

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

1. Список условных обозначений и сокращений	4
2. Введение.....	5
3. Постановка задачи	6
4. Обзор инструментов	7
4.1. Обзор библиотеки Eigen.....	7
4.2. Обзор библиотеки Boost.Multiprecision	8
5. Обзор литературных источников.....	10
5.1. Базовые определения.....	10
5.2. Ортогонализация Грама-Шмидта.....	11
5.3. Алгоритм нахождения ЭНФ для матриц с полным рангом строки	12
5.4. Общий алгоритм нахождения ЭНФ для любых матриц	14
5.5. Пример нахождения ЭНФ	15
5.6. Применение ЭНФ	17
5.7. Определение проблемы ближайшего вектора.....	19
5.8. Жадный метод: алгоритм ближайшей плоскости Бабая	19
5.9. Нерекурсивная реализация	20
5.10. Пример жадного метода	21
5.11. Метод ветвей и границ	21
5.12. Пример метода ветвей и границ	22
5.13. Параллельная реализация метода ветвей и границ	23
6. Обзор существующих решений	24
6.1. WolframAlpha API.....	24
6.2. Numbertheory.org.....	24
6.3. hsnf	26
7. Обзор программной реализации.....	27
7.1. Вспомогательные функции	27
7.2. Ортогонализация Грама-Шмидта.....	29
7.3. Нахождение ЭНФ.....	29
7.4. Решение ПБВ.....	30
8. Заключение.....	32
Список литературы	33
Приложения	34

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++

2. 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, приближительного 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.

3. 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 их эффективно 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.

4. 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.

4.1. 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 стандартной.

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 3×3 .

`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 3.

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 $m \times m$.

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

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.

4.2. 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.

5. Обзор литературных источников

5.1. Базовые определения

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

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Квадратная матрица — матрица, у которой число строк равно числу столбцов $m = n$.

Единичная матрица — матрица, у которой диагональные элементы ($i = j$) равны единице.

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

Вектор — если матрица состоит из одного столбца ($n = 1$), то она называется вектором-столбцом. Если матрица состоит из одной строки ($m = 1$), то она называется вектором-строкой.

Матрицы можно обозначать через вектора-столбцы и через вектора-строки: $\mathbf{A} = \begin{bmatrix} \mathbf{a}_1 & \cdots & \mathbf{a}_n \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1^T \\ \vdots \\ \mathbf{a}_m^T \end{bmatrix}$.

Линейная зависимость/независимость — пусть имеется несколько векторов одной размерности $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k$ и столько же чисел $\alpha_1, \alpha_2, \dots, \alpha_k$. Вектор $\mathbf{y} = \alpha_1 \mathbf{x}_1 + \alpha_2 \mathbf{x}_2 + \dots + \alpha_k \mathbf{x}_k$ называется линейной комбинацией векторов \mathbf{x}_k . Если существуют такие числа $\alpha_i, i = 1, \dots, k$, не все равные нулю, такие, что $\mathbf{y} = \mathbf{0}$, 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 $\mathbf{B} = [\mathbf{b}_1, \dots, \mathbf{b}_n] \in \mathbb{R}^{d \times n}$ — линейно независимые вектора из \mathbb{R}^d . Решетка, генерируемая от \mathbf{B} есть множество

$$\mathcal{L}(\mathbf{B}) = \{\mathbf{B}\mathbf{x} : \mathbf{x} \in \mathbb{Z}^n\} = \left\{ \sum_{i=1}^n x_i \cdot \mathbf{b}_i : \forall i \ x_i \in \mathbb{Z} \right\}$$

всех целочисленных линейных комбинаций столбцов матрицы \mathbf{B} . Матрица \mathbf{B} называется базисом для решетки $\mathcal{L}(\mathbf{B})$. Число n называется рангом решетки. Если $n = d$, то решетка $\mathcal{L}(\mathbf{B})$ называется решеткой полного ранга или полноразмерной решеткой в \mathbb{R}^d .

Определитель решетки — пусть $\mathbf{B} = [\mathbf{b}_1, \dots, \mathbf{b}_n]$ — базис решетки, $\mathbf{B}^* = [\mathbf{b}_1^*, \dots, \mathbf{b}_n^*]$ — ортогонализация Грама-Шмидта для исходного базиса, тогда определитель $\det = \prod_i \|\mathbf{b}_i^*\|$. Определитель решетки не зависит от выбора исходного базиса [2].

Эрмитова нормальная форма [1] — невырожденная матрица $\mathbf{B} = [\mathbf{b}_1, \dots, \mathbf{b}_n] \in \mathbb{R}^{m \times n}$ является Эрмитовой нормальной формой, если

- Существует $1 \leq i_1 < \dots < i_h \leq m$ такое, что $b_{i_j, j} \neq 0 \Rightarrow (j < h) \wedge (i \geq i_j)$ (строго убывающая высота столбца).
- Для всех $k > j, 0 \leq b_{i_j, k} < b_{i_j, j}$, т.е. все элементы в строках i_j приведены по модулю $b_{i_j, j}$.

Проблема ближайшего вектора — дан базис решетки $\mathbf{B} \in \mathbb{R}^{d \times n}$ и целевой вектор $\mathbf{t} \in \mathbb{R}^d$, который не принадлежит решетке, необходимо найти точку решетки $\mathbf{B}\mathbf{x}$ ($\mathbf{x} \in \mathbb{Z}^n$) такую, что расстояние $\|\mathbf{t} - \mathbf{B}\mathbf{x}\|$ минимально [1].

5.2. Ортогонализация Грама-Шмидта

Любой базис \mathbf{B} может быть преобразован в ортогональный базис для того же векторного пространства используя алгоритм ортогонализации Грама-Шмидта [2]. Предположим у нас есть набор векторов $\mathbf{B} = [\mathbf{b}_1, \dots, \mathbf{b}_n]$, $\mathbf{B} \in \mathbb{R}^{m \times n}$. Этот набор необязательно ортогонален или даже линейно независим. Ортогонализацией этого набора векторов является набор векторов $\mathbf{B}^* = [\mathbf{b}_1^*, \dots, \mathbf{b}_n^*] \in \mathbb{R}^{m \times n}$, где

$$\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j < i} \mu_{i,j} \mathbf{b}_j^*, \text{ где } \mu_{i,j} = \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle}, i = 1, \dots, n, j = 1, \dots, i$$

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 $\mathbf{b}_i^* = \mathbf{0}$, то этот вектор линейно зависим от других векторов в наборе и может быть представлен линейной комбинацией этих векторов.

Временная сложность алгоритма $O(N^3)$, 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 $\sum_{j < i} \mu_{i,j} \mathbf{b}_j^*$. Конечный алгоритм выглядит следующим образом:

Input: \mathbf{B}

Output: GS

$\text{GS} \leftarrow \begin{bmatrix} \end{bmatrix}$

$n \leftarrow \mathbf{B}.columns$

for $i \leftarrow 0$ to n **do**

```

bi ← B.column(i)
projections ← 0
for j ← 0 to i do
    bj ← GS.column(j)
    projections ← projections + bj ·  $\frac{\langle \mathbf{b}_i, \mathbf{b}_j \rangle}{\langle \mathbf{b}_j, \mathbf{b}_j \rangle}$ 
end for
GS.push_back(bi − projections)
end for

```

5.3. Алгоритм нахождения ЭНФ для матриц с полным рангом строки

Дана матрица $\mathbf{B} \in \mathbb{Z}^{m \times n}$. Основная идея состоит в том, чтобы найти ЭНФ \mathbf{H} подрешетки от $\mathcal{L}(\mathbf{B})$, и затем обновлять \mathbf{H} , включая столбцы \mathbf{B} один за другим [1]. Предположим, что у нас есть процедура **AddColumn**, которая работает за полиномиальное время и принимает на вход квадратную невырожденную ЭНФ матрицу $\mathbf{H} \in \mathbb{Z}^{m \times m}$ и вектор $\mathbf{b} \in \mathbb{Z}^m$, а возвращает ЭНФ матрицы $[\mathbf{H}|\mathbf{b}]$. Такая процедура должна следить, чтоб выходная матрица подходила под определение ЭНФ, что будет показано в описании этой процедуры. ЭНФ от \mathbf{B} может быть вычислено следующим образом:

1. Применить алгоритм Грама-Шмидта к столбцам \mathbf{B} , чтобы найти m линейно независимых столбцов. Пусть \mathbf{B}' - матрица размера $m \times m$, заданная этими столбцами.
2. Вычислить $d = \det(\mathbf{B}')$, используя алгоритм Грама-Шмидта или любую другую процедуру с полиномиальным временем. Пусть $\mathbf{H}_0 = d \cdot \mathbf{I}$ будет диагональной матрицей с d на диагонали.
3. Для $i = 1, \dots, n$ пусть \mathbf{H}_i – результат применения **AddColumn** к входным \mathbf{H}_{i-1} и \mathbf{b}_i .
4. Вернуть \mathbf{H}_n .

Разберем подпункты:

1. Необходимо найти линейно независимые столбцы матрицы. Их количество всегда будет равно m , т.к. наша матрица полного ранга строки и ранг матрицы равен m , а значит матрица, состоящая из этих столбцов, будет размера $m \times m$. Для нахождения этих строк можно использовать алгоритм ортогонализации Грама-Шмидта: если $\mathbf{b}_i^* = \mathbf{0}$, то i -ая строка является линейной комбинацией других строк, и ее необходимо удалить. Полученная матрица будет названа \mathbf{B}' .
2. Необходимо вычислить d , будем вычислять его по следующей формуле: $d = \sqrt{\prod_i \|\mathbf{b}_i^*\|^2}$ — сумма произведений квадратов длин всех столбцов, полученных после применения ортогонализации Грама-Шмидта. Матрица \mathbf{H}_0 будет единичной матрицей размера $m \times m$, умноженной на определитель. В результате все диагональные элементы будут равны d .

3. Применяем AddColumn к \mathbf{H}_0 и первому столбцу матрицы $\mathbf{B} - \mathbf{b}_0$, получаем \mathbf{H}_1 ; повторяем для всех оставшихся столбцов, получаем \mathbf{H}_n .
4. \mathbf{H}_n является ЭНФ(\mathbf{B}).

Алгоритм AddColumn на вход принимает квадратную невырожденную ЭНФ матрицы $\mathbf{H} \in \mathbb{Z}^{m \times m}$ и вектор $\mathbf{b} \in \mathbb{Z}^m$ и работает следующим образом. Если $m = 0$, то возвращаем \mathbf{H} . В противном случае, пусть $\mathbf{H} = \begin{bmatrix} \mathbf{a} & \mathbf{0}^T \\ \mathbf{h} & \mathbf{H}' \end{bmatrix}$ и $\mathbf{b} = \begin{bmatrix} b \\ \mathbf{b}' \end{bmatrix}$ и дальше:

1. Вычислить $g = \text{НОД}(a, b)$ и целые x, y такие, что $xa + yb = g$, используя расширенный НОД алгоритм.
2. Применить унимодулярное преобразование $\mathbf{U} = \begin{bmatrix} x & (-b/g) \\ y & (a/g) \end{bmatrix}$ к первому столбцу из \mathbf{H} и \mathbf{b} чтобы получить $\begin{bmatrix} a & b \\ \mathbf{h} & \mathbf{b}' \end{bmatrix} \mathbf{U} = \begin{bmatrix} g & 0 \\ \mathbf{h}' & \mathbf{b}'' \end{bmatrix}$.
3. Добавить соответствующий вектор из $\mathcal{L}(\mathbf{H}')$ к \mathbf{b}'' , чтобы сократить его элементы по модулю диагональных элементов из \mathbf{H}' .
4. Рекурсивно вызвать AddColumn на вход \mathbf{H}' и \mathbf{b}'' чтобы получить матрицу \mathbf{H}'' .
5. Добавить соответствующий вектор из $\mathcal{L}(\mathbf{H}'')$ к \mathbf{h}' чтобы сократить его элементы по модулю диагональных элементов из \mathbf{H}'' .
6. Вернуть $\begin{bmatrix} g & \mathbf{0}^T \\ \mathbf{h}' & \mathbf{H}'' \end{bmatrix}$.

Разберем подпункты:

1. Необходимо с помощью расширенного НОД алгоритма найти наибольший общий делитель и целые x, y такие, что $xa + yb = g$.
2. Составляем матрицу $\mathbf{U} = \begin{bmatrix} x & (-b/g) \\ y & (a/g) \end{bmatrix}$ и умножаем ее на матрицу, составленную из первого столбца \mathbf{H} и столбца \mathbf{b} , чтобы получить:

$$\begin{bmatrix} a & b \\ \mathbf{h} & \mathbf{b}' \end{bmatrix} \mathbf{U} = \begin{bmatrix} g & 0 \\ \mathbf{h}' & \mathbf{b}'' \end{bmatrix}$$

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 \mathbf{H}' и \mathbf{b} .
4. Рекурсивно вызываем AddColumn, на вход отправляем \mathbf{H}' и \mathbf{b}'' получаем матрицу \mathbf{H}'' .

5. Вызываем функцию `reduce` к \mathbf{H}'' и \mathbf{h}' .

6. Составляем необходимую матрицу и возвращаем $\begin{bmatrix} \mathbf{g} & \mathbf{0}^T \\ \mathbf{h}' & \mathbf{H}'' \end{bmatrix}$.

5.4. Общий алгоритм нахождения ЭНФ для любых матриц

Данный алгоритм можно применять на произвольных матрицах путем сведения к алгоритму для полного ранга строки [1].

1. Запустить процесс ортогонализации Грама-Шмидта к строкам $\mathbf{r}_1, \dots, \mathbf{r}_m$ из \mathbf{B} , и пусть $K = \{k_1, \dots, k_l\}$ ($k_1 < \dots < k_l$) – это множество индексов, такое, что $\mathbf{r}_{k_i}^* \neq \mathbf{0}$. Определим операцию проецирования $\Pi_K : \mathbb{R}^m \rightarrow \mathbb{R}^l$ при $[\Pi_K(\mathbf{x})]_i = x_{k_i}$. Заметим, что строки \mathbf{r}_k ($k \in K$) линейно независимы и любая строка \mathbf{r}_i ($i \in K$) может быть выражена как линейная комбинация предыдущих строк \mathbf{r}_j ($\{j \in K : j < i\}$). Следовательно, операция проецирования Π_K однозначно определена, когда ограничена к $\mathcal{L}(\mathbf{B})$, и ее инверсия может быть легко вычислена, используя коэффициенты Грама-Шмидта $\mu_{i,j}$.
2. Введем матрицу $\mathbf{B}' = \Pi_K(\mathbf{B})$, которая полного ранга (т.к. все строки линейно независимы), и запустим алгоритм для матриц полного ранга строки, чтобы найти ЭНФ \mathbf{B}'' от \mathbf{B}' .
3. Применить функцию, обратную операции проецирования, Π_K^{-1} к ЭНФ \mathbf{B}'' , чтобы получить матрицу \mathbf{H} , которая является ЭНФ матрицы \mathbf{B} .

Алгоритм прост, но нужно обратить внимание на операцию проецирования и обратную к ней. Для того, чтобы находить результат проецирования напомним функцию `get_linearly_independent_rows_by_gram_schmidt`, которая будет возвращать матрицу \mathbf{B}' , состоящую из линейно независимых строк, а также массив индексов этих строк из исходного массива. К матрице \mathbf{B}' 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 \mathbf{B}' и после чего сложить их, чтобы получить нужную строку, которую необходимо добавить к \mathbf{B}' . Также восстановить строки можно через коэффициенты Грама-Шмидта, для этого на этапе ортогонализации необходимо составить матрицу, состоящую из этих коэффициентов:

$$\mathbf{T} = \begin{bmatrix} 1 & \mu_{2,1} & \cdots & \mu_{n,1} \\ & \ddots & & \vdots \\ & & 1 & \mu_{n,n-1} \\ & & & 1 \end{bmatrix}$$

после чего эту матрицу необходимо умножить на \mathbf{B}' . Получившаяся матрица будет ЭНФ матрицы \mathbf{B} .

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

5.5. Пример нахождения ЭНФ

Рассмотрим нахождение ЭНФ на примере небольшой матрицы размера 2×2 . Получим случайную матрицу $\mathbf{B} = \begin{bmatrix} \mathbf{b}_1^T \\ \vdots \\ \mathbf{b}_m^T \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 1 & 4 \end{bmatrix}$. 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 l линейно независимых строк матрицы \mathbf{B} , используя алгоритм ортогонализации Грама-Шмидта. Обозначим искомую ортого-

нализацию строк за $\mathbf{B}^* = \begin{bmatrix} \mathbf{b}_1^{T*} \\ \vdots \\ \mathbf{b}_m^{T*} \end{bmatrix}$ и найдем их:

1. $\mathbf{b}_1^{T*} = \mathbf{b}_1^T + \sum_{j<1} \mu_{1,j} \mathbf{b}_j^{T*} = \mathbf{b}_1^T = \begin{bmatrix} 2 & 4 \end{bmatrix}$,
2. $\mathbf{b}_2^{T*} = \mathbf{b}_2^T + \sum_{j<2} \mu_{2,j} \mathbf{b}_j^{T*} = \mathbf{b}_2^T + \frac{\langle \mathbf{b}_2^T, \mathbf{b}_1^{T*} \rangle}{\langle \mathbf{b}_1^{T*}, \mathbf{b}_1^{T*} \rangle} \mathbf{b}_1^{T*} = \begin{bmatrix} -\frac{4}{5} & \frac{2}{5} \end{bmatrix}$.

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

Рассмотрим алгоритм для полного ранга строки. Алгоритм принимает на вход матрицу $\mathbf{B} = [\mathbf{b}_1, \dots, \mathbf{b}_n] = \begin{bmatrix} 2 & 4 \\ 1 & 4 \end{bmatrix}$. Требуется найти m линейно независимых строк, используя ортогонализацию Грама-Шмидта. Используем этот алгоритм на строки \mathbf{B} :

1. $\mathbf{b}_1^* = \mathbf{b}_1 + \sum_{j<1} \mu_{1,j} \mathbf{b}_j^* = \mathbf{b}_1 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$,
2. $\mathbf{b}_2^* = \mathbf{b}_2 + \sum_{j<2} \mu_{2,j} \mathbf{b}_j^* = \mathbf{b}_2 + \frac{\langle \mathbf{b}_2, \mathbf{b}_1^* \rangle}{\langle \mathbf{b}_1^*, \mathbf{b}_1^* \rangle} \mathbf{b}_1^* = \begin{bmatrix} -\frac{4}{5} \\ \frac{8}{5} \end{bmatrix}$.

Т.к. матрица полного ранга строки, ее ранг меньше либо равен количеству столбцов и равен количеству строк m . 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 $\mathbf{B}' = \begin{bmatrix} 2 & 4 \\ 1 & 4 \end{bmatrix}$ размера $m \times m$, состоящую из линейно независимых столбцов матрицы \mathbf{B} .

Далее необходимо составить матрицу \mathbf{H}_0 . Для этого необходимо найти определитель решетки $d = \sqrt{(5 \cdot \frac{16}{5})} = 4$ и умножить единичную матрицу размера $m \times m$ на d .

Для $i = 1, \dots, n$ используем AddColumn для каждого \mathbf{H}_{i-1} и \mathbf{b}_i :

$$1. \mathbf{H} = \begin{bmatrix} 4 & 0 \\ 0 & 4 \end{bmatrix}, a = 4, \mathbf{h} = \begin{bmatrix} 0 \end{bmatrix}, \mathbf{H}' = \begin{bmatrix} 4 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, b = 2, \mathbf{b}' = \begin{bmatrix} 1 \end{bmatrix}.$$

Используем расширенный НОД алгоритм, находим $g = 2, x = 0, y = 1$. Составляем матрицу $\mathbf{U} = \begin{bmatrix} 0 & -1 \\ 1 & 2 \end{bmatrix}$, умножаем матрицу, составленную из первого столбца \mathbf{H} и столбца

$$\mathbf{b}$$
 на матрицу \mathbf{U} : $\begin{bmatrix} a & b \\ \mathbf{h} & \mathbf{b}' \end{bmatrix} \mathbf{U} = \begin{bmatrix} g & 0 \\ \mathbf{h}' & \mathbf{b}'' \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix}, \mathbf{h}' = \begin{bmatrix} 1 \end{bmatrix}, \mathbf{b}'' = \begin{bmatrix} 2 \end{bmatrix}.$

Сокращаем \mathbf{b}'' по модулю диагональных элементов из \mathbf{H}' , вычисляя и добавляя соответствующий вектор из $\mathcal{L}(\mathbf{H}')$: $\mathbf{b}'' = \begin{bmatrix} 2 \end{bmatrix}$.

Рекурсивно вызываем AddColumn со входом \mathbf{H}' и \mathbf{b}'' , получаем матрицу $\mathbf{H}'' = \begin{bmatrix} 2 \end{bmatrix}$:

$$\bullet \mathbf{H} = \begin{bmatrix} 4 \end{bmatrix}, a = 4, \mathbf{h} = \begin{bmatrix} \end{bmatrix}, \mathbf{H}' = \begin{bmatrix} \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 2 \end{bmatrix}, b = 2, \mathbf{b}' = \begin{bmatrix} \end{bmatrix}.$$

Находим $g = 2, x = 0, y = 1$. Составляем матрицу $\mathbf{U} = \begin{bmatrix} 0 & -1 \\ 1 & 2 \end{bmatrix}$, умножаем:

$$\begin{bmatrix} a & b \\ \mathbf{h} & \mathbf{b}' \end{bmatrix} \mathbf{U} = \begin{bmatrix} g & 0 \\ \mathbf{h}' & \mathbf{b}'' \end{bmatrix} = \begin{bmatrix} 2 & 0 \end{bmatrix}, \mathbf{h}' = \begin{bmatrix} \end{bmatrix}, \mathbf{b}'' = \begin{bmatrix} \end{bmatrix}.$$

Сокращаем \mathbf{b}'' по модулю диагональных элементов из \mathbf{H}' , вычисляя и добавляя соответствующий вектор из $\mathcal{L}(\mathbf{H}')$: $\mathbf{b}'' = \begin{bmatrix} \end{bmatrix}$.

Рекурсивно вызываем AddColumn со входом \mathbf{H}' и \mathbf{b}'' : произойдет выход из рекурсии по условию и вернется пустая матрица \mathbf{H}'' .

Сокращаем \mathbf{h}' по модулю диагональных элементов из \mathbf{H}'' , вычисляя и добавляя соответствующий вектор из $\mathcal{L}(\mathbf{H}'')$: $\mathbf{h}' = \begin{bmatrix} \end{bmatrix}$.

$$\text{Возвращаем } \begin{bmatrix} g & \mathbf{0}^T \\ \mathbf{h}' & \mathbf{H}'' \end{bmatrix} = \begin{bmatrix} 2 \end{bmatrix}.$$

Сокращаем \mathbf{h}' по модулю диагональных элементов из \mathbf{H}'' , вычисляя и добавляя соответствующий вектор из $\mathcal{L}(\mathbf{H}'')$: $\mathbf{h}' = \begin{bmatrix} 2 \end{bmatrix}$.

$$\text{Возвращаем } \begin{bmatrix} g & \mathbf{0}^T \\ \mathbf{h}' & \mathbf{H}'' \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix}.$$

$$2. \mathbf{H} = \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix}, a = 2, \mathbf{h} = \begin{bmatrix} 1 \end{bmatrix}, \mathbf{H}' = \begin{bmatrix} 2 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 4 \\ 4 \end{bmatrix}, b = 4, \mathbf{b}' = \begin{bmatrix} 4 \end{bmatrix}.$$

Находим $g = 2, x = 0, y = 1$. Составляем матрицу $\mathbf{U} = \begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}$, умножаем: $\begin{bmatrix} \mathbf{a} & \mathbf{b} \\ \mathbf{h} & \mathbf{b}' \end{bmatrix} \mathbf{U} = \begin{bmatrix} g & 0 \\ \mathbf{h}' & \mathbf{b}'' \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix}, \mathbf{h}' = \begin{bmatrix} 1 \end{bmatrix}, \mathbf{b}'' = \begin{bmatrix} 2 \end{bmatrix}$.

Сокращаем \mathbf{b}'' по модулю диагональных элементов из \mathbf{H}' , вычисляя и добавляя соответствующий вектор из $\mathcal{L}(\mathbf{H}')$: $\mathbf{b}'' = \begin{bmatrix} 0 \end{bmatrix}$.

Рекурсивно вызываем AddColumn со входом \mathbf{H}' и \mathbf{b}'' , получаем матрицу $\mathbf{H}'' = \begin{bmatrix} 2 \end{bmatrix}$:

$$\bullet \mathbf{H} = \begin{bmatrix} 2 \end{bmatrix}, a = 2, \mathbf{h} = \begin{bmatrix} \end{bmatrix}, \mathbf{H}' = \begin{bmatrix} \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 0 \end{bmatrix}, b = 0, \mathbf{b}' = \begin{bmatrix} \end{bmatrix}.$$

Находим $g = 2, x = 1, y = 0$. Составляем матрицу $\mathbf{U} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, умножаем:

$$\begin{bmatrix} \mathbf{a} & \mathbf{b} \\ \mathbf{h}' & \mathbf{b}'' \end{bmatrix} \mathbf{U} = \begin{bmatrix} g & 0 \\ \mathbf{h}' & \mathbf{b}'' \end{bmatrix} = \begin{bmatrix} 2 & 0 \end{bmatrix}, \mathbf{h}' = \begin{bmatrix} \end{bmatrix}, \mathbf{b}'' = \begin{bmatrix} \end{bmatrix}.$$

Сокращаем \mathbf{b}'' по модулю диагональных элементов из \mathbf{H}' , вычисляя и добавляя соответствующий вектор из $\mathcal{L}(\mathbf{H}')$: $\mathbf{b}'' = \begin{bmatrix} \end{bmatrix}$.

Рекурсивно вызываем AddColumn со входом \mathbf{H}' и \mathbf{b}'' : произойдет выход из рекурсии по условию и вернется пустая матрица \mathbf{H}'' .

Сокращаем \mathbf{h}' по модулю диагональных элементов из \mathbf{H}'' , вычисляя и добавляя соответствующий вектор из $\mathcal{L}(\mathbf{H}'')$: $\mathbf{h}' = \begin{bmatrix} \end{bmatrix}$.

$$\text{Возвращаем } \begin{bmatrix} g & \mathbf{0}^T \\ \mathbf{h}' & \mathbf{H}'' \end{bmatrix} = \begin{bmatrix} 2 \end{bmatrix}.$$

Сокращаем \mathbf{h}' по модулю диагональных элементов из \mathbf{H}'' , вычисляя и добавляя соответствующий вектор из $\mathcal{L}(\mathbf{H}'')$: $\mathbf{h}' = \begin{bmatrix} 2 \end{bmatrix}$.

$$\text{Возвращаем } \begin{bmatrix} g & \mathbf{0}^T \\ \mathbf{h}' & \mathbf{H}'' \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix}.$$

$$\text{ЭНФ}(\mathbf{B}) = \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix}.$$

5.6. Применение ЭНФ

Будут рассмотрены некоторые проблемы и задачи теории решеток и их решение с помощью ЭНФ[1].

Нахождение базиса. Дан набор рациональных векторов \mathbf{B} , необходимо вычислить базис для $\mathcal{L}(\mathbf{B})$. Проблема решается за полиномиальное время путем вычисления ЭНФ(\mathbf{B}):

$$\mathbf{B} = \begin{bmatrix} 2 & 2 \\ 1 & 0 \end{bmatrix}, \text{ЭНФ}(\mathbf{B}) = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}.$$

Проблема эквивалентности. Дано два базиса \mathbf{B} и \mathbf{B}' . Необходимо узнать, образуют ли они одинаковую решетку $\mathcal{L}(\mathbf{B}) = \mathcal{L}(\mathbf{B}')$. Проблема решается путем вычисления ЭНФ(\mathbf{B}) и ЭНФ(\mathbf{B}') и сравнения их равенства:

$\mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \mathbf{B}' = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, \text{ЭНФ}(\mathbf{B}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \text{ЭНФ}(\mathbf{B}') = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ — образуют одинаковую решетку.

$\mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \mathbf{B}' = \begin{bmatrix} 2 & 2 \\ 1 & 0 \end{bmatrix}, \text{ЭНФ}(\mathbf{B}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \text{ЭНФ}(\mathbf{B}') = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$ — не образуют одинаковой решетки.

Объединение решеток. Дано два базиса \mathbf{B} и \mathbf{B}' . Необходимо найти базис для наименьшей решетки, содержащей обе решетки $\mathcal{L}(\mathbf{B})$ и $\mathcal{L}(\mathbf{B}')$. Такая решетка будет сгенерирована от $[\mathbf{B}|\mathbf{B}']$, и можно легко найти ее базис через ЭНФ:

$$\mathbf{B} = \begin{bmatrix} 2 & 2 \\ 1 & 0 \end{bmatrix}, \mathbf{B}' = \begin{bmatrix} 0 & 1 \\ 2 & 2 \end{bmatrix}, [\mathbf{B}|\mathbf{B}'] = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 0 & 2 & 2 \end{bmatrix}, \text{ЭНФ}([\mathbf{B}|\mathbf{B}']) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}.$$

Проблема включения. Дано два базиса \mathbf{B} и \mathbf{B}' . Необходимо узнать, является ли $\mathcal{L}(\mathbf{B}')$ подрешеткой $\mathcal{L}(\mathbf{B})$, т.е. $\mathcal{L}(\mathbf{B}') \subseteq \mathcal{L}(\mathbf{B})$. Эта проблема сводится к проблемам объединения и эквивалентности: $\mathcal{L}(\mathbf{B}') \subseteq \mathcal{L}(\mathbf{B})$ тогда и только тогда, когда $\mathcal{L}([\mathbf{B}|\mathbf{B}']) = \mathcal{L}(\mathbf{B})$. Для этого необходимо вычислить ЭНФ($[\mathbf{B}|\mathbf{B}']$) и ЭНФ(\mathbf{B}) и сравнения их равенства:

$$\mathbf{B} = \begin{bmatrix} 2 & 2 \\ 1 & 0 \end{bmatrix}, \mathbf{B}' = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, [\mathbf{B}|\mathbf{B}'] = \begin{bmatrix} 2 & 2 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}, \text{ЭНФ}([\mathbf{B}|\mathbf{B}']) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix},$$

$\text{ЭНФ}(\mathbf{B}) = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$ — $\mathcal{L}(\mathbf{B}')$ не является подрешеткой $\mathcal{L}(\mathbf{B})$.

$$\mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \mathbf{B}' = \begin{bmatrix} 2 & 2 \\ 1 & 0 \end{bmatrix}, [\mathbf{B}|\mathbf{B}'] = \begin{bmatrix} 1 & 0 & 2 & 2 \\ 0 & 1 & 1 & 0 \end{bmatrix}, \text{ЭНФ}([\mathbf{B}|\mathbf{B}']) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix},$$

$\text{ЭНФ}(\mathbf{B}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ — $\mathcal{L}(\mathbf{B}')$ является подрешеткой $\mathcal{L}(\mathbf{B})$.

Проблема содержания. Дана решетка \mathbf{B} и вектор \mathbf{v} , необходимо узнать, принадлежит ли вектор решетке ($\mathbf{v} \subseteq \mathcal{L}(\mathbf{B})$). Эта проблема сводится к проблеме включения путем проверки $\mathcal{L}([\mathbf{v}]) \subseteq \mathcal{L}(\mathbf{B})$. Если необходимо проверить содержание нескольких векторов $\mathbf{v}_1, \dots, \mathbf{v}_n$, тогда следует сначала вычислить $\mathbf{H} = \text{ЭНФ}(\mathbf{B})$, и затем проверять, равно ли \mathbf{H} ЭНФ($[\mathbf{H}|\mathbf{v}_i]$) для каждого вектора:

$$\mathbf{B} = \begin{bmatrix} 2 & 2 \\ 1 & 0 \end{bmatrix}, \mathbf{v} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}, [\mathbf{B}|\mathbf{v}] = \begin{bmatrix} 2 & 2 & 2 \\ 1 & 0 & 0 \end{bmatrix}, \text{ЭНФ}([\mathbf{B}|\mathbf{v}]) = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \text{ЭНФ}(\mathbf{B}) = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$$

— вектор $\mathbf{v} \subseteq \mathcal{L}(\mathbf{B})$.

$$\mathbf{B} = \begin{bmatrix} 2 & 2 \\ 1 & 0 \end{bmatrix}, \mathbf{v} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, [\mathbf{B}|\mathbf{v}] = \begin{bmatrix} 2 & 2 & 1 \\ 1 & 0 & 0 \end{bmatrix}, \text{ЭНФ}([\mathbf{B}|\mathbf{v}]) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \text{ЭНФ}(\mathbf{B}) = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$$

— вектор $\mathbf{v} \not\subseteq \mathcal{L}(\mathbf{B})$.

5.7. Определение проблемы ближайшего вектора

Рассмотрим проблему ближайшего вектора[1]: дан базис решетки $\mathbf{B} \in \mathbb{R}^{d \times n}$ и вектор $\mathbf{t} \in \mathbb{R}^d$, найти точку решетки $\mathbf{B}\mathbf{x}$ ($\mathbf{x} \in \mathbb{Z}^n$) такую, что $\|\mathbf{t} - \mathbf{B}\mathbf{x}\|$ (расстояние от точки до решетки) минимально. Это задача оптимизации (минимизации) с допустимыми решениями, заданными всеми целочисленными векторами $\mathbf{x} \in \mathbb{Z}^n$, и целевой функцией $f(\mathbf{x}) = \|\mathbf{t} - \mathbf{B}\mathbf{x}\|$.

Пусть $\mathbf{B} = [\mathbf{B}', \mathbf{b}]$ и $\mathbf{x} = (\mathbf{x}', x)$, где $\mathbf{B}' \in \mathbb{R}^{d \times (n-1)}$, $\mathbf{b} \in \mathbb{R}^d$, $\mathbf{x}' \in \mathbb{Z}^{n-1}$ и $x \in \mathbb{Z}$. Заметим, что если зафиксировать значение x , то задача ПБВ(\mathbf{B}, \mathbf{t}) потребует найти значение $\mathbf{x}' \in \mathbb{Z}^{n-1}$ такое, что

$$\|\mathbf{t} - (\mathbf{B}'\mathbf{x}' + \mathbf{b}x)\| = \|(\mathbf{t} - \mathbf{b}x) - \mathbf{B}'\mathbf{x}'\|$$

минимально. Это также ПБВ (\mathbf{B}', \mathbf{t}') с измененным вектором $\mathbf{t}' = \mathbf{t} - \mathbf{b}x$, и решеткой меньшего размера $\mathcal{L}(\mathbf{B}')$. В частности, пространство решений сейчас состоит из $(n-1)$ целочисленных переменных \mathbf{x}' . Это говорит о том, что можно решить ПБВ путем установки значения x 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:

$$\min_x f(\mathbf{x}) = \text{dist}(\mathbf{t}, \mathcal{L}(\mathbf{B})) \geq \text{dist}(\mathbf{t}, \text{span}(\mathbf{B})) = \|\mathbf{t} \perp \mathbf{B}\|$$

5.8. Жадный метод: алгоритм ближайшей плоскости Бабая

Суть жадного метода состоит в выборе переменных, определяющих пространство решений, по одной, каждый раз выбирая значение, которые выглядят наиболее многообещающим[1]. В нашем случае, выберем значение x , которое дает наименьшее возможное значение для нижней границы $\|\mathbf{t}' \perp \mathbf{B}'\|$. Напомним, что $\mathbf{B} = [\mathbf{B}', \mathbf{b}]$ и $\mathbf{x} = (\mathbf{x}', x)$, и что для любого фиксированного значения x , ПБВ (\mathbf{B}, \mathbf{t}) сводится к ПБВ (\mathbf{B}', \mathbf{t}'), где $\mathbf{t}' = \mathbf{t} - \mathbf{b}x$. Используя $\|\mathbf{t}' \perp \mathbf{B}'\|$ для нижней границы, мы хотим выбрать значение x такое, что

$$\|\mathbf{t}' \perp \mathbf{B}'\| = \|\mathbf{t} - \mathbf{b}x \perp \mathbf{B}'\| = \|(\mathbf{t} \perp \mathbf{B}') - (\mathbf{b} \perp \mathbf{B}')x\|$$

как можно меньше. Это очень простая 1-размерная ПБВ проблема (с решеткой $\mathcal{L}(\mathbf{b} \perp \mathbf{B}')$ и целью $\mathbf{t} \perp \mathbf{B}'$), которая может быть сразу решена установкой

$$x = \left\lfloor \frac{\langle \mathbf{t}, \mathbf{b}^* \rangle}{\|\mathbf{b}^*\|^2} \right\rfloor$$

где $\mathbf{b}^* = \mathbf{b} \perp \mathbf{B}'$ компонента вектора \mathbf{b} , ортогональная другим базисным векторам. Полный алгоритм приведен ниже:

Input: $[\mathbf{B}, \mathbf{b}], \mathbf{t}$

Output: $\begin{cases} 0 & \text{Input} = [], \mathbf{t} \\ c \cdot \mathbf{b} + \text{Greedy}(\mathbf{B}, \mathbf{t} - c \cdot \mathbf{b}) & \text{Input} = [\mathbf{B}, \mathbf{b}], \mathbf{t} \end{cases}$

$\mathbf{b}^* \leftarrow \mathbf{b} \perp \mathbf{B}$

$x \leftarrow \langle \mathbf{t}, \mathbf{b}^* \rangle / \langle \mathbf{b}^*, \mathbf{b}^* \rangle$

$c \leftarrow \lfloor x \rfloor$

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

5.9. Нерекурсивная реализация

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

Input: \mathbf{B}, \mathbf{t}

Output: **result**

$\mathbf{GS} \leftarrow \text{GramSchmidt}(\mathbf{B})$

$n \leftarrow \mathbf{B}.columns$

$result \leftarrow \mathbf{0}$

for $i \leftarrow 0$ to n **do**

$index \leftarrow n - i - 1$

$\mathbf{b} \leftarrow \mathbf{B}.column(index)$

$\mathbf{b}^* \leftarrow \mathbf{GS}.column(index)$

$x \leftarrow \langle \mathbf{t}, \mathbf{b}^* \rangle / \langle \mathbf{b}^*, \mathbf{b}^* \rangle$

$c \leftarrow \lfloor x \rfloor$

$\mathbf{t} \leftarrow \mathbf{t} - c \cdot \mathbf{b}$

$result \leftarrow result + c \cdot \mathbf{b}$

end for

5.10. Пример жадного метода

Рассмотрим пример на простой решетке $\mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ и целевым вектором $\mathbf{t} = \begin{bmatrix} 1.6 \\ 1.6 \end{bmatrix}$.

Представим входную матрицу в виде $[\mathbf{B}, \mathbf{b}]$. На каждом шаге нам необходимо вычислять вектор $\mathbf{b}^* = \mathbf{b} \perp \mathbf{B}$. 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 матрицы и отредактированной целью. Таким образом мы найдем коэффициенты для каждого вектора базиса, и ответ будет суммой умножения коэффициентов на соответствующий вектор базиса:

$$1. [\mathbf{B}, \mathbf{b}] = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \mathbf{t} = \begin{bmatrix} 1.6 \\ 1.6 \end{bmatrix}, \mathbf{b}^* = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, x = 1.6, c = 2, c \cdot \mathbf{b} = \begin{bmatrix} 0 \\ 2 \end{bmatrix}.$$

$$\text{Рекурсивно вызываем метод, на вход отправляем } [\mathbf{B}, \mathbf{b}] = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \mathbf{t} = \mathbf{t} - c \cdot \mathbf{b} = \begin{bmatrix} 0 \\ -0.4 \end{bmatrix}.$$

$$2. [\mathbf{B}, \mathbf{b}] = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \mathbf{t} = \begin{bmatrix} 0 \\ -0.4 \end{bmatrix}, \mathbf{b}^* = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, x = 1.6, c = 2, c \cdot \mathbf{b} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}.$$

$$\text{Рекурсивно вызываем метод, на вход отправляем } [\mathbf{B}, \mathbf{b}] = [], \mathbf{t} = \mathbf{t} - c \cdot \mathbf{b} = \begin{bmatrix} -2 \\ -0.4 \end{bmatrix}.$$

3. Т.к. $[\mathbf{B}, \mathbf{b}] = []$, то возвращаем пустой вектор.

В итоге сумма векторов будет равна $\begin{bmatrix} 2 \\ 2 \end{bmatrix}$ – искомый вектор.

5.11. Метод ветвей и границ

Алгоритм похож на жадный метод, но вместо установки x_n на наиболее подходящее значение (то есть на то, для которого нижняя граница расстояния $\mathbf{t}' \perp \mathbf{B}'$ минимальна), мы ограничиваем множество всех возможных значений для x , 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 x , нам также нужна верхняя граница расстояния от цели до решетки. Ее можно получить несколькими способами. Например, можно просто использовать $\|\mathbf{t}\|$ (расстояние от цели до начала координат) в качестве верхней границы. Но лучше использовать жадный алгоритм, чтобы найти приближенное решение $\mathbf{v} = \text{Greedy}(\mathbf{B}, \mathbf{t})$, и использовать $\|\mathbf{t} - \mathbf{v}\|$ в качестве верхней границы. Как только верхняя граница u установлена, можно ограничить переменную x такими значениями, что $\|\mathbf{t} - x\mathbf{b}\| \perp \mathbf{B}'\| \leq u$.

Количество рекурсивных вызовов будет не больше, чем число

$$T = \prod_i \left\lceil \sqrt{\sum_{i \leq j} (\|\mathbf{b}_i^*\| / \|\mathbf{b}_j^*\|)^2} \right\rceil = m!$$

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

Окончательный алгоритм похож на жадный метод:

Input: $[\mathbf{B}, \mathbf{b}], \mathbf{t}$

Output: $\begin{cases} 0 & \text{Input} = [], \mathbf{t} \\ \text{closest}(V, \mathbf{t}) & \text{Input} = [\mathbf{B}, \mathbf{b}], \mathbf{t} \end{cases}$

$\mathbf{b}^* \leftarrow \mathbf{b} \perp \mathbf{B}$

$\mathbf{v} \leftarrow \text{Greedy}([\mathbf{B}, \mathbf{b}], \mathbf{t})$

$X \leftarrow \{x : \|(\mathbf{t} - x\mathbf{b}) \perp \mathbf{B}\| \leq \|\mathbf{t} - \mathbf{v}\|\}$

$V \leftarrow \{x \cdot \mathbf{b} + \text{Branch\&Bound}(\mathbf{B}, \mathbf{t} - x \cdot \mathbf{b}) : x \in X\}$

где $\text{closest}(V, \mathbf{t})$ выбирает вектор в $V \subset \mathcal{L}(\mathbf{B})$ ближайший к цели \mathbf{t} .

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 X . Его можно найти, используя выражение, выведенное в прошлом алгоритме: $x = \frac{\langle \mathbf{t}, \mathbf{b}^* \rangle}{\|\mathbf{b}^*\|^2}$. С помощью него мы найдем x , который точно удовлетворяет множеству, а затем будем увеличивать/уменьшать до тех пор, пока выполняется условие $\|(\mathbf{t} - x\mathbf{b}) \perp \mathbf{B}\| \leq \|\mathbf{t} - \mathbf{v}\|$.

5.12. Пример метода ветвей и границ

Рассмотрим пример на простой решетке $\mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ и целевым вектором $\mathbf{t} = \begin{bmatrix} 1.6 \\ 1.6 \end{bmatrix}$.

Представим входную матрицу в виде $[\mathbf{B}, \mathbf{b}]$. На каждом шаге нам необходимо вычислять вектор $\mathbf{b}^* = \mathbf{b} \perp \mathbf{B}$. 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 X , подходящее под условие $\|(\mathbf{t} - x\mathbf{b}) \perp \mathbf{B}\| \leq \|\mathbf{t} - \mathbf{v}\|$. Вектор \mathbf{v} найдем с помощью жадного метода. Далее также, как и в жадном методе ищем необходимую сумму векторов, получим множество V , из которого необходимо будет выбрать ближайший к цели \mathbf{t} .

$$[\mathbf{B}, \mathbf{b}] = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \mathbf{t} = \begin{bmatrix} 1.6 \\ 1.6 \end{bmatrix}, \mathbf{b}^* = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \mathbf{v} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}, X = \{2, 3, 1, 0\}.$$

Рекурсивно вызываем метод для каждого $x \in X$, на вход отправляем $[\mathbf{B}, \mathbf{b}] = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$,

$$\mathbf{t} = \mathbf{t} - x \cdot \mathbf{b}.$$

Получаем множество $V = \left\{ \begin{bmatrix} 2 \\ 2 \end{bmatrix}, \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 0 \end{bmatrix} \right\}.$

Ближайший вектор будет равен $\begin{bmatrix} 2 \\ 2 \end{bmatrix}$ – искомый вектор.

5.13. Параллельная реализация метода ветвей и границ

Можно увидеть, что процесс нахождения ближайшего вектора в методе ветвей и границ является деревом: для каждого подходящего значения x из множества X 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 параллельный подход.

Для получения параллельной реализации будем использовать задачи (task) из библиотеки OpenMP. После получения множества X будем находить множество векторов V следующим образом: для каждого значения $x \in X$ 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(V, t)`.

6. Обзор существующих решений

6.1. 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 7×7 , 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:

The screenshot shows the WolframAlpha web interface. At the top, there is a title bar "Convert Matrix to Hermite Normal Form" with a close button. Below it, the input field "Enter Matrix:" contains the input $\{\{12, 19, 28\}, \{7, 11, 16\}\}$. A "Submit" button is located below the input field. The "Input:" section shows the input matrix as a 2x3 grid. The "Result:" section displays the Hermite decomposition $A.U = H$ where $A = \begin{pmatrix} 12 & 19 & 28 \\ 7 & 11 & 16 \end{pmatrix}$, $U = \begin{pmatrix} -11 & 19 \\ 7 & -12 \end{pmatrix}$, and $H = \begin{pmatrix} 1 & 0 & -4 \\ 0 & 1 & 4 \end{pmatrix}$.

Рис. 1: Нахождение ЭНФ с помощью WolframAlpha.

6.2. 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 50×50). Данный сервис можно использовать для отладки на больших размерах матриц, чем при использовании WolframAlpha, и увидеть, как сильно растут числа на больших матрицах.

Пример работы:

row dimension ($2 \leq m \leq 50$): column dimension ($1 \leq n \leq 50$):

Matrix:

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

A:

6	2	4	5	10	3	8	7	5	10
8	7	6	4	6	5	4	9	7	8
4	4	3	1	9	4	8	9	6	6
10	6	1	4	1	3	7	2	7	10
1	8	2	1	6	10	2	6	7	4
6	8	10	5	3	9	7	1	8	2
9	3	10	9	9	5	5	4	6	5
6	1	1	10	8	8	1	5	5	5
10	6	7	3	1	10	7	4	4	2
10	4	8	7	8	9	8	1	8	9

rank(A) = 10

Hermite normal form HNF(A):

1	0	0	0	0	1	0	0	0	22183814
0	1	0	0	0	1	0	0	0	40081888
0	0	1	0	0	1	0	0	0	46402868
0	0	0	1	0	1	0	0	0	26022823
0	0	0	0	1	0	0	0	0	43969157
0	0	0	0	0	2	0	0	0	30212733
0	0	0	0	0	0	1	0	0	21882344
0	0	0	0	0	0	0	1	0	14442922
0	0	0	0	0	0	0	0	1	39143122
0	0	0	0	0	0	0	0	0	64687868

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

Решение ПБВ ограничено размером 25×25 . На вход идет матрица, в которой последняя строка является вектором, для которого надо найти ближайшую точку решетки.

matrix A:

1	0
0	2
5	6

$P = A[3] = (5, 6)$
 \mathcal{L} is the lattice spanned by the first 2 rows of A

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

(8, 12)	(-3, -6)	45
(9, 12)	(-4, -6)	52
(10, 12)	(-5, -6)	61

Also
(5, 6)
is the closest vector of \mathcal{L} to P , with shortest distance squared 0

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

6.3. 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.

```

1  import numpy as np
2  from hsnf import column_style_hermite_normal_form, row_style_hermite_normal_form, smith_normal_form
3
4  # Integer matrix to be decomposed
5  M = np.random.random_integers(1, 10, (4, 4))
6
7  print(M)
8
9  # Row-style hermite normal form
10 H, L = row_style_hermite_normal_form(M)
11
12 print(H)
13
14 # Column-style hermite normal form
15 H, R = column_style_hermite_normal_form(M)
16
17 print(H)
18

```

ПРОБЛЕМЫ ВЫХОДНЫЕ ДАННЫЕ КОНСОЛЬ ОТЛАДКИ ТЕРМИНАЛ

```

[[ 4  8  9  5]
 [ 5 10  4  6]
 [ 8  1  6 10]
 [ 5  1  6  4]]
[[ 1  2  0 314]
 [ 0  3  0 132]
 [ 0  0  1 197]
 [ 0  0  0 336]]
[[ 1  0  0  0]
 [ 0  1  0  0]
 [ 2  0  3  0]
 [326  9  69 336]]
PS C:\programming\test>

```

Рис. 6: Нахождение ЭНФ с помощью hsnf.

7. 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 [10]. 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 `smake` 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 (двухканальный режим 8x2), 2666 МГц. Тестирование проводилось на одинаковых данных.

7.1. Вспомогательные функции

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

`add_column(HNF, column) → [HNF|column]` — функция, используемая при нахождении ЭНФ. Принимает ЭНФ и возвращает `[HNF|column]`.

`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 ортогонализированный 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 T .

`gcd_extended(a, b) → g, x, y` — расширенный НОД алгоритм, возвращает g, x, y такие, что $g = xa + yb$.

`add_column_GMP(HNF, column) → [HNF|column]` — функция, используемая при нахождении ЭНФ. Принимает ЭНФ и возвращает $[HNF|column]$. 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 ортогонализированный 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 T . Использует реализацию больших чисел от GMP.

`gcd_extended_GMP(a, b) → g, x, y` — расширенный НОД алгоритм, возвращает g, x, y такие, что $g = xa + yb$. 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 $vector \perp matrix$.

`closest_vector(matrix, vector) → result_vector` — принимает набор векторов и целевой вектор, возвращает вектор из набора, ближайший к целевому.

7.2. 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

7.3. 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.

7.4. 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.

8. 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

1. Daniele Micciancio. Basic Algorithms. [Электронный ресурс]. — URL: <https://cseweb.ucsd.edu/classes/sp14/cse206A-a/lec4.pdf> (Дата обращения: 16.05.2022).
2. Daniele Micciancio. Point Lattices. [Электронный ресурс]. — URL: <https://cseweb.ucsd.edu/classes/sp14/cse206A-a/lec1.pdf> (Дата обращения: 16.05.2022).
3. 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
4. Голубева Е.А. Линейная алгебра: Учебно-методическое пособие. – Нижний Новгород: Нижегородский госуниверситет, 2022. – 31 с.
5. Документация библиотеки Eigen. [Электронный ресурс]. — URL: <https://eigen.tuxfamily.org/dox/index.html> (Дата обращения: 16.05.2022).
6. Документация библиотеки Boost.Multiprecision. [Электронный ресурс]. — URL: https://www.boost.org/doc/libs/1_79_0/libs/multiprecision/doc/html/index.html (Дата обращения: 16.05.2022).
7. Документация WolframAlpha API. [Электронный ресурс]. — URL: <https://products.wolframalpha.com/simple-api/documentation/> (Дата обращения 16.05.2022).
8. Сервис для проверки Эрмитовой нормальной формы. [Электронный ресурс]. — URL: <http://www.numbertheory.org/php/lllhermite1.html> (Дата обращения: 16.05.2022).
9. Библиотека hsnf. [Электронный ресурс]. — URL: <https://github.com/lan496/hsnf> (Дата обращения: 16.05.2022).
10. Github репозиторий. [Электронный ресурс]. — URL: <https://github.com/DenisOgnev/Lattice-Algorithms> (Дата обращения: 16.05.2022).

Приложения

Приложение А. Исходный код algorithms.hpp

```
1  #ifndef ALGOTITHMS_HPP
2  #define ALGOTITHMS_HPP
3
4  #include <Eigen/Dense>
5  #include <boost/multiprecision/cpp_int.hpp>
6  #ifdef GMP
7  #include <boost/multiprecision/gmp.hpp>
8  #endif
9
10 namespace Algorithms
11 {
12     namespace HNF
13     {
14         Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
15             HNF_full_row_rank(const
16                 Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1> &B);
17         Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1> HNF(const
18             Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1> &B);
19
20         #ifdef GMP
21         Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>
22             HNF_full_row_rank_GMP(const
23                 Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1> &B);
24         Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>
25             HNF_GMP(const Eigen::Matrix<boost::multiprecision::mpz_int,
26                 -1, -1> &B);
27         #endif
28     }
29     namespace CVP
30     {
31         Eigen::VectorXd greedy_recursive(const Eigen::MatrixXd &matrix,
32             const Eigen::VectorXd &target);
33         Eigen::VectorXd greedy(const Eigen::MatrixXd &matrix, const
34             Eigen::VectorXd &target);
35         Eigen::VectorXd branch_and_bound(const Eigen::MatrixXd &matrix,
36             const Eigen::VectorXd &target);
37
38         #ifdef PARALLEL
39         Eigen::VectorXd branch_and_bound_parallel(const Eigen::MatrixXd
40             &matrix, const Eigen::VectorXd &target);
41         #endif
42     }
43     #ifdef PARALLEL
44     Eigen::MatrixXd gram_schmidt_parallel(const Eigen::MatrixXd &matrix,
45         bool delete_zero_rows = true);
46     #endif
47     Eigen::MatrixXd gram_schmidt_sequential(const Eigen::MatrixXd
48         &matrix, bool delete_zero_rows = true);
49 }
50 #endif
```

Приложение Б. Исходный код utils.hpp

```
1  #ifndef UTILS_HPP
2  #define UTILS_HPP
3
```

```

4  #include <Eigen/Dense>
5  #include <vector>
6  #include <boost/multiprecision/cpp_int.hpp>
7  #include <boost/multiprecision/cpp_bin_float.hpp>
8  #ifdef GMP
9  #include <boost/multiprecision/gmp.hpp>
10 #endif
11
12 namespace Utils
13 {
14     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);
15     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);
16     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);
17     Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
        generate_random_matrix(const int m, const int n, int lowest, int
            highest);
18     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);
19     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);
20     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);
21
22     #ifdef GMP
23     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);
24     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);
25     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);
26     Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>
        generate_random_matrix_GMP(const int m, const int n, int lowest,
            int highest);
27     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);
28     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);
29     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);
30     #endif
31
32     Eigen::MatrixXd generate_random_matrix_with_full_column_rank(const

```

```

33     int m, const int n, int lowest, int highest);
Eigen::VectorXd generate_random_vector(const int m, double lowest,
double highest);
34 Eigen::VectorXd projection(const Eigen::MatrixXd &matrix, const
Eigen::VectorXd &vector);
35 Eigen::VectorXd closest_vector(const std::vector<Eigen::VectorXd>
&matrix, const Eigen::VectorXd &vector);
36 }
37
38 #endif

```

Приложение В. Исходный код algorithms.cpp

```

1  #include "algorithms.hpp"
2  #include <iostream>
3  #include "utils.hpp"
4  #include <vector>
5  #include <numeric>
6
7  namespace mp = boost::multiprecision;
8
9  namespace Algorithms
10 {
11     namespace HNF
12     {
13         // Computes HNF of a integer matrix that is full row rank
14         // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
15         // @param B full row rank matrix
16         Eigen::Matrix<mp::cpp_int, -1, -1> HNF_full_row_rank(const
Eigen::Matrix<mp::cpp_int, -1, -1> &B)
17     {
18         int m = static_cast<int>(B.rows());
19         int n = static_cast<int>(B.cols());
20
21         if (m > n)
22         {
23             throw std::invalid_argument("m must be less than or
equal n");
24         }
25         if (m < 1 || n < 1)
26         {
27             throw std::invalid_argument("Matrix is not initialized");
28         }
29         if (B.isZero())
30         {
31             throw std::runtime_error("Matrix is empty");
32         }
33
34         Eigen::Matrix<mp::cpp_int, -1, -1> B_stroke;
35         Eigen::Matrix<mp::cpp_rational, -1, -1> ortogonalized;
36
37         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);
38
39         std::tie(B_stroke, ortogonalized) = result_of_gs;
40
41         mp::cpp_rational t_det = 1.0;
42         for (const Eigen::Vector<mp::cpp_rational, -1> &vec :
ortogonalized.colwise())
43         {
44             t_det *= vec.squaredNorm();
45         }
46         mp::cpp_int det = mp::sqrt(mp::numerator(t_det));
47

```

```

48     Eigen::Matrix<mp::cpp_int, -1, -1> H_temp =
49         Eigen::Matrix<mp::cpp_int, -1, -1>::Identity(m, m) * det;
50     for (int i = 0; i < n; i++)
51     {
52         H_temp = Utils::add_column(H_temp, B.col(i));
53     }
54
55     Eigen::Matrix<mp::cpp_int, -1, -1> H(m, n);
56     H.block(0, 0, H_temp.rows(), H_temp.cols()) = H_temp;
57     if (n > m)
58     {
59         H.block(0, H_temp.cols(), H_temp.rows(), n - m) =
60             Eigen::Matrix<mp::cpp_int, -1,
61                 -1>::Zero(H_temp.rows(), n - m);
62     }
63     return H;
64 }
65
66 // Computes HNF of an arbitrary integer matrix
67 // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
68 // @param B arbitrary matrix
69 Eigen::Matrix<mp::cpp_int, -1, -1> HNF(const
70     Eigen::Matrix<mp::cpp_int, -1, -1> &B)
71 {
72     int m = static_cast<int>(B.rows());
73     int n = static_cast<int>(B.cols());
74
75     if (m < 1 || n < 1)
76     {
77         throw std::invalid_argument("Matrix is not initialized");
78     }
79     if (B.isZero())
80     {
81         throw std::runtime_error("Matrix is empty");
82     }
83
84     Eigen::Matrix<mp::cpp_int, -1, -1> B_stroke;
85     std::vector<int> indicies;
86     std::vector<int> deleted_indicies;
87     Eigen::Matrix<mp::cpp_rational, -1, -1> T;
88     std::tuple<Eigen::Matrix<mp::cpp_int, -1, -1>,
89         std::vector<int>, std::vector<int>,
90         Eigen::Matrix<mp::cpp_rational, -1, -1>> projection =
91         Utils::get_linearly_independent_rows_by_gram_schmidt(B);
92     std::tie(B_stroke, indicies, deleted_indicies, T) =
93         projection;
94
95     Eigen::Matrix<mp::cpp_int, -1, -1> B_double_stroke =
96         HNF_full_row_rank(B_stroke);
97
98     Eigen::Matrix<mp::cpp_int, -1, -1> HNF(B.rows(), B.cols());
99
100     for (int i = 0; i < indicies.size(); i++)
101     {
102         HNF.row(indicies[i]) = B_double_stroke.row(i);
103     }
104
105     //////////////////////////////////////////
106     // First way: just find linear combinations of deleted rows.
107     // More accurate
108
109     // Eigen::Matrix<mp::cpp_bin_float_double, -1, -1>
110     // B_stroke_transposed =
111     // B_stroke.transpose().cast<mp::cpp_bin_float_double>();

```

```

103         // auto QR =
104             B_stroke.cast<mp::cpp_bin_float_double>().colPivHouseholderQr().t
105
106         // for (const auto &indx : deleted_indicies)
107         // {
108             Eigen::Vector<mp::cpp_bin_float_double, -1> vec =
109             B.row(indx).cast<mp::cpp_bin_float_double>();
110             Eigen::RowVector<mp::cpp_bin_float_double, -1> x =
111             QR.solve(vec);
112
113             Eigen::Vector<mp::cpp_bin_float_double, -1> res = x *
114             HNF.cast<mp::cpp_bin_float_double>();
115             for (mp::cpp_bin_float_double &elem : res)
116             {
117                 elem = mp::round(elem);
118             }
119             HNF.row(indx) = res.cast<mp::cpp_int>();
120         }
121         // return HNF;
122         //////////////////////////////////////
123
124         //////////////////////////////////////
125         // Other, the "right" way that is desribed in algorithm.
126         Eigen::Matrix<mp::cpp_bin_float_double, -1, -1> t_HNF =
127             HNF.cast<mp::cpp_bin_float_double>();
128         for (const auto &indx : deleted_indicies)
129         {
130             Eigen::Vector<mp::cpp_bin_float_double, -1> res =
131             Eigen::Vector<mp::cpp_bin_float_double,
132             -1>::Zero(B.cols());
133             for (int i = 0; i < indx; i++)
134             {
135                 res += T(indx,
136                 i).convert_to<mp::cpp_bin_float_double>() *
137                 t_HNF.row(i);
138             }
139             t_HNF.row(indx) = res;
140         }
141
142         return t_HNF.cast<mp::cpp_int>();
143         //////////////////////////////////////
144     }
145
146     #ifndef GMP
147     // Computes HNF of a integer matrix that is full row rank
148     // @return Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>
149     // @param B full row rank matrix
150     Eigen::Matrix<mp::mpz_int, -1, -1> HNF_full_row_rank_GMP(const
151     Eigen::Matrix<mp::mpz_int, -1, -1> &B)
152     {
153         int m = static_cast<int>(B.rows());
154         int n = static_cast<int>(B.cols());
155
156         if (m > n)
157         {
158             throw std::invalid_argument("m must be less than or
159             equal n");
160         }
161         if (m < 1 || n < 1)
162         {
163             throw std::invalid_argument("Matrix is not initialized");
164         }
165         if (B.isZero())
166         {
167             throw std::runtime_error("Matrix is empty");
168         }
169     }
170

```

```

159     }
160
161     Eigen::Matrix<mp::mpz_int, -1, -1> B_stroke;
162     Eigen::Matrix<mp::mpq_rational, -1, -1> ortogonalized;
163
164     std::tuple<Eigen::Matrix<mp::mpz_int, -1, -1>,
165               Eigen::Matrix<mp::mpq_rational, -1, -1>> result_of_gs =
166         Utils::get_linearly_independent_columns_by_gram_schmidt_GMP(B);
167
168     std::tie(B_stroke, ortogonalized) = result_of_gs;
169
170     mp::mpq_rational t_det = 1.0;
171     for (const Eigen::Vector<mp::mpq_rational, -1> &vec :
172          ortogonalized.colwise())
173     {
174         t_det *= vec.squaredNorm();
175     }
176     mp::mpz_int det = mp::sqrt(mp::numerator(t_det));
177
178     Eigen::Matrix<mp::mpz_int, -1, -1> H_temp =
179         Eigen::Matrix<mp::mpz_int, -1, -1>::Identity(m, m) * det;
180
181     for (int i = 0; i < n; i++)
182     {
183         H_temp = Utils::add_column_GMP(H_temp, B.col(i));
184     }
185
186     Eigen::Matrix<mp::mpz_int, -1, -1> H(m, n);
187     H.block(0, 0, H_temp.rows(), H_temp.cols()) = H_temp;
188     if (n > m)
189     {
190         H.block(0, H_temp.cols(), H_temp.rows(), n - m) =
191             Eigen::Matrix<mp::mpz_int, -1,
192                         -1>::Zero(H_temp.rows(), n - m);
193     }
194
195     return H;
196 }
197
198 // Computes HNF of an arbitrary integer matrix
199 // @return Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>
200 // @param B arbitrary matrix
201 Eigen::Matrix<mp::mpz_int, -1, -1> HNF_GMP(const
202 Eigen::Matrix<mp::mpz_int, -1, -1> &B)
203 {
204     int m = static_cast<int>(B.rows());
205     int n = static_cast<int>(B.cols());
206
207     if (m < 1 || n < 1)
208     {
209         throw std::invalid_argument("Matrix is not initialized");
210     }
211     if (B.isZero())
212     {
213         throw std::runtime_error("Matrix is empty");
214     }
215
216     Eigen::Matrix<mp::mpz_int, -1, -1> B_stroke;
217     std::vector<int> indicies;
218     std::vector<int> deleted_indicies;
219     Eigen::Matrix<mp::mpq_rational, -1, -1> T;
220     std::tuple<Eigen::Matrix<mp::mpz_int, -1, -1>,
221               std::vector<int>, std::vector<int>> projection =
222         Utils::get_linearly_independent_rows_by_gram_schmidt_GMP(B);
223     std::tie(B_stroke, indicies, deleted_indicies, T) =

```

```

216         projection;
217     Eigen::Matrix<mp::mpz_int, -1, -1> B_double_stroke =
        HNF_full_row_rank_GMP(B_stroke);
218
219     Eigen::Matrix<mp::mpz_int, -1, -1> HNF(B.rows(), B.cols());
220
221     for (int i = 0; i < indicies.size(); i++)
222     {
223         HNF.row(indicies[i]) = B_double_stroke.row(i);
224     }
225
226     //////////////////////////////////////////
227     // First way: just find linear combinations of deleted rows.
        More accurate
228
229     // Eigen::Matrix<mp::mpf_float_50, -1, -1>
        B_stroke_transposed =
        B_stroke.transpose().cast<mp::mpf_float_50>();
230     // auto QR =
        B_stroke.cast<mp::mpf_float_50>().colPivHouseholderQr().transpose
231
232     // for (const auto &indx : deleted_indicies)
233     // {
234     //     Eigen::Vector<mp::mpf_float_50, -1> vec =
        B.row(indx).cast<mp::mpf_float_50>();
235     //     Eigen::RowVector<mp::mpf_float_50, -1> x =
        QR.solve(vec);
236
237     //     Eigen::Vector<mp::mpf_float_50, -1> res = x *
        HNF.cast<mp::mpf_float_50>();
238     //     for (mp::mpf_float_50 &elem : res)
239     //     {
240     //         elem = mp::round(elem);
241     //     }
242     //     HNF.row(indx) = res.cast<mp::mpz_int>();
243     // }
244     // return HNF;
245     //////////////////////////////////////////
246
247
248     //////////////////////////////////////////
249     // Other, the "right" way that is desribed in algorithm.
250     Eigen::Matrix<mp::mpf_float_50, -1, -1> t_HNF =
        HNF.cast<mp::mpf_float_50>();
251     for (const auto &indx : deleted_indicies)
252     {
253         Eigen::Vector<mp::mpf_float_50, -1> res =
            Eigen::Vector<mp::mpf_float_50, -1>::Zero(B.cols());
254         for (int i = 0; i < indx; i++)
255         {
256             res += T(indx, i).convert_to<mp::mpf_float_50>() *
                t_HNF.row(i);
257         }
258
259         t_HNF.row(indx) = res;
260     }
261
262     return t_HNF.cast<mp::mpz_int>();
263     //////////////////////////////////////////
264 }
265 #endif
266 }
267
268 namespace CVP
269 {
270

```



```

271 Eigen::MatrixXd gram_schmidt_greedy;
272 Eigen::MatrixXd B_greedy;
273 int index_greedy;
274
275 Eigen::MatrixXd gram_schmidt_bb;
276 Eigen::MatrixXd gram_schmidt_bb_parallel;
277
278
279 // Recursive body of greedy algorithm
280 // @return Eigen::VectorXd
281 // @param target vector for which lattice point is being
282 // searched for
283 Eigen::VectorXd greedy_recursive_part(const Eigen::VectorXd
284 &target)
285 {
286     if (index_greedy == 0)
287     {
288         return Eigen::VectorXd::Zero(target.rows());
289     }
290     index_greedy--;
291     Eigen::VectorXd b = B_greedy.col(index_greedy);
292     Eigen::VectorXd b_star =
293         gram_schmidt_greedy.col(index_greedy);
294     double inner1 = std::inner_product(target.data(),
295         target.data() + target.size(), b_star.data(), 0.0);
296     double inner2 = std::inner_product(b_star.data(),
297         b_star.data() + b_star.size(), b_star.data(), 0.0);
298
299     double x = inner1 / inner2;
300     double c = std::round(x);
301
302     Eigen::VectorXd t_res = c * b;
303
304     return t_res + Algorithms::CVP::greedy_recursive_part(target
305 - t_res);
306 }
307
308 // Solves CVP using a recursive greedy algorithm
309 // @return Eigen::VectorXd
310 // @param matrix input rational lattice basis that is linearly
311 // independent
312 // @param target vector for which lattice point is being
313 // searched for
314 Eigen::VectorXd greedy_recursive(const Eigen::MatrixXd &matrix,
315 const Eigen::VectorXd &target)
316 {
317     B_greedy = matrix;
318     #ifdef PARALLEL
319     gram_schmidt_greedy =
320         Algorithms::gram_schmidt_parallel(matrix, false);
321     #else
322     gram_schmidt_greedy =
323         Algorithms::gram_schmidt_sequential(matrix, false);
324     #endif
325     index_greedy = static_cast<int>(matrix.cols());
326
327     return greedy_recursive_part(target);
328 }
329
330
331 // Solves CVP using a non recursive greedy algorithm
332 // @return Eigen::VectorXd
333 // @param matrix input rational lattice basis that is linearly
334 // independent
335 // @param target vector for which lattice point is being
336 // searched for

```

```

325 Eigen::VectorXd greedy(const Eigen::MatrixXd &matrix, const
    Eigen::VectorXd &target)
326 {
327     Eigen::MatrixXd gram_schmidt;
328     #ifdef PARALLEL
329     gram_schmidt = Algorithms::gram_schmidt_parallel(matrix,
        false);
330     #else
331     gram_schmidt = Algorithms::gram_schmidt_sequential(matrix,
        false);
332     #endif
333
334     Eigen::VectorXd result =
        Eigen::VectorXd::Zero(target.rows());
335
336     Eigen::VectorXd t_target = target;
337
338     int n = static_cast<int>(matrix.cols());
339     for (int i = 0; i < matrix.cols(); i++)
340     {
341         int index = n - i - 1;
342         Eigen::VectorXd b = matrix.col(index);
343         Eigen::VectorXd b_star = gram_schmidt.col(index);
344         double inner1 = std::inner_product(t_target.data(),
            t_target.data() + t_target.size(), b_star.data(),
            0.0);
345         double inner2 = std::inner_product(b_star.data(),
            b_star.data() + b_star.size(), b_star.data(), 0.0);
346
347         double x = inner1 / inner2;
348         double c = std::round(x);
349         Eigen::VectorXd t_res = c * b;
350
351         t_target -= t_res;
352         result += t_res;
353     }
354
355     return result;
356 }
357
358
359 // Recursive body of branch and bound algorithm
360 // @return Eigen::VectorXd
361 // @param matrix input rational lattice basis that is linearly
    independent
362 // @param target vector for which lattice point is being
    searched for
363 Eigen::VectorXd branch_and_bound_recursive_part(const
    Eigen::MatrixXd &matrix, const Eigen::VectorXd &target)
364 {
365     if (matrix.cols() == 0)
366     {
367         return Eigen::VectorXd::Zero(target.rows());
368     }
369     Eigen::MatrixXd B = matrix.block(0, 0, matrix.rows(),
        matrix.cols() - 1);
370     Eigen::VectorXd b = matrix.col(B.cols());
371     Eigen::VectorXd b_star = gram_schmidt_bb.col(B.cols());
372
373     Eigen::VectorXd v = Algorithms::CVP::greedy(matrix, target);
374
375     double upper_bound = (target - v).norm();
376
377     double x_middle = std::round(target.dot(b_star) /
        b_star.dot(b_star));
378
379     std::vector<int> X;

```

```

380     X.push_back(static_cast<int>(x_middle));
381
382     bool flag1 = true;
383     bool flag2 = true;
384
385     double x1 = x_middle + 1;
386     double x2 = x_middle - 1;
387
388     while (flag1 || flag2)
389     {
390         if (flag1 && Utils::projection(B, target - x1 *
391             b).norm() <= upper_bound)
392         {
393             X.push_back(static_cast<int>(x1));
394             x1++;
395         }
396         else
397         {
398             flag1 = false;
399         }
400
401         if (flag2 && Utils::projection(B, target - x2 *
402             b).norm() <= upper_bound)
403         {
404             X.push_back(static_cast<int>(x2));
405             x2--;
406         }
407         else
408         {
409             flag2 = false;
410         }
411     }
412
413     std::vector<Eigen::VectorXd> V;
414
415     Eigen::VectorXd t_res;
416     for (const int &x : X)
417     {
418         t_res = x * b +
419             Algorithms::CVP::branch_and_bound_recursive_part(B,
420                 target - x * b);
421         V.push_back(t_res);
422     }
423
424     return Utils::closest_vector(V, target);
425 }
426
427 // Solves CVP using a branch and bound algorithm
428 // @return Eigen::VectorXd
429 // @param matrix input rational lattice basis that is linearly
430 // independent
431 // @param target vector for which lattice point is being
432 // searched for
433 Eigen::VectorXd branch_and_bound(const Eigen::MatrixXd &matrix,
434     const Eigen::VectorXd &target)
435 {
436     #ifndef PARALLEL
437     gram_schmidt_bb = Algorithms::gram_schmidt_parallel(matrix,
438         false);
439     #else
440     gram_schmidt_bb =
441         Algorithms::gram_schmidt_sequential(matrix, false);
442     #endif
443
444     return branch_and_bound_recursive_part(matrix, target);
445 }

```

```

438     }
439
440     #ifndef PARALLEL
441     // Recursive parallel body of branch and bound algorithm
442     // @return Eigen::VectorXd
443     // @param matrix input rational lattice basis that is linearly
444     // independent
445     // @param target vector for which lattice point is being
446     // searched for
447     Eigen::VectorXd branch_and_bound_recursive_part_parallel(const
448     Eigen::MatrixXd &matrix, const Eigen::VectorXd &target)
449     {
450         if (matrix.cols() == 0)
451         {
452             return Eigen::VectorXd::Zero(target.rows());
453         }
454         Eigen::MatrixXd B = matrix.block(0, 0, matrix.rows(),
455         matrix.cols() - 1);
456         Eigen::VectorXd b = matrix.col(B.cols());
457         Eigen::VectorXd b_star =
458         gram_schmidt_bb_parallel.col(B.cols());
459
460         Eigen::VectorXd v = Algorithms::CVP::greedy(matrix, target);
461
462         double upper_bound = (target - v).norm();
463
464         double x_middle = std::round(target.dot(b_star) /
465         b_star.dot(b_star));
466
467         std::vector<int> X;
468         X.push_back(static_cast<int>(x_middle));
469
470         bool flag1 = true;
471         bool flag2 = true;
472
473         double x1 = x_middle + 1;
474         double x2 = x_middle - 1;
475
476         while (flag1 || flag2)
477         {
478             if (flag1 && Utils::projection(B, target - x1 *
479             b).norm() <= upper_bound)
480             {
481                 X.push_back(static_cast<int>(x1));
482                 x1++;
483             }
484             else
485             {
486                 flag1 = false;
487             }
488
489             if (flag2 && Utils::projection(B, target - x2 *
490             b).norm() <= upper_bound)
491             {
492                 X.push_back(static_cast<int>(x2));
493                 x2--;
494             }
495             else
496             {
497                 flag2 = false;
498             }
499         }
500
501         std::vector<Eigen::VectorXd> V;
502
503         Eigen::VectorXd result;

```

```

497     Eigen::VectorXd res;
498     #pragma omp parallel
499     {
500         #pragma omp single nowait
501         {
502             for (const int &x : X)
503             {
504                 #pragma omp task
505                 {
506                     res = x * b +
                        Algorithms::CVP::branch_and_bound_recursive_part_
                        target - x * b);
507                     #pragma omp critical
508                     V.push_back(res);
509                 }
510             }
511             #pragma omp taskwait
512             result = Utils::closest_vector(V, target);
513         }
514     }
515
516     return result;
517 }
518
519 // Solves CVP using a branch and bound parallel algorithm
520 // @return Eigen::VectorXd
521 // @param matrix input rational lattice basis that is linearly
522 // independent
523 // @param target vector for which lattice point is being
524 // searched for
525 Eigen::VectorXd branch_and_bound_parallel(const Eigen::MatrixXd
526 &matrix, const Eigen::VectorXd &target)
527 {
528     gram_schmidt_bb_parallel =
529         Algorithms::gram_schmidt_parallel(matrix, false);
530
531     return branch_and_bound_recursive_part_parallel(matrix,
532 target);
533 }
534 #endif
535 }
536
537 #ifndef PARALLEL
538 // Computes Gram Schmidt orthogonalization
539 // @return Eigen::MatrixXd
540 // @param matrix input matrix
541 // @param normalize indicates whether to normalize output vectors
542 // @param delete_zero_rows indicates whether to delete zero rows
543 Eigen::MatrixXd gram_schmidt_parallel(const Eigen::MatrixXd &matrix,
544 bool delete_zero_rows)
545 {
546     std::vector<Eigen::VectorXd> basis;
547
548     for (const auto &vec : matrix.colwise())
549     {
550         Eigen::VectorXd projections =
551             Eigen::VectorXd::Zero(vec.size());
552
553         #pragma omp parallel for
554         for (int i = 0; i < basis.size(); i++)
555         {
556             Eigen::MatrixXd basis_vector = basis[i];
557             double inner1 = std::inner_product(vec.data(),
558 vec.data() + vec.size(), basis_vector.data(), 0.0);
559             double inner2 = std::inner_product(basis_vector.data(),
560 basis_vector.data() + basis_vector.size(),

```

```

553         basis_vector.data(), 0.0);
554         #pragma omp critical
555         projections += (inner1 / inner2) * basis_vector;
556     }
557
558     Eigen::VectorXd result = vec - projections;
559
560     if (delete_zero_rows)
561     {
562         bool is_all_zero = result.isZero(1e-3);
563         if (!is_all_zero)
564         {
565             basis.push_back(result);
566         }
567     }
568     else
569     {
570         basis.push_back(result);
571     }
572 }
573
574 Eigen::MatrixXd result(matrix.rows(), basis.size());
575
576 for (int i = 0; i < basis.size(); i++)
577 {
578     result.col(i) = basis[i];
579 }
580
581 return result;
582 }
583 #endif
584
585 // Computes Gram Schmidt orthogonalization
586 // @return Eigen::MatrixXd
587 // @param matrix input matrix
588 // @param normalize indicates whether to normalize output vectors
589 // @param delete_zero_rows indicates whether to delete zero rows
590 Eigen::MatrixXd gram_schmidt_sequential(const Eigen::MatrixXd
591 &matrix, bool delete_zero_rows)
592 {
593     std::vector<Eigen::VectorXd> basis;
594
595     for (const auto &vec : matrix.colwise())
596     {
597         Eigen::VectorXd projections =
598             Eigen::VectorXd::Zero(vec.size());
599
600         for (int i = 0; i < basis.size(); i++)
601         {
602             Eigen::MatrixXd basis_vector = basis[i];
603             double inner1 = std::inner_product(vec.data(),
604                 vec.data() + vec.size(), basis_vector.data(), 0.0);
605             double inner2 = std::inner_product(basis_vector.data(),
606                 basis_vector.data() + basis_vector.size(),
607                 basis_vector.data(), 0.0);
608
609             projections += (inner1 / inner2) * basis_vector;
610         }
611
612         Eigen::VectorXd result = vec - projections;
613
614         if (delete_zero_rows)
615         {
616             bool is_all_zero = result.isZero(1e-3);
617             if (!is_all_zero)
618             {

```

```

614         basis.push_back(result);
615     }
616 }
617 else
618 {
619     basis.push_back(result);
620 }
621 }
622
623 Eigen::MatrixXd result(matrix.rows(), basis.size());
624
625 for (int i = 0; i < basis.size(); i++)
626 {
627     result.col(i) = basis[i];
628 }
629
630 return result;
631 }
632 }

```

Приложение Г. Исходный код utils.cpp

```

1  #include "utils.hpp"
2  #include <iostream>
3  #include <random>
4  #include <functional>
5  #include <numeric>
6  #include <vector>
7  #include <stdexcept>
8  #include <string>
9  #include <chrono>
10 #include <algorithm>
11 #include <thread>
12 #include "algorithms.hpp"
13
14 namespace mp = boost::multiprecision;
15
16 namespace Utils
17 {
18     // Function for computing HNF of full row rank matrix
19     // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
20     // @param H HNF
21     // @param b column to be added
22     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)
23     {
24         if (H.rows() == 0)
25         {
26             return H;
27         }
28
29         Eigen::Vector<mp::cpp_int, -1> H_first_col = H.col(0);
30
31         mp::cpp_int a = H_first_col(0);
32         Eigen::Vector<mp::cpp_int, -1> h =
            H_first_col.tail(H_first_col.rows() - 1);
33         Eigen::Matrix<mp::cpp_int, -1, -1> H_stroke = H.block(1, 1,
            H.rows() - 1, H.cols() - 1);
34         mp::cpp_int b = b_column(0);
35         Eigen::Vector<mp::cpp_int, -1> b_stroke =
            b_column.tail(b_column.rows() - 1);
36
37         std::tuple<mp::cpp_int, mp::cpp_int, mp::cpp_int> gcd_result =
            gcd_extended(a, b);

```

```

38     mp::cpp_int g, x, y;
39     std::tie(g, x, y) = gcd_result;
40
41     Eigen::Matrix<mp::cpp_int, 2, 2> U;
42     U << x, -b / g, y, a / g;
43
44
45     Eigen::Matrix<mp::cpp_int, -1, 2> temp_matrix(H.rows(), 2);
46     temp_matrix.col(0) = H_first_col;
47     temp_matrix.col(1) = b_column;
48     Eigen::Matrix<mp::cpp_int, -1, 2> temp_result = temp_matrix * U;
49
50     Eigen::Vector<mp::cpp_int, -1> h_stroke =
51         temp_result.col(0).tail(temp_result.rows() - 1);
52     Eigen::Vector<mp::cpp_int, -1> b_double_stroke =
53         temp_result.col(1).tail(temp_result.rows() - 1);
54
55     b_double_stroke = reduce(b_double_stroke, H_stroke);
56
57     Eigen::Matrix<mp::cpp_int, -1, -1> H_double_stroke =
58         add_column(H_stroke, b_double_stroke);
59
60     h_stroke = reduce(h_stroke, H_double_stroke);
61
62     Eigen::Matrix<mp::cpp_int, -1, -1> result(H.rows(), H.cols());
63
64     result(0, 0) = g;
65     result.col(0).tail(result.cols() - 1) = h_stroke;
66     result.row(0).tail(result.rows() - 1).setZero();
67     result.block(1, 1, H_double_stroke.rows(),
68                 H_double_stroke.cols()) = H_double_stroke;
69
70     return result;
71 }
72
73 // Function for computing HNF, reduces elements of vector modulo
74 // diagonal elements of matrix
75 // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
76 // @param vector vector to be reduced
77 // @param matrix input matrix
78 Eigen::Vector<mp::cpp_int, -1> reduce(const
79     Eigen::Vector<mp::cpp_int, -1> &vector, const
80     Eigen::Matrix<mp::cpp_int, -1, -1> &matrix)
81 {
82     Eigen::Vector<mp::cpp_int, -1> result = vector;
83     for (int i = 0; i < result.rows(); i++)
84     {
85         Eigen::Vector<mp::cpp_int, -1> matrix_column = matrix.col(i);
86         mp::cpp_int t_vec_elem = result(i);
87         mp::cpp_int t_matrix_elem = matrix(i, i);
88
89         mp::cpp_int x;
90         if (t_vec_elem >= 0)
91         {
92             x = (t_vec_elem / t_matrix_elem);
93         }
94         else
95         {
96             x = (t_vec_elem - (t_matrix_elem - 1)) / t_matrix_elem;
97         }
98
99         result -= matrix_column * x;
100     }
101     return result;
102 }

```



```

149 {
150     if (m < 1 || n < 1)
151     {
152         throw std::invalid_argument("Number of rows or columns
153                                     should be greater than one");
154     }
155     if (highest - lowest < 1)
156     {
157         throw std::invalid_argument("highest parameter must be
158                                     greater than lowest parameter by at least one");
159     }
160     std::random_device rd;
161     std::mt19937 gen(rd());
162     std::uniform_int_distribution<int> dis (lowest, highest);
163     Eigen::Matrix<int, -1, -1> matrix = Eigen::Matrix<int, -1,
164                                     -1>::NullaryExpr(m, n, [&]()
165                                                         { return
166                                                         dis(gen);
167                                                         });
168     return matrix.cast<mp::cpp_int>();
169 }
170 #ifdef PARALLEL
171 // Returns matrix that consist of linearly independent columns of
172 // input matrix and othogonalized matrix
173 // @param matrix input matrix
174 // @return std::tuple<Eigen::Matrix<boost::multiprecision::cpp_int,
175 // -1, -1>, Eigen::Matrix<boost::multiprecision::cpp_rational, -1,
176 // -1>>
177 std::tuple<Eigen::Matrix<mp::cpp_int, -1, -1>,
178 Eigen::Matrix<mp::cpp_rational, -1, -1>>
179 get_linearly_independent_columns_by_gram_schmidt(const
180 Eigen::Matrix<mp::cpp_int, -1, -1> &matrix)
181 {
182     std::vector<Eigen::Vector<mp::cpp_rational, -1>> basis;
183     std::vector<int> indexes;
184
185     int counter = 0;
186     for (const Eigen::Vector<mp::cpp_rational, -1> &vec :
187         matrix.cast<mp::cpp_rational>().colwise())
188     {
189         Eigen::Vector<mp::cpp_rational, -1> projections =
190             Eigen::Vector<mp::cpp_rational, -1>::Zero(vec.size());
191
192         #pragma omp parallel for
193         for (int i = 0; i < basis.size(); i++)
194         {
195             Eigen::Vector<mp::cpp_rational, -1> basis_vector =
196                 basis[i];
197             mp::cpp_rational inner1 = std::inner_product(vec.data(),
198                 vec.data() + vec.size(), basis_vector.data(),
199                 mp::cpp_rational(0.0));
200             mp::cpp_rational inner2 =
201                 std::inner_product(basis_vector.data(),
202                     basis_vector.data() + basis_vector.size(),
203                     basis_vector.data(), mp::cpp_rational(0.0));
204
205             mp::cpp_rational coef = inner1 / inner2;
206
207             #pragma omp critical
208             projections += basis_vector * coef;
209         }
210         Eigen::Vector<mp::cpp_rational, -1> result = vec -

```

```

        projections;
197
198     bool is_all_zero = result.isZero(1e-3);
199     if (!is_all_zero)
200     {
201         basis.push_back(result);
202         indexes.push_back(counter);
203     }
204     counter++;
205 }
206
207 Eigen::Matrix<mp::cpp_int, -1, -1> result(matrix.rows(),
    indexes.size());
208 Eigen::Matrix<mp::cpp_rational, -1, -1>
    gram_schmidt(matrix.rows(), basis.size());
209
210 for (int i = 0; i < indexes.size(); i++)
211 {
212     result.col(i) = matrix.col(indexes[i]);
213     gram_schmidt.col(i) = basis[i];
214 }
215 return std::make_tuple(result, gram_schmidt);
216 }
217 #else
218 // Returns matrix that consist of linearly independent columns of
    input matrix and othogonalized matrix
219 // @param matrix input matrix
220 // @return std::tuple<Eigen::Matrix<boost::multiprecision::cpp_int,
    -1, -1>, Eigen::Matrix<boost::multiprecision::cpp_rational, -1,
    -1>>
221 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)
222 {
223     std::vector<Eigen::Vector<mp::cpp_rational, -1>> basis;
224     std::vector<int> indexes;
225
226     int counter = 0;
227     for (const Eigen::Vector<mp::cpp_rational, -1> &vec :
        matrix.cast<mp::cpp_rational>().colwise())
228     {
229         Eigen::Vector<mp::cpp_rational, -1> projections =
            Eigen::Vector<mp::cpp_rational, -1>::Zero(vec.size());
230
231         for (int i = 0; i < basis.size(); i++)
232         {
233             Eigen::Vector<mp::cpp_rational, -1> basis_vector =
                basis[i];
234             mp::cpp_rational inner1 = std::inner_product(vec.data(),
                vec.data() + vec.size(), basis_vector.data(),
                mp::cpp_rational(0.0));
235             mp::cpp_rational inner2 =
                std::inner_product(basis_vector.data(),
                basis_vector.data() + basis_vector.size(),
                basis_vector.data(), mp::cpp_rational(0.0));
236
237             mp::cpp_rational coef = inner1 / inner2;
238
239             projections += basis_vector * coef;
240         }
241
242         Eigen::Vector<mp::cpp_rational, -1> result = vec -
            projections;
243
244         bool is_all_zero = result.isZero(1e-3);
245         if (!is_all_zero)

```

```

246         {
247             basis.push_back(result);
248             indexes.push_back(counter);
249         }
250         counter++;
251     }
252
253     Eigen::Matrix<mp::cpp_int, -1, -1> result(matrix.rows(),
        indexes.size());
254     Eigen::Matrix<mp::cpp_rational, -1, -1>
        gram_schmidt(matrix.rows(), basis.size());
255
256     for (int i = 0; i < indexes.size(); i++)
257     {
258         result.col(i) = matrix.col(indexes[i]);
259         gram_schmidt.col(i) = basis[i];
260     }
261     return std::make_tuple(result, gram_schmidt);
262 }
263 #endif
264
265 #ifdef PARALLEL
266 // 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
267 // @param matrix input matrix
268 // @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>>
269 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)
270 {
271     std::vector<Eigen::Vector<mp::cpp_rational, -1>> basis;
272     std::vector<int> indicies;
273     std::vector<int> deleted_indicies;
274     Eigen::Matrix<mp::cpp_rational, -1, -1> T =
        Eigen::Matrix<mp::cpp_rational, -1,
        -1>::Identity(matrix.rows(), matrix.rows());
275
276     int counter = 0;
277     for (const Eigen::Vector<mp::cpp_rational, -1> &vec :
        matrix.cast<mp::cpp_rational>().rowwise())
278     {
279         Eigen::Vector<mp::cpp_rational, -1> projections =
            Eigen::Vector<mp::cpp_rational, -1>::Zero(vec.size());
280
281         #pragma omp parallel
282         for (int i = 0; i < basis.size(); i++)
283         {
284             Eigen::Vector<mp::cpp_rational, -1> basis_vector =
                basis[i];
285             mp::cpp_rational inner1 = std::inner_product(vec.data(),
                vec.data() + vec.size(), basis_vector.data(),
                mp::cpp_rational(0.0));
286             mp::cpp_rational inner2 =
                std::inner_product(basis_vector.data(),
                basis_vector.data() + basis_vector.size(),
                basis_vector.data(), mp::cpp_rational(0.0));
287
288             mp::cpp_rational u_ij = 0;
289             if (!inner1.is_zero())
290             {
291                 u_ij = inner1 / inner2;
292
293                 #pragma omp critical

```

```

294         {
295             projections += u_ij * basis_vector;
296             T(counter, i) = u_ij;
297         }
298     }
299 }
300
301 Eigen::Vector<mp::cpp_rational, -1> result = vec -
    projections;
302
303 bool is_all_zero = result.isZero(1e-3);
304 if (!is_all_zero)
305 {
306     indicies.push_back(counter);
307 }
308 else
309 {
310     deleted_indicies.push_back(counter);
311 }
312 basis.push_back(result);
313 counter++;
314 }
315
316 Eigen::Matrix<mp::cpp_int, -1, -1> result(indicies.size(),
    matrix.cols());
317 for (int i = 0; i < indicies.size(); i++)
318 {
319     result.row(i) = matrix.row(indicies[i]);
320 }
321 return std::make_tuple(result, indicies, deleted_indicies, T);
322 }
323 #else
324 // 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
325 // @param matrix input matrix
326 // @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>>
327 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)
328 {
329     std::vector<Eigen::Vector<mp::cpp_rational, -1>> basis;
330     std::vector<int> indicies;
331     std::vector<int> deleted_indicies;
332     Eigen::Matrix<mp::cpp_rational, -1, -1> T =
        Eigen::Matrix<mp::cpp_rational, -1,
        -1>::Identity(matrix.rows(), matrix.rows());
333
334     int counter = 0;
335     for (const Eigen::Vector<mp::cpp_rational, -1> &vec :
        matrix.cast<mp::cpp_rational>().rowwise())
336     {
337         Eigen::Vector<mp::cpp_rational, -1> projections =
            Eigen::Vector<mp::cpp_rational, -1>::Zero(vec.size());
338
339         for (int i = 0; i < basis.size(); i++)
340         {
341             Eigen::Vector<mp::cpp_rational, -1> basis_vector =
                basis[i];
342             mp::cpp_rational inner1 = std::inner_product(vec.data(),
                vec.data() + vec.size(), basis_vector.data(),
                mp::cpp_rational(0.0));
343             mp::cpp_rational inner2 =
                std::inner_product(basis_vector.data(),

```

```

        basis_vector.data() + basis_vector.size(),
        basis_vector.data(), mp::cpp_rational(0.0));
344
345     mp::cpp_rational u_ij = 0;
346     if (!inner1.is_zero())
347     {
348         u_ij = inner1 / inner2;
349
350         projections += u_ij * basis_vector;
351         T(counter, i) = u_ij;
352     }
353 }
354
355 Eigen::Vector<mp::cpp_rational, -1> result = vec -
    projections;
356
357 bool is_all_zero = result.isZero(1e-3);
358 if (!is_all_zero)
359 {
360     indicies.push_back(counter);
361 }
362 else
363 {
364     deleted_indicies.push_back(counter);
365 }
366 basis.push_back(result);
367 counter++;
368 }
369
370 Eigen::Matrix<mp::cpp_int, -1, -1> result(indicies.size(),
    matrix.cols());
371 for (int i = 0; i < indicies.size(); i++)
372 {
373     result.row(i) = matrix.row(indicies[i]);
374 }
375 return std::make_tuple(result, indicies, deleted_indicies, T);
376 }
377 #endif
378
379
380 // Extended GCD algorithm, returns tuple of g, x, y such that xa +
    yb = g
381 // @return std::tuple<boost::multiprecision::cpp_int,
    boost::multiprecision::cpp_int, boost::multiprecision::cpp_int>
382 // @param a first number
383 // @param b second number
384 std::tuple<mp::cpp_int, mp::cpp_int, mp::cpp_int>
    gcd_extended(mp::cpp_int a, mp::cpp_int b)
385 {
386     if (a == 0)
387     {
388         return std::make_tuple(b, 0, 1);
389     }
390     mp::cpp_int gcd, x1, y1;
391     std::tie(gcd, x1, y1) = gcd_extended(b % a, a);
392
393     mp::cpp_int x = y1 - (b / a) * x1;
394     mp::cpp_int y = x1;
395
396     return std::make_tuple(gcd, x, y);
397 }
398
399 #ifndef GMP
400 // Function for computing HNF of full row rank matrix
401 // @return Eigen::Matrix<boost::multiprecision::cpp_mpz, -1, -1>
402 // @param H HNF
403 // @param b column to be added

```

```

404 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)
405 {
406     if (H.rows() == 0)
407     {
408         return H;
409     }
410
411     Eigen::Vector<mp::mpz_int, -1> H_first_col = H.col(0);
412
413     mp::mpz_int a = H_first_col(0);
414     Eigen::Vector<mp::mpz_int, -1> h =
415         H_first_col.tail(H_first_col.rows() - 1);
416     Eigen::Matrix<mp::mpz_int, -1, -1> H_stroke = H.block(1, 1,
417         H.rows() - 1, H.cols() - 1);
418     mp::mpz_int b = b_column(0);
419     Eigen::Vector<mp::mpz_int, -1> b_stroke =
420         b_column.tail(b_column.rows() - 1);
421
422     std::tuple<mp::mpz_int, mp::mpz_int, mp::mpz_int> gcd_result =
423         gcd_extended_GMP(a, b);
424     mp::mpz_int g, x, y;
425     std::tie(g, x, y) = gcd_result;
426
427     Eigen::Matrix<mp::mpz_int, 2, 2> U;
428     U << x, -b / g, y, a / g;
429
430     Eigen::Matrix<mp::mpz_int, -1, 2> temp_matrix(H.rows(), 2);
431     temp_matrix.col(0) = H_first_col;
432     temp_matrix.col(1) = b_column;
433     Eigen::Matrix<mp::mpz_int, -1, 2> temp_result = temp_matrix * U;
434
435     Eigen::Vector<mp::mpz_int, -1> h_stroke =
436         temp_result.col(0).tail(temp_result.rows() - 1);
437     Eigen::Vector<mp::mpz_int, -1> b_double_stroke =
438         temp_result.col(1).tail(temp_result.rows() - 1);
439
440     b_double_stroke = reduce_GMP(b_double_stroke, H_stroke);
441
442     Eigen::Matrix<mp::mpz_int, -1, -1> H_double_stroke =
443         add_column_GMP(H_stroke, b_double_stroke);
444
445     h_stroke = reduce_GMP(h_stroke, H_double_stroke);
446
447     Eigen::Matrix<mp::mpz_int, -1, -1> result(H.rows(), H.cols());
448
449     result(0, 0) = g;
450     result.col(0).tail(result.cols() - 1) = h_stroke;
451     result.row(0).tail(result.rows() - 1).setZero();
452     result.block(1, 1, H_double_stroke.rows(),
453         H_double_stroke.cols()) = H_double_stroke;
454
455     return result;
456 }
457
458 // Function for computing HNF, reduces elements of vector modulo
459 // diagonal elements of matrix
460 // @return Eigen::Matrix<boost::multiprecision::cpp_mpf, -1, -1>
461 // @param vector vector to be reduced
462 // @param matrix input matrix
463 Eigen::Vector<mp::mpz_int, -1> reduce_GMP(const
464     Eigen::Vector<mp::mpz_int, -1> &vector, const
465     Eigen::Matrix<mp::mpz_int, -1, -1> &matrix)
466 {

```

```

458 Eigen::Vector<mp::mpz_int, -1> result = vector;
459 for (int i = 0; i < result.rows(); i++)
460 {
461     Eigen::Vector<mp::mpz_int, -1> matrix_column = matrix.col(i);
462     mp::mpz_int t_vec_elem = result(i);
463     mp::mpz_int t_matrix_elem = matrix(i, i);
464
465     mp::mpz_int x;
466     if (t_vec_elem >= 0)
467     {
468         x = (t_vec_elem / t_matrix_elem);
469     }
470     else
471     {
472         x = (t_vec_elem - (t_matrix_elem - 1)) / t_matrix_elem;
473     }
474
475     result -= matrix_column * x;
476 }
477 return result;
478 }
479
480
481 // Generates random matrix with full row rank (all rows are linearly
482 // independent)
483 // @return Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>
484 // @param m number of rows, must be greater than one and less than
485 // or equal to the parameter n
486 // @param n number of columns, must be greater than one and greater
487 // than or equal to the parameter m
488 // @param lowest lowest generated number, must be lower than lowest
489 // parameter by at least one
490 // @param highest highest generated number, must be greater than
491 // lowest parameter by at least one
492 Eigen::Matrix<mp::mpz_int, -1, -1>
493 generate_random_matrix_with_full_row_rank_GMP(const int m, const
494 int n, int lowest, int highest)
495 {
496     if (m > n)
497     {
498         throw std::invalid_argument("m must be less than or equal
499 n");
500     }
501     if (m < 1 || n < 1)
502     {
503         throw std::invalid_argument("Number of rows or columns
504 should be greater than one");
505     }
506     if (highest - lowest < 1)
507     {
508         throw std::invalid_argument("highest parameter must be
509 greater than lowest parameter by at least one");
510     }
511     std::random_device rd;
512     std::mt19937 gen(rd());
513     std::uniform_int_distribution<int> dis (lowest, highest);
514
515     Eigen::Matrix<int, -1, -1> matrix = Eigen::Matrix<int, -1,
516 -1>::NullaryExpr(m, n, [&]()
517 { return
518     dis(gen);
519 });
520
521 Eigen::FullPivLU<Eigen::MatrixXd>
522 lu_decomp(matrix.cast<double>());
523 auto rank = lu_decomp.rank();
524

```



```

511     while (rank != m)
512     {
513         matrix = Eigen::Matrix<int, -1, -1>::NullaryExpr(m, n, [&]()
514             { return dis(gen); });
515
516         lu_decomp.compute(matrix.cast<double>());
517         rank = lu_decomp.rank();
518     }
519
520     return matrix.cast<mp::mpz_int>();
521 }
522
523
524 // Generates random matrix
525 // @return Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>
526 // @param m number of rows, must be greater than one
527 // @param n number of columns, must be greater than one
528 // @param lowest lowest generated number, must be lower than lowest
529 //         parameter by at least one
530 // @param highest highest generated number, must be greater than
531 //         lowest parameter by at least one
532 Eigen::Matrix<mp::mpz_int, -1, -1> generate_random_matrix_GMP(const
533     int m, const int n, int lowest, int highest)
534 {
535     if (m < 1 || n < 1)
536     {
537         throw std::invalid_argument("Number of rows or columns
538             should be greater than one");
539     }
540     if (highest - lowest < 1)
541     {
542         throw std::invalid_argument("highest parameter must be
543             greater than lowest parameter by at least one");
544     }
545
546     std::random_device rd;
547     std::mt19937 gen(rd());
548     std::uniform_int_distribution<int> dis (lowest, highest);
549
550     Eigen::Matrix<int, -1, -1> matrix = Eigen::Matrix<int, -1,
551         -1>::NullaryExpr(m, n, [&]()
552             { return
553                 dis(gen);
554             });
555
556     return matrix.cast<mp::mpz_int>();
557 }
558
559 #ifdef PARALLEL
560 // Returns matrix that consist of linearly independent columns of
561 // input matrix and othogonalized matrix
562 // @param matrix input matrix
563 // @return std::tuple<Eigen::Matrix<boost::multiprecision::mpz_int,
564 //         -1, -1>, Eigen::Matrix<boost::multiprecision::mpq_rational, -1,
565 //         -1>>
566 std::tuple<Eigen::Matrix<mp::mpz_int, -1, -1>,
567     Eigen::Matrix<mp::mpq_rational, -1, -1>>
568 get_linearly_independent_columns_by_gram_schmidt_GMP(const
569     Eigen::Matrix<mp::mpz_int, -1, -1> &matrix)
570 {
571     std::vector<Eigen::Vector<mp::mpq_rational, -1>> basis;
572     std::vector<int> indexes;
573
574     int counter = 0;
575     for (const Eigen::Vector<mp::mpq_rational, -1> &vec :
576         matrix.cast<mp::mpq_rational>().colwise())
577     {

```

```

563 Eigen::Vector<mp::mpq_rational, -1> projections =
    Eigen::Vector<mp::mpq_rational, -1>::Zero(vec.size());
564
565 #pragma omp parallel for
566 for (int i = 0; i < basis.size(); i++)
567 {
568     Eigen::Vector<mp::mpq_rational, -1> basis_vector =
        basis[i];
569     mp::mpq_rational inner1 = std::inner_product(vec.data(),
        vec.data() + vec.size(), basis_vector.data(),
        mp::mpq_rational(0.0));
570     mp::mpq_rational inner2 =
        std::inner_product(basis_vector.data(),
        basis_vector.data() + basis_vector.size(),
        basis_vector.data(), mp::mpq_rational(0.0));
571
572     mp::mpq_rational coef = inner1 / inner2;
573     #pragma omp critical
574     projections += basis_vector * coef;
575 }
576
577 Eigen::Vector<mp::mpq_rational, -1> result = vec -
    projections;
578
579 bool is_all_zero = result.isZero(1e-3);
580 if (!is_all_zero)
581 {
582     basis.push_back(result);
583     indexes.push_back(counter);
584 }
585 counter++;
586 }
587
588 Eigen::Matrix<mp::mpz_int, -1, -1> result(matrix.rows(),
    indexes.size());
589 Eigen::Matrix<mp::mpq_rational, -1, -1>
    gram_schmidt(matrix.rows(), basis.size());
590
591 for (int i = 0; i < indexes.size(); i++)
592 {
593     result.col(i) = matrix.col(indexes[i]);
594     gram_schmidt.col(i) = basis[i];
595 }
596 return std::make_tuple(result, gram_schmidt);
597 }
598 #else
599 // Returns matrix that consist of linearly independent columns of
    input matrix and othogonalized matrix
600 // @param matrix input matrix
601 // @return std::tuple<Eigen::Matrix<boost::multiprecision::mpz_int,
    -1, -1>, Eigen::Matrix<boost::multiprecision::mpq_rational, -1,
    -1>>
602 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)
603 {
604     std::vector<Eigen::Vector<mp::mpq_rational, -1>> basis;
605     std::vector<int> indexes;
606
607     int counter = 0;
608     for (const Eigen::Vector<mp::mpq_rational, -1> &vec :
        matrix.cast<mp::mpq_rational>().colwise())
609     {
610         Eigen::Vector<mp::mpq_rational, -1> projections =
            Eigen::Vector<mp::mpq_rational, -1>::Zero(vec.size());
611

```

```

612     for (int i = 0; i < basis.size(); i++)
613     {
614         Eigen::Vector<mp::mpq_rational, -1> basis_vector =
            basis[i];
615         mp::mpq_rational inner1 = std::inner_product(vec.data(),
            vec.data() + vec.size(), basis_vector.data(),
            mp::mpq_rational(0.0));
616         mp::mpq_rational inner2 =
            std::inner_product(basis_vector.data(),
            basis_vector.data() + basis_vector.size(),
            basis_vector.data(), mp::mpq_rational(0.0));
617
618         mp::mpq_rational coef = inner1 / inner2;
619         projections += basis_vector * coef;
620     }
621
622     Eigen::Vector<mp::mpq_rational, -1> result = vec -
        projections;
623
624     bool is_all_zero = result.isZero(1e-3);
625     if (!is_all_zero)
626     {
627         basis.push_back(result);
628         indexes.push_back(counter);
629     }
630     counter++;
631 }
632
633 Eigen::Matrix<mp::mpz_int, -1, -1> result(matrix.rows(),
    indexes.size());
634 Eigen::Matrix<mp::mpq_rational, -1, -1>
    gram_schmidt(matrix.rows(), basis.size());
635
636 for (int i = 0; i < indexes.size(); i++)
637 {
638     result.col(i) = matrix.col(indexes[i]);
639     gram_schmidt.col(i) = basis[i];
640 }
641 return std::make_tuple(result, gram_schmidt);
642 }
643 #endif
644
645 #ifdef PARALLEL
646 // 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
647 // @param matrix input matrix
648 // @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>>
649 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)
650 {
651     std::vector<Eigen::Vector<mp::mpq_rational, -1>> basis;
652     std::vector<int> indicies;
653     std::vector<int> deleted_indicies;
654     Eigen::Matrix<mp::mpq_rational, -1, -1> T =
        Eigen::Matrix<mp::mpq_rational, -1,
        -1>::Identity(matrix.rows(), matrix.rows());
655
656     int counter = 0;
657     for (const Eigen::Vector<mp::mpq_rational, -1> &vec :
        matrix.cast<mp::mpq_rational>().rowwise())
658     {
659         Eigen::Vector<mp::mpq_rational, -1> projections =

```

```

        Eigen::Vector<mp::mpq_rational, -1>::Zero(vec.size());
660
661 #pragma omp parallel for
662 for (int i = 0; i < basis.size(); i++)
663 {
664     Eigen::Vector<mp::mpq_rational, -1> basis_vector =
        basis[i];
665     mp::mpq_rational inner1 = std::inner_product(vec.data(),
        vec.data() + vec.size(), basis_vector.data(),
        mp::mpq_rational(0.0));
666     mp::mpq_rational inner2 =
        std::inner_product(basis_vector.data(),
        basis_vector.data() + basis_vector.size(),
        basis_vector.data(), mp::mpq_rational(0.0));

667     mp::mpq_rational u_ij = 0;
668     if (!inner1.is_zero())
669     {
670         u_ij = inner1 / inner2;
671         #pragma omp critical
672         {
673             projections += u_ij * basis_vector;
674             T(counter, i) = u_ij;
675         }
676     }
677 }
678
679 Eigen::Vector<mp::mpq_rational, -1> result = vec -
    projections;
680
681 bool is_all_zero = result.isZero(1e-3);
682 if (!is_all_zero)
683 {
684     indicies.push_back(counter);
685 }
686 else
687 {
688     deleted_indicies.push_back(counter);
689 }
690 basis.push_back(result);
691 counter++;
692 }
693
694 Eigen::Matrix<mp::mpz_int, -1, -1> result(indicies.size(),
    matrix.cols());
695 for (int i = 0; i < indicies.size(); i++)
696 {
697     result.row(i) = matrix.row(indicies[i]);
698 }
699 return std::make_tuple(result, indicies, deleted_indicies, T);
700 }
701 #else
702 // Returns matrix that consist of linearly independent rows of input
703 // matrix, indicies of that rows in input matrix, indices of deleted
704 // rows and matrix T, that consists of Gram Schmidt coefficients
705 // @param matrix input matrix
706 // @return std::tuple<Eigen::Matrix<boost::multiprecision::mpz_int,
707 // -1, -1>, std::vector<int>, std::vector<int>,
708 // Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>>
709 std::tuple<Eigen::Matrix<mp::mpz_int, -1, -1>, std::vector<int>,
710 std::vector<int>, Eigen::Matrix<mp::mpq_rational, -1, -1>>
711 get_linearly_independent_rows_by_gram_schmidt_GMP(const
712 Eigen::Matrix<mp::mpz_int, -1, -1> &matrix)
713 {
714     std::vector<Eigen::Vector<mp::mpq_rational, -1>> basis;
715     std::vector<int> indicies;
716     std::vector<int> deleted_indicies;

```

```

711 Eigen::Matrix<mp::mpq_rational, -1, -1> T =
      Eigen::Matrix<mp::mpq_rational, -1,
      -1>::Identity(matrix.rows(), matrix.rows());
712
713 int counter = 0;
714 for (const Eigen::Vector<mp::mpq_rational, -1> &vec :
      matrix.cast<mp::mpq_rational>().rowwise())
715 {
716     Eigen::Vector<mp::mpq_rational, -1> projections =
      Eigen::Vector<mp::mpq_rational, -1>::Zero(vec.size());
717
718     for (int i = 0; i < basis.size(); i++)
719     {
720         Eigen::Vector<mp::mpq_rational, -1> basis_vector =
      basis[i];
721         mp::mpq_rational inner1 = std::inner_product(vec.data(),
      vec.data() + vec.size(), basis_vector.data(),
      mp::mpq_rational(0.0));
722         mp::mpq_rational inner2 =
      std::inner_product(basis_vector.data(),
      basis_vector.data() + basis_vector.size(),
      basis_vector.data(), mp::mpq_rational(0.0));
723
724         mp::mpq_rational u_ij = 0;
725         if (!inner1.is_zero())
726         {
727             u_ij = inner1 / inner2;
728             {
729                 projections += u_ij * basis_vector;
730                 T(counter, i) = u_ij;
731             }
732         }
733     }
734
735     Eigen::Vector<mp::mpq_rational, -1> result = vec -
      projections;
736
737     bool is_all_zero = result.isZero(1e-3);
738     if (!is_all_zero)
739     {
740         indicies.push_back(counter);
741     }
742     else
743     {
744         deleted_indicies.push_back(counter);
745     }
746     basis.push_back(result);
747     counter++;
748 }
749
750 Eigen::Matrix<mp::mpz_int, -1, -1> result(indicies.size(),
      matrix.cols());
751 for (int i = 0; i < indicies.size(); i++)
752 {
753     result.row(i) = matrix.row(indicies[i]);
754 }
755 return std::make_tuple(result, indicies, deleted_indicies, T);
756 }
757 #endif
758
759
760 // Extended GCD algorithm, returns tuple of g, x, y such that xa +
      yb = g
761 // @return std::tuple<boost::multiprecision::mpz_int,
      boost::multiprecision::mpz_int, boost::multiprecision::mpz_int>
762 // @param a first number
763 // @param b second number

```

```

764 std::tuple<mp::mpz_int, mp::mpz_int, mp::mpz_int>
    gcd_extended_GMP(mp::mpz_int a, mp::mpz_int b)
765 {
766     if (a == 0)
767     {
768         return std::make_tuple(b, 0, 1);
769     }
770     mp::mpz_int gcd, x1, y1;
771     std::tie(gcd, x1, y1) = gcd_extended_GMP(b % a, a);
772
773     mp::mpz_int x = y1 - (b / a) * x1;
774     mp::mpz_int y = x1;
775
776     return std::make_tuple(gcd, x, y);
777 }
778 #endif
779
780
781 // Generates random matrix with full column rank
782 // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
783 // @param m number of rows, must be greater than one and greater
784 //         than or equal to the parameter n
785 // @param n number of columns, must be greater than one and lower
786 //         than or equal to the parameter m
787 // @param lowest lowest generated number, must be lower than lowest
788 //         parameter by at least one
789 // @param highest highest generated number, must be greater than
790 //         lowest parameter by at least one
791 Eigen::MatrixXd generate_random_matrix_with_full_column_rank(const
792     int m, const int n, int lowest, int highest)
793 {
794     if (m < n)
795     {
796         throw std::invalid_argument("m must be less than or equal
797             n");
798     }
799     if (m < 1 || n < 1)
800     {
801         throw std::invalid_argument("Number of rows or columns
802             should be greater than one");
803     }
804     if (highest - lowest < 1)
805     {
806         throw std::invalid_argument("highest parameter must be
807             greater than lowest parameter by at least one");
808     }
809     std::random_device rd;
810     std::mt19937 gen(rd());
811     std::uniform_int_distribution<int> dis (lowest, highest);
812     Eigen::MatrixXd matrix = Eigen::MatrixXd::NullaryExpr(m, n, [&]()
813         { return
814             dis(gen);
815         });
816     Eigen::FullPivLU<Eigen::MatrixXd> lu_decomp(matrix);
817     auto rank = lu_decomp.rank();
818     while (rank != n)
819     {
820         matrix = Eigen::MatrixXd::NullaryExpr(m, n, [&]()
821             { return dis(gen); });
822         lu_decomp.compute(matrix);
823         rank = lu_decomp.rank();
824     }

```

```

820         return matrix;
821     }
822
823
824     // Generates random vector
825     // @return Eigen::VectorXd
826     // @param m number of rows, must be greater than one
827     // @param lowest lowest generated number, must be lower than lowest
828     // parameter by at least one
829     // @param highest highest generated number, must be greater than
830     // lowest parameter by at least one
831     Eigen::VectorXd generate_random_vector(const int m, double lowest,
832     double highest)
833     {
834         if (m < 1)
835         {
836             throw std::invalid_argument("Number of rows or columns
837             should be greater than one");
838         }
839         if (highest - lowest < 1)
840         {
841             throw std::invalid_argument("highest parameter must be
842             greater than lowest parameter by at least one");
843         }
844         std::mt19937 gen(std::random_device{}());
845         std::uniform_real_distribution<double> dis(lowest, highest);
846         Eigen::VectorXd vector = Eigen::VectorXd::NullaryExpr(m, [&]()
847         { return
848             dis(gen);
849         });
850
851         return vector;
852     }
853
854 #ifdef PARALLEL
855 // Computes component of a vector perpendicular to a matrix using
856 // equations from Gram Schmidt computing
857 // @return Eigen::VectorXd
858 // @param matrix input matrix
859 // @param vector input vector
860 Eigen::VectorXd projection(const Eigen::MatrixX<double> &matrix, const
861 Eigen::VectorXd &vector)
862 {
863     Eigen::MatrixX<double> t_matrix(matrix.rows(), matrix.cols() + 1);
864     t_matrix << matrix, vector;
865     std::vector<Eigen::VectorXd> basis;
866
867     for (const Eigen::VectorXd &vec : t_matrix.colwise())
868     {
869         Eigen::VectorXd projections =
870             Eigen::VectorXd::Zero(vec.size());
871
872         #pragma omp parallel for
873         for (int i = 0; i < basis.size(); i++)
874         {
875             Eigen::VectorXd basis_vector = basis[i];
876             double inner1 = std::inner_product(vec.data(),
877             vec.data() + vec.size(), basis_vector.data(), 0.0);
878             double inner2 = std::inner_product(basis_vector.data(),
879             basis_vector.data() + basis_vector.size(),
880             basis_vector.data(), 0.0);
881
882             double coef = inner1 / inner2;
883             #pragma omp critical
884             projections += basis_vector * coef;
885         }
886     }
887 }
888 #endif

```



```

874         }
875
876         Eigen::VectorXd t_result = vec - projections;
877
878         basis.push_back(t_result);
879     }
880
881     Eigen::VectorXd result = basis[basis.size() - 1];
882
883     return result;
884 }
885 #else
886 // Computes component of a vector perpendicular to a matrix using
887 // equations from Gram Schmidt computing
888 // @return Eigen::VectorXd
889 // @param matrix input matrix
890 // @param vector input vector
891 Eigen::VectorXd projection(const Eigen::MatrixXd &matrix, const
892     Eigen::VectorXd &vector)
893 {
894     Eigen::MatrixXd t_matrix(matrix.rows(), matrix.cols() + 1);
895     t_matrix << matrix, vector;
896     std::vector<Eigen::VectorXd> basis;
897
898     for (const Eigen::VectorXd &vec : t_matrix.colwise())
899     {
900         Eigen::VectorXd projections =
901             Eigen::VectorXd::Zero(vec.size());
902
903         for (int i = 0; i < basis.size(); i++)
904         {
905             Eigen::VectorXd basis_vector = basis[i];
906             double inner1 = std::inner_product(vec.data(),
907                 vec.data() + vec.size(), basis_vector.data(), 0.0);
908             double inner2 = std::inner_product(basis_vector.data(),
909                 basis_vector.data() + basis_vector.size(),
910                 basis_vector.data(), 0.0);
911
912             double coef = inner1 / inner2;
913             projections += basis_vector * coef;
914         }
915
916         Eigen::VectorXd t_result = vec - projections;
917
918         basis.push_back(t_result);
919     }
920
921     Eigen::VectorXd result = basis[basis.size() - 1];
922
923     return result;
924 }
925 #endif
926
927 // Finds vector that is closest to other vectors in matrix
928 // @return Eigen::VectorXd
929 // @param matrix input matrix
930 // @param vector input vector
931 Eigen::VectorXd closest_vector(const std::vector<Eigen::VectorXd>
932     &matrix, const Eigen::VectorXd &vector)
933 {
934     Eigen::VectorXd closest = matrix[0];
935     for (const auto &v : matrix)
936     {
937         if ((vector - v).norm() <= (vector - closest).norm())
938         {
939             closest = v;
940         }
941     }
942 }

```



```

934         }
935     }
936
937     return closest;
938 }
939 }

```

Приложение Д. Исходный CMakeLists.txt

```

1  cmake_minimum_required(VERSION 3.2)
2  project(LatticeAlgorithms)
3
4  option(BUILD_DOCS "" OFF)
5  option(BUILD_PARALLEL "" OFF)
6  option(BUILD_GMP "" OFF)
7
8  file(GLOB SRC
9       "src/utils.cpp"
10      "src/algorithms.cpp"
11  )
12
13  add_subdirectory(3rdparty/boost_config)
14  add_subdirectory(3rdparty/boost_multiprecision)
15
16  find_package(OpenMP REQUIRED)
17
18  add_library(${PROJECT_NAME} ${SRC})
19
20  target_include_directories(${PROJECT_NAME} PUBLIC include)
21
22  if (BUILD_PARALLEL)
23      target_compile_definitions(${PROJECT_NAME} PUBLIC PARALLEL)
24  endif(BUILD_PARALLEL)
25
26  if (BUILD_GMP)
27      target_compile_definitions(${PROJECT_NAME} PUBLIC GMP)
28  endif(BUILD_GMP)
29
30  target_link_libraries(${PROJECT_NAME} OpenMP::OpenMP_CXX)
31  target_link_libraries(${PROJECT_NAME} gmp libgmp)
32  target_link_libraries(${PROJECT_NAME} Boost::config
33                        Boost::multiprecision)
34
35  if (BUILD_DOCS)
36      add_subdirectory(tex)
37  endif(BUILD_DOCS)

```