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

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, приблизительного решения ПБВ за полиномиальное время и точного решения ПБВ за суперполиномиальное время. Необходимо будет показать, как можно использовать данные алгоритмы на практике. В качестве теоретической базы, откуда взяты основы и описание алгоритмов для программирования, была использована серия лекций по решеткам и решеточным алгоритмам.

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 стандратной.

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

Используемые классы:

Matrix<typename Scalar, int RowsAtCompileTime, int ColsAtCompileTime> — шаблонный класс матриц. Первый параметр шаблона отвечает за тип элементов матрицы, второй параметр за количество строк, третий за количество столбцов. Если количество строк/столбцов неизвестно на этапе компиляции, а будет найдено в процессе выполнения программы, то необходимо ставить количество строк/столбцов равным Eigen::Dynamic, либо -1. Имеет псевдонимы для различных встроенных типов (int, double, float) и размеров матриц (2, 3, 4), например Matrix3d — матрица элементов double размера 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: $\mathbf{A} = \begin{bmatrix} \mathbf{a}_1 & \dots & \mathbf{a}_n \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1^{\mathbf{T}} \\ \vdots \\ \mathbf{a}_m^{\mathbf{T}} \end{bmatrix}$.

Линейная зависимость/независимость — пусть имеется несколько векторов одной размерности $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k$ и столько же чисел $\alpha_1, \alpha_2, \cdots, \alpha_k$. Вектор $\mathbf{y} = \alpha_1\mathbf{x}_1 + \alpha_2\mathbf{x}_2 + \ldots + \alpha_k\mathbf{x}_k$ 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

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

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 $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{n}_n]$ — базис решетки, $\mathbf{B}^* = [\mathbf{b}_1^*, \ldots, \mathbf{n}_n^*]$ — 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 $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{R}^{m \times n}$ является Эрмитовой нормальной формой, если

- Существует $1 \leq i_1 < \ldots < i_h \leq m$ такое, что $b_{i,j} \neq 0 \Rightarrow (j < h) \wedge (i \geq i_j)$ (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 $\mathbf{Bx}$ ($\mathbf{x} \in \mathbb{Z}^n$) такую, что расстояние $||\mathbf{t} - \mathbf{Bx}||$ 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 $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n]$, $\mathbf{B} \in \mathbb{R}^{m \times n}$. Этот набор необязательно ортогонален или даже линейно независим. Ортогонализацией этого набора векторов является набор векторов $\mathbf{B}^* = [\mathbf{b}_1^*, \cdots, \mathbf{b}_n^*] \in \mathbb{R}^{m \times n}$, где

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

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 $\sum_{i<j} \mu_{i,j} \mathbf{b}_j^*$. Конечный алгоритм выглядит следующим образом:

Input: $\mathbf{B}$
Output: $\mathbf{GS}$
  $\mathbf{GS} \leftarrow \begin{bmatrix} & \end{bmatrix}$
  $n \leftarrow \mathbf{B}.columns$
  for $i \leftarrow 0$ to n **do**

$\mathbf{b}_i \leftarrow \mathbf{B}.column(i)$
projections $\leftarrow$ **0**
for $j \leftarrow 0$ to i **do**
$\mathbf{b}_j \leftarrow \mathbf{GS}.column(j)$
projections $\leftarrow$ **projections** $+ \mathbf{b}_j \cdot \frac{\langle \mathbf{b}_i, \mathbf{b}_j \rangle}{\langle \mathbf{b}_j, \mathbf{b}_j \rangle}$
end for
$\mathbf{GS}.push\_back(\mathbf{b}_i - \textbf{projections})$
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 **H** подрешетки от $\mathcal{L}(\mathbf{B})$, и затем обновлять **H**, включая столбцы **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 **B** может быть вычислено следующим образом:

1. Применить алгоритм Грама-Шмидта к столбцам **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 $i = 1, \ldots, n$ 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-Шмитда: 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 форумле: $d = \sqrt{\prod_i \|\mathbf{b}_i^*\|^2}$ — 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 $\mathbf{H} = \begin{bmatrix} a & \mathbf{0}^T \\ \mathbf{h} & \mathbf{H}' \end{bmatrix}$ 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 $\begin{bmatrix} \mathrm{g} & \mathbf{0}^{\mathrm{T}} \\ \mathbf{h}' & \mathbf{H}'' \end{bmatrix}$.

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 $\mathbf{r}_1, \ldots, \mathbf{r}_m$ из $\mathbf{B}$, и пусть $\mathrm{K} = \{\mathrm{k}_1, \ldots, \mathrm{k}_l\}$ $(\mathrm{k}_1 < \ldots < \mathrm{k}_l)$ – это множество индексов, такое, что $\mathbf{r}^*_{\mathrm{k}_i} \neq \mathbf{0}$. Определим операцию проецирования $\prod_{\mathrm{K}} : \mathbb{R}^m \to \mathbb{R}^l$ при $[\prod_{\mathrm{K}}(\mathbf{x})]_i = \mathrm{x}_{\mathrm{k}_i}$. Заметим, что строки $\mathbf{r}_{\mathrm{k}}$ $(\mathrm{k} \in \mathrm{K})$ линейно независимы и любая строка $\mathbf{r}_i$ $(i \in \mathrm{K})$ может быть выражена как линейная комбинация предыдущих строк $\mathbf{r}_j$ $(\{j \in \mathrm{K} : j < i\})$. Следовательно, операция проецирования $\prod_{\mathrm{K}}$ 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 $\mathbf{B}' = \prod_{\mathrm{K}}(\mathbf{B})$, 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, $\prod_{\mathrm{K}}^{-1}$ 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 напишем 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 коээфициенты 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 $\mathbf{B} = \begin{bmatrix} \mathbf{b}_1^{\mathbf{T}} \\ \vdots \\ \mathbf{b}_m^{\mathbf{T}} \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 1 & 4 \end{bmatrix}$. 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 ортогонализацию строк за $\mathbf{B}^* = \begin{bmatrix} \mathbf{b}_1^{\mathbf{T}*} \\ \vdots \\ \mathbf{b}_m^{\mathbf{T}*} \end{bmatrix}$ и найдем их:

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

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 $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] = \begin{bmatrix} 2 & 4 \\ 1 & 4 \end{bmatrix}$. Требуется найти m линейно независимых строк, используя ортогонализацию Грама-Шмидта. Используем этот алгоритм на строки $\mathbf{B}$:

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

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 $i = 1, \ldots, n$ 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 $\mathbf{U}$: $\begin{bmatrix} \text{a} & \text{b} \\ \mathbf{h} & \mathbf{b}' \end{bmatrix} \mathbf{U} = \begin{bmatrix} \text{g} & 0 \\ \mathbf{h}' & \mathbf{b}'' \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix}$, $\mathbf{h}' = \begin{bmatrix} 1 \end{bmatrix}$, $\mathbf{b}'' = \begin{bmatrix} 2 \end{bmatrix}$.

   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 $\mathbf{H}'' = \begin{bmatrix} 2 \end{bmatrix}$:

   - $\mathbf{H} = \begin{bmatrix} 4 \end{bmatrix}$, 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: $\begin{bmatrix} \text{a} & \text{b} \\ \mathbf{h} & \mathbf{b}' \end{bmatrix} \mathbf{U} = \begin{bmatrix} \text{g} & 0 \\ \mathbf{h}' & \mathbf{b}'' \end{bmatrix} = \begin{bmatrix} 2 & 0 \end{bmatrix}$, $\mathbf{h}' = \begin{bmatrix} \end{bmatrix}$, $\mathbf{b}'' = \begin{bmatrix} \end{bmatrix}$.

     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 $\mathcal{L}(\mathbf{H}'')$: $\mathbf{h}' = \begin{bmatrix} \end{bmatrix}$.

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

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

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

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

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 $\mathbf{H}'' = \begin{bmatrix} 2 \end{bmatrix}$:

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

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

  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 $\mathcal{L}(\mathbf{H}'')$: $\mathbf{h}' = \begin{bmatrix} \end{bmatrix}$.

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

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

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

ЭНФ(**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 **B**, необходимо вычислить базис для $\mathcal{L}(\mathbf{B})$. Проблема решается за полиномиальное время путем вычисления ЭНФ($\mathbf{B}$):

$\mathbf{B} = \begin{bmatrix} 2 & 2 \\ 1 & 0 \end{bmatrix}$, 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 однинаковую решетку $\mathcal{L}(\mathbf{B}) = \mathcal{L}(\mathbf{B}')$. Проблема решается путем вычисления ЭНФ($\mathbf{B}$) и ЭНФ($\mathbf{B}'$) и сравнения их равенства:

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

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

$\mathbf{B} = \begin{bmatrix} 2 & 2 \\ 1 & 0 \end{bmatrix}$, $\mathbf{B}' = \begin{bmatrix} 0 & 1 \\ 2 & 2 \end{bmatrix}$, $[\mathbf{B}|\mathbf{B}'] = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 0 & 2 & 2 \end{bmatrix}$, 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:

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

$\mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $\mathbf{B}' = \begin{bmatrix} 2 & 2 \\ 1 & 0 \end{bmatrix}$, $[\mathbf{B}|\mathbf{B}'] = \begin{bmatrix} 1 & 0 & 2 & 2 \\ 0 & 1 & 1 & 0 \end{bmatrix}$, ЭНФ($[\mathbf{B}|\mathbf{B}']$) $= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$, 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 $\mathbf{v}_1, \ldots, \mathbf{v}_n$, тогда следует сначала вычислить $\mathbf{H} =$ ЭНФ($\mathbf{B}$), и затем проверять, равно ли $\mathbf{H}$ ЭНФ($[\mathbf{H}|\mathbf{v}_i]$) для каждого вектора:

$\mathbf{B} = \begin{bmatrix} 2 & 2 \\ 1 & 0 \end{bmatrix}$, $\mathbf{v} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$, $[\mathbf{B}|\mathbf{B}'] = \begin{bmatrix} 2 & 2 & 2 \\ 1 & 0 & 0 \end{bmatrix}$, ЭНФ($[\mathbf{B}|\mathbf{v}]$) $= \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$, ЭНФ($\mathbf{B}$) $= \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$ — вектор $\mathbf{v} \subseteq \mathcal{L}(\mathbf{B})$.

$\mathbf{B} = \begin{bmatrix} 2 & 2 \\ 1 & 0 \end{bmatrix}$, $\mathbf{v} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $[\mathbf{B}|\mathbf{B}'] = \begin{bmatrix} 2 & 2 & 1 \\ 1 & 0 & 0 \end{bmatrix}$, ЭНФ($[\mathbf{B}|\mathbf{v}]$) $= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$, 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 $\mathbf{Bx}$ ($\mathbf{x} \in \mathbb{Z}^n$) такую, что $||\mathbf{t} - \mathbf{Bx}||$ (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 $f(\mathbf{x}) = ||\mathbf{t} - \mathbf{Bx}||$.

Пусть $\mathbf{B} = [\mathbf{B}', \mathrm{b}]$ 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 $\mathbf{x}$ 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:

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

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 выглядит 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

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

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

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

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

$c \leftarrow \lfloor x \rceil$

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**

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

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

$result \leftarrow \mathbf{0}$

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

$index \leftarrow n - i - 1$

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

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

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

$c \leftarrow \lfloor x \rceil$

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

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

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

   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

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

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

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

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

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

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

где 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 $||(\mathbf{t} - x\mathbf{b}) \perp \mathbf{B}|| \le ||\mathbf{t} - \mathbf{v}||$.

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 $|(\mathbf{t} - x\mathbf{b}) \perp \mathbf{B}| \le |\mathbf{t} - \mathbf{v}|$. 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 пареллельный 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:

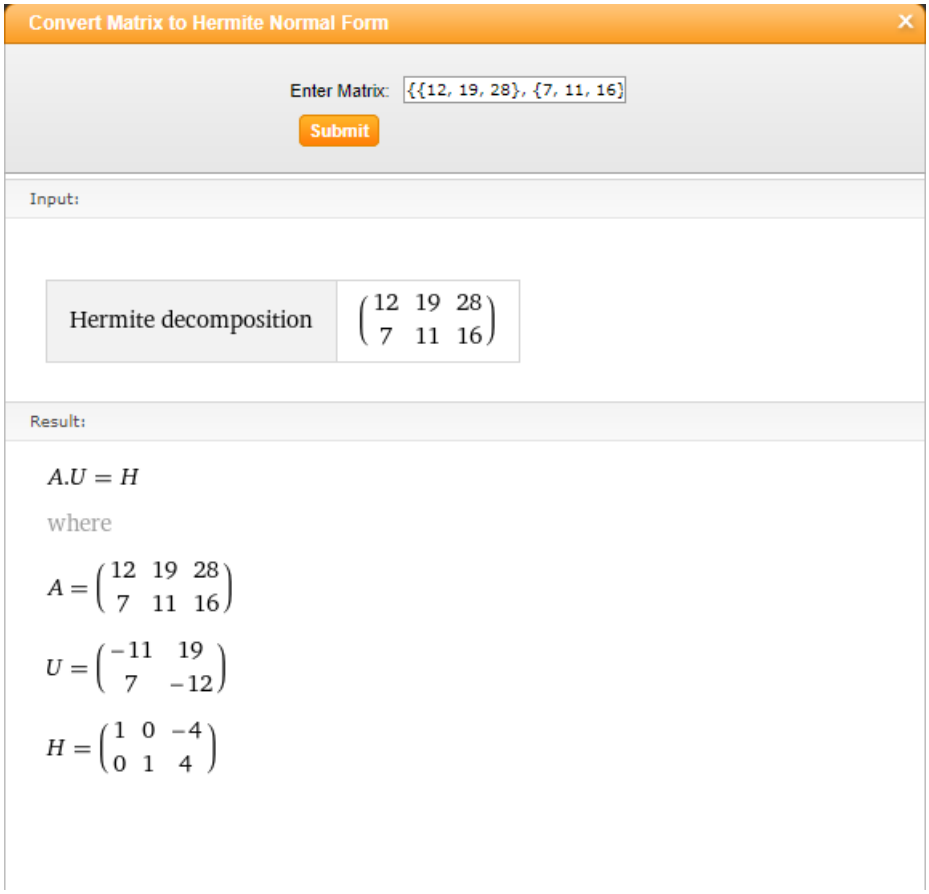

Рис. 1: Нахождение ЭНФ с помощью WolframAlpha.

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 бóльших размерах матриц, чем при использовании WolframAlpha, и увидеть, как сильно растут числа на больших матрицах.

Пример работы:

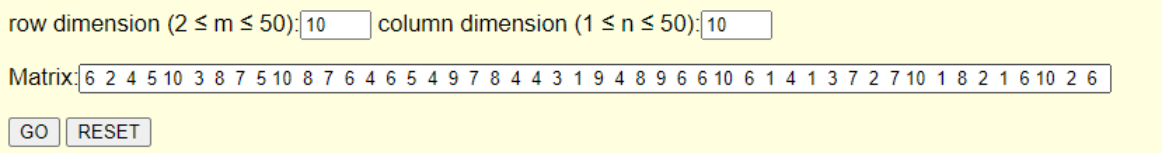

Рис. 2: Нахождение ЭНФ с помощью numbertheory.org. Ввод данных.

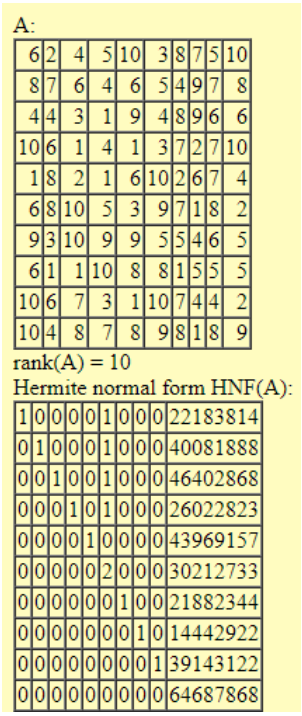

Рис. 3: Нахождение ЭНФ с помощью numbertheory.org. Результат.

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

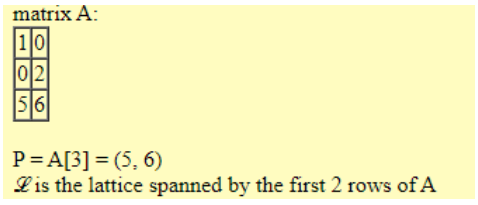

Рис. 4: Решение ПБВ с помощью numbertheory.org. Ввод данных.

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

Also
(5, 6)
is the closest vector of $\mathscr{L}$ 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.

```python
import numpy as np
from hsnf import column_style_hermite_normal_form, row_style_hermite_normal_form, smith_normal_form

# Integer matrix to be decomposed
M = np.random.random_integers(1, 10, (4, 4))

print(M)

# Row-style hermite normal form
H, L = row_style_hermite_normal_form(M)

print(H)

# Column-style hermite normal form
H, R = column_style_hermite_normal_form(M)

print(H)
```

```
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 cmake содержатся файлы для подключения сборок некоторых библиотек через CMake.
- CMakeLists.txt — файл CMake, использующийся для сборки проекта.

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

Программная реализация тестировалась с использованием компилятора G++ версии 6.3.0 в режиме сборки Release на ПК со следующими характеристиками: CPU: Intel(R) Core (TM) i5-9600KF CPU @ 3.70GHz, ОЗУ: DDR4, 16 ГБ (двухканальных режим 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) $\rightarrow$ [HNF|column] — функция, используемая при нахождении ЭНФ. Принимает ЭНФ и возвращает [HNF|column].

reduce(vector, matrix) $\rightarrow$ reduced_vector — функция для сокращения вектора относительно диагональных элементов входного базиса.

generate_random_matrix_with_full_row_rank(m, n, lowest, highest) $\rightarrow$ matrix — возвращает произвольную матрицу заданного размера с полным рангом строки с числами в заданном диапазоне.

generate_random_matrix(m, n, lowest, highest) → matrix — возвращает произвольную матрицу заданного размера с числами в заданном диапазоне.

get_linearly_independent_columns_by_gram_schmidt(matrix) → result_matrix — возвращает линейно независимые столбцы матрицы и ортогонализованный базис.

get_linearly_independent_rows_by_gram_schmidt(matrix) → result_matrix — возвращает линейно независимые строки матрицы, их индексы в исходной матрице, индексы удаленных строк и матрицу **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 ортогонализованный базис. Использует реализацию больших чисел от GMP.

get_linearly_independent_rows_by_gram_schmidt_GMP(matrix) → result_matrix — возвращает линейно независимые строки матрицы, их индексы в исходной матрице, индексы удаленных строк и матрицу **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

```
#ifndef ALGOTITHMS_HPP
#define ALGOTITHMS_HPP

#include <Eigen/Dense>
#include <boost/multiprecision/cpp_int.hpp>
#ifdef GMP
#include <boost/multiprecision/gmp.hpp>
#endif

namespace Algorithms
{
    namespace HNF
    {
        Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
           HNF_full_row_rank(const
           Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1> &B);
        Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1> HNF(const
           Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1> &B);

        #ifdef GMP
        Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>
           HNF_full_row_rank_GMP(const
           Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1> &B);
        Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>
           HNF_GMP(const Eigen::Matrix<boost::multiprecision::mpz_int,
           -1, -1> &B);
        #endif
    }
    namespace CVP
    {
        Eigen::VectorXd greedy_recursive(const Eigen::MatrixXd &matrix,
           const Eigen::VectorXd &target);
        Eigen::VectorXd greedy(const Eigen::MatrixXd &matrix, const
           Eigen::VectorXd &target);
        Eigen::VectorXd branch_and_bound(const Eigen::MatrixXd &matrix,
           const Eigen::VectorXd &target);

        #ifdef PARALLEL
        Eigen::VectorXd branch_and_bound_parallel(const Eigen::MatrixXd
           &matrix, const Eigen::VectorXd &target);
        #endif
    }
    #ifdef PARALLEL
    Eigen::MatrixXd gram_schmidt_parallel(const Eigen::MatrixXd &matrix,
       bool delete_zero_rows = true);
    #endif
    Eigen::MatrixXd gram_schmidt_sequential(const Eigen::MatrixXd
       &matrix, bool delete_zero_rows = true);
}

#endif
```

Приложение Б. Исходный код utils.hpp

```
#ifndef UTILS_HPP
#define UTILS_HPP

```

```
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
```

```
        int m, const int n, int lowest, int highest);
33      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 B. 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 =
                    Eigen::Matrix<mp::cpp_int, -1, -1>::Identity(m, m) * det;
49
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) =
                        Eigen::Matrix<mp::cpp_int, -1,
                        -1>::Zero(H_temp.rows(), n - m);
60              }
61
62              return H;
63          }
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
                Eigen::Matrix<mp::cpp_int, -1, -1> &B)
70          {
71              int m = static_cast<int>(B.rows());
72              int n = static_cast<int>(B.cols());
73
74              if (m < 1 || n < 1)
75              {
76                  throw std::invalid_argument("Matrix is not initialized");
77              }
78              if (B.isZero())
79              {
80                  throw std::runtime_error("Matrix is empty");
81              }
82
83              Eigen::Matrix<mp::cpp_int, -1, -1> B_stroke;
84              std::vector<int> indicies;
85              std::vector<int> deleted_indicies;
86              Eigen::Matrix<mp::cpp_rational, -1, -1> T;
87              std::tuple<Eigen::Matrix<mp::cpp_int, -1, -1>,
                    std::vector<int>, std::vector<int>,
                    Eigen::Matrix<mp::cpp_rational, -1, -1>> projection =
                    Utils::get_linearly_independent_rows_by_gram_schmidt(B);
88              std::tie(B_stroke, indicies, deleted_indicies, T) =
                    projection;
89
90              Eigen::Matrix<mp::cpp_int, -1, -1> B_double_stroke =
                    HNF_full_row_rank(B_stroke);
91
92              Eigen::Matrix<mp::cpp_int, -1, -1> HNF(B.rows(), B.cols());
93
94              for (int i = 0; i < indicies.size(); i++)
95              {
96                  HNF.row(indicies[i]) = B_double_stroke.row(i);
97              }
98
99              ////////////////////////////////////////////
100             // First way: just find linear combinations of deleted rows.
                    More accurate
101
102             // Eigen::Matrix<mp::cpp_bin_float_double, -1, -1>
                    B_stroke_transposed =
                    B_stroke.transpose().cast<mp::cpp_bin_float_double>();
```

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

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

```
                projection;

            Eigen::Matrix<mp::mpz_int, -1, -1> B_double_stroke =
                HNF_full_row_rank_GMP(B_stroke);

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

            for (int i = 0; i < indicies.size(); i++)
            {
                HNF.row(indicies[i]) = B_double_stroke.row(i);
            }

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

            // Eigen::Matrix<mp::mpf_float_50, -1, -1>
                B_stroke_transposed =
                B_stroke.transpose().cast<mp::mpf_float_50>();
            // auto QR =
                B_stroke.cast<mp::mpf_float_50>().colPivHouseholderQr().transpose

            // for (const auto &indx : deleted_indicies)
            // {
            //     Eigen::Vector<mp::mpf_float_50, -1> vec =
                B.row(indx).cast<mp::mpf_float_50>();
            //     Eigen::RowVector<mp::mpf_float_50, -1> x =
                QR.solve(vec);

            //     Eigen::Vector<mp::mpf_float_50, -1> res = x *
                HNF.cast<mp::mpf_float_50>();
            //     for (mp::mpf_float_50 &elem : res)
            //     {
            //         elem = mp::round(elem);
            //     }
            //     HNF.row(indx) = res.cast<mp::mpz_int>();
            // }
            // return HNF;
            ////////////////////////////////////////////


            ////////////////////////////////////////////
            // Other, the "right" way that is desribed in algorithm.
            Eigen::Matrix<mp::mpf_float_50, -1, -1> t_HNF =
                HNF.cast<mp::mpf_float_50>();
            for (const auto &indx : deleted_indicies)
            {
                Eigen::Vector<mp::mpf_float_50, -1> res =
                    Eigen::Vector<mp::mpf_float_50, -1>::Zero(B.cols());
                for (int i = 0; i < indx; i++)
                {
                    res += T(indx, i).convert_to<mp::mpf_float_50>() *
                        t_HNF.row(i);
                }

                t_HNF.row(indx) = res;
            }

            return t_HNF.cast<mp::mpz_int>();
            ////////////////////////////////////////////
        }
        #endif
    }


    namespace CVP
    {
```

```
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
               searched for
282         Eigen::VectorXd greedy_recursive_part(const Eigen::VectorXd
               &target)
283         {
284             if (index_greedy == 0)
285             {
286                 return Eigen::VectorXd::Zero(target.rows());
287             }
288             index_greedy--;
289             Eigen::VectorXd b = B_greedy.col(index_greedy);
290             Eigen::VectorXd b_star =
                   gram_schmidt_greedy.col(index_greedy);
291             double inner1 = std::inner_product(target.data(),
                   target.data() + target.size(), b_star.data(), 0.0);
292             double inner2 = std::inner_product(b_star.data(),
                   b_star.data() + b_star.size(), b_star.data(), 0.0);
293
294             double x = inner1 / inner2;
295             double c = std::round(x);
296
297             Eigen::VectorXd t_res = c * b;
298
299             return t_res + Algorithms::CVP::greedy_recursive_part(target
                   - t_res);
300         }
301
302
303         // Solves CVP using a recursive greedy algorithm
304         // @return Eigen::VectorXd
305         // @param matrix input rational lattice basis that is linearly
               independent
306         // @param target vector for which lattice point is being
               searched for
307         Eigen::VectorXd greedy_recursive(const Eigen::MatrixXd &matrix,
               const Eigen::VectorXd &target)
308         {
309             B_greedy = matrix;
310             #ifdef PARALLEL
311             gram_schmidt_greedy =
                   Algorithms::gram_schmidt_parallel(matrix, false);
312             #else
313             gram_schmidt_greedy =
                   Algorithms::gram_schmidt_sequential(matrix, false);
314             #endif
315             index_greedy = static_cast<int>(matrix.cols());
316
317             return greedy_recursive_part(target);
318         }
319
320
321         // Solves CVP using a non recursive greedy algorithm
322         // @return Eigen::VectorXd
323         // @param matrix input rational lattice basis that is linearly
               independent
324         // @param target vector for which lattice point is being
               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 *
                        b).norm() <= upper_bound)
391                 {
392                     X.push_back(static_cast<int>(x1));
393                     x1++;
394                 }
395                 else
396                 {
397                     flag1 = false;
398                 }
399
400                 if (flag2 && Utils::projection(B, target - x2 *
                        b).norm() <= upper_bound)
401                 {
402                     X.push_back(static_cast<int>(x2));
403                     x2--;
404                 }
405                 else
406                 {
407                     flag2 = false;
408                 }
409             }
410
411             std::vector<Eigen::VectorXd> V;
412
413
414             Eigen::VectorXd t_res;
415             for (const int &x : X)
416             {
417                 t_res = x * b +
                        Algorithms::CVP::branch_and_bound_recursive_part(B,
                        target - x * b);
418                 V.push_back(t_res);
419             }
420
421             return Utils::closest_vector(V, target);
422         }
423
424
425         // Solves CVP using a branch and bound algorithm
426         // @return Eigen::VectorXd
427         // @param matrix input rational lattice basis that is linearly
                independent
428         // @param target vector for which lattice point is being
                searched for
429         Eigen::VectorXd branch_and_bound(const Eigen::MatrixXd &matrix,
                const Eigen::VectorXd &target)
430         {
431             #ifdef PARALLEL
432             gram_schmidt_bb = Algorithms::gram_schmidt_parallel(matrix,
                    false);
433             #else
434             gram_schmidt_bb =
                    Algorithms::gram_schmidt_sequential(matrix, false);
435             #endif
436
437             return branch_and_bound_recursive_part(matrix, target);
```

```
438         }
439
440         #ifdef PARALLEL
441         // Recursive parallel body of branch and bound algorithm
442         // @return Eigen::VectorXd
443         // @param matrix input rational lattice basis that is linearly
                independent
444         // @param target vector for which lattice point is being
                searched for
445         Eigen::VectorXd branch_and_bound_recursive_part_parallel(const
                Eigen::MatrixXd &matrix, const Eigen::VectorXd &target)
446         {
447             if (matrix.cols() == 0)
448             {
449                 return Eigen::VectorXd::Zero(target.rows());
450             }
451             Eigen::MatrixXd B = matrix.block(0, 0, matrix.rows(),
                    matrix.cols() - 1);
452             Eigen::VectorXd b = matrix.col(B.cols());
453             Eigen::VectorXd b_star =
                    gram_schmidt_bb_parallel.col(B.cols());
454
455             Eigen::VectorXd v = Algorithms::CVP::greedy(matrix, target);
456
457             double upper_bound = (target - v).norm();
458
459             double x_middle = std::round(target.dot(b_star) /
                    b_star.dot(b_star));
460
461             std::vector<int> X;
462             X.push_back(static_cast<int>(x_middle));
463
464             bool flag1 = true;
465             bool flag2 = true;
466
467             double x1 = x_middle + 1;
468             double x2 = x_middle - 1;
469
470             while (flag1 || flag2)
471             {
472                 if (flag1 && Utils::projection(B, target - x1 *
                        b).norm() <= upper_bound)
473                 {
474                     X.push_back(static_cast<int>(x1));
475                     x1++;
476                 }
477                 else
478                 {
479                     flag1 = false;
480                 }
481
482                 if (flag2 && Utils::projection(B, target - x2 *
                        b).norm() <= upper_bound)
483                 {
484                     X.push_back(static_cast<int>(x2));
485                     x2--;
486                 }
487                 else
488                 {
489                     flag2 = false;
490                 }
491             }
492
493             std::vector<Eigen::VectorXd> V;
494
495
496             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 
520         // Solves CVP using a branch and bound parallel algorithm
521         // @return Eigen::VectorXd
522         // @param matrix input rational lattice basis that is linearly
                independent
523         // @param target vector for which lattice point is being
                searched for
524         Eigen::VectorXd branch_and_bound_parallel(const Eigen::MatrixXd
               &matrix, const Eigen::VectorXd &target)
525         {
526             gram_schmidt_bb_parallel =
                    Algorithms::gram_schmidt_parallel(matrix, false);
527 
528             return branch_and_bound_recursive_part_parallel(matrix,
                    target);
529         }
530         #endif
531     }
532 
533     #ifdef PARALLEL
534     // Computes Gram Schmidt orthogonalization
535     // @return Eigen::MatrixXd
536     // @param matrix input matrix
537     // @param normalize indicates whether to normalize output vectors
538     // @param delete_zero_rows indicates whether to delete zero rows
539     Eigen::MatrixXd gram_schmidt_parallel(const Eigen::MatrixXd &matrix,
           bool delete_zero_rows)
540     {
541         std::vector<Eigen::VectorXd> basis;
542 
543         for (const auto &vec : matrix.colwise())
544         {
545             Eigen::VectorXd projections =
                    Eigen::VectorXd::Zero(vec.size());
546 
547             #pragma omp parallel for
548             for (int i = 0; i < basis.size(); i++)
549             {
550                 Eigen::MatrixXd basis_vector = basis[i];
551                 double inner1 = std::inner_product(vec.data(),
                        vec.data() + vec.size(), basis_vector.data(), 0.0);
552                 double inner2 = std::inner_product(basis_vector.data(),
                        basis_vector.data() + basis_vector.size(),
```

```
                    basis_vector.data(), 0.0);

                #pragma omp critical
                projections += (inner1 / inner2) * basis_vector;
            }

            Eigen::VectorXd result = vec - projections;

            if (delete_zero_rows)
            {
                bool is_all_zero = result.isZero(1e-3);
                if (!is_all_zero)
                {
                    basis.push_back(result);
                }
            }
            else
            {
                basis.push_back(result);
            }
        }

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

        for (int i = 0; i < basis.size(); i++)
        {
            result.col(i) = basis[i];
        }

        return result;
    }
    #endif

    // Computes Gram Schmidt orthogonalization
    // @return Eigen::MatrixXd
    // @param matrix input matrix
    // @param normalize indicates whether to normalize output vectors
    // @param delete_zero_rows indicates whether to delete zero rows
    Eigen::MatrixXd gram_schmidt_sequential(const Eigen::MatrixXd
       &matrix, bool delete_zero_rows)
    {
        std::vector<Eigen::VectorXd> basis;

        for (const auto &vec : matrix.colwise())
        {
            Eigen::VectorXd projections =
               Eigen::VectorXd::Zero(vec.size());

            for (int i = 0; i < basis.size(); i++)
            {
                Eigen::MatrixXd basis_vector = basis[i];
                double inner1 = std::inner_product(vec.data(),
                   vec.data() + vec.size(), basis_vector.data(), 0.0);
                double inner2 = std::inner_product(basis_vector.data(),
                   basis_vector.data() + basis_vector.size(),
                   basis_vector.data(), 0.0);

                projections += (inner1 / inner2) * basis_vector;
            }

            Eigen::VectorXd result = vec - projections;

            if (delete_zero_rows)
            {
                bool is_all_zero = result.isZero(1e-3);
                if (!is_all_zero)
                {
```

```
                    basis.push_back(result);
                }
            }
            else
            {
                basis.push_back(result);
            }
        }

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

        for (int i = 0; i < basis.size(); i++)
        {
            result.col(i) = basis[i];
        }

        return result;
    }
}
```

Приложение Г. Исходный код utils.cpp

```
#include "utils.hpp"
#include <iostream>
#include <random>
#include <functional>
#include <numeric>
#include <vector>
#include <stdexcept>
#include <string>
#include <chrono>
#include <algorithm>
#include <thread>
#include "algorithms.hpp"

namespace mp = boost::multiprecision;

namespace Utils
{
    // Function for computing HNF of full row rank matrix
    // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
    // @param H HNF
    // @param b column to be added
    Eigen::Matrix<mp::cpp_int, -1, -1> add_column(const
       Eigen::Matrix<mp::cpp_int, -1, -1> &H, const
       Eigen::Vector<mp::cpp_int, -1> &b_column)
    {
        if (H.rows() == 0)
        {
            return H;
        }

        Eigen::Vector<mp::cpp_int, -1> H_first_col = H.col(0);

        mp::cpp_int a = H_first_col(0);
        Eigen::Vector<mp::cpp_int, -1> h =
           H_first_col.tail(H_first_col.rows() - 1);
        Eigen::Matrix<mp::cpp_int, -1, -1> H_stroke = H.block(1, 1,
           H.rows() - 1, H.cols() - 1);
        mp::cpp_int b = b_column(0);
        Eigen::Vector<mp::cpp_int, -1> b_stroke =
           b_column.tail(b_column.rows() - 1);

        std::tuple<mp::cpp_int, mp::cpp_int, mp::cpp_int> gcd_result =
           gcd_extended(a, b);
```

```
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 =
               temp_result.col(0).tail(temp_result.rows() - 1);
51          Eigen::Vector<mp::cpp_int, -1> b_double_stroke =
               temp_result.col(1).tail(temp_result.rows() - 1);
52
53          b_double_stroke = reduce(b_double_stroke, H_stroke);
54
55          Eigen::Matrix<mp::cpp_int, -1, -1> H_double_stroke =
               add_column(H_stroke, b_double_stroke);
56
57          h_stroke = reduce(h_stroke, H_double_stroke);
58
59          Eigen::Matrix<mp::cpp_int, -1, -1> result(H.rows(), H.cols());
60
61          result(0, 0) = g;
62          result.col(0).tail(result.cols() - 1) = h_stroke;
63          result.row(0).tail(result.rows() - 1).setZero();
64          result.block(1, 1, H_double_stroke.rows(),
               H_double_stroke.cols()) = H_double_stroke;
65
66          return result;
67      }
68
69
70      // Function for computing HNF, reduces elements of vector modulo
           diagonal elements of matrix
71      // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
72      // @param vector vector to be reduced
73      // @param matrix input matrix
74      Eigen::Vector<mp::cpp_int, -1> reduce(const
           Eigen::Vector<mp::cpp_int, -1> &vector, const
           Eigen::Matrix<mp::cpp_int, -1, -1> &matrix)
75      {
76          Eigen::Vector<mp::cpp_int, -1> result = vector;
77          for (int i = 0; i < result.rows(); i++)
78          {
79              Eigen::Vector<mp::cpp_int, -1> matrix_column = matrix.col(i);
80              mp::cpp_int t_vec_elem = result(i);
81              mp::cpp_int t_matrix_elem = matrix(i, i);
82
83              mp::cpp_int x;
84              if (t_vec_elem >= 0)
85              {
86                  x = (t_vec_elem / t_matrix_elem);
87              }
88              else
89              {
90                  x = (t_vec_elem - (t_matrix_elem - 1)) / t_matrix_elem;
91              }
92
93              result -= matrix_column * x;
94          }
95          return result;
96      }
97
```

```
98
99      // Generates random matrix with full row rank
100     // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
101     // @param m number of rows, must be greater than one and less than
           or equal to the parameter n
102     // @param n number of columns, must be greater than one and greater
           than or equal to the parameter m
103     // @param lowest lowest generated number, must be lower than lowest
           parameter by at least one
104     // @param highest highest generated number, must be greater than
           lowest parameter by at least one
105     Eigen::Matrix<mp::cpp_int, -1, -1>
           generate_random_matrix_with_full_row_rank(const int m, const int
           n, int lowest, int highest)
106     {
107         if (m > n)
108         {
109             throw std::invalid_argument("m must be less than or equal
                   n");
110         }
111         if (m < 1 || n < 1)
112         {
113             throw std::invalid_argument("Number of rows or columns
                   should be greater than one");
114         }
115         if (highest - lowest < 1)
116         {
117             throw std::invalid_argument("highest parameter must be
                   greater than lowest parameter by at least one");
118         }
119         std::random_device rd;
120         std::mt19937 gen(rd());
121         std::uniform_int_distribution<int> dis (lowest, highest);
122
123         Eigen::Matrix<int, -1, -1> matrix = Eigen::Matrix<int, -1,
               -1>::NullaryExpr(m, n, [&]()
124                                                                 { return
                                                                       dis(gen);
                                                                       });
125
126         Eigen::FullPivLU<Eigen::MatrixXd>
               lu_decomp(matrix.cast<double>());
127         auto rank = lu_decomp.rank();
128
129         while (rank != m)
130         {
131             matrix = Eigen::Matrix<int, -1, -1>::NullaryExpr(m, n, [&]()
132                                                     { return dis(gen); });
133
134             lu_decomp.compute(matrix.cast<double>());
135             rank = lu_decomp.rank();
136         }
137
138         return matrix.cast<mp::cpp_int>();
139     }
140
141
142     // Generates random matrix
143     // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
144     // @param m number of rows, must be greater than one
145     // @param n number of columns, must be greater than one
146     // @param lowest lowest generated number, must be lower than lowest
           parameter by at least one
147     // @param highest highest generated number, must be greater than
           lowest parameter by at least one
148     Eigen::Matrix<mp::cpp_int, -1, -1> generate_random_matrix(const int
           m, const int n, int lowest, int highest)
```

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

```
                projections;

            bool is_all_zero = result.isZero(1e-3);
            if (!is_all_zero)
            {
                basis.push_back(result);
                indexes.push_back(counter);
            }
            counter++;
        }

        Eigen::Matrix<mp::cpp_int, -1, -1> result(matrix.rows(),
            indexes.size());
        Eigen::Matrix<mp::cpp_rational, -1, -1>
            gram_schmidt(matrix.rows(), basis.size());

        for (int i = 0; i < indexes.size(); i++)
        {
            result.col(i) = matrix.col(indexes[i]);
            gram_schmidt.col(i) = basis[i];
        }
        return std::make_tuple(result, gram_schmidt);
    }
    #else
    // Returns matrix that consist of linearly independent columns of
        input matrix and othogonalized matrix
    // @param matrix input matrix
    // @return std::tuple<Eigen::Matrix<boost::multiprecision::cpp_int,
        -1, -1>, Eigen::Matrix<boost::multiprecision::cpp_rational, -1,
        -1>>
    std::tuple<Eigen::Matrix<mp::cpp_int, -1, -1>,
        Eigen::Matrix<mp::cpp_rational, -1, -1>>
        get_linearly_independent_columns_by_gram_schmidt(const
        Eigen::Matrix<mp::cpp_int, -1, -1> &matrix)
    {
        std::vector<Eigen::Vector<mp::cpp_rational, -1>> basis;
        std::vector<int> indexes;

        int counter = 0;
        for (const Eigen::Vector<mp::cpp_rational, -1> &vec :
            matrix.cast<mp::cpp_rational>().colwise())
        {
            Eigen::Vector<mp::cpp_rational, -1> projections =
                Eigen::Vector<mp::cpp_rational, -1>::Zero(vec.size());

            for (int i = 0; i < basis.size(); i++)
            {
                Eigen::Vector<mp::cpp_rational, -1> basis_vector =
                    basis[i];
                mp::cpp_rational inner1 = std::inner_product(vec.data(),
                    vec.data() + vec.size(), basis_vector.data(),
                    mp::cpp_rational(0.0));
                mp::cpp_rational inner2 =
                    std::inner_product(basis_vector.data(),
                    basis_vector.data() + basis_vector.size(),
                    basis_vector.data(), mp::cpp_rational(0.0));

                mp::cpp_rational coef = inner1 / inner2;

                projections += basis_vector * coef;
            }

            Eigen::Vector<mp::cpp_rational, -1> result = vec -
                projections;

            bool is_all_zero = result.isZero(1e-3);
            if (!is_all_zero)
```

```
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));

                mp::cpp_rational u_ij = 0;
                if (!inner1.is_zero())
                {
                    u_ij = inner1 / inner2;

                    projections += u_ij * basis_vector;
                    T(counter, i) = u_ij;
                }
            }

            Eigen::Vector<mp::cpp_rational, -1> result = vec -
                projections;

            bool is_all_zero = result.isZero(1e-3);
            if (!is_all_zero)
            {
                indicies.push_back(counter);
            }
            else
            {
                deleted_indicies.push_back(counter);
            }
            basis.push_back(result);
            counter++;
        }

        Eigen::Matrix<mp::cpp_int, -1, -1> result(indicies.size(),
            matrix.cols());
        for (int i = 0; i < indicies.size(); i++)
        {
            result.row(i) = matrix.row(indicies[i]);
        }
        return std::make_tuple(result, indicies, deleted_indicies, T);
    }
    #endif


    // Extended GCD algorithm, returns tuple of g, x, y such that xa +
        yb = g
    // @return std::tuple<boost::multiprecision::cpp_int,
        boost::multiprecision::cpp_int, boost::multiprecision::cpp_int>
    // @param a first number
    // @param b second number
    std::tuple<mp::cpp_int, mp::cpp_int, mp::cpp_int>
        gcd_extended(mp::cpp_int a, mp::cpp_int b)
    {
        if (a == 0)
        {
            return std::make_tuple(b, 0, 1);
        }
        mp::cpp_int gcd, x1, y1;
        std::tie(gcd, x1, y1) = gcd_extended(b % a, a);

        mp::cpp_int x = y1 - (b / a) * x1;
        mp::cpp_int y = x1;

        return std::make_tuple(gcd, x, y);
    }

    #ifdef GMP
    // Function for computing HNF of full row rank matrix
    // @return Eigen::Matrix<boost::multiprecision::cpp_mpz, -1, -1>
    // @param H HNF
    // @param b column to be added
```

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

```
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
        independent)
482     // @return Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>
483     // @param m number of rows, must be greater than one and less than
        or equal to the parameter n
484     // @param n number of columns, must be greater than one and greater
        than or equal to the parameter m
485     // @param lowest lowest generated number, must be lower than lowest
        parameter by at least one
486     // @param highest highest generated number, must be greater than
        lowest parameter by at least one
487     Eigen::Matrix<mp::mpz_int, -1, -1>
        generate_random_matrix_with_full_row_rank_GMP(const int m, const
        int n, int lowest, int highest)
488     {
489         if (m > n)
490         {
491             throw std::invalid_argument("m must be less than or equal
                n");
492         }
493         if (m < 1 || n < 1)
494         {
495             throw std::invalid_argument("Number of rows or columns
                should be greater than one");
496         }
497         if (highest - lowest < 1)
498         {
499             throw std::invalid_argument("highest parameter must be
                greater than lowest parameter by at least one");
500         }
501         std::random_device rd;
502         std::mt19937 gen(rd());
503         std::uniform_int_distribution<int> dis (lowest, highest);
504 
505         Eigen::Matrix<int, -1, -1> matrix = Eigen::Matrix<int, -1,
            -1>::NullaryExpr(m, n, [&]()
506                                                                 { return
                                                                    dis(gen);
                                                                    });
507 
508         Eigen::FullPivLU<Eigen::MatrixXd>
            lu_decomp(matrix.cast<double>());
509         auto rank = lu_decomp.rank();
510 
```

```
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
        parameter by at least one
529     // @param highest highest generated number, must be greater than
        lowest parameter by at least one
530     Eigen::Matrix<mp::mpz_int, -1, -1> generate_random_matrix_GMP(const
        int m, const int n, int lowest, int highest)
531     {
532         if (m < 1 || n < 1)
533         {
534             throw std::invalid_argument("Number of rows or columns
                should be greater than one");
535         }
536         if (highest - lowest < 1)
537         {
538             throw std::invalid_argument("highest parameter must be
                greater than lowest parameter by at least one");
539         }
540 
541         std::random_device rd;
542         std::mt19937 gen(rd());
543         std::uniform_int_distribution<int> dis (lowest, highest);
544 
545         Eigen::Matrix<int, -1, -1> matrix = Eigen::Matrix<int, -1,
            -1>::NullaryExpr(m, n, [&]()
546                                                                 { return
                                                                    dis(gen);
                                                                    });
547 
548         return matrix.cast<mp::mpz_int>();
549     }
550 
551     #ifdef PARALLEL
552     // Returns matrix that consist of linearly independent columns of
        input matrix and othogonalized matrix
553     // @param matrix input matrix
554     // @return std::tuple<Eigen::Matrix<boost::multiprecision::mpz_int,
        -1, -1>, Eigen::Matrix<boost::multiprecision::mpq_rational, -1,
        -1>>
555     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)
556     {
557         std::vector<Eigen::Vector<mp::mpq_rational, -1>> basis;
558         std::vector<int> indexes;
559 
560         int counter = 0;
561         for (const Eigen::Vector<mp::mpq_rational, -1> &vec :
            matrix.cast<mp::mpq_rational>().colwise())
562         {
```

```
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());

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

                mp::mpq_rational u_ij = 0;
                if (!inner1.is_zero())
                {
                    u_ij = inner1 / inner2;
                    #pragma omp critical
                    {
                        projections += u_ij * basis_vector;
                        T(counter, i) = u_ij;
                    }
                }
            }

            Eigen::Vector<mp::mpq_rational, -1> result = vec -
                projections;

            bool is_all_zero = result.isZero(1e-3);
            if (!is_all_zero)
            {
                indicies.push_back(counter);
            }
            else
            {
                deleted_indicies.push_back(counter);
            }
            basis.push_back(result);
            counter++;
        }

        Eigen::Matrix<mp::mpz_int, -1, -1> result(indicies.size(),
            matrix.cols());
        for (int i = 0; i < indicies.size(); i++)
        {
            result.row(i) = matrix.row(indicies[i]);
        }
        return std::make_tuple(result, indicies, deleted_indicies, T);
    }
    #else
    // Returns matrix that consist of linearly independent rows of input
        matrix, indicies of that rows in input matrix, indices of deleted
        rows and martix T, that consists of Gram Schmidt coefficients
    // @param matrix input matrix
    // @return std::tuple<Eigen::Matrix<boost::multiprecision::mpz_int,
        -1, -1>, std::vector<int>, std::vector<int>,
        Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>>
    std::tuple<Eigen::Matrix<mp::mpz_int, -1, -1>, std::vector<int>,
        std::vector<int>, Eigen::Matrix<mp::mpq_rational, -1, -1>>
        get_linearly_independent_rows_by_gram_schmidt_GMP(const
        Eigen::Matrix<mp::mpz_int, -1, -1> &matrix)
    {
        std::vector<Eigen::Vector<mp::mpq_rational, -1>> basis;
        std::vector<int> indicies;
       std::vector<int> deleted_indicies;
```

```
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
           than or equal to the parameter n
784     // @param n number of columns, must be greater than one and lower
           than or equal to the parameter m
785     // @param lowest lowest generated number, must be lower than lowest
           parameter by at least one
786     // @param highest highest generated number, must be greater than
           lowest parameter by at least one
787     Eigen::MatrixXd generate_random_matrix_with_full_column_rank(const
           int m, const int n, int lowest, int highest)
788     {
789         if (m < n)
790         {
791             throw std::invalid_argument("m must be less than or equal
                   n");
792         }
793         if (m < 1 || n < 1)
794         {
795             throw std::invalid_argument("Number of rows or columns
                   should be greater than one");
796         }
797         if (highest - lowest < 1)
798         {
799             throw std::invalid_argument("highest parameter must be
                   greater than lowest parameter by at least one");
800         }
801         std::random_device rd;
802         std::mt19937 gen(rd());
803         std::uniform_int_distribution<int> dis (lowest, highest);
804
805         Eigen::MatrixXd matrix = Eigen::MatrixXd::NullaryExpr(m, n, [&]()
806                                                               { return
                                                                  dis(gen);
                                                                  });
807
808         Eigen::FullPivLU<Eigen::MatrixXd> lu_decomp(matrix);
809         auto rank = lu_decomp.rank();
810
811         while (rank != n)
812         {
813             matrix = Eigen::MatrixXd::NullaryExpr(m, n, [&]()
814                                                   { return dis(gen); });
815
816             lu_decomp.compute(matrix);
817             rank = lu_decomp.rank();
818         }
819
```

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

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

```
            }
        }

        return closest;
    }
}
```

Приложение Д. Исходный CMakeLists.txt

```
cmake_minimum_required(VERSION 3.2)
project(LatticeAlgorithms)

option(BUILD_DOCS "" OFF)
option(BUILD_PARALLEL "" OFF)
option(BUILD_GMP "" OFF)

file(GLOB SRC
     "src/utils.cpp"
     "src/algorithms.cpp"
)

add_subdirectory(3rdparty/boost_config)
add_subdirectory(3rdparty/boost_multiprecision)

find_package(OpenMP REQUIRED)

add_library(${PROJECT_NAME} ${SRC})

target_include_directories(${PROJECT_NAME} PUBLIC include)

if (BUILD_PARALLEL)
     target_compile_definitions(${PROJECT_NAME} PUBLIC PARALLEL)
endif(BUILD_PARALLEL)

if (BUILD_GMP)
     target_compile_definitions(${PROJECT_NAME} PUBLIC GMP)
endif(BUILD_GMP)

target_link_libraries(${PROJECT_NAME} OpenMP::OpenMP_CXX)
target_link_libraries(${PROJECT_NAME} gmp libgmp)
target_link_libraries(${PROJECT_NAME} Boost::config
   Boost::multiprecision)

if (BUILD_DOCS)
     add_subdirectory(tex)
endif(BUILD_DOCS)
```