#### МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

**«Национальный исследовательский**

**Нижегородский государственный университет им. Н.И. Лобачевского» (ННГУ)**

**Институт информационных технологий, математики и механики Кафедра: алгебры, геометрии и дискретной математики** Направление подготовки: «Программная инженерия»

Профиль подготовки: «Разработка программно-информационных систем»

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА**

на тему:

**«Алгоритмы для нахождения Эрмитовой нормальной формы и решения проблемы ближайшего вектора решетки»**

**Выполнил(а):** студент(ка) группы

Д.В. Огнев

Подпись

**Научный руководитель:**

Доцент, кандидат физико- математических наук

С.И. Веселов Подпись

Нижний Новгород 2022

Аннотация

Тема выпускной квалификационной работы бакалавра — <<Алгоритмы для нахождения Эрмитовой нормальной формы и решения проблемы ближайшего вектора решетки>>.

Ключевые слова: решетки, задачи теории решеток, Эрмитова нормальная форма, проблема ближайшего вектора.

Данная работа посвящена изучению задач теории решеток и методов их решения. В работе изложены основные понятия, связанные с решетками, исследованы алгоритмы для нахождения Эрмитовой нормальной формы и решения проблемы ближайшего вектора и разработана программная реализация разобранных алгоритмов.

Целью работы является программная реализация алгоритмов для решения задач теории решеток. Для успешного достижения цели поставленной цели необходимо разобрать теоретические основы алгоритмов, определить необходимые программные инструменты, научиться эффективно их использовать и получить программную реализацию.

Объем работы — 33 страницы, 8 таблиц, 6 рисунков, 5 приложений, 10 литературных источников.

Содержание

[Аннотация 2](#_Toc106217071)

[Содержание 3](#_Toc106217072)

[1. Список условных обозначений и сокращений 5](#_Toc106217073)

[2. Введение 6](#_Toc106217074)

[3. Постановка задачи 7](#_Toc106217075)

[4. Обзор инструментов 8](#_Toc106217076)

[4.1. Обзор библиотеки Eigen 8](#_Toc106217077)

[4.2. Обзор библиотеки Boost.Multiprecision 9](#_Toc106217078)

[5. Обзор литературных источников 11](#_Toc106217079)

[5.1. Базовые определения 11](#_Toc106217080)

[5.2. Ортогонализация Грама-Шмидта 12](#_Toc106217081)

[5.3. Алгоритм нахождения ЭНФ для матриц с полным рангом строки 13](#_Toc106217082)

[5.4. Общий алгоритм нахождения ЭНФ для любых матриц 15](#_Toc106217083)

[5.5. Пример нахождения ЭНФ 16](#_Toc106217084)

[5.6. Применение ЭНФ 18](#_Toc106217085)

[5.7. Определение проблемы ближайшего вектора 19](#_Toc106217086)

[5.8. Жадный метод: алгоритм ближайшей плоскости Бабая 20](#_Toc106217087)

[5.9. Нерекурсивная реализация 21](#_Toc106217088)

[5.10. Пример жадного метода 21](#_Toc106217089)

[5.11. Метод ветвей и границ 22](#_Toc106217090)

[5.12. Пример метода ветвей и границ 23](#_Toc106217091)

[5.13. Параллельная реализация метода ветвей и границ 23](#_Toc106217092)

[6. Обзор существующих решений 24](#_Toc106217093)

[6.1. WolframAlpha API 24](#_Toc106217094)

[6.2. Numbertheory.org 24](#_Toc106217095)

[6.3. hsnf 26](#_Toc106217096)

[7. Обзор программной реализации 27](#_Toc106217097)

[7.1. Вспомогательные функции 27](#_Toc106217098)

[7.2. Ортогонализация Грама-Шмидта 29](#_Toc106217099)

[7.3. Нахождение ЭНФ 29](#_Toc106217100)

[7.4. Решение ПБВ 30](#_Toc106217101)

[8. Заключение 32](#_Toc106217102)

[Список литературы 33](#_Toc106217103)

[Приложения 34](#_Toc106217104)

[Приложение А. Исходный код algorithms.hpp 34](#_Toc106217105)

[Приложение Б. Исходный код utils.hpp 34](#_Toc106217106)

[Приложение В. Исходный код algorithms.cpp 36](#_Toc106217107)

[Приложение Г. Исходный код utils.cpp 47](#_Toc106217108)

[Приложение Д. Исходный CMakeLists.txt 65](#_Toc106217109)

1. Список условных обозначений и сокращений

ПБВ (CVP) — проблема ближайшего вектора (closest vector problem)

ЭНФ (HNF) — Эрмитова нормальная форма (Hermite normal form)

API — application programming interface

B&B — branch and bound

GMP — GNU Multiprecision Library

G++ — GNU C++

1. Введение

Криптография — наука, которая занимается методами преобразования (шифрования) с целью обеспечения конфиденциальности, целостности данных, аутентификации и защиты информации от незаконных пользователей. Самыми известными вычислительно трудными задачами считаются проблема вычисления дискретного логарифма и факторизация (разложение на множители) целых чисел. Для этих задач неизвестны эффективные (работающие за полиномиальное время) алгоритмы. С развитием квантовых компьютеров было показано существование полиномиальных алгоритмов решения задач дискретного логарифмирования и разложения числа на множители на квантовых вычислителях [3], что заставляет искать задачи, для которых неизвестны эффективные квантовые алгоритмы. В области постквантовой криптографии фаворитом можно назвать криптографию на решетках, т. к. считается, что она устойчива к квантовым компьютерам. Поэтому изучение задач теорий решеток является основной целью при построении устойчивых криптосистем на решетках.

Предметом исследования данной работы являются алгоритмы для нахождения Эрмитовой нормальной формы и решения проблемы ближайшего вектора. Целью работы является получение программной реализации алгоритмов для нахождения ЭНФ за полиномиальное время, приблизительного решения ПБВ за полиномиальное время и точного решения ПБВ за суперполиномиальное время. Необходимо будет показать, как можно использовать данные алгоритмы на практике. В качестве теоретической базы, откуда взяты основы и описание алгоритмов для программирования, была использована серия лекций по решеткам и решеточным алгоритмам.

1. Постановка задачи

Цель работы — реализовать алгоритмы для нахождения ЭНФ и решения ПБВ за полиномиальное и суперполиномиальное время. Для достижения этой цели необходимо решить следующие задачи:

* Изучить теоретические основы для программирования алгоритмов.
* Найти необходимые инструменты для программной реализации, научиться эффективно их использовать.
* Написать программную реализацию, в которой будут реализованы разобранные алгоритмы. Программная реализация должна быть кроссплатформенной и разработана как отдельная библиотека.
* Полученную библиотеку использовать для решения задач теории решеток и показать, как можно применять ее на практике.

1. Обзор инструментов

Для программной реализации был выбран язык C++. Приоритет этому языку был отдан из-за его скорости, статической типизации, большому количеству написанных библиотек и обширной стандартной библиотеке. Сборка проекта осуществляется с помощью системы сборки CMake, при сборке можно указать флаги

* BUILD\_DOCS — используется для сборки документа выпускной квалификационной работы, написанной в формате Latex;
* BUILD\_PARALLEL — используется для сборки параллельной реализации алгоритма ортогонализации Грама-Шмидта и branch and bound;
* BUILD\_GMP — для использования библиотеки GMP.

Для работы с матрицами была выбрана библиотека Eigen, для работы с большими числами используется часть библиотеки Boost — Boost.Multiprecision, которая подключается в режиме Standalone. Используется встроенная в Boost реализация больших чисел и реализация от GMP.

Используется система контроля версий Git и сервис Github, все исходные файлы проекта доступны в онлайн репозитории. Для подключения Boost.Multiprecision используются модули Git.

* 1. Обзор библиотеки Eigen

Eigen - библиотека для работы с линейной алгеброй, предоставляет шаблонные классы для работы с матрицами и векторами. Является header-only библиотекой и не требует отдельной компиляции, для работы не требует других библиотек, кроме стандратной.

Все необходимые классы находятся в заголовочном файле Eigen/Dense и подключаются директивой #include <Eigen/Dense>, для их использования необходимо указывать пространство имен Eigen, например Eigen::Matrix2d [5].

Используемые классы:

Matrix<typename Scalar, int RowsAtCompileTime, int ColsAtCompileTime> — шаблонный класс матриц. Первый параметр шаблона отвечает за тип элементов матрицы, второй параметр за количество строк, третий за количество столбцов. Если количество строк/столбцов неизвестно на этапе компиляции, а будет найдено в процессе выполнения программы, то необходимо ставить количество строк/столбцов равным Eigen::Dynamic, либо -1. Имеет псевдонимы для различных встроенных типов (int, double, float) и размеров матриц (2, 3, 4), например Matrix3d — матрица элементов double размера .

Vector и RowVector — вектор-столбец и вектора-строка соответственно, являются псевдонимами класса матриц, в которых количество строк/столбцов равно единице. Используются псевдонимы для различных встроенных типов (int, float, double) и размеров векторов (2, 3, 4), например Vector2f — вектор, состоящий из элементов float размера .

С матрицами и векторами можно производить различные арифметические действия, например складывать и вычитать между собой, умножать и делить между собой и на число. Все действия должны осуществляться по правилам линейной алгебры.

Используемые методы:

matrix.rows() — получение количества строк.

matrix.cols() — получение количества столбцов.

vector.norm() — длина вектора.

vector.squaredNorm() — квадрат длины вектора.

matrix << elems — comma-инициализация матрицы, можно инициализировать матрицу через скалярные типы, матрицы и векторы.

Eigen::MatrixXd::Identity(m, m) — получение единичной матрицы размера .

Eigen::VectorXd::Zero(m) — получение нулевого вектора размера .

matrix.row(index) — получение строки матрицы по индексу.

matrix.col(index) — получение столбца матрицы по индексу.

matrix.row(index) = vector — установить строку матрицы значениями вектора.

matrix.col(index) = vector — установить столбец матрицы значениями вектора.

matrix.block(startRow, startCol, endRow, endCol) — получение подматрицы по индексам.

matrix.block(startRow, startCol, endRow, endCol) = elem — установка блока матрицы по индексам значением elem.

matrix.cast<type>() — привести матрицу к типу type.

vector1.dot(vector2) — скалярное произведение двух векторов.

vector.tail(size) — получить с конца вектора size элементов.

matrix(i, j) — получение элемента матрицы по индексам.

vector(i) — получение элемента вектора по индексу.

matrix(i, j) = elem — установка элемента матрицы по индексам значением elem.

vector(i) = elem — установка элемента вектора по индексу значением elem.

for (const Eigen::VectorXd &vector : matrix.colwise()) — цикл по столбцам матрицы.

for (const Eigen::VectorXd &vector : matrix.rowwise()) — цикл по строкам матрицы.

* 1. Обзор библиотеки Boost.Multiprecision

Boost.Multiprecision — часть библиотеки Boost, подключается в режиме Standalone, что позволяет не подключать основную библиотеку и не использовать модули, которые не требуются, в конечном итоге уменьшив итоговый размер программы. Все классы находятся в пространстве имен boost::multiprecision. Для подключения библиотеки используется директива #include <boost::multiprecision/cpp\_тип.hpp>. Если при сборке CMake будет указан флаг BUILD\_GMP=ON, то будет использована обертка от Boost над библиотекой GMP. Классы, связанные с GMP, подключаются с помощью #include <boost/multiprecision/gmp.hpp>. В документации Boost сказано, что реализация GMP работает быстрее, что будет видно показано далее.

Библиотека предоставляет классы для работы с целыми, рациональными числами и числами с плавающей запятой неограниченной точности. Размер этих чисел ограничен только количеством оперативной памяти [6].

Используемые классы:

cpp\_int — класс целых чисел.

cpp\_rational — класс рациональных чисел.

cpp\_bin\_float\_double — класс чисел с плавающей запятой с увеличенной точностью.

mpz\_int — класс целых чисел, использующий реализацию GMP.

mpq\_rational — класс рациональных чисел, использующий реализацию GMP.

mpf\_float\_50 — класс чисел с плавающей запятой, использующий реализацию GMP.

Используемые методы:

sqrt(int) — квадратный корень из целого числа.

numerator(rational) — числитель рационального числа.

denominator(rational) — знаменатель рационального числа.

1. Обзор литературных источников
   1. Базовые определения

Матрица [4] –– прямоугольная таблица чисел, содержащая строк и столбцов. Обозначается полужирной заглавной буквой, а ее элементы — строчными с двумя индексами (строка и столбец). При программировании использовалась стандартная структура хранения матриц:

Квадратная матрица –– матрица, у которой число строк равно числу столбцов .

Единичная матрица –– матрица, у которой диагональные элементы равны единице.

Невырожденная матрица–– квадратная матрица, определитель которой отличен от нуля.

Вектор –– если матрица состоит из одного столбца , то она называется вектором-столбцом. Если матрица состоит из одной строки , то она называется вектором-строкой. Матрицы можно обозначать через вектора-столбцы и через вектора-строки: .

Линейная зависимость/независимость –– пусть имеется несколько векторов одной размерности и столько же чисел . Вектор называется линейной комбинацией векторов . Если существуют такие числа , не все равные нулю, такие, что , то такой набор векторов называется линейно зависимым. В противном случае векторы называются линейно независимыми [4].

Ранг матрицы –– максимальное число линейно независимых векторов. Матрица называется матрицей с полным рангом строки, когда все строки матрицы линейно независимы. Матрица называется матрицей с полным рангом столбца, когда все столбцы матрицы линейно независимы.

Решетка — пусть — линейно независимые вектора из . Решетка, генерируемая от есть множество

всех целочисленных линейных комбинаций столбцов матрицы . Матрица называется базисом для решетки . Число называется рангом решетки. Если , то решетка называется решеткой полного ранга или полноразмерной решеткой в .

Определитель решетки — пусть — базис решетки, — ортогонализация Грама-Шмидта для исходного базиса, тогда определитель . Определитель решетки не зависит от выбора исходного базиса [1].

Эрмитова нормальная форма [4] — невырожденная матрица является Эрмитовой нормальной формой, если

* Существует такое, что (строго убывающая высота столбца).
* Для всех , т.е. все элементы в строках приведены по модулю .

Проблема ближайшего вектора — дан базис решетки и целевой вектор , который не принадлежит решетке, необходимо найти точку решетки () такую, что расстояние минимально [4].

* 1. Ортогонализация Грама-Шмидта

Любой базис может быть преобразован в ортогональный базис для того же векторного пространства используя алгоритм ортогонализации Грама-Шмидта (lec1?). Предположим у нас есть набор векторов , . Этот набор необзятельно ортогонален или даже линейно независим. Ортогонализацией этого набора векторов является набор векторов , где

Полученный набор векторов может не являться базисом для решетки, сгенерированной от исходного набора векторов, т.к. точки этой решетки могут не входить в решетку от ортогонализованного базиса. Этот набор также обладает важным свойством, которое мы будем использовать: если вектор , то этот вектор линейно зависим от других векторов в наборе и может быть представлен линейной комбинацией этих векторов.

Временная сложность алгоритма , т.к. у нас имеется цикл, вложенный в цикл, в котором 2 скалярных произведения и сумма векторов. Для процесса ортогонализации Грама-Шмидта нельзя сделать параллельную реализацию, так как каждая следующая итерация требует данные, найденные на предыдущем шаге. Но можно ускорить ее нахождение, путем параллельного нахождения суммы . Конечный алгоритм выглядит следующим образом:

**Input:**

**Output:**

**← [ ]**

.columns

**for** i ← 0 to n **do**

**←** .column(i)

**← 0**

**for** j ← 0 to i **do**

**←** .column(j)

**projections ←**

**end for**

**GS.**push\_back()

**end for**

* 1. Алгоритм нахождения ЭНФ для матриц с полным рангом строки

Дана матрица . Основная идея состоит в том, чтобы найти ЭНФ подрешетки от , и затем обновлять , включая столбцы один за другим (**lec4?**). Предположим, что у нас есть процедура AddColumn, которая работает за полиномиальное время и принимает на вход квадратную невырожденную ЭНФ матрицу и вектор , а возвращает ЭНФ матрицы . Такая процедура должна следить, чтоб выходная матрица подоходила под определение ЭНФ, что будет показано в описании этой процедуры. ЭНФ от может быть вычислено следующим образом:

1. Применить алгоритм Грама-Шмидта к столбцам , чтобы найти линейно независимых столбцов. Пусть - матрица размера , заданная этими столбцами.
2. Вычислить , используя алгоритм Грама-Шмидта или любую другую процедуру с полиномиальным временем. Пусть будет диагональной матрицей с на диагонали.
3. Для пусть – результат применения AddColumn к входным и .
4. Вернуть .

Разберем подпункты:

1. Необходимо найти линейно независимые столбцы матрицы. Их количество всегда будет равно , т.к. наша матрица полного ранга строки и ранг матрицы равен , а значит матрица, состоящая из этих столбцов, будет размера . Для нахождения этих строк можно использовать алгоритм ортогонализации Грама-Шмитда: если , то -ая строка является линейной комбинацией других строк, и ее необходимо удалить. Полученная матрица будет названа .
2. Необходимо вычислить , будем вычислять его по следующей форумле: — сумма произведений квадратов длин всех столбцов, полученных после применения ортогонализации Грама-Шмидта. Матрица будет единичной матрицей размера , умноженной на определитель. В результате все диагональные элементы будут равны .
3. Применяем AddColumn к и первому столбцу матрицы , получаем ; повторяем для всех оставшихся столбцов, получаем .
4. является ЭНФ().

Алгоритм AddColumn на вход принимает квадратную невырожденную ЭНФ матрицы и вектор и работает следующим образом. Если , то возвращаем . В противном случае, пусть и и дальше:

1. Вычислить и целые такие, что , используя расширенный НОД алгоритм.
2. Применить унимодулярное преобразование к первому столбцу из и чтобы получить .
3. Добавить соответствующий вектор из к , чтобы сократить его элементы по модулю диагональных элементов из .
4. Рекурсивно вызвать AddColumn на вход и чтобы получить матрицу .
5. Добавить соответствующий вектор из к чтобы сократить его элементы по модулю диагональных элементов из .
6. Вернуть .

Разберем подпункты:

1. Необходимо с помощью расширенного НОД алгоритма найти наибольший общий делитель и целые такие, что .
2. Составляем матрицу и умножаем ее на матрицу, составленную из первого столбца и столбца , чтобы получить:
3. Функция reduce должна принимать на вход матрицу и вектор и получать необходимый вектор из решетки от матрицы на входе, чтобы сократить элементы вектора по модулю диагональных элементов из матрицы. Применяем функцию reduce к и .
4. Рекурсивно вызываем AddColumn, на вход отправляем и получаем матрицу .
5. Вызываем функцию reduce к и .
6. Составляем необходимую матрицу и возвращаем .
   1. Общий алгоритм нахождения ЭНФ для любых матриц

Данный алгоритм можно применять на произвольных матрицах путем сведения к алгоритму для полного ранга строки [2].

1. Запустить процесс ортогонализации Грама-Шмидта к строкам из , и пусть – это множество индексов, такое, что . Определим операцию проецирования при . Заметим, что строки линейно независимы и любая строка может быть выражена как линейная комбинация предыдущих строк . Следовательно, операция проецирования однозначно определена, когда ограничена к , и ее инверсия может быть легко вычислена, используя коэффициенты Грама-Шмидта .
2. Введем матрицу , которая полного ранга (т.к. все строки линейно независимы), и запустим алгоритм для матриц полного ранга строки, чтобы найти ЭНФ от .
3. Применить функцию, обратную операции проецирования, к ЭНФ , чтобы получить матрицу , которая является ЭНФ матрицы .

Алгоритм прост, но нужно обратить внимание на операцию проецирования и обратную к ней. Для того, чтобы находить результат проецирования напишем функцию get\_linearly\_independent\_rows\_by\_gram\_schmidt, которая будет возвращать матрицу , состоящую из линейно независимых строк, а также массив индексов этих строк из исходного массива. К матрице применяется алгоритм нахождения ЭНФ для матриц с полным рангом, разобранный в прошлом разделе. Далее необходимо восстановить удаленные строки. Т.к. они являются линейной комбинацией линейно независимых строк, то мы можем найти коэффициенты, на которые нужно умножить строки из матрицы и после чего сложить их, чтобы получить нужную строку, которую необходимо добавить к . Также восстановить строки можно через коээфициенты Грама-Шмидта, для этого на этапе ортогонализации необходимо составить матрицу, состояющую из этих коэффициентов:

после чего эту матрицу необходимо умножить на . Получившаяся матрица будет ЭНФ матрицы .

Количество рекурсивных вызовов будет равно , т.к. мы вызываем процедуру AddColumn для каждого столбца и для каждого столбца рекурсивно вызываем ее до тех пор, пока количество строк не будет равно нулю.

* 1. Пример нахождения ЭНФ

Рассмотрим нахождение ЭНФ на примере небольшой матрицы размера . Получим случайную матрицу . Т.к. мы получаем случайную матрицу, то не можем заранее знать, матрица с полным рангом строки или нет, поэтому будем использовать общий алгоритм. Первый шаг алгоритма требует от нас найти линейно независимых строк матрицы , используя алгоритм ортогонализации Грама-Шмидта. Обозначим искомую ортогонализацию строк за и найдем их:

1. ,
2. .

Нулевых строк нет, значит матрица полностью состоит из линейно независимых строк, матрица будет содержать в себе все строки из . Далее алгоритм требует от нас найти ЭНФ от матрицы , используя алгоритм для полного ранга строки.

Рассмотрим алгоритм для полного ранга строки. Алгоритм принимает на вход матрицу . Требуется найти линейно независимых строк, используя ортогонализацию Грама-Шмидта. Используем этот алгоритм на строки :

1. ,
2. .

Т.к. матрица полного ранга строки, ее ранг меньше либо равен количеству столбцов и равен количеству строк . Используя алгоритм Грама-Шмидта на столбцы матрицы мы удаляем линейно зависимые столбцы, и, если количество столбцов больше либо равно количества строк, то количество столбцов становится равно количеству строк. Получаем матрицу размера , состоящую из линейно независимых столбцов матрицы .

Далее необходимо составить матрицу . Для этого необходимо найти определитель решетки и умножить единичную матрицу размера на .

Для используем AddColumn для каждого и :

* + - 1. , , , , , , .

Используем расширенный НОД алгоритм, находим , , . Составляем матрицу , умножаем матрицу, составленную из первого столбца и столбца на матрицу : , , .

Сокращаем по модулю диагональных элементов из , вычисляя и добавляя соответствующий вектор из : .

Рекурсивно вызываем AddColumn со входом и , получаем матрицу :

* , , , , , , .

Находим , , . Составляем матрицу , умножаем: , , .

Сокращаем по модулю диагональных элементов из , вычисляя и добавляя соответствующий вектор из : .

Рекурсивно вызываем AddColumn со входом и : произойдет выход из рекурсии по условию и вернется пустая матрица .

Сокращаем по модулю диагональных элементов из , вычисляя и добавляя соответствующий вектор из : .

Возвращаем .

Сокращаем по модулю диагональных элементов из , вычисляя и добавляя соответствующий вектор из : .

Возвращаем .

* + - 1. , , , , , , .

Находим , , . Составляем матрицу , умножаем: , , .

Сокращаем по модулю диагональных элементов из , вычисляя и добавляя соответствующий вектор из : .

Рекурсивно вызываем AddColumn со входом и , получаем матрицу :

* , , , , , , .

Находим , , . Составляем матрицу , умножаем: , , .

Сокращаем по модулю диагональных элементов из , вычисляя и добавляя соответствующий вектор из : .

Рекурсивно вызываем AddColumn со входом и : произойдет выход из рекурсии по условию и вернется пустая матрица .

Сокращаем по модулю диагональных элементов из , вычисляя и добавляя соответствующий вектор из : .

Возвращаем .

Сокращаем по модулю диагональных элементов из , вычисляя и добавляя соответствующий вектор из : .

Возвращаем .

ЭНФ() = .

* 1. Применение ЭНФ

Будут рассмотрены некоторые проблемы и задачи теории решеток и их решение с помощью ЭНФ [2].

**Нахождение базиса.** Дан набор рациональных векторов , необходимо вычислить базис для . Проблема решается за полиномиальное время путем вычисления :

, .

**Проблема эквивалентности.** Дано два базиса и . Необходимо узнать, образуют ли они однинаковую решетку . Проблема решается путем вычисления и и сравнения их равенства:

, , , — образуют одинаковую решетку.

, , , — не образуют одинаковой решетки.

**Объединение решеток.** Дано два базиса и . Необходимо найти базис для наименьшей решетки, содержащей обе решетки и . Такая решетка будет сгенерирована от , и можно легко найти ее базис через ЭНФ:

, , , .

**Проблема включения.** Дано два базиса и . Необходимо узнать, является ли подрешеткой , т.е. . Эта проблема сводится к проблемам объединения и эквивалентности: тогда и только тогда, когда . Для этого необходимо вычислить и и сравнения их равенства:

, , , , — не является подрешеткой .

, , , , — является подрешеткой .

**Проблема содержания.** Дана решетка и вектор , необходимо узнать, принадлежит ли вектор решетке (). Эта проблема сводится к проблеме включения путем проверки . Если необходимо проверить содержание нескольких векторов , тогда следует сначала вычислить , и затем проверять, равно ли для каждого вектора:

, , , , — вектор .

, , , , — вектор .

* 1. Определение проблемы ближайшего вектора

Рассмотрим проблему ближайшего вектора [2]: дан базис решетки и вектор , найти точку решетки такую, что (расстояние от точки до решетки) минимально. Это задача оптимизации (минимизации) с допустимыми решениями, заданными всеми целочисленными векторами , и целевой функцией .

Пусть и , где , , и . Заметим, что если зафиксировать значение , то задача потребует найти значение такое, что

минимально. Это также ПБВ с измененным вектором , и решеткой меньшего размера . В частности, пространство решений сейчас состоит из целочисленных переменных . Это говорит о том, что можно решить ПБВ путем установки значения по одной координате за раз. Есть несколько способов превратить этот подход к уменьшению размерности в алгоритм, используя некоторые стандартные методы алгоритмического программирования. Простейшие методы:

* + - 1. Жадный метод, который выдает приближенные значения, но работает за полиномиальное время.
      2. Метод ветвей и границ, который выдает точное решение за суперэкспоненциальное время.

Оба метода основаны на очень простой нижней оценке целевой функции:

* 1. Жадный метод: алгоритм ближайшей плоскости Бабая

Суть жадного метода состоит в выборе переменных, определяющих пространство решений, по одной, каждый раз выбирая значение, которые выглядит наиболее многообещающим [2]. В нашем случае, выберем значение x, которое дает наименьшее возможное значение для нижней границы . Напомним, что и , и что для любого фиксированного значения , ПБВ сводится к ПБВ , где . Используя для нижней границы, мы хотим выбрать значение такое, что

как можно меньше. Это очень простая 1-размерная ПБВ проблема (с решеткой и целью , которая может быть сразу решена установкой

где компонента вектора , ортогональная другим базисным векторам. Полный алгоритм приведен ниже:

**Input: [B, b], t**

**Output:**

←

←

←

Количество рекурсивных вызовов будет равно размеру столбцов n входной матрицы, т.к. мы ищем x для каждого столбца.

* 1. Нерекурсивная реализация

Легко заметить, что можно заменить рекурсию на цикл и таким образом получить нерекурсивную версию алгоритма:

**Input:**

**Output:**

← GramSchmidt()

← .columns

←

**for** i ← 0 to n **do**

index ← n − i − 1

← .column(index)

← .column(index)

←

←

t ← t − c \* **b**

**result** ← **result** + c \* **b**

**end for**

* 1. Пример жадного метода

Рассмотрим пример на простой решетке и целевым вектором .

Представим входную матрицу в виде . На каждом шагу нам необходимо вычислять вектор . Эти вектора можно заранее вычислить через алгоритм Грама-Шмидта. В нашем случае вектора уже перпендикулярны друг другу. Смысл алгоритма заключается в установлении одной координаты за раз, для этого мы берем крайний вектор базиса, находим коэффициент, на который его надо умножить, и складываем с результатом рекурсии текущего алгоритма со входом уменьшенной матрицу и отредактированной целью. Таким образом мы найдем коэффициенты для каждого вектора базиса, и ответ будет суммой умножения коэффициентов на соответствующий вектор базиса:

, , , , , .

Рекурсивно вызываем метод, на вход отправляем , .

, , , , , .

Рекурсивно вызываем метод, на вход отправляем , .

Т.к. , то возвращаем пустой вектор.

В итоге сумма векторов будет равна – искомый вектор.

* 1. Метод ветвей и границ

Алгоритм похож на жадный метод, но вместо установки на наиболее подходящее значение (то есть на то, для которого нижняя граница расстояния минимальна), мы ограничиваем множество всех возможных значений для , и затем мы переходим на каждую из них для решения каждой соответствующей подзадачи независимо. В заключении, мы выбираем наилучшее возможное решение среди возвращенных всеми ветками.

Чтобы ограничить значения, которые может принимать , нам также нужна верхняя граница расстояния от цели до решетки. Ее можно получить несколькими способами. Например, можно просто использовать (расстояние от цели до начала координат) в качестве верхней границы. Но лучше использовать жадный алгоритм, чтобы найти приближенное решение , и использовать в качестве верхней границы. Как только верхняя граница установлена, можно ограничить переменную такими значениями, что .

Количество рекурсивных вызовов будет не больше, чем число

В процессе временного тестирования алгоритма будет видно, что чем больше число строк , тем сильнее возрастает время выполнения алгоритма.

Окончательный алгоритм похож на жадный метод:

где выбирает вектор в ближайший к цели .

Как и для жадного алгоритма, производительность (в данном случае время выполнения) метода Ветвей и Границ может быть очень низкой, если мы сперва не сократим базис входной решетки (например, используя LLL-алгоритм).

Сложность алгоритма заключается в нахождении множества . Его можно найти, используя выражение, выведенное в прошлом алгоритме: . С помощью него мы найдем , который точно удовлетворяет множеству, а затем будем увеличивать/уменьшать до тех пор, пока выполняется условие .

* 1. Пример метода ветвей и границ

Рассмотрим пример на простой решетке и целевым вектором .

Представим входную матрицу в виде . На каждом шаге нам необходимо вычислять вектор . Заранее вычислим их с помощью алгоритма Грама-Шмидта. В нашем случае вектора уже перпендикулярны друг другу. Смысл алгоритма также заключается в установлении одной координаты за раз, но вместо самого перспективного варианта мы будем строить множество , подходящее под условие . Вектор найдем с помощью жадного метода. Далее также, как и в жадном методе ищем необходимую сумму векторов, получим множество , из которого необходимо будет выбрать ближайший к цели .

, , , , .

Рекурсивно вызываем метод для каждого , на вход отправляем , .

Получаем множество .

Ближайший вектор будет равен – искомый вектор.

* 1. Параллельная реализация метода ветвей и границ

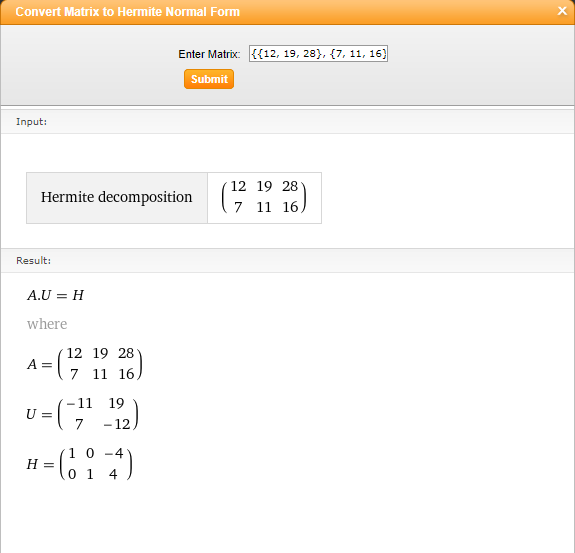
Можно увидеть, что процесс нахождения ближайшего вектора в методе ветвей и границ является деревом: для каждого подходящего значения из множества мы запускаем подзадачу, используя тот же алгоритм с решеткой меньшей размерности, и так до тех пор, пока у нас не закончатся векторы в базисе. При таком подходе сложно уйти от рекурсии, т.к. каждая подзадача использует свою версию целевого вектора, но каждую такую задачу можно решать независимо от другой, в чем и заключается пареллельный подход.

Для получения параллельной реализации будем использовать задачи (task) из библиотеки OpenMP. После получения множества будем находить множество векторов следующим образом: для каждого значения будем создавать свою задачу, которая помещается в специальный пул, после чего свободные потоки берут из него задачи и выполняют работу параллельно. В качестве синхронизации используется директива #pragma omp taskwait, она указывается перед вызовом closest(, **t**).

1. Обзор существующих решений
   1. WolframAlpha API

WolframAlpha Webservice API [7] предоставляет web-based API, позволяющий интегрировать свои вычислительные возможности в разрабатываемое приложение. API реализован в стиле REST и использует HTTP GET запросы. Возвращает результат в формате XML структуры. Главное его достоинство — легкость интеграции и простота использования. Главный недостаток – размер входных матриц сильно ограничен, максимальный размер , что делает его непригодным для использования на практике, но пригодным для проверки результатов при программировании и отладке. Также можно использовать веб-версию WolframAlpha.

Пример работы:



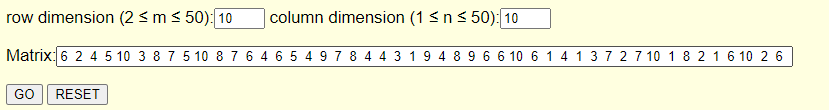
#### Рис. 1: Нахождение ЭНФ с помощью WolframAlpha.

* 1. Numbertheory.org

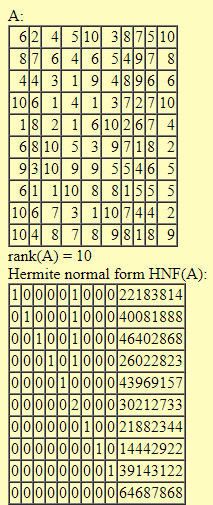
Сайт numbertheory.org предоставляет сервис [8], в котором реализованы различные алгоритмы на решетках, в том числе нахождение ЭНФ и решение ПБВ.

Для нахождения ЭНФ необходимо указать количество строк, столбцов и саму входную матрицу. Недостатком является низкая эффективность при большой входной матрице, а также ограничение на ее размер (максимально ). Данный сервис можно использовать для отладки на бо́льших размерах матриц, чем при использовании WolframAlpha, и увидеть, как сильно растут числа на больших матрицах.

Пример работы:

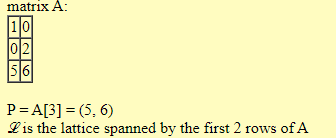


#### Рис. 2: Нахождение ЭНФ с помощью numbertheory.org. Ввод данных.

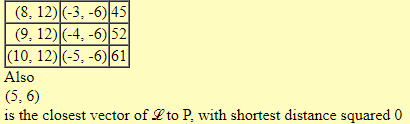


#### Рис. 3: Нахождение ЭНФ с помощью numbertheory.org. Результат.

Решение ПБВ ограничено размером . На вход идет матрица, в которой последняя строка является вектором, для которого надо найти ближайшую точку решетки.



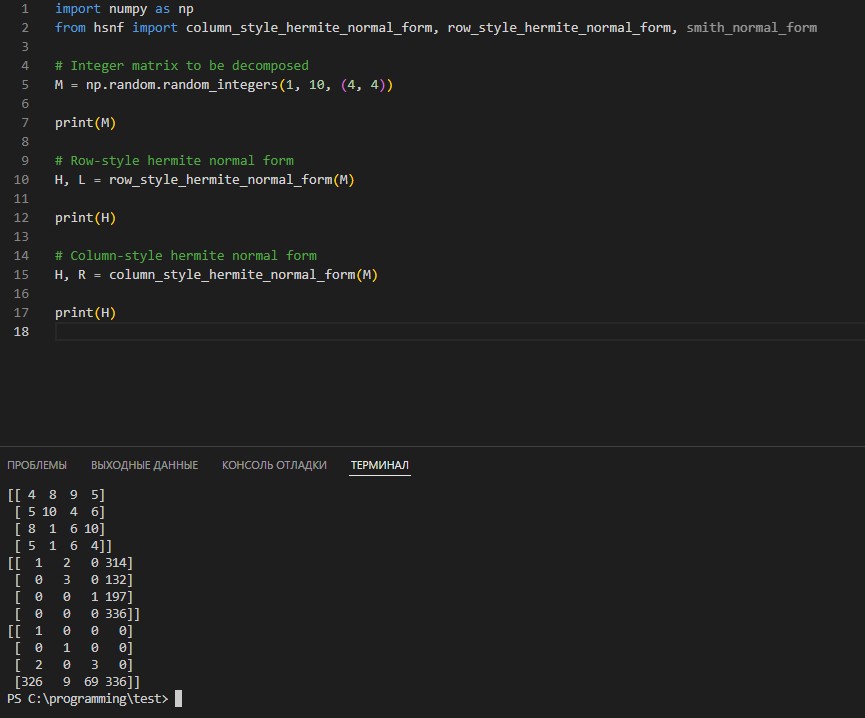
#### Рис. 4: Решение ПБВ с помощью numbertheory.org. Ввод данных.



#### Рис. 5: Решение ПБВ с помощью numbertheory.org. Результат.

* 1. hsnf

hsnf — библиотека для расчета Эрмитовой нормальной формы и нормальной формы Смита [9]. Написана на языке Python, легко интегрируется в программу. Главный минус — при больших размерах матриц выводит неправильные результаты, что делает его непригодным для применения на практике.



#### Рис. 6: Нахождение ЭНФ с помощью hsnf.

1. Обзор программной реализации

В ходе выполнения выпускной квалификационной работы была получена реализация описанных алгоритмов на языке C++. Для хранения исходного кода используется система контроля версий Git и сервис Github, где был создан репозиторий (**Repository?**). Программная реализация должна использоваться как подключаемая библиотека. Структура проекта следующая:

* В папке src содержатся файлы с исходным кодом в формате .cpp.
* В папке include содержатся подключаемые header файлы .hpp.
* В папке tex содержатся исходные .tex файлы документа выпускной квалификационной работы.
* В папке docs содержатся отчеты прошлых семестров.
* В папке 3rdparty содержатся модули Git.
* В папке cmake содержатся файлы для подключения сборок некоторых библиотек через CMake.
* CMakeLists.txt — файл CMake, использующийся для сборки проекта.

Проект автоматически собирается с помощью системы сборки CMake. Информация по сборке описана в README репозитория. По умолчанию отключена сборка документа выпускной квалификационной работы.

Программная реализация тестировалась с использованием компилятора G++ версии 6.3.0 в режиме сборки Release на ПК со следующими характеристиками: CPU: Intel(R) Core (TM) i5-9600KF CPU @ 3.70GHz, ОЗУ: DDR4, 16 ГБ (двухканальных режим 8х2), 2666 МГц. Тестирование проводилось на одинаковых данных.

* 1. Вспомогательные функции

Вспомогательные функции находятся в файле utils.cpp в пространстве имен Utils:

add\_column(HNF, column) — функция, используемая при нахождении ЭНФ. Принимает ЭНФ и возвращает .

reduce(vector, matrix) reduced\_vector — функция для сокращения вектора относительно диагональных элементов входного базиса.

generate\_random\_matrix\_with\_full\_row\_rank(m, n, lowest, highest) matrix — возвращает произвольную матрицу заданного размера с полным рангом строки с числами в заданном диапазоне.

generate\_random\_matrix(m, n, lowest, highest) matrix — возвращает произвольную матрицу заданного размера с числами в заданном диапазоне.

get\_linearly\_independent\_columns\_by\_gram\_schmidt(matrix) result\_matrix — возвращает линейно независимые столбцы матрицы и ортогонализованный базис.

get\_linearly\_independent\_rows\_by\_gram\_schmidt(matrix) result\_matrix — возвращает линейно независимые строки матрицы, их индексы в исходной матрице, индексы удаленных строк и матрицу .

gcd\_extended(a, b) g, x, y — расширенный НОД алгоритм, возвращает g, x, y такие, что .

add\_column\_GMP(HNF, column) — функция, используемая при нахождении ЭНФ. Принимает ЭНФ и возвращает . Использует реализацию больших чисел от GMP.

reduce\_GMP(vector, matrix) reduced\_vector — функция для сокращения вектора относительно диагональных элементов входного базиса. Использует реализацию больших чисел от GMP.

generate\_random\_matrix\_with\_full\_row\_rank\_GMP(m, n, lowest, highest) matrix — возвращает произвольную матрицу заданного размера с полным рангом строки с числами в заданном диапазоне. Использует реализацию больших чисел от GMP.

generate\_random\_matrix\_GMP(m, n, lowest, highest) matrix — возвращает произвольную матрицу заданного размера с числами в заданном диапазоне. Использует реализацию больших чисел от GMP.

get\_linearly\_independent\_columns\_by\_gram\_schmidt\_GMP(matrix) result\_matrix — возвращает линейно независимые столбцы матрицы и ортогонализованный базис. Использует реализацию больших чисел от GMP.

get\_linearly\_independent\_rows\_by\_gram\_schmidt\_GMP(matrix) result\_matrix — возвращает линейно независимые строки матрицы, их индексы в исходной матрице, индексы удаленных строк и матрицу . Использует реализацию больших чисел от GMP.

gcd\_extended\_GMP(a, b) g, x, y — расширенный НОД алгоритм, возвращает g, x, y такие, что . Использует реализацию больших чисел от GMP.

generate\_random\_matrix\_with\_full\_column\_rank(m, n, lowest, highest) matrix — возвращает произвольную матрицу заданного размера с полным рангом столбца с числами в заданном диапазоне.

generate\_random\_vector(m, lowest, highest) vector — возвращает случайный вектор заданного размера с числами в заданном диапазоне.

projection(matrix, vector) result\_vector — возвращает .

closest\_vector(matrix, vector) result\_vector — принимает набор векторов и целевой вектор, возвращает вектор из набора, ближайший к целевому.

* 1. Ортогонализация Грама-Шмидта

Реализация находится в файле algorithms.cpp в пространстве имен Utils и содержит 2 функции:

1. gram\_schmidt\_sequential(matrix, delete\_zero\_rows) result\_GS — принимает на вход матрицу и флаг, указывающий, следует ли удалять нулевые строки, и возвращает ортогонализацию Грама-Шмидта.
2. gram\_schmidt\_parallel(matrix, delete\_zero\_rows) result\_GS — принимает на вход матрицу и флаг, указывающий, следует ли удалять нулевые строки, и возвращает ортогонализацию Грама-Шмидта, вычисленную параллельным путем.

#### Таблица 1: Время нахождения ортогонализации Грама-Шмидта

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| m | 50 | 200 | 600 | 1000 | 2500 | 5000 |
| n | 50 | 200 | 600 | 1000 | 2500 | 5000 |
| Время, сек | 0.001 | 0.013 | 0.35 | 1.57 | 24.1 | 191.7 |

#### Таблица 2: Время параллельного нахождения ортогонализации Грама-Шмидта

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| m | 50 | 200 | 600 | 1000 | 2500 | 5000 |
| n | 50 | 200 | 600 | 1000 | 2500 | 5000 |
| Время, сек | 0.002 | 0.02 | 0.28 | 1.5 | 12.3 | 85.4 |

* 1. Нахождение ЭНФ

В ходе работы была получена реализация с использованием библиотеки Boost.Multiprecision. Реализация находится в файле algorithms.cpp в пространстве имен Algorithms::HNF и состоит из 4 функций:

1. HNF\_full\_row\_rank(matrix) result\_HNF — принимает на вход матрицу с полным рангом строки и возвращает ее ЭНФ. Использует встроенную реализацию больших чисел Boost.Multiprecision.
2. HNF(matrix) result\_HNF — принимает на вход матрицу и возвращает ее ЭНФ. Использует встроенную реализацию больших чисел Boost.Multiprecision.
3. HNF\_full\_row\_rank\_GMP(matrix) result\_HNF — принимает на вход матрицу с полным рангом строки и возвращает ее ЭНФ. Использует реализацию больших чисел от GMP.
4. HNF\_GMP(matrix) result\_HNF — принимает на вход матрицу и возвращает ее ЭНФ. Использует реализацию больших чисел от GMP.

#### Таблица 3: Время работы ЭНФ

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| m | 5 | 10 | 17 | 25 | 35 | 50 | 75 | 100 | 100 | 125 |
| n | 5 | 10 | 17 | 25 | 35 | 50 | 75 | 100 | 125 | 100 |
| Время, сек | 0.001 | 0.005 | 0.05 | 0.24 | 1.03 | 4.27 | 23.2 | 78.3 | 117.1 | 104.7 |

#### Таблица 4: Время работы ЭНФ с использованием GMP

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| m | 5 | 10 | 17 | 25 | 35 | 50 | 75 | 100 | 100 | 125 |
| n | 5 | 10 | 17 | 25 | 35 | 50 | 75 | 100 | 125 | 100 |
| Время, сек | 0.002 | 0.01 | 0.06 | 0.22 | 0.85 | 3.35 | 17.9 | 59.6 | 84.2 | 71.23 |

По временам видно, что чем больше размер входной матрицы, тем сильнее идет замедление по времени. На матрицах больших размеров следует использовать реализацию, которая использует библиотеку GMP.

* 1. Решение ПБВ

Реализация находится в файле algorithms.cpp в пространстве имен Algorithms::CVP и состоит из 4 функций:

1. greedy\_recursive(matrix, vector) vector — рекурсивный Greedy алгоритм, принимает на вход базис решетки и целевой вектор, возвращает вектор решетки, примерно ближайший к целевому.
2. greedy(matrix, vector) vector — последовательный Greedy алгоритм, принимает на вход базис решетки и целевой вектор, возвращает вектор решетки, примерно ближайший к целевому.
3. branch\_and\_bound(matrix, vector) vector — рекурсивный Branch and Bound алгоритм, принимает на вход базис решетки и целевой вектор, возвращает вектор решетки, ближайший к целевому.
4. greedy\_recursive(matrix, vector) vector — параллельный рекурсивный Branch and Bound алгоритм, принимает на вход базис решетки и целевой вектор, возвращает вектор решетки, ближайший к целевому.

#### Таблица 5: Время работы рекурсивного Greedy

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| m | 12 | 20 | 50 | 100 | 150 | 250 | 500 | 1000 | 1500 | 2500 | 3500 | 5000 |
| n | 12 | 20 | 50 | 100 | 150 | 250 | 500 | 1000 | 1500 | 2500 | 3500 | 5000 |
| Время, сек | 0.002 | 0.003 | 0.004 | 0.006 | 0.1 | 0.027 | 0.2 | 0.9 | 2.9 | 13.4 | 29.2 | 78.8 |

#### Таблица 6: Время работы нерекурсивного Greedy

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| m | 12 | 20 | 50 | 100 | 150 | 250 | 500 | 1000 | 1500 | 2500 | 3500 | 5000 |
| n | 12 | 20 | 50 | 100 | 150 | 250 | 500 | 1000 | 1500 | 2500 | 3500 | 5000 |
| Время, сек | 0.002 | 0.003 | 0.004 | 0.007 | 0.01 | 0.027 | 0.2 | 0.9 | 2.9 | 13.2 | 29 | 78.6 |

#### Таблица 7: Время работы нерекурсивного Greedy

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| m | 3 | 7 | 9 | 11 | 15 |
| n | 3 | 7 | 9 | 11 | 11 |
| Время, сек | 0.002 | 0.061 | 1.65 | 9.4 | 20.2 |

#### Таблица 8: Время работы параллельного Branch and Bound

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| m | 3 | 7 | 9 | 11 | 12 | 13 |
| n | 3 | 7 | 9 | 11 | 12 | 13 |
| Время, сек | 0.001 | 0.01 | 0.2 | 1.6 | 16.1 | 91.2 |

По временам видна заметная разница в скорости выполнения алгоритмов. Можно заметить, что сложность точного вычисления ПБВ сильно растет с увеличением количества столбцов базиса.

1. Заключение

В современной криптографии на решетках используются большие размерности базисов, что требует нахождения эффективных алгоритмов, которые помогут решать различные задачи теории решеток. Полученные в ходе выполнения выпускной квалификационной работы бакалавра алгоритмы, кроме метода Ветвей и границ, можно использовать на практике на относительно больших размерах решеток.

В ходе выполнения выпускной квалификационной работы бакалавра была написана библиотека, в которой реализованы алгоритмы для нахождения ЭНФ и решения ПБВ на языке C++. Полученную библиотеку можно подключать и использовать в других проектах.

Был создан Github репозиторий, который содержит в себе все исходные файлы программы, подключенные библиотеки и .tex файлы выпускной квалификационной работы. Программная реализация использует CMake для автоматической сборки исходного кода и .pdf документа.

Был получен опыт работы с языком C++, библиотеками для работы с линейной алгеброй и числами высокой точности, системой контроля версий Git, системой сборки CMake и написанием отчетов в формате .tex.

Список литературы

#### Daniele Micciancio. Basic Algorithms. [Электронный ресурс]. — URL: https://cseweb.ucsd.edu/classes/sp14/cse206A-a/lec4.pdf (Дата обращения: 16.05.2022).

#### Daniele Micciancio. Point Lattices. [Электронный ресурс]. — URL: https://cseweb.ucsd.edu/classes/sp14/cse206A-a/lec1.pdf (Дата обращения: 16.05.2022).

#### Shor P. W. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer // Foundations of Computer Science : Conference Publications. — 1997. — С. 1484–1509

#### Голубева Е.А. Линейная алгебра: Учебно-методическое пособие. – Нижний Новгород: Нижегородский госуниверситет, 2022. – 31 с.

#### Документация библиотеки Eigen. [Электронный ресурс]. — URL: https://eigen.tuxfamily.org/dox/index.html (Дата обращения: 16.05.2022).

#### Документация библиотеки Boost.Multiprecision. [Электронный ресурс]. — URL: [https://www.boost.org/doc/libs/1\_79\_0/libs/multiprecision/doc/html/index.html](http://www.boost.org/doc/libs/1_79_0/libs/multiprecision/doc/html/index.html) (Дата обра­щения: 16.05.2022).

#### Документация WolframAlpha API. [Электронный ресурс]. — URL: https://products.wolframalpha.com/simple-api/documentation/ (Дата обращения 16.05.2022).

#### Сервис для проверки Эрмитовой нормальной формы. [Электронный ресурс]. — URL:<http://www.numbertheory.org/php/lllhermite1.html> (Дата обращения: 16.05.2022).

#### Библиотека hsnf. [Электронный ресурс]. — URL: https://github.com/lan496/hsnf (Дата обра­щения: 16.05.2022).

#### Github репозиторий. [Электронный ресурс]. — URL: https://github.com/DenisOgnev/LatticeAlgorithms (Дата обращения: 16.05.2022).

Приложения

Приложение А. Исходный код algorithms.hpp

#ifndef ALGOTITHMS\_HPP

#define ALGOTITHMS\_HPP

#include <Eigen/Dense>

#include <boost/multiprecision/cpp\_int.hpp>

#ifdef GMP

#include <boost/multiprecision/gmp.hpp>

#endif

namespace Algorithms

{

    namespace HNF

    {

        Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1> HNF\_full\_row\_rank(const Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1> &B);

        Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1> HNF(const Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1> &B);

        #ifdef GMP

        Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1> HNF\_full\_row\_rank\_GMP(const Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1> &B);

        Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1> HNF\_GMP(const Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1> &B);

        #endif

    }

    namespace CVP

    {

        Eigen::VectorXd greedy\_recursive(const Eigen::MatrixXd &matrix, const Eigen::VectorXd &target);

        Eigen::VectorXd greedy(const Eigen::MatrixXd &matrix, const Eigen::VectorXd &target);

        Eigen::VectorXd branch\_and\_bound(const Eigen::MatrixXd &matrix, const Eigen::VectorXd &target);

        #ifdef PARALLEL

        Eigen::VectorXd branch\_and\_bound\_parallel(const Eigen::MatrixXd &matrix, const Eigen::VectorXd &target);

        #endif

    }

    #ifdef PARALLEL

    Eigen::MatrixXd gram\_schmidt\_parallel(const Eigen::MatrixXd &matrix, bool delete\_zero\_rows = true);

    #endif

    Eigen::MatrixXd gram\_schmidt\_sequential(const Eigen::MatrixXd &matrix, bool delete\_zero\_rows = true);

}

#endif

Приложение Б. Исходный код utils.hpp

#ifndef UTILS\_HPP

#define UTILS\_HPP

#include <Eigen/Dense>

#include <vector>

#include <boost/multiprecision/cpp\_int.hpp>

#include <boost/multiprecision/cpp\_bin\_float.hpp>

#ifdef GMP

#include <boost/multiprecision/gmp.hpp>

#endif

namespace Utils

{

    Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1> add\_column(const Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1> &H, const Eigen::Vector<boost::multiprecision::cpp\_int, -1> &b\_column);

    Eigen::Vector<boost::multiprecision::cpp\_int, -1> reduce(const Eigen::Vector<boost::multiprecision::cpp\_int, -1> &vector, const Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1> &matrix);

    Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1> generate\_random\_matrix\_with\_full\_row\_rank(const int m, const int n, int lowest, int highest);

    Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1> generate\_random\_matrix(const int m, const int n, int lowest, int highest);

    std::tuple<Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1>, Eigen::Matrix<boost::multiprecision::cpp\_rational, -1, -1>> get\_linearly\_independent\_columns\_by\_gram\_schmidt(const Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1> &matrix);

    std::tuple<Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1>, std::vector<int>, std::vector<int>, Eigen::Matrix<boost::multiprecision::cpp\_rational, -1, -1>> get\_linearly\_independent\_rows\_by\_gram\_schmidt(const Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1> &matrix);

    std::tuple<boost::multiprecision::cpp\_int, boost::multiprecision::cpp\_int, boost::multiprecision::cpp\_int> gcd\_extended(boost::multiprecision::cpp\_int a, boost::multiprecision::cpp\_int b);

    #ifdef GMP

    Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1> add\_column\_GMP(const Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1> &H, const Eigen::Vector<boost::multiprecision::mpz\_int, -1> &b\_column);

    Eigen::Vector<boost::multiprecision::mpz\_int, -1> reduce\_GMP(const Eigen::Vector<boost::multiprecision::mpz\_int, -1> &vector, const Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1> &matrix);

    Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1> generate\_random\_matrix\_with\_full\_row\_rank\_GMP(const int m, const int n, int lowest, int highest);

    Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1> generate\_random\_matrix\_GMP(const int m, const int n, int lowest, int highest);

    std::tuple<Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1>, Eigen::Matrix<boost::multiprecision::mpq\_rational, -1, -1>> get\_linearly\_independent\_columns\_by\_gram\_schmidt\_GMP(const Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1> &matrix);

    std::tuple<Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1>, std::vector<int>, std::vector<int>, Eigen::Matrix<boost::multiprecision::mpq\_rational, -1, -1>> get\_linearly\_independent\_rows\_by\_gram\_schmidt\_GMP(const Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1> &matrix);

    std::tuple<boost::multiprecision::mpz\_int, boost::multiprecision::mpz\_int, boost::multiprecision::mpz\_int> gcd\_extended\_GMP(boost::multiprecision::mpz\_int a, boost::multiprecision::mpz\_int b);

    #endif

    Eigen::MatrixXd generate\_random\_matrix\_with\_full\_column\_rank(const int m, const int n, int lowest, int highest);

    Eigen::VectorXd generate\_random\_vector(const int m, double lowest, double highest);

    Eigen::VectorXd projection(const Eigen::MatrixXd &matrix, const Eigen::VectorXd &vector);

    Eigen::VectorXd closest\_vector(const std::vector<Eigen::VectorXd> &matrix, const Eigen::VectorXd &vector);

}

#endif

Приложение В. Исходный код algorithms.cpp

#include "algorithms.hpp"

#include <iostream>

#include "utils.hpp"

#include <vector>

#include <numeric>

namespace mp = boost::multiprecision;

namespace Algorithms

{

    namespace HNF

    {

        // Computes HNF of a integer matrix that is full row rank

        // @return Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1>

        // @param B full row rank matrix

        Eigen::Matrix<mp::cpp\_int, -1, -1> HNF\_full\_row\_rank(const Eigen::Matrix<mp::cpp\_int, -1, -1> &B)

        {

            int m = static\_cast<int>(B.rows());

            int n = static\_cast<int>(B.cols());

            if (m > n)

            {

                throw std::invalid\_argument("m must be less than or equal n");

            }

            if (m < 1 || n < 1)

            {

                throw std::invalid\_argument("Matrix is not initialized");

            }

            if (B.isZero())

            {

                throw std::runtime\_error("Matrix is empty");

            }

            Eigen::Matrix<mp::cpp\_int, -1, -1> B\_stroke;

            Eigen::Matrix<mp::cpp\_rational, -1, -1> ortogonalized;

            std::tuple<Eigen::Matrix<mp::cpp\_int, -1, -1>, Eigen::Matrix<mp::cpp\_rational, -1, -1>> result\_of\_gs = Utils::get\_linearly\_independent\_columns\_by\_gram\_schmidt(B);

            std::tie(B\_stroke, ortogonalized) = result\_of\_gs;

            mp::cpp\_rational t\_det = 1.0;

            for (const Eigen::Vector<mp::cpp\_rational, -1> &vec : ortogonalized.colwise())

            {

                t\_det \*= vec.squaredNorm();

            }

            mp::cpp\_int det = mp::sqrt(mp::numerator(t\_det));

            Eigen::Matrix<mp::cpp\_int, -1, -1> H\_temp = Eigen::Matrix<mp::cpp\_int, -1, -1>::Identity(m, m) \* det;

            for (int i = 0; i < n; i++)

            {

                H\_temp = Utils::add\_column(H\_temp, B.col(i));

            }

            Eigen::Matrix<mp::cpp\_int, -1, -1> H(m, n);

            H.block(0, 0, H\_temp.rows(), H\_temp.cols()) = H\_temp;

            if (n > m)

            {

                H.block(0, H\_temp.cols(), H\_temp.rows(), n - m) = Eigen::Matrix<mp::cpp\_int, -1, -1>::Zero(H\_temp.rows(), n - m);

            }

            return H;

        }

        // Computes HNF of an arbitrary integer matrix

        // @return Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1>

        // @param B arbitrary matrix

        Eigen::Matrix<mp::cpp\_int, -1, -1> HNF(const Eigen::Matrix<mp::cpp\_int, -1, -1> &B)

        {

            int m = static\_cast<int>(B.rows());

            int n = static\_cast<int>(B.cols());

            if (m < 1 || n < 1)

            {

                throw std::invalid\_argument("Matrix is not initialized");

            }

            if (B.isZero())

            {

                throw std::runtime\_error("Matrix is empty");

            }

            Eigen::Matrix<mp::cpp\_int, -1, -1> B\_stroke;

            std::vector<int> indicies;

            std::vector<int> deleted\_indicies;

            Eigen::Matrix<mp::cpp\_rational, -1, -1> T;

            std::tuple<Eigen::Matrix<mp::cpp\_int, -1, -1>, std::vector<int>, std::vector<int>, Eigen::Matrix<mp::cpp\_rational, -1, -1>> projection = Utils::get\_linearly\_independent\_rows\_by\_gram\_schmidt(B);

            std::tie(B\_stroke, indicies, deleted\_indicies, T) = projection;

            Eigen::Matrix<mp::cpp\_int, -1, -1> B\_double\_stroke = HNF\_full\_row\_rank(B\_stroke);

            Eigen::Matrix<mp::cpp\_int, -1, -1> HNF(B.rows(), B.cols());

            for (int i = 0; i < indicies.size(); i++)

            {

                HNF.row(indicies[i]) = B\_double\_stroke.row(i);

            }

            /////////////////////////////////////////////

            // First way: just find linear combinations of deleted rows. More accurate

            // Eigen::Matrix<mp::cpp\_bin\_float\_double, -1, -1> B\_stroke\_transposed = B\_stroke.transpose().cast<mp::cpp\_bin\_float\_double>();

            // auto QR = B\_stroke.cast<mp::cpp\_bin\_float\_double>().colPivHouseholderQr().transpose();

            // for (const auto &indx : deleted\_indicies)

            // {

            //     Eigen::Vector<mp::cpp\_bin\_float\_double, -1> vec = B.row(indx).cast<mp::cpp\_bin\_float\_double>();

            //     Eigen::RowVector<mp::cpp\_bin\_float\_double, -1> x = QR.solve(vec);

            //     Eigen::Vector<mp::cpp\_bin\_float\_double, -1> res = x \* HNF.cast<mp::cpp\_bin\_float\_double>();

            //     for (mp::cpp\_bin\_float\_double &elem : res)

            //     {

            //         elem = mp::round(elem);

            //     }

            //     HNF.row(indx) = res.cast<mp::cpp\_int>();

            // }

            // return HNF;

            /////////////////////////////////////////////

            /////////////////////////////////////////////

            // Other, the "right" way that is desribed in algorithm.

            Eigen::Matrix<mp::cpp\_bin\_float\_double, -1, -1> t\_HNF = HNF.cast<mp::cpp\_bin\_float\_double>();

            for (const auto &indx : deleted\_indicies)

            {

                Eigen::Vector<mp::cpp\_bin\_float\_double, -1> res = Eigen::Vector<mp::cpp\_bin\_float\_double, -1>::Zero(B.cols());

                for (int i = 0; i < indx; i++)

                {

                    res += T(indx, i).convert\_to<mp::cpp\_bin\_float\_double>() \* t\_HNF.row(i);

                }

                t\_HNF.row(indx) = res;

            }

            return t\_HNF.cast<mp::cpp\_int>();

            /////////////////////////////////////////////

        }

        #ifdef GMP

        // Computes HNF of a integer matrix that is full row rank

        // @return Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1>

        // @param B full row rank matrix

        Eigen::Matrix<mp::mpz\_int, -1, -1> HNF\_full\_row\_rank\_GMP(const Eigen::Matrix<mp::mpz\_int, -1, -1> &B)

        {

            int m = static\_cast<int>(B.rows());

            int n = static\_cast<int>(B.cols());

            if (m > n)

            {

                throw std::invalid\_argument("m must be less than or equal n");

            }

            if (m < 1 || n < 1)

            {

                throw std::invalid\_argument("Matrix is not initialized");

            }

            if (B.isZero())

            {

                throw std::runtime\_error("Matrix is empty");

            }

            Eigen::Matrix<mp::mpz\_int, -1, -1> B\_stroke;

            Eigen::Matrix<mp::mpq\_rational, -1, -1> ortogonalized;

            std::tuple<Eigen::Matrix<mp::mpz\_int, -1, -1>, Eigen::Matrix<mp::mpq\_rational, -1, -1>> result\_of\_gs = Utils::get\_linearly\_independent\_columns\_by\_gram\_schmidt\_GMP(B);

            std::tie(B\_stroke, ortogonalized) = result\_of\_gs;

            mp::mpq\_rational t\_det = 1.0;

            for (const Eigen::Vector<mp::mpq\_rational, -1> &vec : ortogonalized.colwise())

            {

                t\_det \*= vec.squaredNorm();

            }

            mp::mpz\_int det = mp::sqrt(mp::numerator(t\_det));

            Eigen::Matrix<mp::mpz\_int, -1, -1> H\_temp = Eigen::Matrix<mp::mpz\_int, -1, -1>::Identity(m, m) \* det;

            for (int i = 0; i < n; i++)

            {

                H\_temp = Utils::add\_column\_GMP(H\_temp, B.col(i));

            }

            Eigen::Matrix<mp::mpz\_int, -1, -1> H(m, n);

            H.block(0, 0, H\_temp.rows(), H\_temp.cols()) = H\_temp;

            if (n > m)

            {

                H.block(0, H\_temp.cols(), H\_temp.rows(), n - m) = Eigen::Matrix<mp::mpz\_int, -1, -1>::Zero(H\_temp.rows(), n - m);

            }

            return H;

        }

        // Computes HNF of an arbitrary integer matrix

        // @return Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1>

        // @param B arbitrary matrix

        Eigen::Matrix<mp::mpz\_int, -1, -1> HNF\_GMP(const Eigen::Matrix<mp::mpz\_int, -1, -1> &B)

        {

            int m = static\_cast<int>(B.rows());

            int n = static\_cast<int>(B.cols());

            if (m < 1 || n < 1)

            {

                throw std::invalid\_argument("Matrix is not initialized");

            }

            if (B.isZero())

            {

                throw std::runtime\_error("Matrix is empty");

            }

            Eigen::Matrix<mp::mpz\_int, -1, -1> B\_stroke;

            std::vector<int> indicies;

            std::vector<int> deleted\_indicies;

            Eigen::Matrix<mp::mpq\_rational, -1, -1> T;

            std::tuple<Eigen::Matrix<mp::mpz\_int, -1, -1>, std::vector<int>, std::vector<int>, Eigen::Matrix<mp::mpq\_rational, -1, -1>> projection = Utils::get\_linearly\_independent\_rows\_by\_gram\_schmidt\_GMP(B);

            std::tie(B\_stroke, indicies, deleted\_indicies, T) = projection;

            Eigen::Matrix<mp::mpz\_int, -1, -1> B\_double\_stroke = HNF\_full\_row\_rank\_GMP(B\_stroke);

            Eigen::Matrix<mp::mpz\_int, -1, -1> HNF(B.rows(), B.cols());

            for (int i = 0; i < indicies.size(); i++)

            {

                HNF.row(indicies[i]) = B\_double\_stroke.row(i);

            }

            /////////////////////////////////////////////

            // First way: just find linear combinations of deleted rows. More accurate

            // Eigen::Matrix<mp::mpf\_float\_50, -1, -1> B\_stroke\_transposed = B\_stroke.transpose().cast<mp::mpf\_float\_50>();

            // auto QR = B\_stroke.cast<mp::mpf\_float\_50>().colPivHouseholderQr().transpose();

            // for (const auto &indx : deleted\_indicies)

            // {

            //     Eigen::Vector<mp::mpf\_float\_50, -1> vec = B.row(indx).cast<mp::mpf\_float\_50>();

            //     Eigen::RowVector<mp::mpf\_float\_50, -1> x = QR.solve(vec);

            //     Eigen::Vector<mp::mpf\_float\_50, -1> res = x \* HNF.cast<mp::mpf\_float\_50>();

            //     for (mp::mpf\_float\_50 &elem : res)

            //     {

            //         elem = mp::round(elem);

            //     }

            //     HNF.row(indx) = res.cast<mp::mpz\_int>();

            // }

            // return HNF;

            /////////////////////////////////////////////

            /////////////////////////////////////////////

            // Other, the "right" way that is desribed in algorithm.

            Eigen::Matrix<mp::mpf\_float\_50, -1, -1> t\_HNF = HNF.cast<mp::mpf\_float\_50>();

            for (const auto &indx : deleted\_indicies)

            {

                Eigen::Vector<mp::mpf\_float\_50, -1> res = Eigen::Vector<mp::mpf\_float\_50, -1>::Zero(B.cols());

                for (int i = 0; i < indx; i++)

                {

                    res += T(indx, i).convert\_to<mp::mpf\_float\_50>() \* t\_HNF.row(i);

                }

                t\_HNF.row(indx) = res;

            }

            return t\_HNF.cast<mp::mpz\_int>();

            /////////////////////////////////////////////

        }

        #endif

    }

    namespace CVP

    {

        Eigen::MatrixXd gram\_schmidt\_greedy;

        Eigen::MatrixXd B\_greedy;

        int index\_greedy;

        Eigen::MatrixXd gram\_schmidt\_bb;

        Eigen::MatrixXd gram\_schmidt\_bb\_parallel;

        // Recursive body of greedy algorithm

        // @return Eigen::VectorXd

        // @param target vector for which lattice point is being searched for

        Eigen::VectorXd greedy\_recursive\_part(const Eigen::VectorXd &target)

        {

            if (index\_greedy == 0)

            {

                return Eigen::VectorXd::Zero(target.rows());

            }

            index\_greedy--;

            Eigen::VectorXd b = B\_greedy.col(index\_greedy);

            Eigen::VectorXd b\_star = gram\_schmidt\_greedy.col(index\_greedy);

            double inner1 = std::inner\_product(target.data(), target.data() + target.size(), b\_star.data(), 0.0);

            double inner2 = std::inner\_product(b\_star.data(), b\_star.data() + b\_star.size(), b\_star.data(), 0.0);

            double x = inner1 / inner2;

            double c = std::round(x);

            Eigen::VectorXd t\_res = c \* b;

            return t\_res + Algorithms::CVP::greedy\_recursive\_part(target - t\_res);

        }

        // Solves CVP using a recursive greedy algorithm

        // @return Eigen::VectorXd

        // @param matrix input rational lattice basis that is linearly independent

        // @param target vector for which lattice point is being searched for

        Eigen::VectorXd greedy\_recursive(const Eigen::MatrixXd &matrix, const Eigen::VectorXd &target)

        {

            B\_greedy = matrix;

            #ifdef PARALLEL

            gram\_schmidt\_greedy = Algorithms::gram\_schmidt\_parallel(matrix, false);

            #else

            gram\_schmidt\_greedy = Algorithms::gram\_schmidt\_sequential(matrix, false);

            #endif

            index\_greedy = static\_cast<int>(matrix.cols());

            return greedy\_recursive\_part(target);

        }

        // Solves CVP using a non recursive greedy algorithm

        // @return Eigen::VectorXd

        // @param matrix input rational lattice basis that is linearly independent

        // @param target vector for which lattice point is being searched for

        Eigen::VectorXd greedy(const Eigen::MatrixXd &matrix, const Eigen::VectorXd &target)

        {

            Eigen::MatrixXd gram\_schmidt;

            #ifdef PARALLEL

            gram\_schmidt = Algorithms::gram\_schmidt\_parallel(matrix, false);

            #else

            gram\_schmidt = Algorithms::gram\_schmidt\_sequential(matrix, false);

            #endif

            Eigen::VectorXd result = Eigen::VectorXd::Zero(target.rows());

            Eigen::VectorXd t\_target = target;

            int n = static\_cast<int>(matrix.cols());

            for (int i = 0; i < matrix.cols(); i++)

            {

                int index = n - i - 1;

                Eigen::VectorXd b = matrix.col(index);

                Eigen::VectorXd b\_star = gram\_schmidt.col(index);

                double inner1 = std::inner\_product(t\_target.data(), t\_target.data() + t\_target.size(), b\_star.data(), 0.0);

                double inner2 = std::inner\_product(b\_star.data(), b\_star.data() + b\_star.size(), b\_star.data(), 0.0);

                double x = inner1 / inner2;

                double c = std::round(x);

                Eigen::VectorXd t\_res = c \* b;

                t\_target -= t\_res;

                result += t\_res;

            }

            return result;

        }

        // Recursive body of branch and bound algorithm

        // @return Eigen::VectorXd

        // @param matrix input rational lattice basis that is linearly independent

        // @param target vector for which lattice point is being searched for

        Eigen::VectorXd branch\_and\_bound\_recursive\_part(const Eigen::MatrixXd &matrix, const Eigen::VectorXd &target)

        {

            if (matrix.cols() == 0)

            {

                return Eigen::VectorXd::Zero(target.rows());

            }

            Eigen::MatrixXd B = matrix.block(0, 0, matrix.rows(), matrix.cols() - 1);

            Eigen::VectorXd b = matrix.col(B.cols());

            Eigen::VectorXd b\_star = gram\_schmidt\_bb.col(B.cols());

            Eigen::VectorXd v = Algorithms::CVP::greedy(matrix, target);

            double upper\_bound = (target - v).norm();

            double x\_middle = std::round(target.dot(b\_star) / b\_star.dot(b\_star));

            std::vector<int> X;

            X.push\_back(static\_cast<int>(x\_middle));

            bool flag1 = true;

            bool flag2 = true;

            double x1 = x\_middle + 1;

            double x2 = x\_middle - 1;

            while (flag1 || flag2)

            {

                if (flag1 && Utils::projection(B, target - x1 \* b).norm() <= upper\_bound)

                {

                    X.push\_back(static\_cast<int>(x1));

                    x1++;

                }

                else

                {

                    flag1 = false;

                }

                if (flag2 && Utils::projection(B, target - x2 \* b).norm() <= upper\_bound)

                {

                    X.push\_back(static\_cast<int>(x2));

                    x2--;

                }

                else

                {

                    flag2 = false;

                }

            }

            std::vector<Eigen::VectorXd> V;

            Eigen::VectorXd t\_res;

            for (const int &x : X)

            {

                t\_res = x \* b + Algorithms::CVP::branch\_and\_bound\_recursive\_part(B, target - x \* b);

                V.push\_back(t\_res);

            }

            return Utils::closest\_vector(V, target);

        }

        // Solves CVP using a branch and bound algorithm

        // @return Eigen::VectorXd

        // @param matrix input rational lattice basis that is linearly independent

        // @param target vector for which lattice point is being searched for

        Eigen::VectorXd branch\_and\_bound(const Eigen::MatrixXd &matrix, const Eigen::VectorXd &target)

        {

            #ifdef PARALLEL

            gram\_schmidt\_bb = Algorithms::gram\_schmidt\_parallel(matrix, false);

            #else

            gram\_schmidt\_bb = Algorithms::gram\_schmidt\_sequential(matrix, false);

            #endif

            return branch\_and\_bound\_recursive\_part(matrix, target);

        }

        #ifdef PARALLEL

        // Recursive parallel body of branch and bound algorithm

        // @return Eigen::VectorXd

        // @param matrix input rational lattice basis that is linearly independent

        // @param target vector for which lattice point is being searched for

        Eigen::VectorXd branch\_and\_bound\_recursive\_part\_parallel(const Eigen::MatrixXd &matrix, const Eigen::VectorXd &target)

        {

            if (matrix.cols() == 0)

            {

                return Eigen::VectorXd::Zero(target.rows());

            }

            Eigen::MatrixXd B = matrix.block(0, 0, matrix.rows(), matrix.cols() - 1);

            Eigen::VectorXd b = matrix.col(B.cols());

            Eigen::VectorXd b\_star = gram\_schmidt\_bb\_parallel.col(B.cols());

            Eigen::VectorXd v = Algorithms::CVP::greedy(matrix, target);

            double upper\_bound = (target - v).norm();

            double x\_middle = std::round(target.dot(b\_star) / b\_star.dot(b\_star));

            std::vector<int> X;

            X.push\_back(static\_cast<int>(x\_middle));

            bool flag1 = true;

            bool flag2 = true;

            double x1 = x\_middle + 1;

            double x2 = x\_middle - 1;

            while (flag1 || flag2)

            {

                if (flag1 && Utils::projection(B, target - x1 \* b).norm() <= upper\_bound)

                {

                    X.push\_back(static\_cast<int>(x1));

                    x1++;

                }

                else

                {

                    flag1 = false;

                }

                if (flag2 && Utils::projection(B, target - x2 \* b).norm() <= upper\_bound)

                {

                    X.push\_back(static\_cast<int>(x2));

                    x2--;

                }

                else

                {

                    flag2 = false;

                }

            }

            std::vector<Eigen::VectorXd> V;

            Eigen::VectorXd result;

            Eigen::VectorXd res;

            #pragma omp parallel

            {

                #pragma omp single nowait

                {

                    for (const int &x : X)

                    {

                        #pragma omp task

                        {

                            res = x \* b + Algorithms::CVP::branch\_and\_bound\_recursive\_part\_parallel(B, target - x \* b);

                            #pragma omp critical

                            V.push\_back(res);

                        }

                    }

                    #pragma omp taskwait

                    result = Utils::closest\_vector(V, target);

                }

            }

            return result;

        }

        // Solves CVP using a branch and bound parallel algorithm

        // @return Eigen::VectorXd

        // @param matrix input rational lattice basis that is linearly independent

        // @param target vector for which lattice point is being searched for

        Eigen::VectorXd branch\_and\_bound\_parallel(const Eigen::MatrixXd &matrix, const Eigen::VectorXd &target)

        {

            gram\_schmidt\_bb\_parallel = Algorithms::gram\_schmidt\_parallel(matrix, false);

            return branch\_and\_bound\_recursive\_part\_parallel(matrix, target);

        }

        #endif

    }

    #ifdef PARALLEL

    // Computes Gram Schmidt orthogonalization

    // @return Eigen::MatrixXd

    // @param matrix input matrix

    // @param normalize indicates whether to normalize output vectors

    // @param delete\_zero\_rows indicates whether to delete zero rows

    Eigen::MatrixXd gram\_schmidt\_parallel(const Eigen::MatrixXd &matrix, bool delete\_zero\_rows)

    {

        std::vector<Eigen::VectorXd> basis;

        for (const auto &vec : matrix.colwise())

        {

            Eigen::VectorXd projections = Eigen::VectorXd::Zero(vec.size());

            #pragma omp parallel for

            for (int i = 0; i < basis.size(); i++)

            {

                Eigen::MatrixXd basis\_vector = basis[i];

                double inner1 = std::inner\_product(vec.data(), vec.data() + vec.size(), basis\_vector.data(), 0.0);

                double inner2 = std::inner\_product(basis\_vector.data(), basis\_vector.data() + basis\_vector.size(), basis\_vector.data(), 0.0);

                #pragma omp critical

                projections += (inner1 / inner2) \* basis\_vector;

            }

            Eigen::VectorXd result = vec - projections;

            if (delete\_zero\_rows)

            {

                bool is\_all\_zero = result.isZero(1e-3);

                if (!is\_all\_zero)

                {

                    basis.push\_back(result);

                }

            }

            else

            {

                basis.push\_back(result);

            }

        }

        Eigen::MatrixXd result(matrix.rows(), basis.size());

        for (int i = 0; i < basis.size(); i++)

        {

            result.col(i) = basis[i];

        }

        return result;

    }

    #endif

    // Computes Gram Schmidt orthogonalization

    // @return Eigen::MatrixXd

    // @param matrix input matrix

    // @param normalize indicates whether to normalize output vectors

    // @param delete\_zero\_rows indicates whether to delete zero rows

    Eigen::MatrixXd gram\_schmidt\_sequential(const Eigen::MatrixXd &matrix, bool delete\_zero\_rows)

    {

        std::vector<Eigen::VectorXd> basis;

        for (const auto &vec : matrix.colwise())

        {

            Eigen::VectorXd projections = Eigen::VectorXd::Zero(vec.size());

            for (int i = 0; i < basis.size(); i++)

            {

                Eigen::MatrixXd basis\_vector = basis[i];

                double inner1 = std::inner\_product(vec.data(), vec.data() + vec.size(), basis\_vector.data(), 0.0);

                double inner2 = std::inner\_product(basis\_vector.data(), basis\_vector.data() + basis\_vector.size(), basis\_vector.data(), 0.0);

                projections += (inner1 / inner2) \* basis\_vector;

            }

            Eigen::VectorXd result = vec - projections;

            if (delete\_zero\_rows)

            {

                bool is\_all\_zero = result.isZero(1e-3);

                if (!is\_all\_zero)

                {

                    basis.push\_back(result);

                }

            }

            else

            {

                basis.push\_back(result);

            }

        }

        Eigen::MatrixXd result(matrix.rows(), basis.size());

        for (int i = 0; i < basis.size(); i++)

        {

            result.col(i) = basis[i];

        }

        return result;

    }

}

Приложение Г. Исходный код utils.cpp

#include "utils.hpp"

#include <iostream>

#include <random>

#include <functional>

#include <numeric>

#include <vector>

#include <stdexcept>

#include <string>

#include <chrono>

#include <algorithm>

#include <thread>

#include "algorithms.hpp"

namespace mp = boost::multiprecision;

namespace Utils

{

    // Function for computing HNF of full row rank matrix

    // @return Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1>

    // @param H HNF

    // @param b column to be added

    Eigen::Matrix<mp::cpp\_int, -1, -1> add\_column(const Eigen::Matrix<mp::cpp\_int, -1, -1> &H, const Eigen::Vector<mp::cpp\_int, -1> &b\_column)

    {

        if (H.rows() == 0)

        {

            return H;

        }

        Eigen::Vector<mp::cpp\_int, -1> H\_first\_col = H.col(0);

        mp::cpp\_int a = H\_first\_col(0);

        Eigen::Vector<mp::cpp\_int, -1> h = H\_first\_col.tail(H\_first\_col.rows() - 1);

        Eigen::Matrix<mp::cpp\_int, -1, -1> H\_stroke = H.block(1, 1, H.rows() - 1, H.cols() - 1);

        mp::cpp\_int b = b\_column(0);

        Eigen::Vector<mp::cpp\_int, -1> b\_stroke = b\_column.tail(b\_column.rows() - 1);

        std::tuple<mp::cpp\_int, mp::cpp\_int, mp::cpp\_int> gcd\_result = gcd\_extended(a, b);

        mp::cpp\_int g, x, y;

        std::tie(g, x, y) = gcd\_result;

        Eigen::Matrix<mp::cpp\_int, 2, 2> U;

        U << x, -b / g, y, a / g;

        Eigen::Matrix<mp::cpp\_int, -1, 2> temp\_matrix(H.rows(), 2);

        temp\_matrix.col(0) = H\_first\_col;

        temp\_matrix.col(1) = b\_column;

        Eigen::Matrix<mp::cpp\_int, -1, 2> temp\_result = temp\_matrix \* U;

        Eigen::Vector<mp::cpp\_int, -1> h\_stroke = temp\_result.col(0).tail(temp\_result.rows() - 1);

        Eigen::Vector<mp::cpp\_int, -1> b\_double\_stroke = temp\_result.col(1).tail(temp\_result.rows() - 1);

        b\_double\_stroke = reduce(b\_double\_stroke, H\_stroke);

        Eigen::Matrix<mp::cpp\_int, -1, -1> H\_double\_stroke = add\_column(H\_stroke, b\_double\_stroke);

        h\_stroke = reduce(h\_stroke, H\_double\_stroke);

        Eigen::Matrix<mp::cpp\_int, -1, -1> result(H.rows(), H.cols());

        result(0, 0) = g;

        result.col(0).tail(result.cols() - 1) = h\_stroke;

        result.row(0).tail(result.rows() - 1).setZero();

        result.block(1, 1, H\_double\_stroke.rows(), H\_double\_stroke.cols()) = H\_double\_stroke;

        return result;

    }

    // Function for computing HNF, reduces elements of vector modulo diagonal elements of matrix

    // @return Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1>

    // @param vector vector to be reduced

    // @param matrix input matrix

    Eigen::Vector<mp::cpp\_int, -1> reduce(const Eigen::Vector<mp::cpp\_int, -1> &vector, const Eigen::Matrix<mp::cpp\_int, -1, -1> &matrix)

    {

        Eigen::Vector<mp::cpp\_int, -1> result = vector;

        for (int i = 0; i < result.rows(); i++)

        {

            Eigen::Vector<mp::cpp\_int, -1> matrix\_column = matrix.col(i);

            mp::cpp\_int t\_vec\_elem = result(i);

            mp::cpp\_int t\_matrix\_elem = matrix(i, i);

            mp::cpp\_int x;

            if (t\_vec\_elem >= 0)

            {

                x = (t\_vec\_elem / t\_matrix\_elem);

            }

            else

            {

                x = (t\_vec\_elem - (t\_matrix\_elem - 1)) / t\_matrix\_elem;

            }

            result -= matrix\_column \* x;

        }

        return result;

    }

    // Generates random matrix with full row rank

    // @return Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1>

    // @param m number of rows, must be greater than one and less than or equal to the parameter n

    // @param n number of columns, must be greater than one and greater than or equal to the parameter m

    // @param lowest lowest generated number, must be lower than lowest parameter by at least one

    // @param highest highest generated number, must be greater than lowest parameter by at least one

    Eigen::Matrix<mp::cpp\_int, -1, -1> generate\_random\_matrix\_with\_full\_row\_rank(const int m, const int n, int lowest, int highest)

    {

        if (m > n)

        {

            throw std::invalid\_argument("m must be less than or equal n");

        }

        if (m < 1 || n < 1)

        {

            throw std::invalid\_argument("Number of rows or columns should be greater than one");

        }

        if (highest - lowest < 1)

        {

            throw std::invalid\_argument("highest parameter must be greater than lowest parameter by at least one");

        }

        std::random\_device rd;

        std::mt19937 gen(rd());

        std::uniform\_int\_distribution<int> dis (lowest, highest);

        Eigen::Matrix<int, -1, -1> matrix = Eigen::Matrix<int, -1, -1>::NullaryExpr(m, n, [&]()

                                                              { return dis(gen); });

        Eigen::FullPivLU<Eigen::MatrixXd> lu\_decomp(matrix.cast<double>());

        auto rank = lu\_decomp.rank();

        while (rank != m)

        {

            matrix = Eigen::Matrix<int, -1, -1>::NullaryExpr(m, n, [&]()

                                                  { return dis(gen); });

            lu\_decomp.compute(matrix.cast<double>());

            rank = lu\_decomp.rank();

        }

        return matrix.cast<mp::cpp\_int>();

    }

    // Generates random matrix

    // @return Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1>

    // @param m number of rows, must be greater than one

    // @param n number of columns, must be greater than one

    // @param lowest lowest generated number, must be lower than lowest parameter by at least one

    // @param highest highest generated number, must be greater than lowest parameter by at least one

    Eigen::Matrix<mp::cpp\_int, -1, -1> generate\_random\_matrix(const int m, const int n, int lowest, int highest)

    {

        if (m < 1 || n < 1)

        {

            throw std::invalid\_argument("Number of rows or columns should be greater than one");

        }

        if (highest - lowest < 1)

        {

            throw std::invalid\_argument("highest parameter must be greater than lowest parameter by at least one");

        }

        std::random\_device rd;

        std::mt19937 gen(rd());

        std::uniform\_int\_distribution<int> dis (lowest, highest);

        Eigen::Matrix<int, -1, -1> matrix = Eigen::Matrix<int, -1, -1>::NullaryExpr(m, n, [&]()

                                                              { return dis(gen); });

        return matrix.cast<mp::cpp\_int>();

    }

    #ifdef PARALLEL

    // Returns matrix that consist of linearly independent columns of input matrix and othogonalized matrix

    // @param matrix input matrix

    // @return std::tuple<Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1>, Eigen::Matrix<boost::multiprecision::cpp\_rational, -1, -1>>

    std::tuple<Eigen::Matrix<mp::cpp\_int, -1, -1>, Eigen::Matrix<mp::cpp\_rational, -1, -1>> get\_linearly\_independent\_columns\_by\_gram\_schmidt(const Eigen::Matrix<mp::cpp\_int, -1, -1> &matrix)

    {

        std::vector<Eigen::Vector<mp::cpp\_rational, -1>> basis;

        std::vector<int> indexes;

        int counter = 0;

        for (const Eigen::Vector<mp::cpp\_rational, -1> &vec : matrix.cast<mp::cpp\_rational>().colwise())

        {

            Eigen::Vector<mp::cpp\_rational, -1> projections = Eigen::Vector<mp::cpp\_rational, -1>::Zero(vec.size());

            #pragma omp parallel for

            for (int i = 0; i < basis.size(); i++)

            {

                Eigen::Vector<mp::cpp\_rational, -1> basis\_vector = basis[i];

                mp::cpp\_rational inner1 = std::inner\_product(vec.data(), vec.data() + vec.size(), basis\_vector.data(), mp::cpp\_rational(0.0));

                mp::cpp\_rational inner2 = std::inner\_product(basis\_vector.data(), basis\_vector.data() + basis\_vector.size(), basis\_vector.data(), mp::cpp\_rational(0.0));

                mp::cpp\_rational coef = inner1 / inner2;

                #pragma omp critical

                projections += basis\_vector \* coef;

            }

            Eigen::Vector<mp::cpp\_rational, -1> result = vec - projections;

            bool is\_all\_zero = result.isZero(1e-3);

            if (!is\_all\_zero)

            {

                basis.push\_back(result);

                indexes.push\_back(counter);

            }

            counter++;

        }

        Eigen::Matrix<mp::cpp\_int, -1, -1> result(matrix.rows(), indexes.size());

        Eigen::Matrix<mp::cpp\_rational, -1, -1> gram\_schmidt(matrix.rows(), basis.size());

        for (int i = 0; i < indexes.size(); i++)

        {

            result.col(i) = matrix.col(indexes[i]);

            gram\_schmidt.col(i) = basis[i];

        }

        return std::make\_tuple(result, gram\_schmidt);

    }

    #else

    // Returns matrix that consist of linearly independent columns of input matrix and othogonalized matrix

    // @param matrix input matrix

    // @return std::tuple<Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1>, Eigen::Matrix<boost::multiprecision::cpp\_rational, -1, -1>>

    std::tuple<Eigen::Matrix<mp::cpp\_int, -1, -1>, Eigen::Matrix<mp::cpp\_rational, -1, -1>> get\_linearly\_independent\_columns\_by\_gram\_schmidt(const Eigen::Matrix<mp::cpp\_int, -1, -1> &matrix)

    {

        std::vector<Eigen::Vector<mp::cpp\_rational, -1>> basis;

        std::vector<int> indexes;

        int counter = 0;

        for (const Eigen::Vector<mp::cpp\_rational, -1> &vec : matrix.cast<mp::cpp\_rational>().colwise())

        {

            Eigen::Vector<mp::cpp\_rational, -1> projections = Eigen::Vector<mp::cpp\_rational, -1>::Zero(vec.size());

            for (int i = 0; i < basis.size(); i++)

            {

                Eigen::Vector<mp::cpp\_rational, -1> basis\_vector = basis[i];

                mp::cpp\_rational inner1 = std::inner\_product(vec.data(), vec.data() + vec.size(), basis\_vector.data(), mp::cpp\_rational(0.0));

                mp::cpp\_rational inner2 = std::inner\_product(basis\_vector.data(), basis\_vector.data() + basis\_vector.size(), basis\_vector.data(), mp::cpp\_rational(0.0));

                mp::cpp\_rational coef = inner1 / inner2;

                projections += basis\_vector \* coef;

            }

            Eigen::Vector<mp::cpp\_rational, -1> result = vec - projections;

            bool is\_all\_zero = result.isZero(1e-3);

            if (!is\_all\_zero)

            {

                basis.push\_back(result);

                indexes.push\_back(counter);

            }

            counter++;

        }

        Eigen::Matrix<mp::cpp\_int, -1, -1> result(matrix.rows(), indexes.size());

        Eigen::Matrix<mp::cpp\_rational, -1, -1> gram\_schmidt(matrix.rows(), basis.size());

        for (int i = 0; i < indexes.size(); i++)

        {

            result.col(i) = matrix.col(indexes[i]);

            gram\_schmidt.col(i) = basis[i];

        }

        return std::make\_tuple(result, gram\_schmidt);

    }

    #endif

    #ifdef PARALLEL

    // Returns matrix that consist of linearly independent rows of input matrix, indicies of that rows in input matrix, indices of deleted rows and martix T, that consists of Gram Schmidt coefficients

    // @param matrix input matrix

    // @return std::tuple<Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1>, std::vector<int>, std::vector<int>, Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1>>

    std::tuple<Eigen::Matrix<mp::cpp\_int, -1, -1>, std::vector<int>, std::vector<int>, Eigen::Matrix<mp::cpp\_rational, -1, -1>> get\_linearly\_independent\_rows\_by\_gram\_schmidt(const Eigen::Matrix<mp::cpp\_int, -1, -1> &matrix)

    {

        std::vector<Eigen::Vector<mp::cpp\_rational, -1>> basis;

        std::vector<int> indicies;

        std::vector<int> deleted\_indicies;

        Eigen::Matrix<mp::cpp\_rational, -1, -1> T = Eigen::Matrix<mp::cpp\_rational, -1, -1>::Identity(matrix.rows(), matrix.rows());

        int counter = 0;

        for (const Eigen::Vector<mp::cpp\_rational, -1> &vec : matrix.cast<mp::cpp\_rational>().rowwise())

        {

            Eigen::Vector<mp::cpp\_rational, -1> projections = Eigen::Vector<mp::cpp\_rational, -1>::Zero(vec.size());

            #pragma omp parallel

            for (int i = 0; i < basis.size(); i++)

            {

                Eigen::Vector<mp::cpp\_rational, -1> basis\_vector = basis[i];

                mp::cpp\_rational inner1 = std::inner\_product(vec.data(), vec.data() + vec.size(), basis\_vector.data(), mp::cpp\_rational(0.0));

                mp::cpp\_rational inner2 = std::inner\_product(basis\_vector.data(), basis\_vector.data() + basis\_vector.size(), basis\_vector.data(), mp::cpp\_rational(0.0));

                mp::cpp\_rational u\_ij = 0;

                if (!inner1.is\_zero())

                {

                    u\_ij = inner1 / inner2;

                    #pragma omp critical

                    {

                        projections += u\_ij \* basis\_vector;

                        T(counter, i) = u\_ij;

                    }

                }

            }

            Eigen::Vector<mp::cpp\_rational, -1> result = vec - projections;

            bool is\_all\_zero = result.isZero(1e-3);

            if (!is\_all\_zero)

            {

                indicies.push\_back(counter);

            }

            else

            {

                deleted\_indicies.push\_back(counter);

            }

            basis.push\_back(result);

            counter++;

        }

        Eigen::Matrix<mp::cpp\_int, -1, -1> result(indicies.size(), matrix.cols());

        for (int i = 0; i < indicies.size(); i++)

        {

            result.row(i) = matrix.row(indicies[i]);

        }

        return std::make\_tuple(result, indicies, deleted\_indicies, T);

    }

    #else

    // Returns matrix that consist of linearly independent rows of input matrix, indicies of that rows in input matrix, indices of deleted rows and martix T, that consists of Gram Schmidt coefficients

    // @param matrix input matrix

    // @return std::tuple<Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1>, std::vector<int>, std::vector<int>, Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1>>

    std::tuple<Eigen::Matrix<mp::cpp\_int, -1, -1>, std::vector<int>, std::vector<int>, Eigen::Matrix<mp::cpp\_rational, -1, -1>> get\_linearly\_independent\_rows\_by\_gram\_schmidt(const Eigen::Matrix<mp::cpp\_int, -1, -1> &matrix)

    {

        std::vector<Eigen::Vector<mp::cpp\_rational, -1>> basis;

        std::vector<int> indicies;

        std::vector<int> deleted\_indicies;

        Eigen::Matrix<mp::cpp\_rational, -1, -1> T = Eigen::Matrix<mp::cpp\_rational, -1, -1>::Identity(matrix.rows(), matrix.rows());

        int counter = 0;

        for (const Eigen::Vector<mp::cpp\_rational, -1> &vec : matrix.cast<mp::cpp\_rational>().rowwise())

        {

            Eigen::Vector<mp::cpp\_rational, -1> projections = Eigen::Vector<mp::cpp\_rational, -1>::Zero(vec.size());

            for (int i = 0; i < basis.size(); i++)

            {

                Eigen::Vector<mp::cpp\_rational, -1> basis\_vector = basis[i];

                mp::cpp\_rational inner1 = std::inner\_product(vec.data(), vec.data() + vec.size(), basis\_vector.data(), mp::cpp\_rational(0.0));

                mp::cpp\_rational inner2 = std::inner\_product(basis\_vector.data(), basis\_vector.data() + basis\_vector.size(), basis\_vector.data(), mp::cpp\_rational(0.0));

                mp::cpp\_rational u\_ij = 0;

                if (!inner1.is\_zero())

                {

                    u\_ij = inner1 / inner2;

                    projections += u\_ij \* basis\_vector;

                    T(counter, i) = u\_ij;

                }

            }

            Eigen::Vector<mp::cpp\_rational, -1> result = vec - projections;

            bool is\_all\_zero = result.isZero(1e-3);

            if (!is\_all\_zero)

            {

                indicies.push\_back(counter);

            }

            else

            {

                deleted\_indicies.push\_back(counter);

            }

            basis.push\_back(result);

            counter++;

        }

        Eigen::Matrix<mp::cpp\_int, -1, -1> result(indicies.size(), matrix.cols());

        for (int i = 0; i < indicies.size(); i++)

        {

            result.row(i) = matrix.row(indicies[i]);

        }

        return std::make\_tuple(result, indicies, deleted\_indicies, T);

    }

    #endif

    // Extended GCD algorithm, returns tuple of g, x, y such that xa + yb = g

    // @return std::tuple<boost::multiprecision::cpp\_int, boost::multiprecision::cpp\_int, boost::multiprecision::cpp\_int>

    // @param a first number

    // @param b second number

    std::tuple<mp::cpp\_int, mp::cpp\_int, mp::cpp\_int> gcd\_extended(mp::cpp\_int a, mp::cpp\_int b)

    {

        if (a == 0)

        {

            return std::make\_tuple(b, 0, 1);

        }

        mp::cpp\_int gcd, x1, y1;

        std::tie(gcd, x1, y1) = gcd\_extended(b % a, a);

        mp::cpp\_int x = y1 - (b / a) \* x1;

        mp::cpp\_int y = x1;

        return std::make\_tuple(gcd, x, y);

    }

    #ifdef GMP

    // Function for computing HNF of full row rank matrix

    // @return Eigen::Matrix<boost::multiprecision::cpp\_mpz, -1, -1>

    // @param H HNF

    // @param b column to be added

    Eigen::Matrix<mp::mpz\_int, -1, -1> add\_column\_GMP(const Eigen::Matrix<mp::mpz\_int, -1, -1> &H, const Eigen::Vector<mp::mpz\_int, -1> &b\_column)

    {

        if (H.rows() == 0)

        {

            return H;

        }

        Eigen::Vector<mp::mpz\_int, -1> H\_first\_col = H.col(0);

        mp::mpz\_int a = H\_first\_col(0);

        Eigen::Vector<mp::mpz\_int, -1> h = H\_first\_col.tail(H\_first\_col.rows() - 1);

        Eigen::Matrix<mp::mpz\_int, -1, -1> H\_stroke = H.block(1, 1, H.rows() - 1, H.cols() - 1);

        mp::mpz\_int b = b\_column(0);

        Eigen::Vector<mp::mpz\_int, -1> b\_stroke = b\_column.tail(b\_column.rows() - 1);

        std::tuple<mp::mpz\_int, mp::mpz\_int, mp::mpz\_int> gcd\_result = gcd\_extended\_GMP(a, b);

        mp::mpz\_int g, x, y;

        std::tie(g, x, y) = gcd\_result;

        Eigen::Matrix<mp::mpz\_int, 2, 2> U;

        U << x, -b / g, y, a / g;

        Eigen::Matrix<mp::mpz\_int, -1, 2> temp\_matrix(H.rows(), 2);

        temp\_matrix.col(0) = H\_first\_col;

        temp\_matrix.col(1) = b\_column;

        Eigen::Matrix<mp::mpz\_int, -1, 2> temp\_result = temp\_matrix \* U;

        Eigen::Vector<mp::mpz\_int, -1> h\_stroke = temp\_result.col(0).tail(temp\_result.rows() - 1);

        Eigen::Vector<mp::mpz\_int, -1> b\_double\_stroke = temp\_result.col(1).tail(temp\_result.rows() - 1);

        b\_double\_stroke = reduce\_GMP(b\_double\_stroke, H\_stroke);

        Eigen::Matrix<mp::mpz\_int, -1, -1> H\_double\_stroke = add\_column\_GMP(H\_stroke, b\_double\_stroke);

        h\_stroke = reduce\_GMP(h\_stroke, H\_double\_stroke);

        Eigen::Matrix<mp::mpz\_int, -1, -1> result(H.rows(), H.cols());

        result(0, 0) = g;

        result.col(0).tail(result.cols() - 1) = h\_stroke;

        result.row(0).tail(result.rows() - 1).setZero();

        result.block(1, 1, H\_double\_stroke.rows(), H\_double\_stroke.cols()) = H\_double\_stroke;

        return result;

    }

    // Function for computing HNF, reduces elements of vector modulo diagonal elements of matrix

    // @return Eigen::Matrix<boost::multiprecision::cpp\_mpz, -1, -1>

    // @param vector vector to be reduced

    // @param matrix input matrix

    Eigen::Vector<mp::mpz\_int, -1> reduce\_GMP(const Eigen::Vector<mp::mpz\_int, -1> &vector, const Eigen::Matrix<mp::mpz\_int, -1, -1> &matrix)

    {

        Eigen::Vector<mp::mpz\_int, -1> result = vector;

        for (int i = 0; i < result.rows(); i++)

        {

            Eigen::Vector<mp::mpz\_int, -1> matrix\_column = matrix.col(i);

            mp::mpz\_int t\_vec\_elem = result(i);

            mp::mpz\_int t\_matrix\_elem = matrix(i, i);

            mp::mpz\_int x;

            if (t\_vec\_elem >= 0)

            {

                x = (t\_vec\_elem / t\_matrix\_elem);

            }

            else

            {

                x = (t\_vec\_elem - (t\_matrix\_elem - 1)) / t\_matrix\_elem;

            }

            result -= matrix\_column \* x;

        }

        return result;

    }

    // Generates random matrix with full row rank (all rows are linearly independent)

    // @return Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1>

    // @param m number of rows, must be greater than one and less than or equal to the parameter n

    // @param n number of columns, must be greater than one and greater than or equal to the parameter m

    // @param lowest lowest generated number, must be lower than lowest parameter by at least one

    // @param highest highest generated number, must be greater than lowest parameter by at least one

    Eigen::Matrix<mp::mpz\_int, -1, -1> generate\_random\_matrix\_with\_full\_row\_rank\_GMP(const int m, const int n, int lowest, int highest)

    {

        if (m > n)

        {

            throw std::invalid\_argument("m must be less than or equal n");

        }

        if (m < 1 || n < 1)

        {

            throw std::invalid\_argument("Number of rows or columns should be greater than one");

        }

        if (highest - lowest < 1)

        {

            throw std::invalid\_argument("highest parameter must be greater than lowest parameter by at least one");

        }

        std::random\_device rd;

        std::mt19937 gen(rd());

        std::uniform\_int\_distribution<int> dis (lowest, highest);

        Eigen::Matrix<int, -1, -1> matrix = Eigen::Matrix<int, -1, -1>::NullaryExpr(m, n, [&]()

                                                              { return dis(gen); });

        Eigen::FullPivLU<Eigen::MatrixXd> lu\_decomp(matrix.cast<double>());

        auto rank = lu\_decomp.rank();

        while (rank != m)

        {

            matrix = Eigen::Matrix<int, -1, -1>::NullaryExpr(m, n, [&]()

                                                  { return dis(gen); });

            lu\_decomp.compute(matrix.cast<double>());

            rank = lu\_decomp.rank();

        }

        return matrix.cast<mp::mpz\_int>();

    }

    // Generates random matrix

    // @return Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1>

    // @param m number of rows, must be greater than one

    // @param n number of columns, must be greater than one

    // @param lowest lowest generated number, must be lower than lowest parameter by at least one

    // @param highest highest generated number, must be greater than lowest parameter by at least one

    Eigen::Matrix<mp::mpz\_int, -1, -1> generate\_random\_matrix\_GMP(const int m, const int n, int lowest, int highest)

    {

        if (m < 1 || n < 1)

        {

            throw std::invalid\_argument("Number of rows or columns should be greater than one");

        }

        if (highest - lowest < 1)

        {

            throw std::invalid\_argument("highest parameter must be greater than lowest parameter by at least one");

        }

        std::random\_device rd;

        std::mt19937 gen(rd());

        std::uniform\_int\_distribution<int> dis (lowest, highest);

        Eigen::Matrix<int, -1, -1> matrix = Eigen::Matrix<int, -1, -1>::NullaryExpr(m, n, [&]()

                                                              { return dis(gen); });

        return matrix.cast<mp::mpz\_int>();

    }

    #ifdef PARALLEL

    // Returns matrix that consist of linearly independent columns of input matrix and othogonalized matrix

    // @param matrix input matrix

    // @return std::tuple<Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1>, Eigen::Matrix<boost::multiprecision::mpq\_rational, -1, -1>>

    std::tuple<Eigen::Matrix<mp::mpz\_int, -1, -1>, Eigen::Matrix<mp::mpq\_rational, -1, -1>> get\_linearly\_independent\_columns\_by\_gram\_schmidt\_GMP(const Eigen::Matrix<mp::mpz\_int, -1, -1> &matrix)

    {

        std::vector<Eigen::Vector<mp::mpq\_rational, -1>> basis;

        std::vector<int> indexes;

        int counter = 0;

        for (const Eigen::Vector<mp::mpq\_rational, -1> &vec : matrix.cast<mp::mpq\_rational>().colwise())

        {

            Eigen::Vector<mp::mpq\_rational, -1> projections = Eigen::Vector<mp::mpq\_rational, -1>::Zero(vec.size());

            #pragma omp parallel for

            for (int i = 0; i < basis.size(); i++)

            {

                Eigen::Vector<mp::mpq\_rational, -1> basis\_vector = basis[i];

                mp::mpq\_rational inner1 = std::inner\_product(vec.data(), vec.data() + vec.size(), basis\_vector.data(), mp::mpq\_rational(0.0));

                mp::mpq\_rational inner2 = std::inner\_product(basis\_vector.data(), basis\_vector.data() + basis\_vector.size(), basis\_vector.data(), mp::mpq\_rational(0.0));

                mp::mpq\_rational coef = inner1 / inner2;

                #pragma omp critical

                projections += basis\_vector \* coef;

            }

            Eigen::Vector<mp::mpq\_rational, -1> result = vec - projections;

            bool is\_all\_zero = result.isZero(1e-3);

            if (!is\_all\_zero)

            {

                basis.push\_back(result);

                indexes.push\_back(counter);

            }

            counter++;

        }

        Eigen::Matrix<mp::mpz\_int, -1, -1> result(matrix.rows(), indexes.size());

        Eigen::Matrix<mp::mpq\_rational, -1, -1> gram\_schmidt(matrix.rows(), basis.size());

        for (int i = 0; i < indexes.size(); i++)

        {

            result.col(i) = matrix.col(indexes[i]);

            gram\_schmidt.col(i) = basis[i];

        }

        return std::make\_tuple(result, gram\_schmidt);

    }

    #else

    // Returns matrix that consist of linearly independent columns of input matrix and othogonalized matrix

    // @param matrix input matrix

    // @return std::tuple<Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1>, Eigen::Matrix<boost::multiprecision::mpq\_rational, -1, -1>>

    std::tuple<Eigen::Matrix<mp::mpz\_int, -1, -1>, Eigen::Matrix<mp::mpq\_rational, -1, -1>> get\_linearly\_independent\_columns\_by\_gram\_schmidt\_GMP(const Eigen::Matrix<mp::mpz\_int, -1, -1> &matrix)

    {

        std::vector<Eigen::Vector<mp::mpq\_rational, -1>> basis;

        std::vector<int> indexes;

        int counter = 0;

        for (const Eigen::Vector<mp::mpq\_rational, -1> &vec : matrix.cast<mp::mpq\_rational>().colwise())

        {

            Eigen::Vector<mp::mpq\_rational, -1> projections = Eigen::Vector<mp::mpq\_rational, -1>::Zero(vec.size());

            for (int i = 0; i < basis.size(); i++)

            {

                Eigen::Vector<mp::mpq\_rational, -1> basis\_vector = basis[i];

                mp::mpq\_rational inner1 = std::inner\_product(vec.data(), vec.data() + vec.size(), basis\_vector.data(), mp::mpq\_rational(0.0));

                mp::mpq\_rational inner2 = std::inner\_product(basis\_vector.data(), basis\_vector.data() + basis\_vector.size(), basis\_vector.data(), mp::mpq\_rational(0.0));

                mp::mpq\_rational coef = inner1 / inner2;

                projections += basis\_vector \* coef;

            }

            Eigen::Vector<mp::mpq\_rational, -1> result = vec - projections;

            bool is\_all\_zero = result.isZero(1e-3);

            if (!is\_all\_zero)

            {

                basis.push\_back(result);

                indexes.push\_back(counter);

            }

            counter++;

        }

        Eigen::Matrix<mp::mpz\_int, -1, -1> result(matrix.rows(), indexes.size());

        Eigen::Matrix<mp::mpq\_rational, -1, -1> gram\_schmidt(matrix.rows(), basis.size());

        for (int i = 0; i < indexes.size(); i++)

        {

            result.col(i) = matrix.col(indexes[i]);

            gram\_schmidt.col(i) = basis[i];

        }

        return std::make\_tuple(result, gram\_schmidt);

    }

    #endif

    #ifdef PARALLEL

    // Returns matrix that consist of linearly independent rows of input matrix, indicies of that rows in input matrix, indices of deleted rows and martix T, that consists of Gram Schmidt coefficients

    // @param matrix input matrix

    // @return std::tuple<Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1>, std::vector<int>, std::vector<int>, Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1>>

    std::tuple<Eigen::Matrix<mp::mpz\_int, -1, -1>, std::vector<int>, std::vector<int>, Eigen::Matrix<mp::mpq\_rational, -1, -1>> get\_linearly\_independent\_rows\_by\_gram\_schmidt\_GMP(const Eigen::Matrix<mp::mpz\_int, -1, -1> &matrix)

    {

        std::vector<Eigen::Vector<mp::mpq\_rational, -1>> basis;

        std::vector<int> indicies;

        std::vector<int> deleted\_indicies;

        Eigen::Matrix<mp::mpq\_rational, -1, -1> T = Eigen::Matrix<mp::mpq\_rational, -1, -1>::Identity(matrix.rows(), matrix.rows());

        int counter = 0;

        for (const Eigen::Vector<mp::mpq\_rational, -1> &vec : matrix.cast<mp::mpq\_rational>().rowwise())

        {

            Eigen::Vector<mp::mpq\_rational, -1> projections = Eigen::Vector<mp::mpq\_rational, -1>::Zero(vec.size());

            #pragma omp parallel for

            for (int i = 0; i < basis.size(); i++)

            {

                Eigen::Vector<mp::mpq\_rational, -1> basis\_vector = basis[i];

                mp::mpq\_rational inner1 = std::inner\_product(vec.data(), vec.data() + vec.size(), basis\_vector.data(), mp::mpq\_rational(0.0));

                mp::mpq\_rational inner2 = std::inner\_product(basis\_vector.data(), basis\_vector.data() + basis\_vector.size(), basis\_vector.data(), mp::mpq\_rational(0.0));

                mp::mpq\_rational u\_ij = 0;

                if (!inner1.is\_zero())

                {

                    u\_ij = inner1 / inner2;

                    #pragma omp critical

                    {

                        projections += u\_ij \* basis\_vector;

                        T(counter, i) = u\_ij;

                    }

                }

            }

            Eigen::Vector<mp::mpq\_rational, -1> result = vec - projections;

            bool is\_all\_zero = result.isZero(1e-3);

            if (!is\_all\_zero)

            {

                indicies.push\_back(counter);

            }

            else

            {

                deleted\_indicies.push\_back(counter);

            }

            basis.push\_back(result);

            counter++;

        }

        Eigen::Matrix<mp::mpz\_int, -1, -1> result(indicies.size(), matrix.cols());

        for (int i = 0; i < indicies.size(); i++)

        {

            result.row(i) = matrix.row(indicies[i]);

        }

        return std::make\_tuple(result, indicies, deleted\_indicies, T);

    }

    #else

    // Returns matrix that consist of linearly independent rows of input matrix, indicies of that rows in input matrix, indices of deleted rows and martix T, that consists of Gram Schmidt coefficients

    // @param matrix input matrix

    // @return std::tuple<Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1>, std::vector<int>, std::vector<int>, Eigen::Matrix<boost::multiprecision::mpz\_int, -1, -1>>

    std::tuple<Eigen::Matrix<mp::mpz\_int, -1, -1>, std::vector<int>, std::vector<int>, Eigen::Matrix<mp::mpq\_rational, -1, -1>> get\_linearly\_independent\_rows\_by\_gram\_schmidt\_GMP(const Eigen::Matrix<mp::mpz\_int, -1, -1> &matrix)

    {

        std::vector<Eigen::Vector<mp::mpq\_rational, -1>> basis;

        std::vector<int> indicies;

        std::vector<int> deleted\_indicies;

        Eigen::Matrix<mp::mpq\_rational, -1, -1> T = Eigen::Matrix<mp::mpq\_rational, -1, -1>::Identity(matrix.rows(), matrix.rows());

        int counter = 0;

        for (const Eigen::Vector<mp::mpq\_rational, -1> &vec : matrix.cast<mp::mpq\_rational>().rowwise())

        {

            Eigen::Vector<mp::mpq\_rational, -1> projections = Eigen::Vector<mp::mpq\_rational, -1>::Zero(vec.size());

            for (int i = 0; i < basis.size(); i++)

            {

                Eigen::Vector<mp::mpq\_rational, -1> basis\_vector = basis[i];

                mp::mpq\_rational inner1 = std::inner\_product(vec.data(), vec.data() + vec.size(), basis\_vector.data(), mp::mpq\_rational(0.0));

                mp::mpq\_rational inner2 = std::inner\_product(basis\_vector.data(), basis\_vector.data() + basis\_vector.size(), basis\_vector.data(), mp::mpq\_rational(0.0));

                mp::mpq\_rational u\_ij = 0;

                if (!inner1.is\_zero())

                {

                    u\_ij = inner1 / inner2;

                    {

                        projections += u\_ij \* basis\_vector;

                        T(counter, i) = u\_ij;

                    }

                }

            }

            Eigen::Vector<mp::mpq\_rational, -1> result = vec - projections;

            bool is\_all\_zero = result.isZero(1e-3);

            if (!is\_all\_zero)

            {

                indicies.push\_back(counter);

            }

            else

            {

                deleted\_indicies.push\_back(counter);

            }

            basis.push\_back(result);

            counter++;

        }

        Eigen::Matrix<mp::mpz\_int, -1, -1> result(indicies.size(), matrix.cols());

        for (int i = 0; i < indicies.size(); i++)

        {

            result.row(i) = matrix.row(indicies[i]);

        }

        return std::make\_tuple(result, indicies, deleted\_indicies, T);

    }

    #endif

    // Extended GCD algorithm, returns tuple of g, x, y such that xa + yb = g

    // @return std::tuple<boost::multiprecision::mpz\_int, boost::multiprecision::mpz\_int, boost::multiprecision::mpz\_int>

    // @param a first number

    // @param b second number

    std::tuple<mp::mpz\_int, mp::mpz\_int, mp::mpz\_int> gcd\_extended\_GMP(mp::mpz\_int a, mp::mpz\_int b)

    {

        if (a == 0)

        {

            return std::make\_tuple(b, 0, 1);

        }

        mp::mpz\_int gcd, x1, y1;

        std::tie(gcd, x1, y1) = gcd\_extended\_GMP(b % a, a);

        mp::mpz\_int x = y1 - (b / a) \* x1;

        mp::mpz\_int y = x1;

        return std::make\_tuple(gcd, x, y);

    }

    #endif

    // Generates random matrix with full column rank

    // @return Eigen::Matrix<boost::multiprecision::cpp\_int, -1, -1>

    // @param m number of rows, must be greater than one and greater than or equal to the parameter n

    // @param n number of columns, must be greater than one and lower than or equal to the parameter m

    // @param lowest lowest generated number, must be lower than lowest parameter by at least one

    // @param highest highest generated number, must be greater than lowest parameter by at least one

    Eigen::MatrixXd generate\_random\_matrix\_with\_full\_column\_rank(const int m, const int n, int lowest, int highest)

    {

        if (m < n)

        {

            throw std::invalid\_argument("m must be less than or equal n");

        }

        if (m < 1 || n < 1)

        {

            throw std::invalid\_argument("Number of rows or columns should be greater than one");

        }

        if (highest - lowest < 1)

        {

            throw std::invalid\_argument("highest parameter must be greater than lowest parameter by at least one");

        }

        std::random\_device rd;

        std::mt19937 gen(rd());

        std::uniform\_int\_distribution<int> dis (lowest, highest);

        Eigen::MatrixXd matrix = Eigen::MatrixXd::NullaryExpr(m, n, [&]()

                                                              { return dis(gen); });

        Eigen::FullPivLU<Eigen::MatrixXd> lu\_decomp(matrix);

        auto rank = lu\_decomp.rank();

        while (rank != n)

        {

            matrix = Eigen::MatrixXd::NullaryExpr(m, n, [&]()

                                                  { return dis(gen); });

            lu\_decomp.compute(matrix);

            rank = lu\_decomp.rank();

        }

        return matrix;

    }

    // Generates random vector

    // @return Eigen::VectorXd

    // @param m number of rows, must be greater than one

    // @param lowest lowest generated number, must be lower than lowest parameter by at least one

    // @param highest highest generated number, must be greater than lowest parameter by at least one

    Eigen::VectorXd generate\_random\_vector(const int m, double lowest, double highest)

    {

        if (m < 1)

        {

            throw std::invalid\_argument("Number of rows or columns should be greater than one");

        }

        if (highest - lowest < 1)

        {

            throw std::invalid\_argument("highest parameter must be greater than lowest parameter by at least one");

        }

        std::mt19937 gen(std::random\_device{}());

        std::uniform\_real\_distribution<double> dis(lowest, highest);

        Eigen::VectorXd vector = Eigen::VectorXd::NullaryExpr(m, [&]()

                                                           { return dis(gen); });

        return vector;

    }

    #ifdef PARALLEL

    // Computes component of a vector perpendicular to a matrix using equations from Gram Schmidt computing

    // @return Eigen::VectorXd

    // @param matrix input matrix

    // @param vector input vector

    Eigen::VectorXd projection(const Eigen::MatrixXd &matrix, const Eigen::VectorXd &vector)

    {

        Eigen::MatrixXd t\_matrix(matrix.rows(), matrix.cols() + 1);

        t\_matrix << matrix, vector;

        std::vector<Eigen::VectorXd> basis;

        for (const Eigen::VectorXd &vec : t\_matrix.colwise())

        {

            Eigen::VectorXd projections = Eigen::VectorXd::Zero(vec.size());

            #pragma omp parallel for

            for (int i = 0; i < basis.size(); i++)

            {

                Eigen::VectorXd basis\_vector = basis[i];

                double inner1 = std::inner\_product(vec.data(), vec.data() + vec.size(), basis\_vector.data(), 0.0);

                double inner2 = std::inner\_product(basis\_vector.data(), basis\_vector.data() + basis\_vector.size(), basis\_vector.data(), 0.0);

                double coef = inner1 / inner2;

                #pragma omp critical

                projections += basis\_vector \* coef;

            }

            Eigen::VectorXd t\_result = vec - projections;

            basis.push\_back(t\_result);

        }

        Eigen::VectorXd result = basis[basis.size() - 1];

        return result;

    }

    #else

    // Computes component of a vector perpendicular to a matrix using equations from Gram Schmidt computing

    // @return Eigen::VectorXd

    // @param matrix input matrix

    // @param vector input vector

    Eigen::VectorXd projection(const Eigen::MatrixXd &matrix, const Eigen::VectorXd &vector)

    {

        Eigen::MatrixXd t\_matrix(matrix.rows(), matrix.cols() + 1);

        t\_matrix << matrix, vector;

        std::vector<Eigen::VectorXd> basis;

        for (const Eigen::VectorXd &vec : t\_matrix.colwise())

        {

            Eigen::VectorXd projections = Eigen::VectorXd::Zero(vec.size());

            for (int i = 0; i < basis.size(); i++)

            {

                Eigen::VectorXd basis\_vector = basis[i];

                double inner1 = std::inner\_product(vec.data(), vec.data() + vec.size(), basis\_vector.data(), 0.0);

                double inner2 = std::inner\_product(basis\_vector.data(), basis\_vector.data() + basis\_vector.size(), basis\_vector.data(), 0.0);

                double coef = inner1 / inner2;

                projections += basis\_vector \* coef;

            }

            Eigen::VectorXd t\_result = vec - projections;

            basis.push\_back(t\_result);

        }

        Eigen::VectorXd result = basis[basis.size() - 1];

        return result;

    }

    #endif

    // Finds vector that is closest to other vectors in matrix

    // @return Eigen::VectorXd

    // @param matrix input matrix

    // @param vector input vector

    Eigen::VectorXd closest\_vector(const std::vector<Eigen::VectorXd> &matrix, const Eigen::VectorXd &vector)

    {

        Eigen::VectorXd closest = matrix[0];

        for (const auto &v : matrix)

        {

            if ((vector - v).norm() <= (vector - closest).norm())

            {

                closest = v;

            }

        }

        return closest;

    }

}

Приложение Д. Исходный CMakeLists.txt

cmake\_minimum\_required(VERSION 3.2)

project(LatticeAlgorithms)

option(BUILD\_DOCS "" OFF)

option(BUILD\_PARALLEL "" OFF)

option(BUILD\_GMP "" OFF)

file(GLOB SRC

     "src/utils.cpp"

     "src/algorithms.cpp"

)

add\_subdirectory(3rdparty/boost\_config)

add\_subdirectory(3rdparty/boost\_multiprecision)

find\_package(OpenMP REQUIRED)

add\_library(${PROJECT\_NAME} ${SRC})

target\_include\_directories(${PROJECT\_NAME} PUBLIC include)

if (BUILD\_PARALLEL)

     target\_compile\_definitions(${PROJECT\_NAME} PUBLIC PARALLEL)

endif(BUILD\_PARALLEL)

if (BUILD\_GMP)

     target\_compile\_definitions(${PROJECT\_NAME} PUBLIC GMP)

endif(BUILD\_GMP)

target\_link\_libraries(${PROJECT\_NAME} OpenMP::OpenMP\_CXX)

target\_link\_libraries(${PROJECT\_NAME} gmp libgmp)

target\_link\_libraries(${PROJECT\_NAME} Boost::config Boost::multiprecision)

if (BUILD\_DOCS)

     add\_subdirectory(tex)

endif(BUILD\_DOCS)