,
formSolution: Boolean = false): VectorResultWithStatus
```

Jacobi method implementation.

In numerical linear algebra, the Jacobi method is an iterative algorithm for determining the solutions of a strictly diagonally dominant system of linear equations. Each diagonal element is solved for, and an approximate value is plugged in. The process is then iterated until it converges. This algorithm is a stripped-down version of the Jacobi transformation method of matrix diagonalization. The method is named after Carl Gustav Jacob Jacobi.

Asymptotic complexity: O(n^3)

See Also:https://en.wikipedia.org/wiki/Jacobi_method, https://ru.wikipedia.org/wiki/Метод_Якоби

Return

This method returns approximate solution of the input system which is wrapped into VectorResultWithStatus object. This object also contains solution of vector and array representation, successful flag, error-exception object if unsuccess, and solution object if needed.

Parameters

inputA	is the input matrix of the system.
inputB	is the input vector of the right side of the system.
initialApproximation	is the input initial approximation array. If the user does not pass their default values, then the following defaults will be used: inputB vector divided on diagonal elements of the matrix inputA. Formula: 'Xi = Bi / Aii for i 0...n-1'
eps	is the input required precision of the result. The user can use, for example, 'eps = 0.001' if he need quickly solution with low error. If the user does not pass their required precision, then will be used default machine precision as the most accurate precision.
formSolution	is the flag, that says that the method need to form a solution object with the text of a detailed solution.

Рисунок 2.6.1 – Описание метода «solveSystemByJacobiMethod» в формате kotlin-html

Пример сгенерированной документации в формате JavaDoc представлен на рисунке 2.6.2.

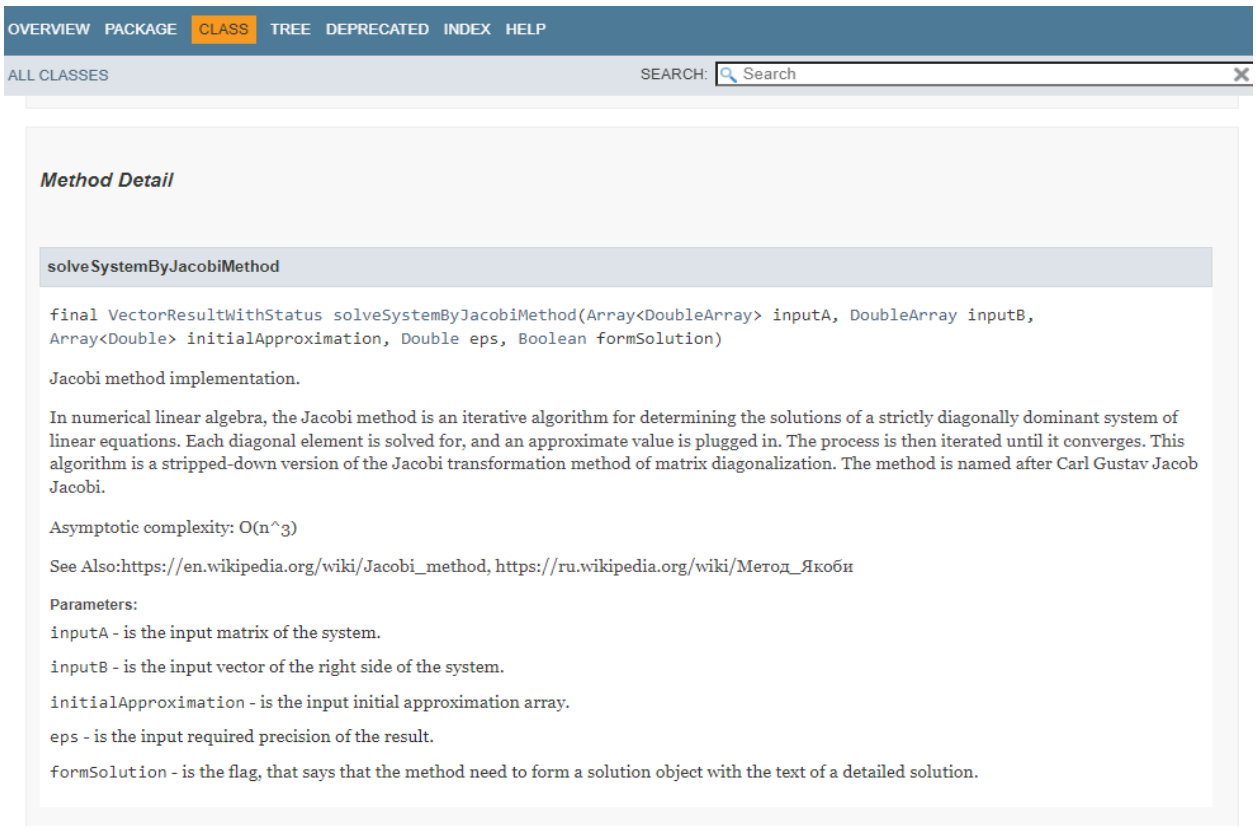


Рисунок 2.6.2 – Описание метода «solveSystemByJacobiMethod» в формате JavaDoc

Таким образом, документация реализована в виде комментариев к коду на языке KDoc, а с помощью инструмента Dokka генерируются веб-страницы, которые при необходимости можно разместить на сайте или же импортировать в GitHub документацию к репозиторию.

3. Основные принципы разработки и архитектурные решения

3.1. Выбор названия библиотеки: «KNML – Kotlin Numeric Methods Library»

В качестве названия библиотеки было решено использовать «Kotlin Numeric Methods Library», с аббревиатурой «KNML». Перед выбором данного названия был проведен анализ в сети интернет на существование других библиотек с таким названием. Выявить библиотеки для Kotlin с таким названием не удалось.

3.2. Выбор инженерного подхода для разработки библиотеки

Существует два подхода для решения различных задач:

- научно-исследовательский — программист с помощью научно-исследовательского подхода что-то оптимизирует или изобретает с точки зрения науки, например, придумывает новые алгоритмы и т.д, в рамках данной курсовой работы можно попытаться придумать и реализовать новые алгоритмы, но все же промышленная библиотека численных методов подразумевает реализацию уже известных и проверенных мировым научным сообществом методов;
- инженерный — программист реализовывает библиотеку, используя уже существующие научные материалы в рамках рассматриваемой предметной области, вполне возможно окажется так, что реализация в рамках инженерного подхода будет обладать рядом преимуществ, например, будет сфокусирована на достижении конкретных целей, к примеру, формирование подробного решения.

В рамках данной курсовой работы используется инженерный подход. В качестве готового научного материала были использованы уже известные и проверенные наукой методы численной алгебры.

Реализация численных методов на Kotlin, а особенно совокупности методов в виде библиотеки является во многом творческой задачей т.к. необходимо реализовать все методы так, чтобы их реализации соответствовали единому стилю программирования и предоставляли одинаковый API, причем в максимально возможно clean-code стиле, предоставляя документацию к сущностям и методам библиотеки.

3.3. Выбор стиля API библиотеки. Шаблон проектирования «Посредник»

Существует два способа предоставления клиентского API.

Первый способ: шаблон проектирования «Посредник».

Пусть в проекте есть два класса, представляющие различные сущности, например, Matrix и Vector и требуется реализовать операцию умножения матрицы на вектор. Для этого можно завести отдельный дополнительный класс, в котором будут храниться «операции», выполняемые над независимыми друг от друга сущностями. Таким образом, в коде данный API с точки зрения клиента будет выглядеть примерно так:

“result = matrixMultiplicateVector(m, v)”.

Данный подход имеет преимущество: мы отделяем классы-сущности от логики работы с ними, вынося дополнительный функционал в отдельное место, в результате чего код становится более гибким для масштабирования и поддержки. Рассмотренный подход отделения дополнительного функционала от сущностей похож на шаблон проектирования «Посредник».

Второй способ: функционал в виде методов класса.

Можно обойтись без дополнительного класса с функционалом и реализовывать операции внутри классов-сущностей, например, реализовать метод умножения матрицы на вектор внутри класса Matrix.

Таким образом, в коде данный API будет выглядеть примерно так:

“result = m.matrixMultiplicateVector(v)”.

Недостаток данного подхода заключается в том, что классы-сущности могут содержать в себе много дополнительных методов, которые в принципе, по своей природе, невозможно отнести к конкретной сущности. В результате, бизнес-логика классов тесно переплетается с деталями реализации сторонних классов. Такой код довольно сложно понимать и поддерживать.

В данном курсовом проекте было принято решение совместить приведенные два способа формирования API:

1) В некоторых языках программирования, прежде чем использовать какую-то экзотическую функцию, необходимо это делать с помощью обращения к некоторому объекту. Во многих языках популярные функции интегрировали в язык и их можно вызывать из любого места просто вызвав функцию, без обращения к сторонним объектам. В разрабатываемой библиотеке роль таких функций играют функции, использование которых не должно подразумевать преждевременное обращение к лишним сущностям-объектам и было бы неплохо их

вызывать из любого места кода, без обращения к объектам. Такие функции было решено определять в специальных Kotlin файлах, на самом верхнем уровне (вне классов). Данное решение дополнительно подкрепляется тем, что Kotlin поддерживает процедурный стиль программирования и для функций не обязательно создавать отдельный класс. Таким образом, в результате выбора первого способа не обязательно создавать дополнительный класс, а можно просто объявлять нужные функции-операции в обычном файле, на самом верхнем уровне, как это принято делать в языке СИ. Пример вызова одной из таких реализованных функций представлен на рисунке 3.3.1.

```
val resultVector: Vector = matrixMultiplicateVector(m1, v1)
```

Рисунок 3.3.1 – Вызов функции умножения матрицы на вектор

2) Для каждого численного метода был реализовал свой отдельный класс, отвечающий за работу конкретного численного метода и содержащий реализацию основного алгоритма, вспомогательные методы, необходимые для работы основного алгоритма и общедоступные API методы, которые позволяют прикладному программисту работать с объектом класса. Таким образом, каждый численный метод реализован в классе, который не содержит посторонних данных (свойств, полей) и действий (методов), не относящихся конкретно к численному методу. Таким образом, исключаются недостатки второго подхода. Пример вызова численного метода Зейделя представлен на рисунке 3.3.2.

```
val result: VectorResultWithStatus = SeidelMethod().solveSystemBySeidelMethod(
    A,
    B,
    initialApproximation = Array( size: 4) {1.0},
    eps = 0.01,
    formSolution = true
)
```

Рисунок 3.3.2 – Вызов метода Зейделя решения СЛАУ

3.4. API входных параметров для численных методов

Т.к. в библиотеке дополнительно реализованы вспомогательные классы Vector и Matrix для удобства пользователей, то необходимо реализовать поддержку как обычных массивов, так и вспомогательных классов, в качестве входных данных для численных методов.

Например, в численный метод Якоби пользователь имеет возможность передать входные данные в виде массивов или же в виде классов Vector и Matrix. Данная поддержка реализовывается с помощью перегрузок функций, а поддержка нескольких вариантов входных параметров делает библиотеку более универсальной.

Кроме того, т.к. язык Kotlin позволяет объявлять значения по умолчанию для входных параметров, то отпала необходимость перегружать функции много раз, как если бы библиотека реализовывалась на Java. На примере реализованного метода Якоби достаточно было реализовать лишь две перегрузки, благодаря использованию значений по умолчанию.

Т.к. различные численные методы подразумевают в качестве входных данных различные не обязательные параметры, такие как, точность требуемого решения или выбор начального приближения, то для них была предоставлена реализация по умолчанию: если пользователь не передаст какой-то параметр в вызов численного метода, то по возможности будет использовано запрограммированное значение по умолчанию, например, алгоритм выбора начального приближения.

Для численных методов, требующих передачи на вход значения точности требуемого решения epsilon, было реализовано значение по умолчанию, причем машинной точности. В данном случае под машинной точностью подразумевается такое значение epsilon, при котором язык программирования будет считать две различные переменные с плавающей точкой одинаковыми по значению. Код реализации вычисления машинной точности представлен на рисунке:

```
fun getMachineEps(): Double {
    var eps = 1.0
    while (1 + eps > 1) {
        eps /= 2.0
    }
    return eps
}
```

Рисунок 3.4.1 – Реализация алгоритма получения машинной точности

В данном алгоритме начальное значение (eps = 1.0) делится на 2.0 до тех пор, пока JVM не станет считать (1 + eps) и 1 одинаковыми числами.

3.5. API возвращаемых значений из численных методов

Т.к. язык Kotlin не позволяет возвращать из функций или методов более одного значения, было принято решение реализовать и использовать специальные классы-обертки, объекты которых хранят в себе результаты работы численных методов.

В библиотеке такими классами являются:

- VectorResultWithStatus;
- MatrixResultWithStatus;
- DoubleResultWithStatus.

Возвращаемые численными методами объекты приведенных классов хранят в себе различную информацию в следующих свойствах (в Kotlin нет полей):

1) решение задачи в виде одномерного вектора (реализованный тип Vector) или матрицы (реализованный тип Matrix) или числа с плавающей точкой (тип Double);
2) решение задачи в виде одномерного массива или многомерного массива;
3) флаг, несущий информацию об успешно отработанном численном методе или о неудаче (например, произошло исключение или пользователь передал на вход метода некорректные данные);
4) свойство, хранящее исключение, если оно возникнет;
5) объект подробного решения, если необходим, данный объект содержит в себе строку по умолчанию с подробным решением, а также сохраненные промежуточные данные, используемые в процессе работы алгоритма численного метода.

Таким образом, в возвращаемом объекте содержится решение в двух формах: в обычном массиве или же в пользовательском Vector или Matrix объекте и пользователь может использовать удобную для него форму.

Для классов VectorResultWithStatus и MatrixResultWithStatus были реализованы методы equals и hashCode, что позволит корректно сравнивать друг с другом объекты данных типов. Пример объявления свойств для класса MatrixResultWithStatus представлен на рисунке 3.5.1.

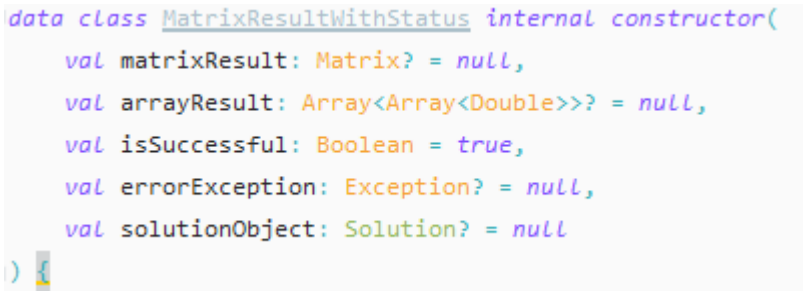


Рисунок 3.5.1 – Структура класса MatrixResultWithStatus

Стоит отметить, что стандартная библиотека Kotlin предоставляет классы Pair и Triple, которые можно было бы использовать для возврата нескольких значений из функций. Но хорошим тоном в Kotlin является определение своих классов-данных и их объявление с ключевым словом data, как показано на рисунке 3.5.1, а хранимым данным приходится предоставлять имена, что делает код более читаемым. Благодаря пометке класса ключевым словом data компилятор автоматически генерирует различные полезные методы, такие как toString, clone, equals, hashCode и другие.

3.6. Обработка ошибок и исключений в реализации численных методов

Многие численные методы имеют различные ограничения на входные данные, например, метод Якоби для решения СЛАУ позволяет работать над системами только с диагональным преобладанием. Если пользователь библиотеки по незнанию или случайно предоставит неверные входные данные, то в работе алгоритма может произойти одна из ситуаций:

- будет предоставлено решение, но неверное;
- программа завершится аварийно из-за брошенного исключения.

Чтобы предотвратить возникновение подобных ситуаций, важно продумать стратегию борьбы с возможными ошибками и исключениями.

При разработке данной библиотеки было принято решение использовать следующие приемы, на примере реализации метода Якоби.

1) в основном классе метода Якоби реализовывается private функция, закрытая для пользователей библиотеки, выполняющая работу основного алгоритма. В данной функции проводятся все возможные проверки (на корректность входных данных, условия сходимости

метода и т.д.). В случае неудовлетворения какой-то проверке или возникновения ошибки, она выбрасывает исключение с подробным описанием ошибки, и работа метода не продолжается. В случае, если алгоритм отработал успешно, из метода возвращается результат работы в виде объекта VectorResultWithStatus, в который помещается решение в различных формах, информация об успешном выполнении метода, объект подробного решения, если требуется (иначе null);

2) дополнительно в классе объявляются public API перегруженные методы, доступные пользователям библиотеки. Данные методы в своей работе вызывают основную функцию алгоритма, принимают результат в виде объекта VectorResultWithStatus и передают его далее пользователю, а также обрабатывают возможные исключения. В случае, если основной алгоритм выбросил исключение, то метод-обертка отлавливает его, обрабатывает, и в качестве результата работы численного метода возвращает объект VectorResultWithStatus, в поле errorException которого помещается пойманное исключение, значение свойства isSuccessful устанавливается в false, а все остальные свойства имею значения null;
3) таким образом, объекты VectorResultWithStatus могут иметь только два состояния:
    3.1) если значение свойства isSuccessful=false, то объект обязательно содержит внутри себя объект брошенного исключения, в свойстве errorException, а все остальные свойства гарантированно равны null и никакого «мусора» в них храниться не может;
    3.2) если значение свойства isSuccessful=true, то свойство errorException гарантированно равно null. А значение свойства solutionObject с подробным решением либо заполнено, либо имеет значение null, в зависимости от того, передавал ли пользователь специальный флаг в вызов численного метода.

Пример заполнения свойств возвращаемого объекта-результата MatrixResultWithStatus представлен на рисунке 3.6.1.

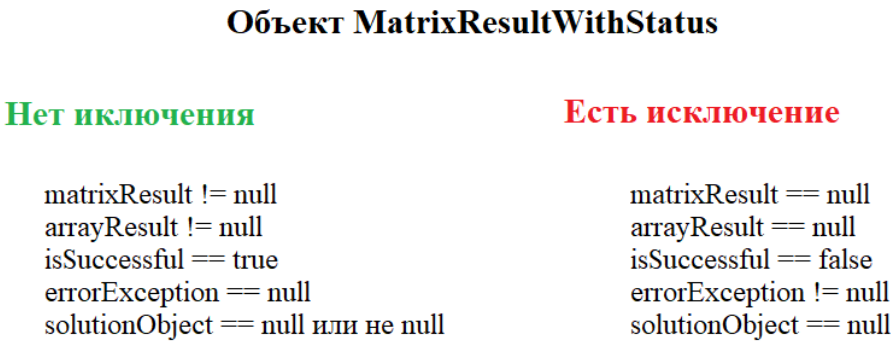


Рисунок 3.6.1 – Заполнение свойств возвращаемого объекта-результата MatrixResultWithStatus

Таким образом, в библиотеке реализованы проверки на все возможные необходимые и достаточные условия сходимости численных методов, соответствия размеров входных данных и т.д., в зависимости от передаваемых пользователем входных данных. Программа пользователя гарантированно не завершится аварийно из-за возможного брошенного исключения, т.к. по сути, библиотека обрабатывает исключение вместо пользователя, и хранит его в возвращаемом объекте errorException, из которого пользователь сможет достать объект-исключения и вытащить из него всю требуемую информацию с подсказками.

На рисунке 3.6.2 представлен пример перегруженного API метода Якоби и его документирование с помощью KDoc комментария. На рисунке 3.6.2 демонстрируется обработка исключения, которое возможно будет выброшено основным методом алгоритма.

```
/**
 * Jacobi method implementation.
 *
 * In numerical linear algebra, the Jacobi method is an iterative algorithm for determining the solutions of a strictly diagonally dominant system of linear equations.
 * Each diagonal element is solved for, and an approximate value is plugged in. The process is then iterated until it converges.
 * This algorithm is a stripped-down version of the Jacobi transformation method of matrix diagonalization.
 * The method is named after Carl Gustav Jacob Jacobi.
 *
 * Asymptotic complexity: O(n^3)
 *
 * **See Also:** [https://en.wikipedia.org/wiki/Jacobi_method], [https://ru.wikipedia.org/wiki/Метод_Якоби]
 *
 * @param [inputA] is the input matrix of the system.
 * @param [inputB] is the input vector of the right side of the system.
 * @param [initialApproximation] is the input initial approximation array.
 * If the user does not pass their default values, then the following defaults will be used:
 * [inputB] vector divided on diagonal elements of the matrix [inputA].
 * Formula: 'Xi = Bi / Aii for i 0...n-1'
 * @param [eps] is the input required precision of the result.
 * The user can use, for example, 'eps = 0.001' if he need quickly solution with low error.
 * If the user does not pass their required precision, then will be used default machine precision as the most accurate precision.
 * @param [formSolution] is the flag, that says that the method need to form a solution object with the text of a detailed solution.
 *
 * @return This method returns approximate solution of the input system which is wrapped into [VectorResultWithStatus] object.
 * This object also contains solution of vector and array representation, successful flag, error-exception object if unsuccess, and solution object if needed.
 */
fun solveSystemByJacobiMethod(
    inputA: Array<Array<Double>>,
    inputB: Array<Double>,
    initialApproximation: Array<Double>? = null,
    eps: Double? = null,
    formSolution: Boolean = false
): VectorResultWithStatus {
    return try {
        runSolvingSystemByJacobiMethod(
            inputA,
            inputB,
            initialApproximation,
            eps: eps ?: getMachineEps(),
            formSolution
        )
    } catch (e: Exception) {
        VectorResultWithStatus( vectorResult: null,  arrayResult: null,  isSuccessful: false, e,  solutionObject: null)
    }
}
```

Рисунок 3.6.2 – Реализация одного из API метода Зейделя и его документирование

Единственное неудобство при таком подходе: пользователю потребуется постоянно проверять флаг isSuccessful, прежде чем использовать результаты, т.к. они могут содержать null в случае неудачной работы алгоритма, как это делается в языке GoLang, или же напрямую использовать результаты, предварительно выполнив проверку на null, к счастью, Kotlin позволяет это делать красиво, с помощью Элвис-оператора.

В качестве альтернативы можно было бы не обрабатывать исключения за пользователя, а заставить это делать его самостоятельно, но в таком случае пользователю пришлось бы нагромождать свой код конструкциями try-catch, а если он не будет делать этого, то программа может завершиться аварийно.

3.7. Формирование текстовых подробных решений

Дополнительной возможностью библиотеки является формирование подробных решений. Такие подробные решения, с текстовым объяснением часто демонстрируются пользователям различных онлайн калькуляторов. Реализованная библиотека позволяет получать такие решения по требованию пользователя при вызове определенного численного метода, передав в качестве аргумента флаг formSolution=true. Если такой флаг будет передан, то в возвращаемом численным методом объекте VectorResultWithStatus и тому подобных будет располагаться объект подробного решения, в свойстве solutionObject. Если флаг не передается, то свойство solutionObject будет содержать null.

Объект подробного решения solutionObject содержит строку типа String, содержащее сформированный текст подробного решения по умолчанию.

На рисунке 3.7.1 представлен пример вызова численного метода Зейделя для решения СЛАУ и его входные данные.

```
val A: Array<Array<Double>> =  arrayOf(
    arrayOf(115.0, -20.0, -75.0),
    arrayOf(15.0, -50.0, -5.0),
    arrayOf(6.0, 2.0, 20.0)
)
val B: Array<Double> = arrayOf(20.0, -40.0, 28.0)

val result: VectorResultWithStatus = SeidelMethod().solveSystemBySeidelMethod(
    A,
    B,
    initialApproximation = Array( size: 3) {1.0},
    eps = 0.01,
    formSolution = true
)
```

Рисунок 3.7.1 – Пример вызова метода Зейделя для решения СЛАУ и его входные значения

На рисунке 3.7.2 представлено распечатанное подробное решение численного метода Якоби.

```
The fully system solving solution of the Gauss-Seidel iterative method.
Checking the dimensions of the input matrix and vector...
The dimensions of the input data correspond to each other.
Checking the sufficient condition for the convergence of the Seidel method:
needed diagonal dominance of the input matrix A...
Sufficient condition is satisfied.
Input values of the system matrix A are: elems=[
    115.0   -20.0   -75.0
    15.0    -50.0   -5.0
    6.0     2.0     20.0
].
The dimension of the system is 3x3.
The vector B of the right-hand side of the equations values are: elems=[20.0    -40.0   28.0].
Initial approximation vector values are: elems=[1.0     1.0     1.0]
The Seidel algorithm will run until precision eps = 0.01 is reached.
The new vector of the approximate solution will be recalculated until the norm is still smaller than eps.
Calculated X0 = 0.0 on the 1 iteration.
Calculated X1 = 0.0 on the 1 iteration.
Calculated X2 = 0.0 on the 1 iteration.
Calculated norm value = 3.0.
The new vector of the approximate solution will be recalculated until the norm is still smaller than eps.
Calculated X0 = 1.0 on the 2 iteration.
Calculated X1 = 1.0 on the 2 iteration.
Calculated X2 = 1.0 on the 2 iteration.
Calculated norm value = 0.0.
The norm is smaller than eps, required precision has achieved on the 2 iteration.
The approximate solution vector is elems=[1.0   1.0     1.0].
```

Рисунок 3.7.2 – Пример сформированного решения в виде строки по умолчанию

Такую строку с подробным решением можно использовать для разработки онлайн-калькуляторов или Android-приложений, просто подставляя в нужное место View полученную строку с решением. Если потребуется, то можно вручную выполнить лексический анализ строки решения и внести необходимые изменения.

Сформированная строка решения по умолчанию содержит недостаток: если пользователю захочется изменить информативный текст строки и выводить значения в каком-то другом визуальном виде, то ему придется выполнить синтаксический анализ строки и извлечь из нее значения. Если пользователю не обязательно настраивать визуальный вид, то данное решение является хорошим, поскольку можно просто подставлять строку в нужное View. Но лучшим решением было бы предоставить возможность пользователю извлекать данные решения более удобным способом, например, из XML или JSON файла или же из объекта, в котором будет сохраняться пересчитываемое состояние алгоритма. Реализация данной возможности является одним из направлений дальнейшего развития библиотеки и ее планируется реализовать в рамках дипломного проекта.

3.8. Использование преимущественно классических массивов в работе алгоритмов

Хотя в библиотеке и реализованы различные утилитарные классы, такие как Matrix и Vector, в реализации алгоритмов численных методов было принято решение использовать классические массивы, встроенные и оптимизированные в Kotlin, для достижения максимальной производительности. Таким образом исключаются различные недостатки, связанные с накоплением лишних ссылок в памяти, фреймами вызовов методов и т.д. на стеке JVM.

Еще одной предпосылкой для отказа от использования лишних вспомогательных классов, например, Matrix и Vector, стало продуманное заранее обстоятельство, заключающееся в том, что API классов Matrix и Vector может в будущем измениться, а также при импортировании разрабатываемой библиотеки в состав некоторой другой, от реализованных вспомогательных классов возможно придется отказаться т.к. с большой долью вероятности они уже будут реализованы в другой библиотеке, возможно с другой логикой и функционалом.

3.9. Копирование входных данных с целью предотвращения изменения внешнего окружения работой численных методов

В Kotlin все является объектом, в нем нет примитивных типов, а также как и в Java, все объекты передаются по ссылке, хотя сами ссылки все же передаются по значению т.е. копируются, но указывают на один и тот же участок памяти. Данная особенность языка приводит к тому, что при передаче некоторого объекта в параметр функции, функция, работая с этим параметром может изменить его и это изменение отразится на переданном объекте вне функции т.к. его копирования не происходит. Эту особенность языка пришлось обязательно учесть при реализации численных методов. Для избегания аномалии изменения внешнего окружения, каждый реализованный численный метод перед своей основной работой производит полное копирование входных параметров, а алгоритм работает с копиями входных параметров. Таким образом предотвращается побочный эффект изменения внешнего окружения, но с дополнительной затратой на производительность.

На рисунке 3.9.1 представлен пример полного копирования входных массивов с помощью использования метода clone.

```
val A: Array<Array<Double>> = inputA.map { it.clone() }.toTypedArray()
val B: Array<Double> = inputB.clone()
```

Рисунок 3.9.1 – Осуществление копирования входных данных в численный метод Зейделя

3.10. Структура проекта

Ниже представлена файловая структура проекта.

Основной файл build.gradle, содержит в себе необходимые зависимости и параметры конфигурации проекта.

Иерархия пакетов содержит ряд пакетов:

- util – в этом пакете подразумевается хранение «дополнительных» классов и функций, которые могут быть полезны для реализации численных методов, такие как транспонирование матрицы, умножение матриц и другие;
- integralmethods – этом пакете размещаются реализации численных методов для интегрирования;
- systemsolvingmethods – в этом пакете размещаются реализации численных методов для решения систем алгебраический уравнений;
- othermethods – в этом пакете подразумевается размещение отдельных методов, не подходящие под классификацию для хранения в других пакетах.

Файловая структура проекта была спроектирована таким образом, чтобы в будущем было удобно добавлять другие пакеты, в которых планируется хранение численных методов для реализации других задач, например, методы поиска экстремумов. Таким образом, в будущем библиотеку будет удобно развивать и расширять. Структура основных пакетов представлена на рисунке 3.10.1.

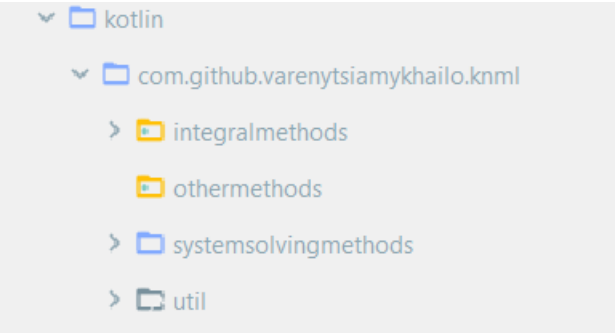


Рисунок 3.10.1 – Структура основных пакетов проекта

В отдельном каталоге test размещаются реализованные юнит-тесты, в соответствующих пакетах.

4. Тестирование библиотеки

4.1. Модульное тестирование (Юнит-тестирование)

Модульное тестирование (англ. Unit Testing) – это тип тестирования программного обеспечения, при котором тестируются отдельные модули или компоненты программного обеспечения. Его цель заключается в том, чтобы проверить, что каждая единица программного кода работает корректно. Данный вид тестирования выполняется разработчиками на этапе разработки приложения. Модульные тесты изолируют часть кода и проверяют его работоспособность. Качественное модульное тестирование на этапе разработки экономит время и средства выделенные на разработку, т.к. позволяет вовремя выявить логические ошибки кода при его дальнейшей модернизации и развитии.

Логика тестирования всех реализованных численных методов похожа и подразумевает реализацию следующих функций-тестов:

- функции, контролирующие правильность полученных результатов в результате работы какого-то конкретного численного метода;
- функции, контролирующие факт того, что работа конкретного численного метода не изменяет внешнее окружение (объекты, переданные в качестве параметра функции);
- функции, которые тестируют работу численных методов на неправильных входных данных и убеждаются в правильности генерируемых объектов-исключений для конкретных ошибок, с соответствующими описаниями. Например, при передаче в численный метод входных данных, не удовлетворяющих необходимому условию работы метода, должен генерироваться объект-исключение с соответствующим, осмысленным описанием.

Соответствующие тестируемые функции реализовывались в соответствующих классах для каждого разрабатываемого численного метода, с использованием фреймворка юнит-тестирования JUnit 5.8.1. Данные тесты выполняются автоматически при попытке скомпилировать библиотеку. Если какой-то тест не выполняется успешно, то компиляция не осуществляется, а разработчик с помощью специального инструмента может посмотреть статистику по проведенному тестированию и выявить проблемные участки кода. Таким образом, продукт частично застрахован от непредусмотренных появлений ошибок в работе кода, вследствие его модификации и развития. Программные продукты, разрабатываемые с помощью принципа «разработка через тестирование» считаются более надежными.

Дополнительным преимуществом наличия юнит-текстов к классам в проекте является тот факт, что юнит-тесты могут содержать демонстрацию использования API библиотеки. По реализованным юнит-тестам новый

разработчик проекта или пользователь может ознакомиться с API библиотеки и использовать юнит-тесты в качестве примеров для вызова определенных методов библиотеки.

На рисунке 4.1.1 представлена структура классов, содержащих юнит-тесты. В каждом классе содержится около 5-10 реализованных тест-функций.

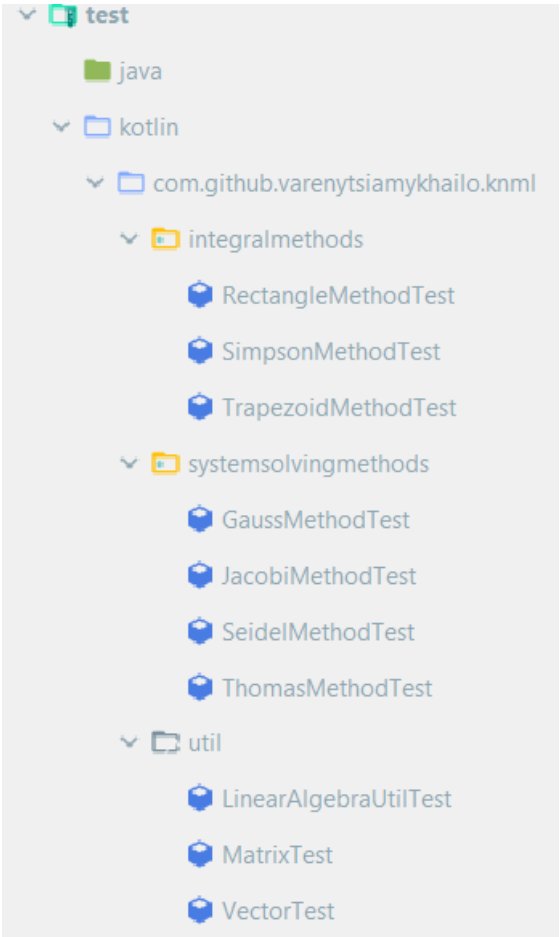


Рисунок 4.1.1 – Структура классов, осуществляющих юнит-тестирование

4.2. Профилирование методов и тестирование производительности

Профилирование — это сбор характеристик программы во время ее выполнения. При профилировании замеряется время выполнения и количество вызовов отдельных функций численных методов. При помощи этого инструмента разработчик может найти наиболее медленные участки кода и провести их оптимизацию.

Профилирование является одним из способов провести тестирование производительности разработанных численных методов. Для профилирования

численных методов был разработан программный компонент, который вызывает конкретный тестируемый численный метод на различных объемах входных данных и в качестве результата тестирования выводит график среднего времени работы тестируемого метода для конкретного объема входных данных. Данный график так же выводит асимптотическую сложность протестированного метода, что позволяет сравнить ее с теоретически известными данными об асимптотике тех или иных численных методов и убедиться в правильности их имплементации.

Алгоритм работы программы-профилировщика:

1) выбирается конкретный тестируемый численный метод;
2) для соответствующего метода генерируется набор входных данных, начиная с малого размера, разработчик прописывает логику генерации входных данных для каждого численного метода отдельно, т.к. каждый метод обладает своими условиями сходимости;
3) метод запускается много раз на определенном размере входных данных и подсчитывается его среднее время выполнения, а результаты запоминаются;
4) генерируется новый набор входных данных, в два раза больше объема предыдущего набора;
5) возврат к шагу 3, генерация и тестирование нового большего объема данных выполняется до тех пор, пока время работы метода не превысит определенную отметку времени;
6) по полученным данным строится график зависимости времени работы метода от объема входных данных, позволяющий оценить его производительность, а также изложить в документации рекомендуемый максимальный объем входных данных для конкретного численного метода.

Было решено останавливать профилирование метода, когда его работа на конкретном объеме данных превышает одну секунду. На рисунке 4.2.1 представлен результат профилирования метода Якоби в виде графика.

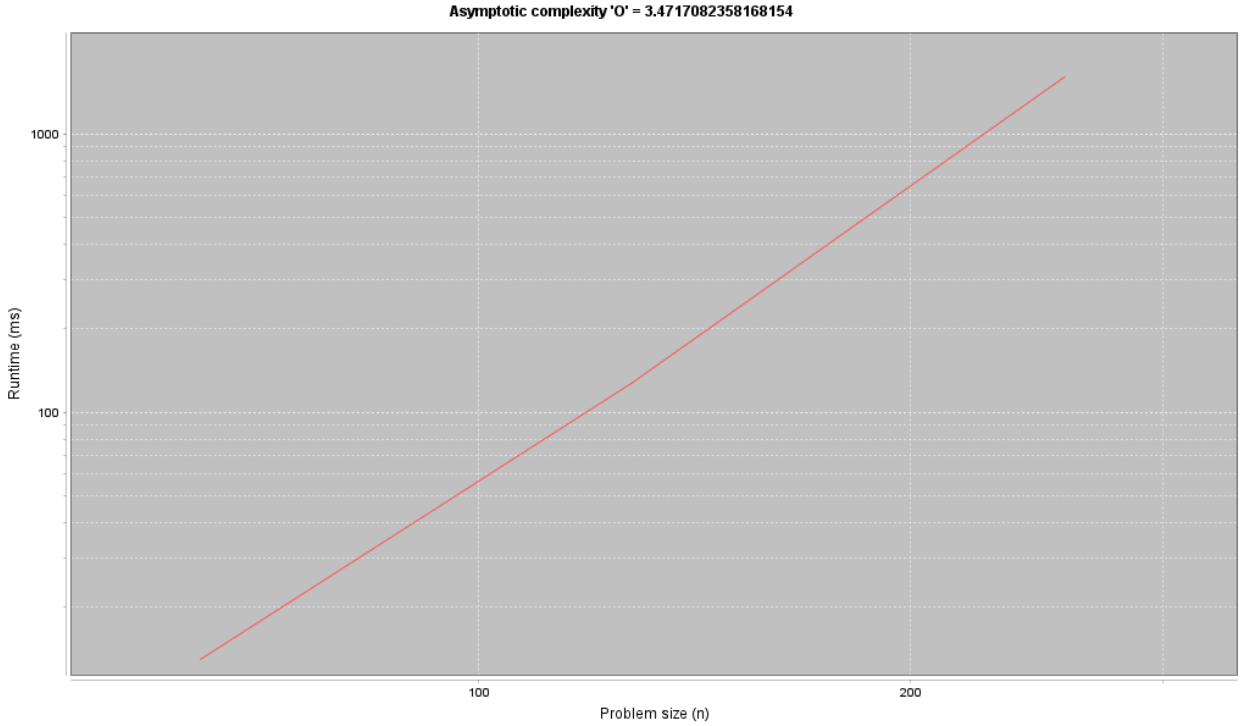


Рисунок 4.2.1 – Результаты профилирования метода Якоби

На графике видна зависимость размера входных данных, в данном случае квадратной матрицы, и времени выполнения метода на конкретном объеме данных. Тангенс угла наклона кривой указывает на асимптотическую сложность работы метода, ее значение выводится сверху графика. Стоит отметить, что результаты приведенной асимптотической сложности являются примерными и имеют погрешность, связанную с аппаратными особенностями машины или, например, с работой сборщика мусора в JVM, на которой выполняется тестирование. В первом запуске результирующее значение может быть 2.8, а в следующем 3.4, на одних и тех же входных данных, поэтому следует округлять полученное значение до целого, ближайшего к истине. Из практических наблюдений установлено, что погрешность варьируется от -0.8 до 0.8, от истинного значения. Если бы алгоритм численного метода был реализован в корне неверно, то отклонение было бы явно большим и заметным от ожидаемого. Кроме того, полученная асимптотическая сложность сравнивается с предсказанной асимптотической сложностью, полученной в рамках ручного теоретического анализа кода алгоритма.

4.3. Теоретический анализ кода алгоритмов

Прогноз асимптотической сложности реализуемого алгоритма можно получить с помощью теоретического анализа кода алгоритма. Самым

популярным способом такой оценки является анализ иерархии вложенности циклов, лежащих в основе работы алгоритма. Например, если в основе работы алгоритма лежит цикл тройной вложенности, и количество итераций каждого из вложенного цикла зависит от размера входных данных, то вероятнее всего, что асимптотическая сложность алгоритма кубическая. При этом важно не забывать о теоремах сложения и умножения операций, представленных на рисунке 4.3.1.

Если $f \in O(n)$ и $g \in O(1)$, то $f + g \in O(n)$.

Если $f \in O(n)$ и $g \in O(n)$, то $f + g \in O(n)$.

Если $f \in O(n)$ и k — константа, то $kf \in O(n)$.

Если $f \in O(n)$, то $nf \in O(n^2)$.

Рисунок 4.3.1 – Теоремы сложения и умножения операций для оценки асимптотической сложности алгоритма

Полученные данные в рамках профилирования и теоретического анализа сравниваются между собой, а также с теоретически известными науке данными об асимптотике конкретного метода, затем производится вывод об успешности реализации метода.

5. Реализация итерационного метода Якоби для решения СЛАУ

5.1. Основная идея алгоритма

Метод Якоби [9.40] – итерационный метод решения СЛАУ. Под итерационным алгоритмом понимается алгоритм, в работе которого результирующие значения пересчитываются на основе предыдущих результатов итераций до тех пор, пока не будет достигнута определенная точность.

Асимптотическая сложность алгоритма равна $O(n^3)$.

5.2. Детали реализации

Так как данный алгоритм относится к классу алгоритмов для решения СЛАУ, то он размещается в подпакете systemsolvingmethods. Реализация метода Якоби описана в классе JacobiMethod.kt. Класс содержит два перегруженных API-метода для прикладного пользователя, с описанной на KDoc документацией, со следующими сигнатурами (см. рисунок 5.2.1):

```
fun solveSystemByJacobiMethod(
    inputA: Array<Array<Double>>,
    inputB: Array<Double>,
    initialApproximation: Array<Double>? = null,
    eps: Double? = null,
    formSolution: Boolean = false
): VectorResultWithStatus {
```

```
fun solveSystemByJacobiMethod(
    inputA: Matrix,
    inputB: Vector,
    initialApproximation: Vector? = null,
    eps: Double? = null,
    formSolution: Boolean = false
): VectorResultWithStatus {
```

Рисунок 5.2.1 – Сигнатуры API функций для метода Якоби

Приведенные на изображениях методы позволяют передавать в качестве входных данных классические массивы или же объекты классов Vector и Matrix, и требуемую точность eps, значением по умолчанию которой является машинная точность. В качестве возвращаемого значения возвращается объект класса VectorResultWithStatus.

Дополнительно класс JacobiMethod.kt содержит:

– private-метод, в котором осуществляется основная работа алгоритма и формирование текста подробного решения, данный метод также осуществляет копирование входных данных и проверку на их

корректность: размерность входных данных и соответствие достаточному условию сходимости;
- private-метод вычисляющий норму, которая контролирует точность полученного решения на определенном этапе итерации;
- private-метод для генерации значения по умолчанию для параметра initialApproximation: возвращает начальное приближение на основе переданного пользователем вектора B.

5.3. Модульное тестирование (Юнит-тестирование)

Юнит-тесты для данного тестируемого класса описаны в классе JacobiMethodTest.kt и содержат тестирующие функции согласно логике, описанной в главе 4.1.

5.4. Профилирование метода, тестирование производительности

Результаты профилирования метода представлены на рисунке 5.4.1:

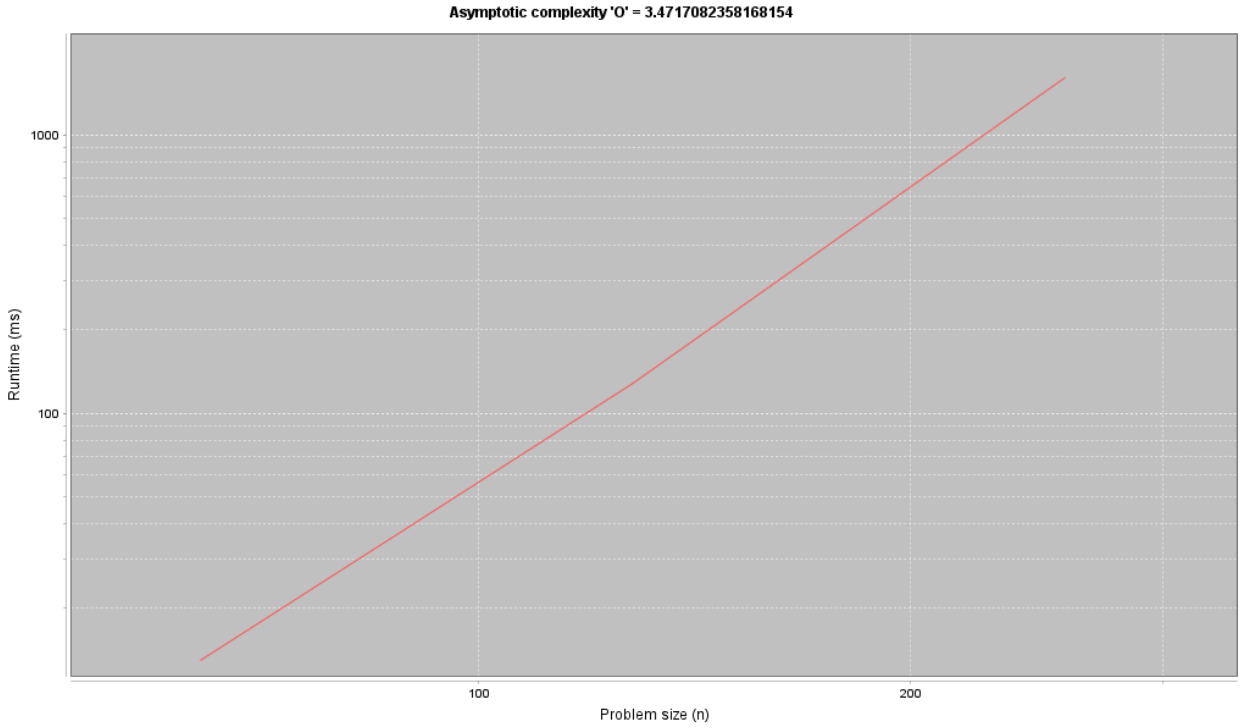


Рисунок 5.4.1 – Результаты профилирования метода Якоби

Согласно полученным результатам, асимптотическая сложность реализованного метода Якоби с высокой долью вероятности является

кубической $O(n^3)$. При решении системы, состоящей более чем из 200 уравнений, метод работает более одной секунды. Таким образом, данным методом рекомендуется воспользоваться, если СЛАУ не превышает 200 уравнений, иначе время ожидания будет ощутимым.

5.5. Теоретический анализ кода алгоритма

В основе алгоритма лежат циклы тройной вложенности, каждый из которых непрерывно зависит от размера входных данных. Таким образом, с теоретической точки зрения асимптотическую сложность алгоритма метода Якоби можно оценить, как кубическую, что подтверждает полученные результаты на этапе профилирования и соответствует научной теоретической оценке.

Полученные данные об асимптотической сложности реализованного метода Якоби совпадают с теоретическими научными оценками, что позволяет говорить об успешной реализации метода.

6. Реализация итерационного метода Зейделя для решения СЛАУ

6.1. Основная идея алгоритма

Метод Зейделя [9.41] – итерационный метод решения СЛАУ, являющийся модернизацией метода Якоби. В отличии от метода Якоби, метод Зейделя требует значительно меньше итераций. Под итерационным алгоритмом понимается алгоритм, в работе которого результирующие значения пересчитываются на основе предыдущих результатов итераций до тех пор, пока не будет достигнута определенная точность.

Асимптотическая сложность алгоритма равна $O(n^3)$.

6.2. Детали реализации

Так как данный алгоритм относится к классу алгоритмов для решения СЛАУ, то он размещается в подпакете systemsolvingmethods. Реализация метода Зейделя описана в классе SeidelMethod.kt. Класс содержит два перегруженных API-метода для прикладного пользователя, с описанной на KDoc документацией, со следующими сигнатурами (см. рисунок 6.2.1):

```
fun solveSystemBySeidelMethod(
    inputA: Array<Array<Double>>,
    inputB: Array<Double>,
    initialApproximation: Array<Double>? = null,
    eps: Double? = null,
    formSolution: Boolean = false
): VectorResultWithStatus {
```

```
fun solveSystemBySeidelMethod(
    inputA: Matrix,
    inputB: Vector,
    initialApproximation: Vector? = null,
    eps: Double? = null,
    formSolution: Boolean = false
): VectorResultWithStatus {
```

Рисунок 6.2.1 – Сигнатуры API функций для метода Зейделя

Приведенные на изображениях методы позволяют передавать в качестве входных данных классические массивы или же объекты классов Vector и Matrix, и требуемую точность eps, значением по умолчанию которой является машинная точность. В качестве возвращаемого значения возвращается объект класса VectorResultWithStatus.

Дополнительно класс SeidelMethod.kt содержит:

– private-метод, в котором осуществляется основная работа алгоритма и формирование текста подробного решения, данный метод также осуществляет копирование входных данных и проверку на их

корректность: размерность входных данных и соответствие достаточному условию сходимости;
- private-метод вычисляющий норму, которая контролирует точность полученного решения на определенном этапе итерации;
- private-метод для генерации значения по умолчанию для параметра initialApproximation: возвращает начальное приближение на основе переданного пользователем вектора B.

6.3. Модульное тестирование (Юнит-тестирование)

Юнит-тесты для данного тестируемого класса описаны в классе SeidelMethodTest.kt и содержат тестирующие функции согласно логике, описанной в главе 4.1.

6.4. Профилирование метода, тестирование производительности

Результаты профилирования метода представлены на рисунке 6.4.1:

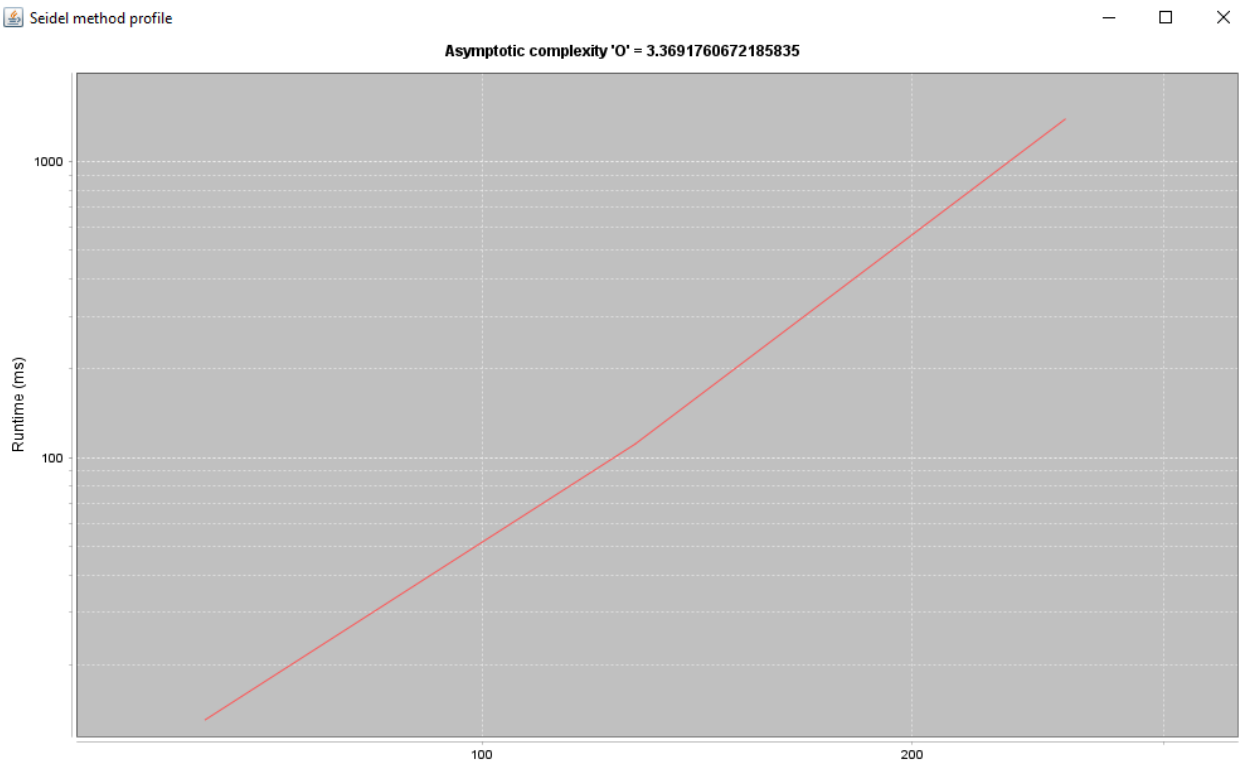


Рисунок 6.4.1 – Результаты профилирования метода Зейделя

Согласно полученным результатам, асимптотическая сложность реализованного метода Зейделя с высокой долью вероятности является кубической $O(n^3)$. При решении системы, состоящей более чем из 200 уравнений, метод работает более одной секунды. Таким образом, данным методом рекомендуется воспользоваться, если СЛАУ не превышает 200 уравнений, иначе время ожидания будет ощутимым.

6.5. Теоретический анализ кода алгоритма

В основе алгоритма лежат циклы тройной вложенности, каждый из которых непрерывно зависит от размера входных данных. Таким образом, с теоретической точки зрения асимптотическую сложность алгоритма метода Зейделя можно оценить, как кубическую, что подтверждает полученные результаты на этапе профилирования и соответствует научной теоретической оценке.

Полученные данные об асимптотической сложности реализованного метода Зейделя совпадают с теоретическими научными оценками, что позволяет говорить об успешной реализации метода.

7. Реализация метода Томаса (прогонки или алгоритм трехдиагональной матрицы) для решения СЛАУ

7.1. Основная идея алгоритма

Метод прогонки (tridiagonal matrix algorithm) или алгоритм Томаса (англ. Thomas algorithm) [9.31] используется для решения систем линейных уравнений вида $Ax=F$, где A — трёхдиагональная матрица. Данный метод представляет собой вариант метода последовательного исключения неизвестных.

A – трехдиагональная матрица, если все элементы стоящие не на главной и смежных с ней диагоналях равны нулю.

Данный метод является частным случаем метода Гаусса и, как и метод Гаусса, состоит из прямого и обратного хода. В отличии от метода Гаусса, асимптотическая сложность которого $O(n^3)$, асимптотическая сложность метода прогонки равна $O(n)$.

7.2. Детали реализации

Так как данный алгоритм относится к классу алгоритмов для решения СЛАУ, то он размещается в подпакете systemsolvingmethods. Реализация метода прогонки описана в классе ThomasMethod.kt. Класс содержит два перегруженных API-метода для прикладного пользователя, с описанной на KDoc документацией, со следующими сигнатурами (см. рисунок 7.2.1):

```
fun solveSystemByThomasMethod(
    inputA: Array<Array<Double>>,
    inputB: Array<Double>,
    formSolution: Boolean = false,
    increasePerformanceByIgnoringInputDataChecking: Boolean = false
): VectorResultWithStatus {

fun solveSystemByThomasMethod(
    inputA: Matrix,
    inputB: Vector,
    formSolution: Boolean = false,
    increasePerformanceByIgnoringInputDataChecking: Boolean = false
): VectorResultWithStatus {
```

Рисунок 7.2.1 – Сигнатуры API функций для метода прогонки

Приведенные на изображениях методы позволяют передавать в качестве входных данных классические массивы или же объекты классов Vector и Matrix. В качестве возвращаемого значения возвращается объект класса VectorResultWithStatus.

Дополнительно класс ThomasMethod.kt содержит private-метод, в котором осуществляется основная работа алгоритма и формирование текста подробного решения. Данный метод также осуществляет копирование входных данных и проверку на их корректность: размерность входных данных и проверку на условие сходимости: матрица A должна быть трехдиагональной.

По умолчанию реализованный метод производит проверку над входной матрицей A и не запускает работу алгоритма, если матрица A не является трехдиагональной. Такая проверка приводит к тому, что асимптотическая сложность метода является $O(n^2)$ т.е. проверка на корректность входных данных требует больше операций, по сравнению с работой основного алгоритма, сложность которого $O(n)$. Поэтому для прикладных пользователей была предоставлена возможность отключить проверку на трехдиагональность матрицы A с помощью передаваемого флага “increasePerfomanceByIgnoringDataChacking” в качестве аргумента вызова метода. При активации данного флага, метод будет выполняться за $O(n)$, но пользователь должен быть уверен, что он передает действительно трехдиагональную матрицу, иначе возвращаемый объект типа VectorResultWithStatus скорее всего возвратит объект возникшего исключения в процессе работы алгоритма.

7.3. Модульное тестирование (Юнит-тестирование)

Юнит-тесты для данного тестируемого класса описаны в классе ThomasMethodTest.kt и содержат тестирующие функции согласно логике, описанной в главе 4.1.

7.4. Профилирование метода, тестирование производительности

Результаты профилирования метода с флагом “increasePerfomanceByIgnoringDataChacking = true” представлены на рисунке 7.4.1:

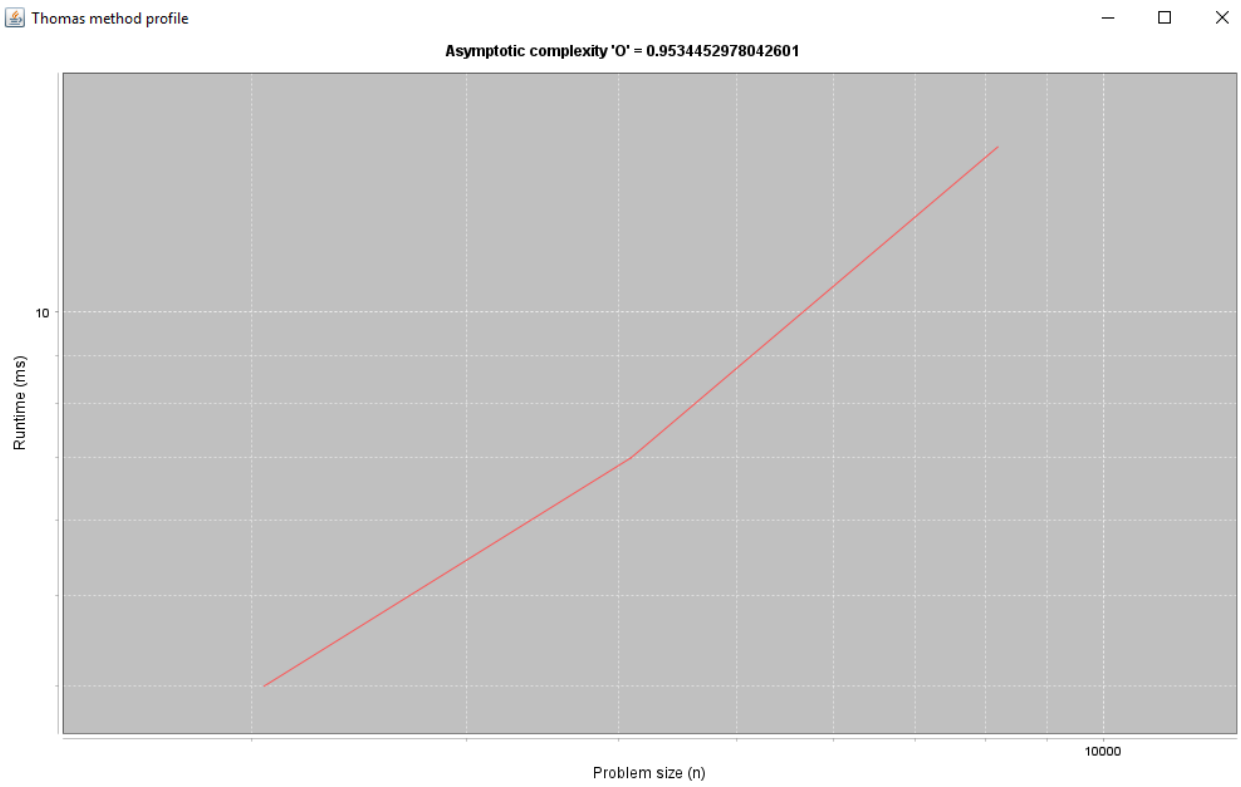


Рисунок 7.4.1 – Результаты профилирования метода прогонки

Согласно полученным результатам, асимптотическая сложность реализованного метода Томаса с высокой долью вероятности является линейной *O(n)*. Алгоритм справляется с системой размера 8192 x 8192 за 15 миллисекунд, что позволяет говорить об высокой эффективности данного алгоритма за счет его линейной асимптотической сложности, конечно если выполнено условие того, что матрица является трехдиаганальной.

7.5. Теоретический анализ кода алгоритма

В основе алгоритма лежат два цикла одинарной вложенности: для прямого хода и обратного, каждый из которых непрерывно зависит от размера входных данных. Последовательная комбинация выполнения двух циклов одинарной вложенности приводит к линейной асимптотической сложности. Таким образом, с теоретической точки зрения асимптотическую сложность алгоритма метода Томаса можно оценить, как линейную *O(n)*, что подтверждает полученные результаты на этапе профилирования и соответствует научной теоретической оценке.

Полученные данные об асимптотической сложности реализованного метода Томаса совпадают с теоретическими научными оценками, что позволяет говорить об успешной реализации метода.

8. Реализация метода Гаусса для решения СЛАУ

8.1. Основная идея алгоритма

Метод Гаусса [9.28] является классическим методом для решения СЛАУ, представляя собой метод последовательного исключения переменных.

Данный метод состоит из двух этапов:

- прямой ход: приведение СЛАУ к треугольному виду;
- обратный ход: вычисление неизвестных.

Асимптотическая сложность алгоритма равна $O(n^3)$.

8.2. Детали реализации

Так как данный алгоритм относится к классу алгоритмов для решения СЛАУ, то он размещается в подпакете systemsolvingmethods. Реализация метода Гаусса описана в классе GaussMethod.kt. Класс содержит два перегруженных API-метода для прикладного пользователя, с описанной на KDoc документацией, со следующими сигнатурами (см. рисунок 8.2.1):

```
fun solveSystemByGaussClassicMethod(
    inputA: Array<Array<Double>>,
    inputB: Array<Double>,
    formSolution: Boolean = false
): VectorResultWithStatus {
```

```
fun solveSystemByGaussClassicMethod(
    inputA: Matrix,
    inputB: Vector,
    formSolution: Boolean = false
): VectorResultWithStatus {
```

Рисунок 8.2.1 – Сигнатуры API функций для метода Гаусса

Приведенные на изображениях методы позволяют передавать в качестве входных данных классические массивы или же объекты классов Vector и Matrix. В качестве возвращаемого значения возвращается объект класса VectorResultWithStatus.

Дополнительно класс GaussMethod.kt содержит private-метод, в котором осуществляется основная работа алгоритма и формирование подробного решения. Данный метод также осуществляет копирование входных данных и проверку на их корректность.

8.3. Модульное тестирование (Юнит-тестирование)

Юнит-тесты для данного тестируемого класса описаны в классе GaussMethodTest.kt и содержат тестирующие функции согласно логике, описанной в главе 4.1.

8.4. Профилирование метода, тестирование производительности

Результаты профилирования метода представлены на рисунке 8.4.1:

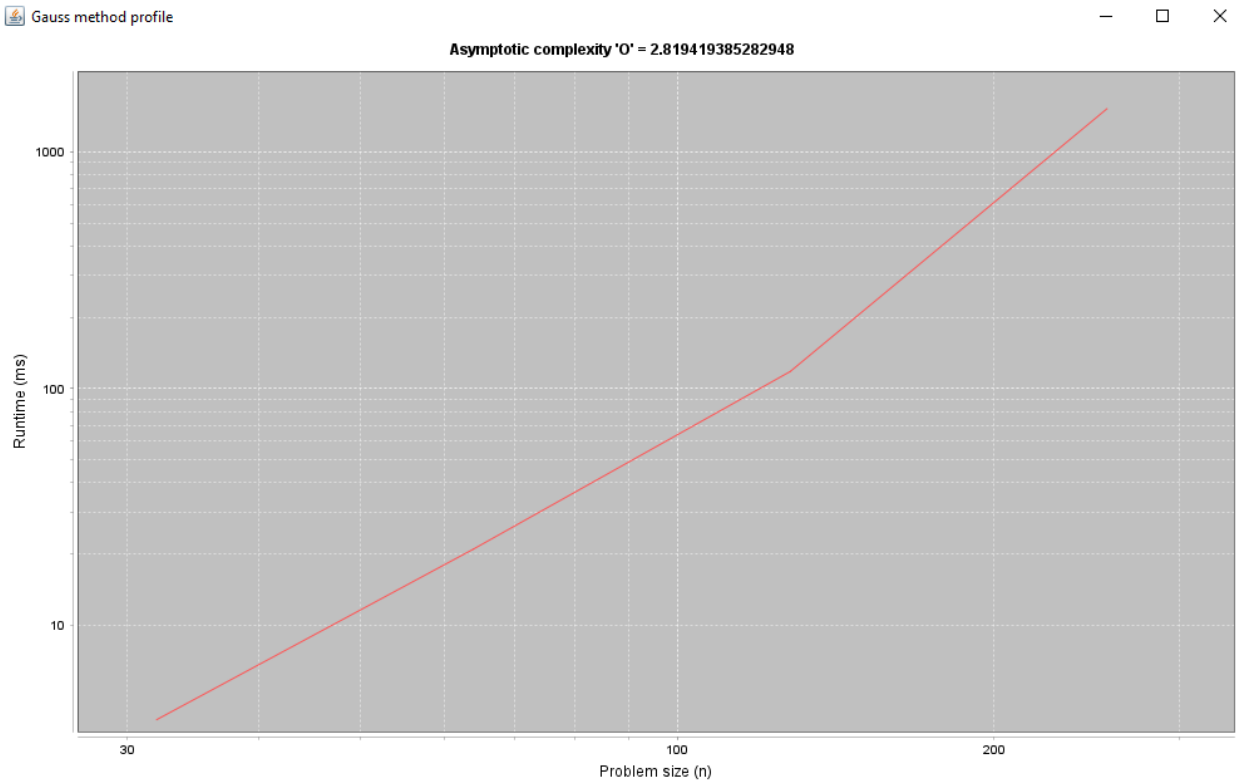


Рисунок 8.4.1 – Результаты профилирования метода Гаусса

Согласно полученным результатам, асимптотическая сложность реализованного метода Гаусса с высокой долью вероятности является кубической $O(n^3)$.

8.5. Теоретический анализ кода алгоритма

В основе алгоритма лежат циклы тройной вложенности, каждый из которых непрерывно зависит от размера входных данных. Таким образом, с теоретической точки зрения асимптотическую сложность алгоритма метода Гаусса можно оценить, как кубическую, что подтверждает полученные

результаты на этапе профилирования и соответствует научной теоретической оценке.

Полученные данные об асимптотической сложности реализованного метода Гаусса совпадают с теоретическими оценками, что позволяет говорить об успешной реализации метода.

9. Реализация метода средних прямоугольников для интегрирования одномерной функции

9.1. Основная идея алгоритма

Методы численного интегрирования [9.79] позволяют получать приближённое значение определённого интеграла от функции на отрезке с помощью разложения интеграла в ряд.

Метод разложения интеграла в ряд подразумевает разбиение отрезка интегрирования на k равных частей, на каждой из которой вычисляется приближённое значение интеграла – площадь соответствующего прямоугольника. Полученные значения суммируются, а результатом является приближенное значение интеграла (площадь фигуры) на отрезке. Соответственно, с увеличением числа отрезков разбиения увеличивается точность получаемого значения.

9.2. Алгоритм контроля точности, уточнение по Ричардсону

Для того, чтобы вычислять значения интегралов с заданной точностью с помощью метода средних прямоугольников был применен алгоритм аппроксимации Ричардсона.

Таким образом, задачу интегрирования можно решить с помощью комбинации методов средних прямоугольников и аппроксимации Ричардсона.

9.3. Детали реализации

Так как данный алгоритм относится к классу алгоритмов для интегрирования функций, то он размещается в подпакете integralmethods. Реализация метода средних прямоугольников описана в классе RectangleMethod.kt. Класс содержит API-метод для прикладного пользователя, с описанной на KDoc документацией, со следующей сигнатурой (см. рисунок 9.3.1):

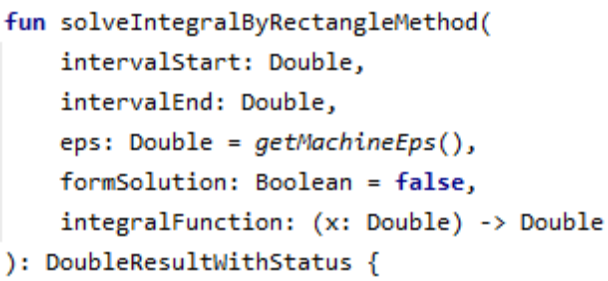


Рисунок 9.3.1 – Сигнатура API функций для метода интегрирования средними прямоугольниками

Приведенный на изображении метод позволяет передавать в качестве входных данных отрезок интегрирования, требуемую точность eps (значением по умолчанию которой является машинная точность), флаг для формирования объекта с подробным решением и функцию интегрирования. В качестве возвращаемого значения возвращается объект класса DoubleResultWithStatus.

Возвращаемый объект класса DoubleResultWithStatus в качестве подробного решения содержит объект класса RectangleMethodSolution, в котором содержится строка подробного решения по умолчанию, а также подсчитываемые значения в процессе работы алгоритма, в соответствующих полях.

Дополнительно класс RectangleMethod.kt содержит private-методы, в которых осуществляется работа алгоритмов и формирование объекта подробного решения, если задан соответствующий флаг.

9.4. Модульное тестирование (Юнит-тестирование)

Юнит-тесты для данного тестируемого класса описаны в классе RectangleMethodTest.kt и содержат тестирующие функции согласно логике, описанной в главе 4.1.

9.5. Профилирование метода, тестирование производительности

Результаты профилирования метода представлены на рисунке 9.5.1:

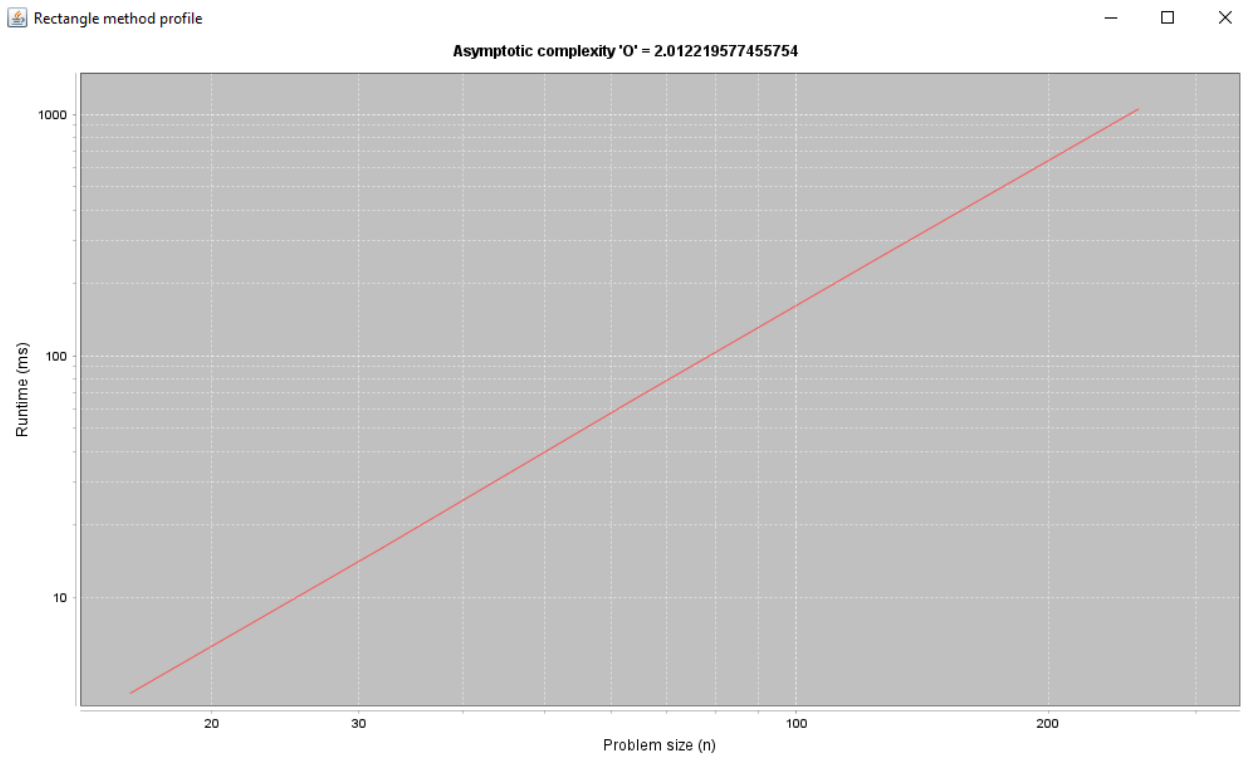


Рисунок 9.5.1 – Результаты профилирования метода интегрирования средними прямоугольниками

Для оценки асимптотической сложности алгоритма, в качестве размера задачи была выбрана длина отрезка интегрирования, которая удваивалась при каждом следующем замере времени выполнения.

Так как асимптотическая сложность алгоритма интегрирования зависит от типа интегрируемой функции, то для того, чтобы оценить сложность реализованного алгоритма, его следует проверять на интегрируемой функции, асимптотическая сложность вычисления которой является константной. В качестве такой функции была выбрана функция $f(x)=x$.

Согласно полученным результатам, асимптотическая сложность реализованного метода средних прямоугольников с высокой долью вероятности является квадратичной $O(n^2)$, в случае, если в качестве интегрируемой функции используется функция с константной асимптотической сложностью.

9.6. Теоретический анализ кода алгоритма

В основе алгоритма лежат циклы двойной вложенности:

- для алгоритма аппроксимации Ричардсона: используется один цикл и его асимптотическая сложность оценивается в линейную $O(n)$;

– для подсчета интеграла методом средних прямоугольников: используется один цикл и его асимптотическая сложность оценивается в линейную $O(n)$.

Каждый из которых непрерывно зависит от размера входных данных. Циклы двойной вложенности приводят к квадратичной $O(n^2)$ асимптотической сложности. Но внутри второго вложенного цикла вызывается передаваемая пользователем функция интегрирования, которая имеет свою уникальную асимптотическую сложность, которую необходимо скомбинировать с полученной квадратичной асимптотической сложностью для реализованного метода интегрирования. Таким образом, с теоретической точки зрения асимптотическую сложность алгоритма метода средних прямоугольников можно оценить, как $O(n^2 * T(f))$, где $T(f)$ – асимптотическая сложность подынтегральной функции. Полученный результат подтверждает полученные результаты на этапе профилирования, в котором использовалась подынтегральная функция с константной асимптотической сложностью.

Полученные данные об асимптотической сложности реализованного метода средних прямоугольников совпадают с теоретическими оценками, что позволяет говорить об успешной реализации метода.

10. Реализация метода трапеций для интегрирования одномерной функции

10.1. Основная идея алгоритма

Методы численного интегрирования позволяют получать приближённое значение определённого интеграла от функции на отрезке с помощью разложения интеграла в ряд.

Метод разложения интеграла в ряд подразумевает разбиение отрезка интегрирования на k равных частей, на каждой из которой вычисляется приближённое значение интеграла – площадь соответствующей трапеции. Полученные значения суммируются, а результатом является приближенное значение интеграла (площадь фигуры) на отрезке. Соответственно, с увеличением числа отрезков разбиения увеличивается точность получаемого значения.

10.2. Алгоритм контроля точности, уточнение по Ричардсону

Для того, чтобы вычислять значения интегралов с заданной точностью с помощью метода трапеций был применен алгоритм аппроксимации Ричардсона.

Таким образом, задачу интегрирования можно решить с помощью комбинации методов трапеций и аппроксимации Ричардсона.

10.3. Детали реализации

Так как данный алгоритм относится к классу алгоритмов для интегрирования функций, то он размещается в подпакете integralmethods. Реализация метода трапеций описана в классе TrapezoidMethod.kt. Класс содержит API-метод для прикладного пользователя, с описанной на KDoc документацией, со следующей сигнатурой (см. рисунок 10.3.1):

```
fun solveIntegralByTrapezoidMethod(
    intervalStart: Double,
    intervalEnd: Double,
    eps: Double = getMachineEps(),
    formSolution: Boolean = false,
    integralFunction: (x: Double) -> Double
): DoubleResultWithStatus {
```

Рисунок 10.3.1 – Сигнатура API функций для метода интегрирования трапециями

Приведенный на изображении метод позволяет передавать в качестве входных данных отрезок интегрирования, требуемую точность eps (значением по умолчанию которой является машинная точность), флаг для формирования объекта с подробным решением и функцию интегрирования. В качестве возвращаемого значения возвращается объект класса DoubleResultWithStatus.

Возвращаемый объект класса DoubleResultWithStatus в качестве подробного решения содержит объект класса TrapezoidMethodSolution, в котором содержится строка подробного решения по умолчанию, а также подсчитываемые значения в процессе работы алгоритма, в соответствующих полях.

Дополнительно класс TrapezoidMethod.kt содержит private-методы, в которых осуществляется работа алгоритмов и формирование объекта подробного решения, если задан соответствующий флаг.

10.4. Модульное тестирование (Юнит-тестирование)

Юнит-тесты для данного тестируемого класса описаны в классе TrapezoidMethodTest.kt и содержат тестирующие функции согласно логике, описанной в главе 4.1.

10.5. Профилирование метода, тестирование производительности

Результаты профилирования метода представлены на рисунке 10.5.1:

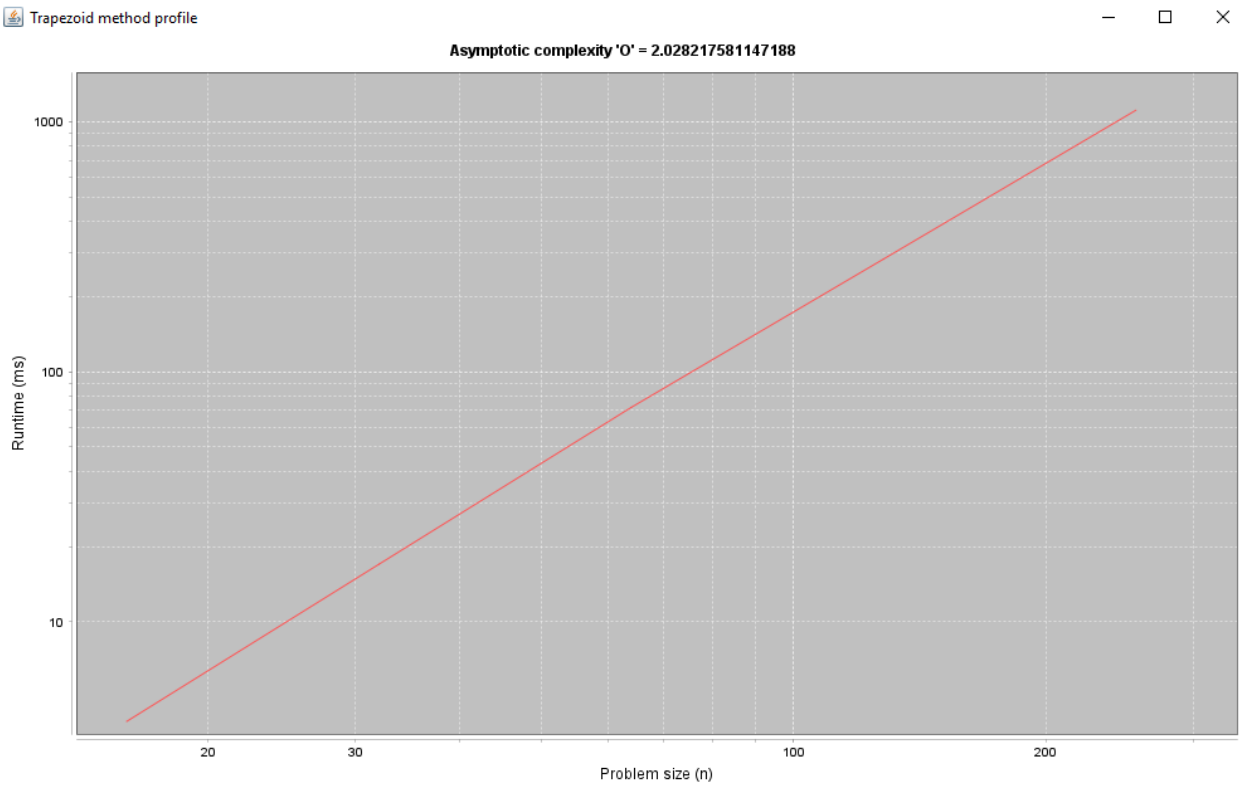


Рисунок 10.5.1 – Результаты профилирования метода интегрирования трапециями

Для оценки асимптотической сложности алгоритма, в качестве размера задачи была выбрана длина отрезка интегрирования, которая удваивалась при каждом следующем замере времени выполнения.

Так как асимптотическая сложность алгоритма интегрирования зависит от типа интегрируемой функции, то для того, чтобы оценить сложность реализованного алгоритма, его следует проверять на интегрируемой функции, асимптотическая сложность вычисления которой является константной. В качестве такой функции была выбрана функция $f(x)=x$.

Согласно полученным результатам, асимптотическая сложность реализованного метода трапеций с высокой долью вероятности является квадратичной $O(n^2)$, в случае, если в качестве интегрируемой функции используется функция с константной асимптотической сложностью.

10.6. Теоретический анализ кода алгоритма

В основе алгоритма лежат циклы двойной вложенности:

– для алгоритма аппроксимации Ричардсона: используется один цикл и его асимптотическая сложность оценивается в линейную $O(n)$;

– для подсчета интеграла методом трапеций: используется один цикл и его асимптотическая сложность оценивается в линейную *O(n)*.

Каждый из которых непрерывно зависит от размера входных данных. Циклы двойной вложенности приводят к квадратичной $O(n^2)$ асимптотической сложности. Но внутри второго вложенного цикла вызывается передаваемая пользователем функция интегрирования, которая имеет свою уникальную асимптотическую сложность, которую необходимо скомбинировать с полученной квадратичной асимптотической сложностью для реализованного метода интегрирования. Таким образом, с теоретической точки зрения асимптотическую сложность алгоритма метода трапеций можно оценить, как $O(n^2 * T(f))$, где $T(f)$ – асимптотическая сложность подынтегральной функции. Полученный результат подтверждает полученные результаты на этапе профилирования, в котором использовалась подынтегральная функция с константной асимптотической сложностью.

Полученные данные об асимптотической сложности реализованного метода трапеций совпадают с теоретическими оценками, что позволяет говорить об успешной реализации метода.

11. Реализация метода Симпсона для интегрирования одномерной функции

11.1. Основная идея алгоритма

Методы численного интегрирования позволяют получать приближённое значение определённого интеграла от функции на отрезке с помощью разложения интеграла в ряд.

Метод разложения интеграла в ряд подразумевает разбиение отрезка интегрирования на k равных частей, на каждой из которой вычисляется приближённое значение интеграла – площадь соответствующей кривой трапеции. Полученные значения суммируются, а результатом является приближенное значение интеграла (площадь фигуры) на отрезке. Соответственно, с увеличением числа отрезков разбиения увеличивается точность получаемого значения.

Метод Симпсона основан на приближении некоторой кривой интерполяционным многочленом второй степени к графику исходной функции на каждом отрезке разбиения.

11.2. Алгоритм контроля точности, уточнение по Ричардсону

Для того, чтобы вычислять значения интегралов с заданной точностью с помощью метода Симпсона был применен алгоритм аппроксимации Ричардсона, для которого необходимо пересчитывать в цикле следующее значение.

Таким образом, задачу интегрирования можно решить с помощью комбинации методов Симпсона и аппроксимации Ричардсона.

11.3. Детали реализации

Так как данный алгоритм относится к классу алгоритмов для интегрирования функций, то он размещается в подпакете integralmethods. Реализация метода Симпсона описана в классе SimpsonMethod.kt. Класс содержит API-метод для прикладного пользователя, с описанной на KDoc документацией, со следующей сигнатурой (см. рисунок 11.3.1):

```
fun solveIntegralBySimpsonMethod(
    intervalStart: Double,
    intervalEnd: Double,
    eps: Double = getMachineEps(),
    formSolution: Boolean = false,
    integralFunction: (x: Double) -> Double
): DoubleResultWithStatus {
```

Рисунок 11.3.1 – Сигнатура API функций для метода интегрирования трапециями

Приведенный на изображении метод позволяет передавать в качестве входных данных отрезок интегрирования, требуемую точность eps, значением по умолчанию которой является машинная точность, флаг для формирования объекта с подробным решением и функцию интегрирования. В качестве возвращаемого значения возвращается объект класса DoubleResultWithStatus.

Возвращаемый объект класса DoubleResultWithStatus в качестве подробного решения содержит объект класса SimpsonMethodSolution, в котором содержится строка подробного решения по умолчанию, а также подсчитываемые значения в процессе работы алгоритма, в соответствующих полях.

Дополнительно класс SimpsonMethod.kt содержит private-методы, в которых осуществляется работа алгоритмов и формирование объекта подробного решения, если задан соответствующий флаг.

11.4. Модульное тестирование (Юнит-тестирование)

Юнит-тесты для данного тестируемого класса описаны в классе SimpsonMethodTest.kt и содержат тестирующие функции согласно логике, описанной в главе 4.1.

11.5. Профилирование метода, тестирование производительности

Результаты профилирования метода представлены на рисунке 11.5.1:

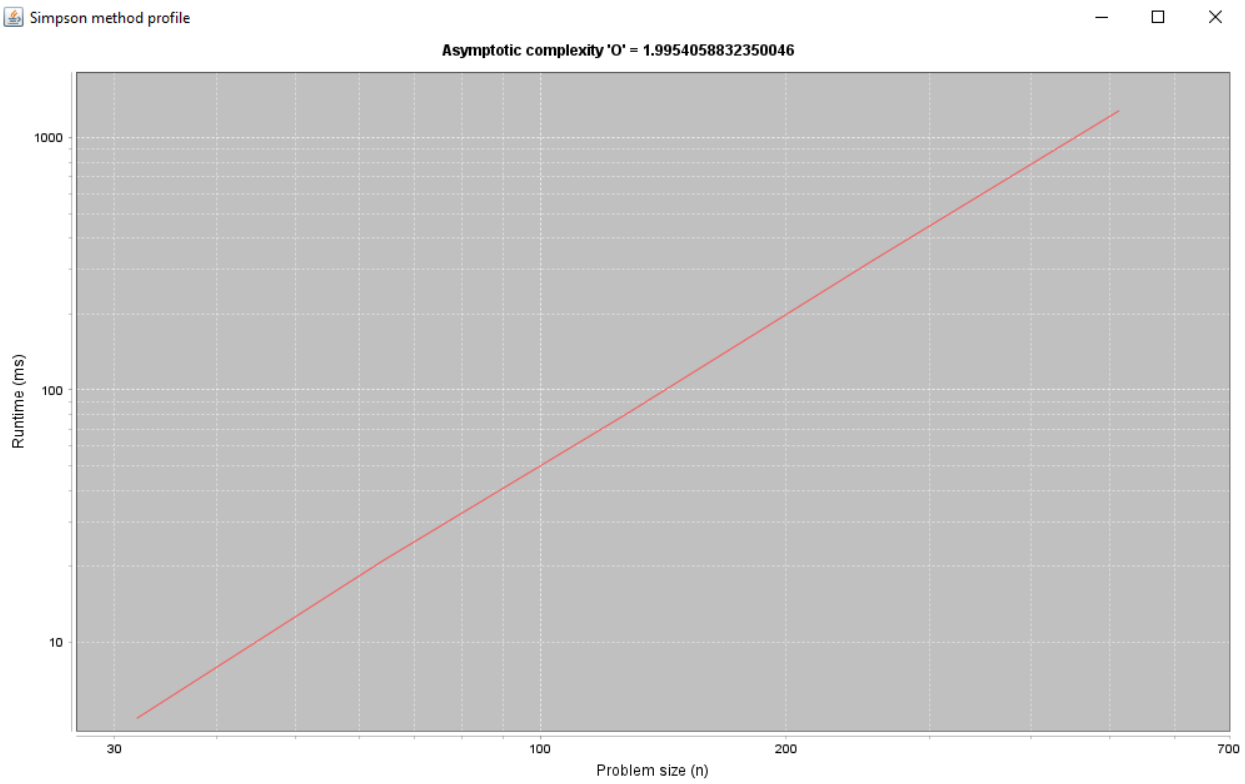


Рисунок 11.5.1 – Результаты профилирования метода интегрирования трапециями

Для оценки асимптотической сложности алгоритма, в качестве размера задачи была выбрана длина отрезка интегрирования, которая удваивалась при каждом следующем замере времени выполнения.

Так как асимптотическая сложность алгоритма интегрирования зависит от типа интегрируемой функции, то для того, чтобы оценить сложность реализованного алгоритма, его следует проверять на интегрируемой функции, асимптотическая сложность вычисления которой является константной. В качестве такой функции была выбрана функция $f(x)=x$.

Согласно полученным результатам, асимптотическая сложность реализованного метода Симпсона с высокой долью вероятности является квадратичной $O(n^2)$, в случае, если в качестве интегрируемой функции используется функция с константной асимптотической сложностью.

11.6. Теоретический анализ кода алгоритма

В основе алгоритма лежат циклы двойной вложенности:

– для алгоритма аппроксимации Ричардсона: используется один цикл и его асимптотическая сложность оценивается в линейную $O(n)$;

– для подсчета интеграла методом Симпсона: используется один цикл и его асимптотическая сложность оценивается в линейную *O(n)*.

Каждый из которых непрерывно зависит от размера входных данных. Циклы двойной вложенности приводят к квадратичной $O(n^2)$ асимптотической сложности. Но внутри второго вложенного цикла вызывается передаваемая пользователем функция интегрирования, которая имеет свою уникальную асимптотическую сложность, которую необходимо скомбинировать с полученной квадратичной асимптотической сложностью для реализованного метода интегрирования. Таким образом, с теоретической точки зрения асимптотическую сложность алгоритма метода Симпсона можно оценить, как $O(n^2 * T(f))$, где $T(f)$ – асимптотическая сложность подынтегральной функции. Полученный результат подтверждает полученные результаты на этапе профилирования, в котором использовалась подынтегральная функция с константной асимптотической сложностью.

Полученные данные об асимптотической сложности реализованного метода Симпсона совпадают с теоретическими научными оценками, что позволяет говорить об успешной реализации метода.

12. Диаграммы реализованных классов и функций

С помощью среды разработки IntelliJ IDEA были сгенерированы диаграммы реализованных классов и функций.

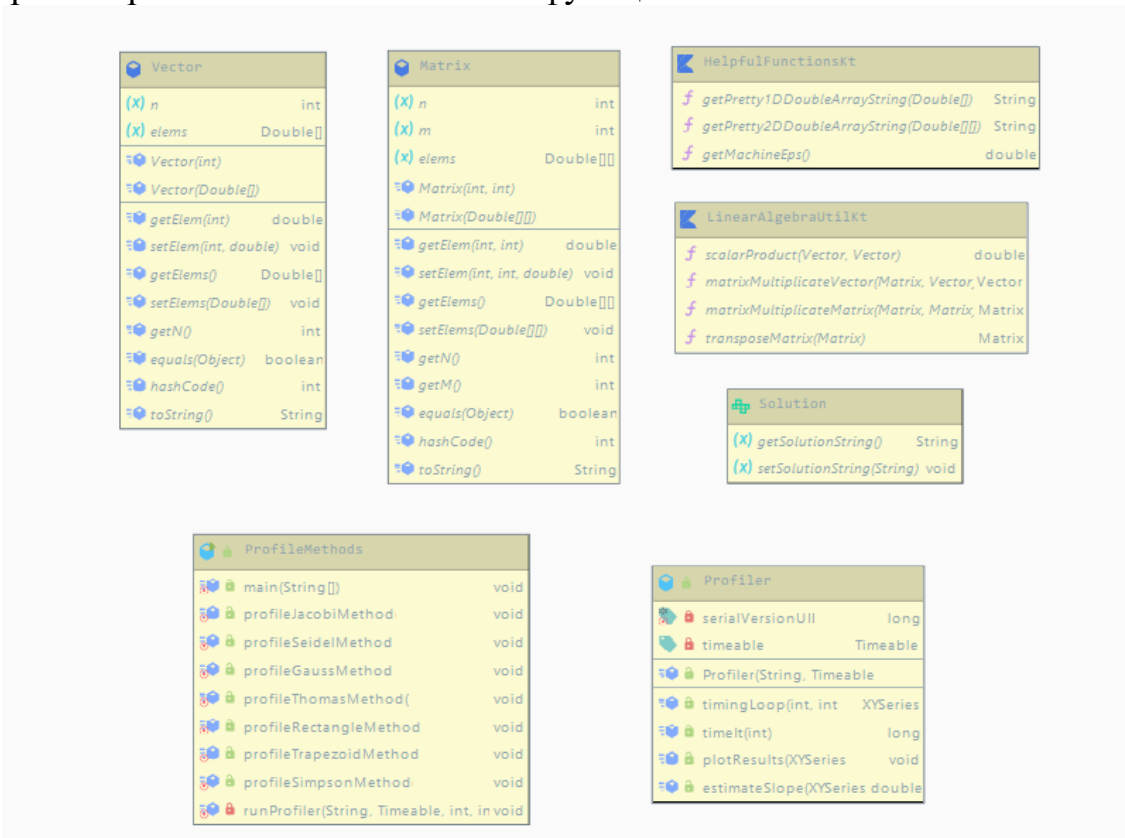


Рисунок 12.1 – Вспомогательные классы и функции

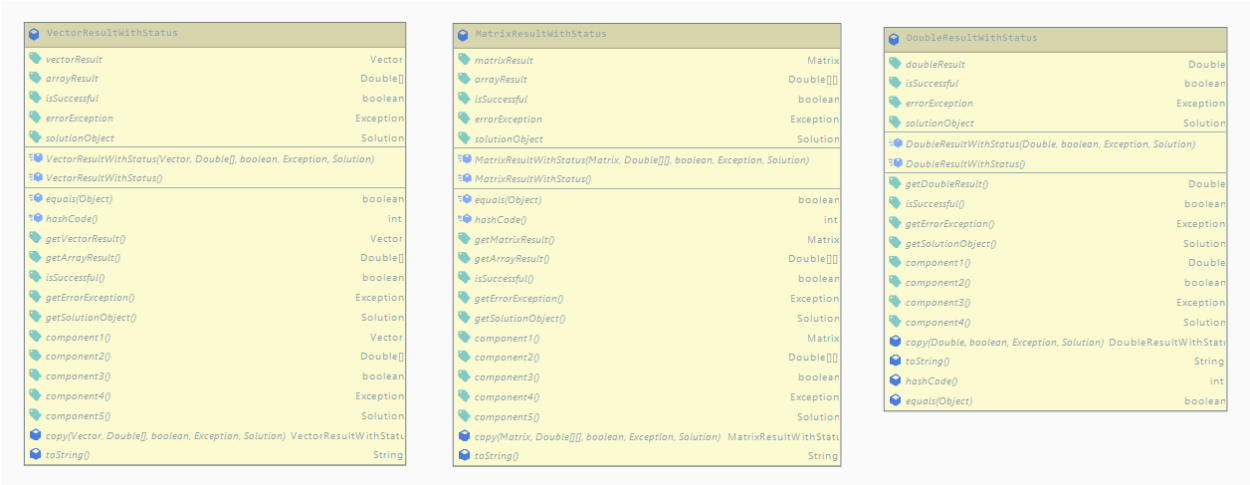


Рисунок 12.2 – Классы, хранящие результаты работы методов

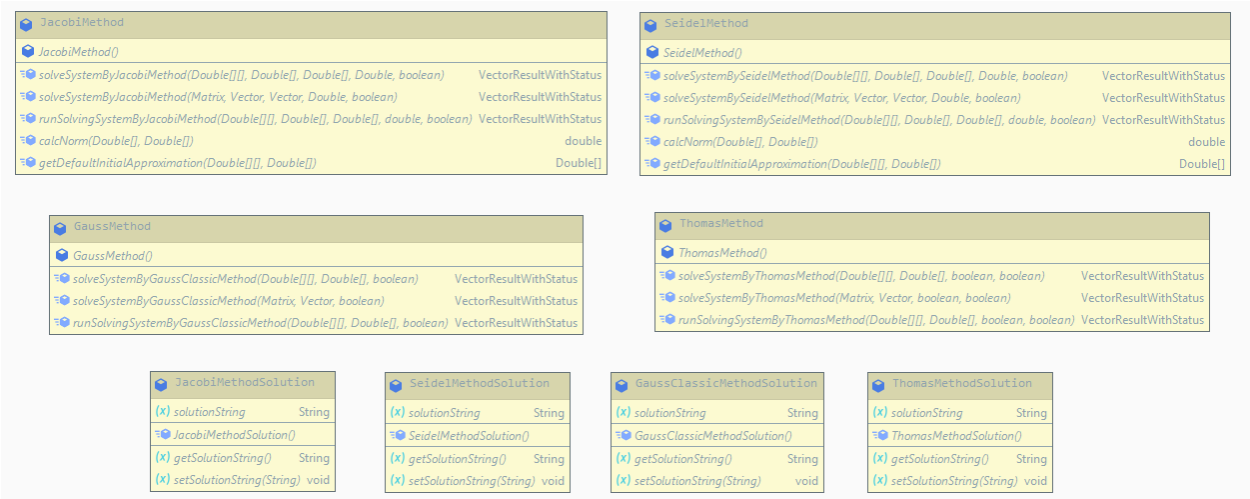


Рисунок 12.3 – Методы решения СЛАУ

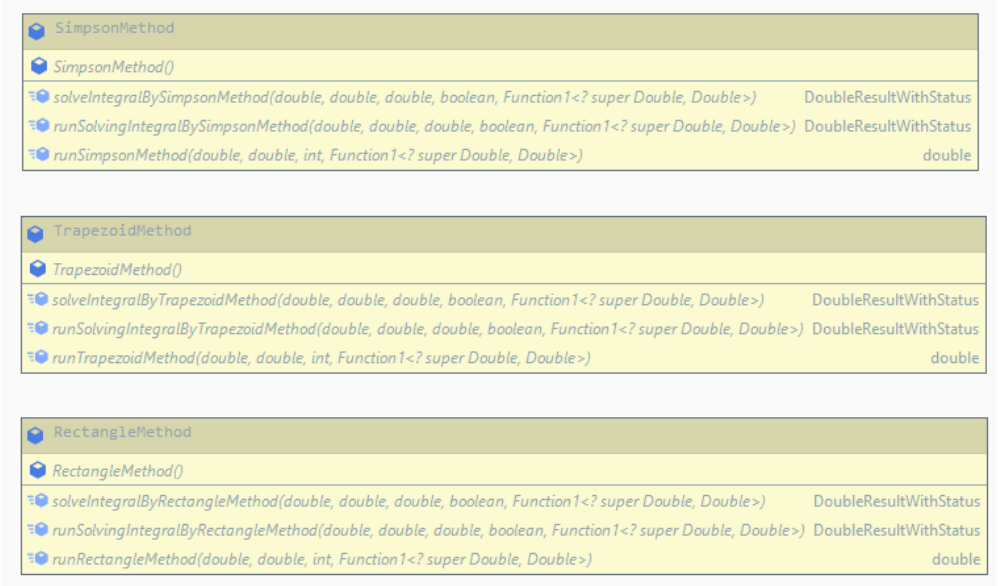


Рисунок 12.4 – Методы интегрирования

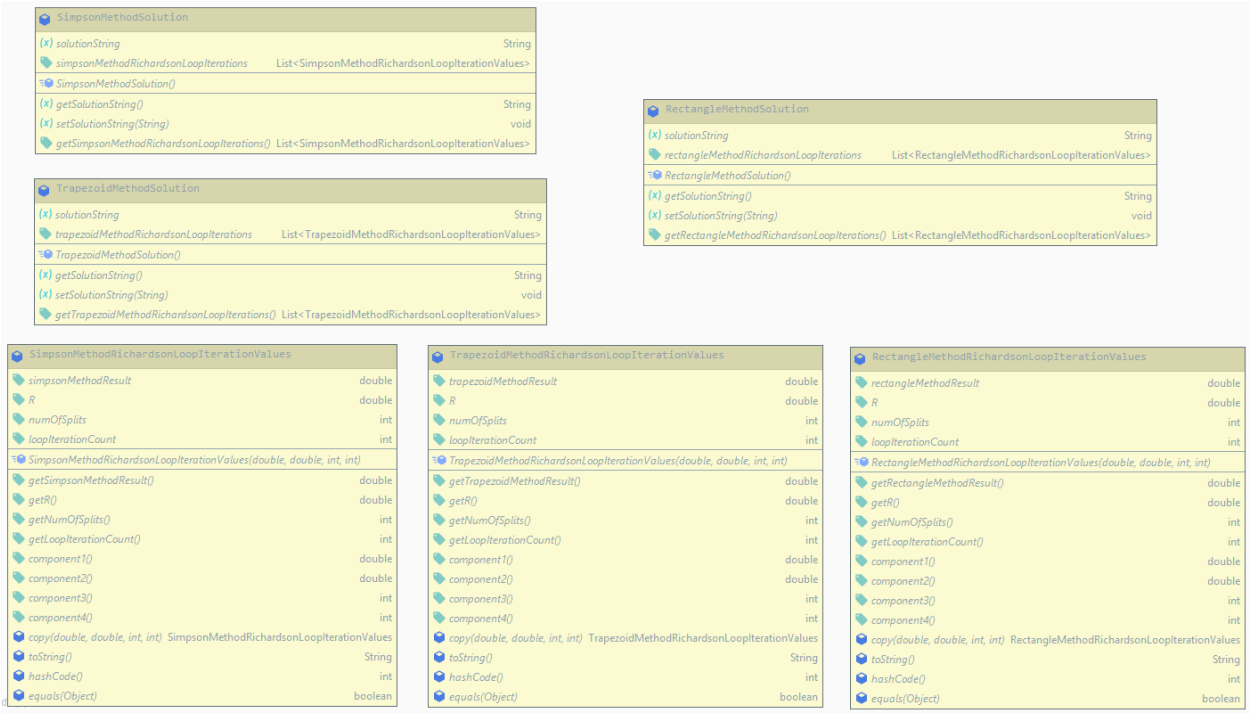


Рисунок 12.5 – Методы интегрирования: классы для подробного решения

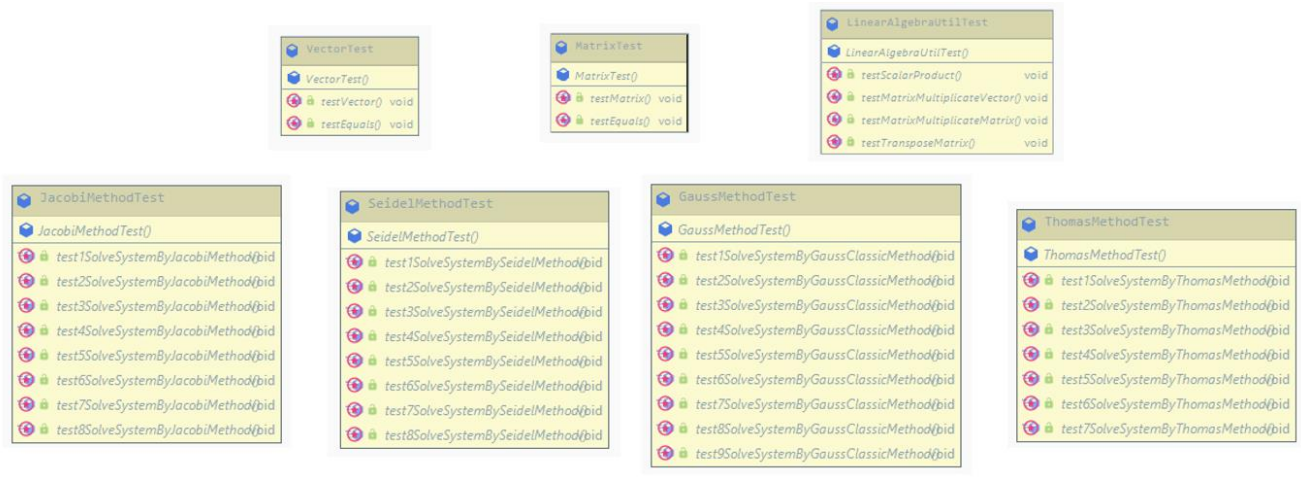


Рисунок 12.6 – Юнит-тесты

13. Исследование возможностей масштабирования

Разрабатываемую библиотеку можно использовать для разработки различных приложений и других проектов. Пример возможной схемы использования библиотеки на примере разработки Web-приложений, сервисов и Android-приложений представлен на рисунке 13.1.

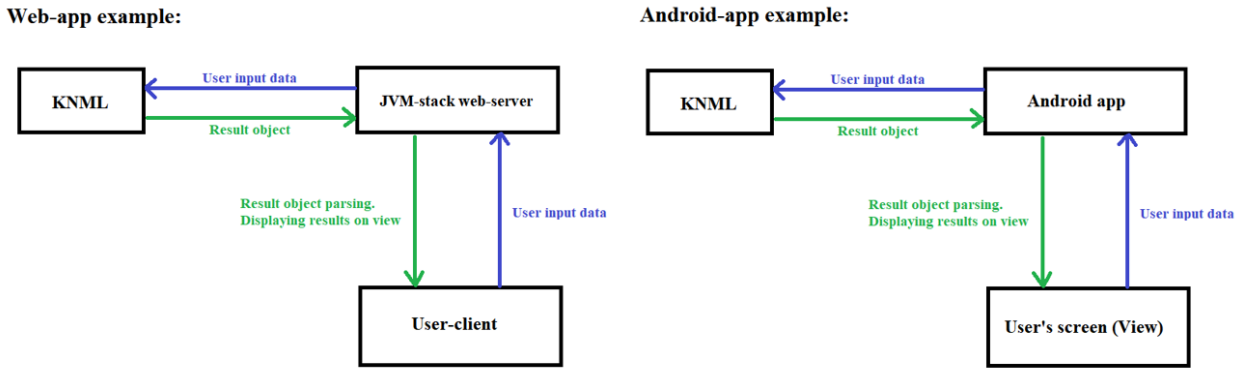


Рисунок 13.1 – Схема использования библиотеки для разработки приложений

Дополнительно разрабатываемую библиотеку можно объединять с другими библиотеками. В качестве демонстрации возможности объединения, библиотека была объединена с библиотекой kotlin-math [10]. Объединение было осуществлено на основе объединения модулей проектов двух библиотек, в результате чего получился мультимодульный проект: JAR архив объединенной библиотеки.

Из объединенной библиотеки удалось вызвать методы из двух библиотек, в проекте на языке программирования Java. Таким образом, удалось продемонстрировать возможность внедрения библиотеки в другие библиотеки, а также ее использование в других JVM языках.

ЗАКЛЮЧЕНИЕ

Разрабатываемая библиотека вносит вклад в развитие языка Kotlin, так как реализации подобных библиотек выявить не удалось.

В ходе разработки библиотеки было разработано около 4000 строк программного кода. Разрабатываемую библиотеку можно использовать в различных целях:

- использование на практикуме по ЧМЛА и других курсах кафедры;
- возможность трансляции знаний для других студентов;
- возможности разработки информационной системы для дистанционного обучения;
- разработка калькуляторов с подробным решением.

Развитие проекта планируется продолжить в рамках дипломной работы.

СПИСОК ЛИТЕРАТУРЫ

1. Сервис репозиториев GitHub: сайт // URL: https://github.com/ (дата обращения 28.12.2021)
2. Библиотека numpy: сайт // URL: https://numpy.org (дата обращения 28.12.2021)
3. Среда разработки IntelliJ IDEA Ultimate: сайт // URL: https://www.jetbrains.com/idea/ (дата обращения 28.12.2021)
4. Система сборки Ant: сайт // URL: https://ant.apache.org (дата обращения 28.12.2021)
5. Система сборки Maven: сайт // URL: https://maven.apache.org (дата обращения 28.12.2021)
6. Система сборки Gradle: сайт // URL: https://gradle.org (дата обращения 28.12.2021)
7. Фреймворк юнит-тестирования JUnit5: сайт // URL: https://junit.org/junit5/ (дата обращения 28.12.2021)
8. Инструмент генерации документации Dokka: сайт // URL: https://kotlin.github.io/dokka/ (дата обращения 28.12.2021)
9. Косарев В. И. 12 лекций по вычислительной математике (вводный курс) – Изд. 3-е, испр. и доп. – М.: Физматкнига, 2013. – 240с. ISBN 978-5-89155-214-2.
10. Библиотека kotlin-math: сайт // URL: https://github.com/romainguy/kotlin-math (дата обращения 28.12.2021)