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 механники

Кафедра: алгебры, геометрии и дискретной математики

Направление подготовки: «Программная инженерия»
Профиль подготовки: «Разработка программно-информационных систем»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

на тему:
«Нахождение Эрмитовой нормальной формы и решение проблемы ближайшего вектора решетки»

Выполнил(а): студент(ка) группы ____
_____ Д.В. Огнев

Подпись

Научный руководитель:
Доцент, кандидат физико-математических наук

_____ С.И. Весёлов

Подпись

Нижний Новгород
2022

Аннотация (ДОПИСАТЬ)

Тема выпускной квалификационной работы бакалавра – «Нахождение Эрмитовой нормальной формы и решение проблемы ближайшего вектора решетки».

Ключевые слова: решетки, 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, и разбор алгоритмов для нахождения Эрмитовой нормальной формы и решения ближайшего вектора решетки.

Целью работы является программная реализация алгоритмов для решения задач.

Объем работы - ...

Содержание

1. Список условных обозначений и сокращений (TODO)

ПБВ (CVP) – проблема ближайшего вектора (Closest vector problem)

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

B&B – Branch and bound

2. Введение (TODO)

Криптография занимается разработкой методов преобразования (шифрования) информации с целью ее зашиты 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, что заставляет искать задачи, для которых неизвестны эффективные квантовые алгоритмы. В области постквантовой криптографии фаворитом считается криптография на решетках. Считается, что такая криптография устойчива к квантовым компьютерам.

Объектом 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 основы, откуда взяты теоретические основы и описание алгоритмов для программирования, будем использовать серию лекций по основам алгоритмов на решетках и их применении.

3. Основные определения (TODO)

Матрица – прямоугольная таблица чисел, состоящая из n столбцов и m строк. Обозначается полужирной заглавной буквой, а ее элементы - строчными с двумя индексами (строка и столбец). При программировании использовалась стандартная структура хранения матриц:

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

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 неравен нулю.

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. Будем обозначать матрицы через вектора-столбцы, то есть $\mathbf{A} = [\mathbf{a}_1, \cdots, \mathbf{a}_n]$. Вектора-строки будем обознать $\mathbf{a}^{\mathbf{T}}$.

Линейная зависимость и независимость – пусть имеется несколько векторов одной размерности $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k$ и столько же чисел $\alpha_1, \alpha_2, \cdots, \alpha_k$. Вектор $\mathbf{y} = \alpha_1 x_1 + \alpha_2 x_2 + \ldots + \alpha_k x_k$ называется линейной комбинацией векторов $\mathbf{x}_k$. Если существуют такие числа $\alpha_i, i = 1, \ldots, k$, не все равные нулю, такие, что $\mathbf{y} = 0$, то такой набор векторов называется линейно зависимым. В противном случае векторы называются линейно независимыми.

Ранг матрицы – максимальное число линейно независимых векторов. Матрица называется матрицей полного ранга строки, когда все строки матрицы линейно независимы. Матрица называется матрицей полного ранга столбца, когда все столбцы матрицы линейно независимы.

Решетка - пусть $\mathbf{B} = [\mathbf{b}_1, ..., \mathbf{b}_n] \in \mathbb{R}^{d \times n}$ - 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 $\mathbb{R}^d$.

Эрмитова нормальная форма - 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 $b_{i_j,j}$.

4. Постановка задачи (TODO)

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 программу, в которой будут реализованы разобранные алгоритмы. Полученная программа должна быть использована как подключаемая библиотека.
- Полученную библиотеку использовать для решения задач теории решеток и найти практическое применение.

5. Обзор литературных источников (TODO)

В результате работы был получен перевод статьи, описывающей необходимые алгоритмы и их применение.

6. Обзор инструментов (TODO)

Для программной реализации был выбран язык C++. Приоритет этому яызку отдается из-за его скорости, статической типизации и большому количеству написанных библиотек. Сборка проекта осуществляется с помощью системы сборки CMake, при сборке она автоматически собирает документ выпускной квалификационной работы, написанный в формате LaTeX. 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 система контроля версий Git и сервис Github, все исходные файлы проекта доступны в онлайн репозитории. Для Boost.Multiprecision используются модули Git.

6.1. Обзор библиотеки Eigen

Eigen - шаблонная библиотека для работы с линейной алгеброй. Предоставляет классы и методы для работы с матрицами, векторами и связанными алгоритмами. Является header-only библиотекой, не требует отдельной компиляции и линковки. Для работы не требует других библиотек, кроме стандратной.

Все необходимые классы находятся в заголовочном файле `Eigen/Dense` и подключается командой `#include <Eigen/Dense>`. Все используемые классы находятся в пространстве имен Eigen.

Используемые классы:

`Matrix<typename Scalar, int RowsAtCompileTime, int ColsAtCompileTime>` – шаблонный класс матрицы. Первый параметр шаблона - тип элементов матрицы, второй параметр – количество строк, третий – количество столбцов. Если количество строк/столбцов неизвестно на стадии компиляции, а будет найдено в процессе выполнения программы, то необходимо ставить количество строк/столбцов равным `Eigen::Dynamic`, либо `-1`. Имеет псевдонимы для различных типов и размеров матриц, например `Matrix3d` – матрица элементов `double` размера `3x3`.

`Vector` и `RowVector` – псевдонимы класса матриц, в которых количество строк/столбцов равно единице. Используются псевдонимы для различных типов и размеров векторов, например `Vector2f` – вектор, состоящий из элементов `float` размера `3`.

Матрицы и вектора можно складывать и вычитать между собой, умножать и делить между собой и на скаляр.

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 вставлять скалярные типы, матрицы, вектора.

`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())` – перебор матрицы по столбцам.

`for (const Eigen::VectorXd &vector : matrix.rowwise())` – перебор матрицы по строкам.

6.2. Обзор библиотеки Boost.Multiprecision

Multiprecision – часть библиотеки Boost. Подключается в режиме Standalone и не требует подключения основной библиотеки. Все классы находятся в пространстве имен `boost::multiprecision`. Для подключения используется директива `#include <boost/multiprecision/cpp_тип.hpp>`.

Библиотека предоставляет классы для работы с целыми, рациональными числами и числами с плавающей запятой, которые имеют большую точность, чем встроенные в C++ типы данных. Точность и размер чисел ограничен количеством оперативной памяти.

Используемые классы:

`cpp_int` – класс целых чисел.

`cpp_rational` – класс рациональных чисел.

`cpp_bin_float_double` – класс чисел с плавающей запятой с увеличенной точностью.

Используемые методы:

`sqrt(int)` – квадратный корень из целого числа.

`numerator(rational)` – числитель рационального числа.

`denominator(rational)` – знаменатель рационального числа.

7. Нахождение ЭНФ (TODO)

Будет разобрано два алгоритма - общий и алгоритм для матриц полного ранга строки, который используется в общем алгоритме.

7.1. Алгоритм для матриц полного ранга строки

Дана матрица $\mathbf{B} \in \mathbb{Z}^{m \times n}$. Предположим, что у нас есть процедура AddColumn, которая работает за полиномиальное время и принимает на вход квадратную невырожденную ЭНФ матрицы $\mathbf{H} \in \mathbb{Z}^{m \times m}$ и вектор $\mathbf{b}$, а возвращает ЭНФ матрицы $[\mathbf{H}|\mathbf{b}]$. ЭНФ от $\mathbf{B}$ может быть вычислена следующим образом:

1. Применить алгоритм Грама-Шмидта к столбцам $\mathbf{B}$, чтобы найти m линейно независимых столбцов. Пусть $\mathbf{B}$' - матрица размера $m \times m$, заданная этими столбцами.
2. Вычислить $d = \det(\mathbf{B}')$, используя алгоритм Грама-Шмидта или любую другую процедуру с полиномиальным временем. Пусть $\mathbf{H}_0 = d \cdot \mathbf{I}$ будет диагональной матрицей с d на диагонали.
3. Для $i = 1, \ldots, n$ пусть $\mathbf{H}_i$ это результат применения AddColumn ко входу $\mathbf{H}_{i-1}$ 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, а значит матрица, состоящая из этих столбцов, будет размера $m \times m$. Для нахождения этих строк можно использовать алгоритм ортогонализации Грама-Шмитда: если $\mathbf{b}_i^* = 0$, то i-ая строка является линейной комбинацией других строк, и ее необходимо удалить. Реализация данного алгоритма находится в пространстве имен Utils в функции get_linearly_independent_columns_by_gram_schmidt. Полученная матрица будет названа $\mathbf{B}$'.
2. Для вычисления det напишем функцию det_by_gram_schmidt, которая принимает на вход матрицу и вычисляет det по формуле $d = \prod_i \|\mathbf{b}_i^*\|$ - сумма произведений длин всех элементов, 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 функцию AddColumn (реализация находится в функции add_column) к $\mathbf{H}_0$ и первому столбцу матрицы $\mathbf{B} - \mathbf{b}_0$, получаем $\mathbf{H}_1$; повторяем для всех 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 тут ничего не надо делать, и мы можем сразу вернуть $\mathbf{H}$. 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 $g = НОД\,(a, b)$ 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. Функция extended_gcd принимает a, b , вычисляет 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 принимает на вход матрицу и вектор, получает 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 $\begin{bmatrix} g & \mathbf{0}^T \\ \mathbf{h}' & \mathbf{H}'' \end{bmatrix}$

7.2. Общий алгоритм для любых матриц

1. Запустить процесс ортогонализации Грамма-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 $\mathbf{r}^*_{\mathrm{k}_i} \neq 0$. 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 другая строка может быть выражена как линейная комбинация предыдущих строк $\mathbf{r}_j\,(\{j \in \mathrm{K} : j < i\})$. Следовательно, $\prod_{\mathrm{K}}$ однозначна, когда ограничена к $\mathcal{L}(\mathbf{B})$, и ее инверсия может быть легко вычислена, используя коэффициенты Грама-Шмидта $\mu_{i,j}$.

2. Определить новую матрицу $\mathbf{B}' = \prod_{\mathrm{K}}(\mathbf{B})$, которая полного ранга, и запустить алгоритм, данный в предыдущем пункте, чтобы найти ЭНФ $\mathbf{B}''$ от $\mathbf{B}'$.

3. Применить функцию обратную операции проецирования, $\prod_{\mathrm{K}}^{-1}$, к ЭНФ, определенной в предыдущем шаге $(\mathbf{B}'')$, к данной матрице $\mathbf{H}$. Легко заметить, что $\mathcal{L}\,(\mathbf{H})\,\mathcal{L}\,(\mathbf{B})$ и $\mathbf{H}$ входят в ЭНФ. Следовательно, $\mathbf{H}$ является ЭНФ $\mathbf{B}$.

Алгоритм прост, но вызывает вопрос операция проецирования и обратная 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, данный 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 $\mathbf{B}'$.

7.3. Пример нахождения ЭНФ

7.4. Сложность алгоритма

7.5. Обзор программной реализации

7.6. Применение

8. Решение ПБВ (TODO)

Будет разобрано два алгоритма - жадный метод, работающий за полиномиальное время, но дающий приближенное решение, и метод ветвей и границ, работающий за суперполиномиальное время, но точно решающий проблему ближайшего вектора.

8.1. Определение проблемы

Рассмотрим проблему ближайшего вектора (ПБВ): 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 экземпляр 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:

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

8.2. 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. 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

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

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

Greedy$([], \mathbf{t}) = 0$

Greedy$([\mathbf{B}, \mathbf{b}], \mathbf{t}) = c \cdot \mathbf{b} +$ Greedy$(\mathbf{B}, \mathbf{t} - c \cdot \mathbf{b})$

где $\mathbf{b}^* = \mathbf{b} \perp \mathbf{B}$

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

$c = \lfloor x \rceil$.

8.3. Пример жадного метода

8.4. Метод ветвей и границ

Структура похожа на жадный алгоритм, но вместо жадной 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 $\mathbf{v} =$ 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 $(\mathbf{t} - x\mathbf{b}) \perp \mathbf{B}'|| \leq u$.

Окончательный алгоритм похож на жадный метод и описан ниже:

Branch&Bound$([], \mathbf{t}) = 0$

Branch&Bound$([\mathbf{B}, \mathbf{b}], \mathbf{t}) =$ closest$(V, \mathbf{t})$

где $\mathbf{b}^* = \mathbf{b} \perp \mathbf{B}$

$\mathbf{v} =$ Greedy$(\mathbf{B}, \mathbf{t})$

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

$V = x \cdot \mathbf{b} +$ 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 метода Ветвей и Границ может быть произвольно плохой, если мы сперва не сократим базисы.

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

8.5. Пример метода ветвей и границ

8.6. Сложность алгоритмов

8.7. Обзор программной реализации

8.8. Применение

9. Обзор программной реализации (TODO)

10. Заключение (TODO)

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. Point Lattices. [Электронный ресурс]. — URL: https://cseweb.ucsd.edu/classes/sp14/cse206A-a/lec1.pdf (Дата обращения: 16.05.2022).
2. Daniele Micciancio. Basic Algorithms. [Электронный ресурс]. — URL: https://cseweb.ucsd.edu/classes/sp14/cse206A-a/lec4.pdf (Дата обращения: 16.05.2022).
3. Документация библиотеки Eigen. [Электронный ресурс]. — URL: https://eigen.tuxfamily.org/dox/index.html (Дата обращения: 16.05.2022).
4. 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).

Приложения (TODO)

Algorithms.cpp

```cpp
#include "algorithms.hpp"
#include <iostream>
#include "utils.hpp"
#include <vector>
#include <numeric>

namespace mp = boost::multiprecision;

namespace Algorithms
{
    namespace HNF
    {
        // Computes HNF of a integer matrix that is full row rank
        // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
        // @param B full row rank matrix
        Eigen::Matrix<mp::cpp_int, -1, -1> HNF_full_row_rank(const
            Eigen::Matrix<mp::cpp_int, -1, -1> &B)
        {
            int m = static_cast<int>(B.rows());
            int n = static_cast<int>(B.cols());

            if (m > n)
            {
                throw std::invalid_argument("m must be less than or
                    equal n");
            }
            if (m < 1 || n < 1)
            {
                throw std::invalid_argument("Matrix is not initialized");
            }
            if (B.isZero())
            {
                throw std::exception("Matrix is empty");
            }

            Eigen::Matrix<mp::cpp_int, -1, -1> B_stroke;
            Eigen::Matrix<mp::cpp_rational, -1, -1> ortogonalized;

            std::tuple<Eigen::Matrix<mp::cpp_int, -1, -1>,
                Eigen::Matrix<mp::cpp_rational, -1, -1>> result_of_gs =
                Utils::get_linearly_independent_columns_by_gram_schmidt(B);

            std::tie(B_stroke, ortogonalized) = result_of_gs;

            mp::cpp_rational t_det = 1.0;
            for (const Eigen::Vector<mp::cpp_rational, -1> &vec :
                ortogonalized.colwise())
            {
                t_det *= vec.squaredNorm();
            }
            mp::cpp_int det = mp::sqrt(mp::numerator(t_det));

            Eigen::Matrix<mp::cpp_int, -1, -1> H_temp =
                Eigen::Matrix<mp::cpp_int, -1, -1>::Identity(m, m) * det;

            for (int i = 0; i < n; i++)
            {
                H_temp = Utils::add_column(H_temp, B.col(i));
            }
```

```
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          // Computes HNF of an arbitrary integer matrix
66          // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
67          // @param B arbitrary matrix
68          Eigen::Matrix<mp::cpp_int, -1, -1> HNF(const
                Eigen::Matrix<mp::cpp_int, -1, -1> &B)
69          {
70              int m = static_cast<int>(B.rows());
71              int n = static_cast<int>(B.cols());
72
73              if (m < 1 || n < 1)
74              {
75                  throw std::invalid_argument("Matrix is not initialized");
76              }
77              if (B.isZero())
78              {
79                  throw std::exception("Matrix is empty");
80              }
81
82              Eigen::Matrix<mp::cpp_int, -1, -1> B_stroke;
83              std::vector<int> indicies;
84              std::vector<int> deleted_indicies;
85              Eigen::Matrix<mp::cpp_rational, -1, -1> T;
86              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);
87              std::tie(B_stroke, indicies, deleted_indicies, T) =
                    projection;
88
89              Eigen::Matrix<mp::cpp_int, -1, -1> B_double_stroke =
                    HNF_full_row_rank(B_stroke);
90
91              Eigen::Matrix<mp::cpp_int, -1, -1> HNF(B.rows(), B.cols());
92
93              for (int i = 0; i < indicies.size(); i++)
94              {
95                  HNF.row(indicies[i]) = B_double_stroke.row(i);
96              }
97
98              ////////////////////////////////////////////////////////////////
99              // First way: just find linear combinations of deleted rows.
                    More accurate
100
101             // Eigen::Matrix<mp::cpp_bin_float_double, -1, -1>
                    B_stroke_transposed =
                    B_stroke.transpose().cast<mp::cpp_bin_float_double>();
102             // auto QR =
                    B_stroke.cast<mp::cpp_bin_float_double>().colPivHouseholderQr().t
103
104             // for (const auto &indx : deleted_indicies)
105             // {
106             //     Eigen::Vector<mp::cpp_bin_float_double, -1> vec =
                    B.row(indx).cast<mp::cpp_bin_float_double>();
107             //     Eigen::RowVector<mp::cpp_bin_float_double, -1> x =
```

```
                QR.solve(vec);
108
109             //      Eigen::Vector<mp::cpp_bin_float_double, -1> res = x *
                    HNF.cast<mp::cpp_bin_float_double>();
110             //      for (mp::cpp_bin_float_double &elem : res)
111             //      {
112             //          elem = mp::round(elem);
113             //      }
114             //      HNF.row(indx) = res.cast<mp::cpp_int>();
115             // }
116             // return HNF;
117             ////////////////////////////////////////////////////////////////////
118
119
120             ////////////////////////////////////////////////////////////////////
121             // Other, the "right" way that is desribed in algorithm.
                    Have small numerical errors
122             Eigen::Matrix<mp::cpp_bin_float_double, -1, -1> t_HNF =
                    HNF.cast<mp::cpp_bin_float_double>();
123             for (const auto &indx : deleted_indicies)
124             {
125                 Eigen::Vector<mp::cpp_bin_float_double, -1> res =
                        Eigen::Vector<mp::cpp_bin_float_double,
                        -1>::Zero(B.cols());
126                 for (int i = 0; i < indx; i++)
127                 {
128                     res += T(indx,
                            i).convert_to<mp::cpp_bin_float_double>() *
                            t_HNF.row(i);
129                 }
130
131                 t_HNF.row(indx) = res;
132             }
133
134             return t_HNF.cast<mp::cpp_int>();
135             ////////////////////////////////////////////////////////////////////
136         }
137     }
138     namespace CVP
139     {
140         Eigen::MatrixXd gram_schmidt_greedy;
141         Eigen::MatrixXd B_greedy;
142         int index_greedy;
143
144         Eigen::MatrixXd gram_schmidt_bb;
145
146         // Recursive body of greedy algorithm
147         // @return Eigen::VectorXd
148         // @param target vector for which lattice point is being
                searched for
149         Eigen::VectorXd greedy_recursive(const Eigen::VectorXd &target)
150         {
151             if (index_greedy == 0)
152             {
153                 return Eigen::VectorXd::Zero(target.rows());
154             }
155             index_greedy--;
156             Eigen::VectorXd b = B_greedy.col(index_greedy);
157             Eigen::VectorXd b_star =
                    gram_schmidt_greedy.col(index_greedy);
158             double x = target.dot(b_star) / b_star.dot(b_star);
159             double c = std::round(x);
160
161             return c * b + Algorithms::CVP::greedy_recursive(target - c
                    * b);
162         }
163
```

```
164
165         // Solves CVP using a greedy algorithm
166         // @return Eigen::VectorXd
167         // @param matrix input rational lattice basis that is linearly
               independent
168         // @param target vector for which lattice point is being
               searched for
169         Eigen::VectorXd greedy(const Eigen::MatrixXd &matrix, const
               Eigen::VectorXd &target)
170         {
171             B_greedy = matrix;
172             gram_schmidt_greedy = Algorithms::gram_schmidt(matrix,
                   false);
173             index_greedy = static_cast<int>(matrix.cols());
174
175             return greedy_recursive(target);
176         }
177
178         // Recursive body of branch and bound algorithm
179         // @return Eigen::VectorXd
180         // @param matrix input rational lattice basis that is linearly
               independent
181         // @param target vector for which lattice point is being
               searched for
182         Eigen::VectorXd branch_and_bound_recursive(const Eigen::MatrixXd
               &matrix, const Eigen::VectorXd &target)
183         {
184             if (matrix.cols() == 0)
185             {
186                 return Eigen::VectorXd::Zero(target.rows());
187             }
188             Eigen::MatrixXd B = matrix.block(0, 0, matrix.rows(),
                   matrix.cols() - 1);
189             Eigen::VectorXd b = matrix.col(matrix.cols() - 1);
190             Eigen::VectorXd b_star = gram_schmidt_bb.col(matrix.cols() -
                   1);
191
192             Eigen::VectorXd v = Algorithms::CVP::greedy(B, target);
193             double upper_bound = (target - v).norm();
194
195             double x_middle = std::round(target.dot(b_star) /
                   b_star.dot(b_star));
196
197             std::vector<int> X;
198             X.push_back(static_cast<int>(x_middle));
199
200             bool flag1 = true;
201             bool flag2 = true;
202
203             double x1 = x_middle + 1;
204             double x2 = x_middle - 1;
205             while (flag1 || flag2)
206             {
207                 #pragma omp parallel sections
208                 {
209                     #pragma omp section
210                     {
211                         if (flag1 && Utils::projection(B, target - x1 *
                               b).norm() <= upper_bound)
212                         {
213                             #pragma omp critical
214                             X.push_back(static_cast<int>(x1));
215                             x1++;
216                         }
217                         else
218                         {
219                             flag1 = false;
```

```
                    }
                }
                #pragma omp section
                {
                    if (flag2 && Utils::projection(B, target - x2 *
                       b).norm() <= upper_bound)
                    {
                        #pragma omp critical
                        X.push_back(static_cast<int>(x2));
                        x2--;
                    }
                    else
                    {
                        flag2 = false;
                    }
                }
            }
        }

        // #pragma omp parallel sections
        // {
        //     #pragma omp section
        //     {
        //         double x = x_middle + 1;
        //         while (Utils::projection(B, target - x *
           b).norm() <= upper_bound)
        //         {
        //             #pragma omp critical
        //             X.push_back(static_cast<int>(x));
        //             x++;
        //         }
        //     }
        //     #pragma omp section
        //     {
        //         double x = x_middle - 1;
        //         while (Utils::projection(B, target - x *
           b).norm() <= upper_bound)
        //         {
        //             #pragma omp critical
        //             X.push_back(static_cast<int>(x));
        //             x--;
        //         }
        //     }
        // }

        // double x = x_middle + 1;
        // while (Utils::projection(B, target - x * b).norm() <=
           upper_bound)
        // {
        //     X.push_back(static_cast<int>(x));
        //     x++;
        // }
        // x = x_middle - 1;
        // while (Utils::projection(B, target - x * b).norm() <=
           upper_bound)
        // {
        //     X.push_back(static_cast<int>(x));
        //     x--;
        // }

        std::vector<Eigen::VectorXd> V;
        for (const int &x : X)
        {
            Eigen::VectorXd res = x * b +
               Algorithms::CVP::branch_and_bound(B, target - x * b);
            V.push_back(res);
        }
```

```
281             return Utils::closest_vector(V, target);
282         }
283 
284         // Solves CVP using a branch and bound algorithm
285         // @return Eigen::VectorXd
286         // @param matrix input rational lattice basis that is linearly
                independent
287         // @param target vector for which lattice point is being
                searched for
288         Eigen::VectorXd branch_and_bound(const Eigen::MatrixXd &matrix,
                const Eigen::VectorXd &target)
289         {
290             gram_schmidt_bb = Algorithms::gram_schmidt(matrix, false);
291 
292             return branch_and_bound_recursive(matrix, target);
293         }
294     }
295     // Computes Gram Schmidt orthogonalization
296     // @return Eigen::MatrixXd
297     // @param matrix input matrix
298     // @param normalize indicates whether to normalize output vectors
299     // @param delete_zero_rows indicates whether to delete zero rows
300     Eigen::MatrixXd gram_schmidt(const Eigen::MatrixXd &matrix, bool
            delete_zero_rows)
301     {
302         std::vector<Eigen::VectorXd> basis;
303 
304         for (const auto &vec : matrix.colwise())
305         {
306             Eigen::VectorXd projections =
                    Eigen::VectorXd::Zero(vec.size());
307 
308             for (int i = 0; i < basis.size(); i++)
309             {
310                 double inner1;
311                 double inner2;
312                 Eigen::MatrixXd basis_vector = basis[i];
313                 #pragma omp parallel sections
314                 {
315                     #pragma omp section
316                     {
317                         inner1 = std::inner_product(vec.data(),
                                vec.data() + vec.size(), basis_vector.data(),
                                0.0);
318                     }
319                     #pragma omp section
320                     {
321                         inner2 = std::inner_product(basis_vector.data(),
                                basis_vector.data() + basis_vector.size(),
                                basis_vector.data(), 0.0);
322                     }
323                 }
324                 projections += (inner1 / inner2) * basis_vector;
325             }
326 
327             Eigen::VectorXd result = vec - projections;
328 
329             if (delete_zero_rows)
330             {
331                 bool is_all_zero = result.isZero(1e-3);
332                 if (!is_all_zero)
333                 {
334                     basis.push_back(result);
335                 }
336             }
337             else
338             {
```

```
                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;
    }
}
```

Utils.cpp

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

namespace mp = boost::multiprecision;

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

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

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

        std::tuple<mp::cpp_int, mp::cpp_int, mp::cpp_int> gcd_result =
           gcd_extended(a, b);
        mp::cpp_int g, x, y;
        std::tie(g, x, y) = gcd_result;

        Eigen::Matrix<mp::cpp_int, 2, 2> U;
        U << x, -b / g, y, a / g;

```

```
43
44          Eigen::Matrix<mp::cpp_int, -1, 2> temp_matrix(H.rows(), 2);
45          temp_matrix.col(0) = H_first_col;
46          temp_matrix.col(1) = b_column;
47          Eigen::Matrix<mp::cpp_int, -1, 2> temp_result = temp_matrix * U;
48
49          Eigen::Vector<mp::cpp_int, -1> h_stroke =
               temp_result.col(0).tail(temp_result.rows() - 1);
50          Eigen::Vector<mp::cpp_int, -1> b_double_stroke =
               temp_result.col(1).tail(temp_result.rows() - 1);
51
52          b_double_stroke = reduce(b_double_stroke, H_stroke);
53
54          Eigen::Matrix<mp::cpp_int, -1, -1> H_double_stroke =
               add_column(H_stroke, b_double_stroke);
55
56          h_stroke = reduce(h_stroke, H_double_stroke);
57
58          Eigen::Matrix<mp::cpp_int, -1, -1> result(H.rows(), H.cols());
59
60          result(0, 0) = g;
61          result.col(0).tail(result.cols() - 1) = h_stroke;
62          result.row(0).tail(result.rows() - 1).setZero();
63          result.block(1, 1, H_double_stroke.rows(),
               H_double_stroke.cols()) = H_double_stroke;
64
65          return result;
66      }
67
68      // Function for computing HNF, reduces elements of vector modulo
           diagonal elements of matrix
69      // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
70      // @param vector vector to be reduced
71      // @param matrix input matrix
72      Eigen::Vector<mp::cpp_int, -1> reduce(const
           Eigen::Vector<mp::cpp_int, -1> &vector, const
           Eigen::Matrix<mp::cpp_int, -1, -1> &matrix)
73      {
74          Eigen::Vector<mp::cpp_int, -1> result = vector;
75          for (int i = 0; i < result.rows(); i++)
76          {
77              Eigen::Vector<mp::cpp_int, -1> matrix_column = matrix.col(i);
78              mp::cpp_int t_vec_elem = result(i);
79              mp::cpp_int t_matrix_elem = matrix(i, i);
80
81              mp::cpp_int x;
82              if (t_vec_elem >= 0)
83              {
84                  x = (t_vec_elem / t_matrix_elem);
85              }
86              else
87              {
88                  x = (t_vec_elem - (t_matrix_elem - 1)) / t_matrix_elem;
89              }
90
91              result -= matrix_column * x;
92          }
93          return result;
94      }
95
96      // Generates random matrix with full row rank (all rows are linearly
           independent)
97      // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
98      // @param m number of rows, must be greater than one and less than
           or equal to the parameter n
99      // @param n number of columns, must be greater than one and greater
           than or equal to the parameter m
```

```
100     // @param lowest lowest generated number, must be lower than lowest
            parameter by at least one
101     // @param highest highest generated number, must be greater than
            lowest parameter by at least one
102     Eigen::Matrix<mp::cpp_int, -1, -1>
            generate_random_matrix_with_full_row_rank(const int m, const int
            n, int lowest, int highest)
103     {
104         if (m > n)
105         {
106             throw std::invalid_argument("m must be less than or equal
                    n");
107         }
108         if (m < 1 || n < 1)
109         {
110             throw std::invalid_argument("Number of rows or columns
                    should be greater than one");
111         }
112         if (highest - lowest < 1)
113         {
114             throw std::invalid_argument("highest parameter must be
                    greater than lowest parameter by at least one");
115         }
116         std::random_device rd;
117         std::mt19937 gen(rd());
118         std::uniform_int_distribution<int> dis (lowest, highest);
119
120         Eigen::Matrix<int, -1, -1> matrix = Eigen::Matrix<int, -1,
                -1>::NullaryExpr(m, n, [&]()
121                                                                 { return
                                                                    dis(gen);
                                                                    });
122
123         Eigen::FullPivLU<Eigen::MatrixXd>
                lu_decomp(matrix.cast<double>());
124         auto rank = lu_decomp.rank();
125
126         while (rank != m)
127         {
128             matrix = Eigen::Matrix<int, -1, -1>::NullaryExpr(m, n, [&]()
129                                                 { return dis(gen); });
130
131             lu_decomp.compute(matrix.cast<double>());
132             rank = lu_decomp.rank();
133         }
134
135         return matrix.cast<mp::cpp_int>();
136     }
137
138     // Generates random matrix
139     // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
140     // @param m number of rows, must be greater than one
141     // @param n number of columns, must be greater than one
142     // @param lowest lowest generated number, must be lower than lowest
            parameter by at least one
143     // @param highest highest generated number, must be greater than
            lowest parameter by at least one
144     Eigen::Matrix<mp::cpp_int, -1, -1> generate_random_matrix(const int
            m, const int n, int lowest, int highest)
145     {
146         if (m < 1 || n < 1)
147         {
148             throw std::invalid_argument("Number of rows or columns
                    should be greater than one");
149         }
150         if (highest - lowest < 1)
151         {
```

```
152             throw std::invalid_argument("highest parameter must be
                    greater than lowest parameter by at least one");
153         }
154
155         std::random_device rd;
156         std::mt19937 gen(rd());
157         std::uniform_int_distribution<int> dis (lowest, highest);
158
159         Eigen::Matrix<int, -1, -1> matrix = Eigen::Matrix<int, -1,
                -1>::NullaryExpr(m, n, [&]()
160                                                                 { return
                                                                    dis(gen);
                                                                    });
161
162         return matrix.cast<mp::cpp_int>();
163     }
164
165     // Returns matrix that consist of linearly independent columns of
            input matrix and othogonalized matrix
166     // @param matrix input matrix
167     // @return std::tuple<Eigen::Matrix<boost::multiprecision::cpp_int,
            -1, -1>, Eigen::Matrix<boost::multiprecision::cpp_rational, -1,
            -1>>
168     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)
169     {
170         std::vector<Eigen::Vector<mp::cpp_rational, -1>> basis;
171         std::vector<int> indexes;
172
173         int counter = 0;
174         for (const Eigen::Vector<mp::cpp_rational, -1> &vec :
                matrix.cast<mp::cpp_rational>().colwise())
175         {
176             Eigen::Vector<mp::cpp_rational, -1> projections =
                    Eigen::Vector<mp::cpp_rational, -1>::Zero(vec.size());
177
178             for (int i = 0; i < basis.size(); i++)
179             {
180                 Eigen::Vector<mp::cpp_rational, -1> basis_vector =
                        basis[i];
181                 mp::cpp_rational inner1;
182                 mp::cpp_rational inner2;
183                 #pragma omp parallel sections
184                 {
185                     #pragma omp section
186                     {
187                         inner1 = std::inner_product(vec.data(),
                                vec.data() + vec.size(), basis_vector.data(),
                                mp::cpp_rational(0.0));
188                     }
189                     #pragma omp section
190                     {
191                         inner2 = std::inner_product(basis_vector.data(),
                                basis_vector.data() + basis_vector.size(),
                                basis_vector.data(), mp::cpp_rational(0.0));
192                     }
193                 }
194                 mp::cpp_rational coef = inner1 / inner2;
195                 projections += basis_vector * coef;
196             }
197
198             Eigen::Vector<mp::cpp_rational, -1> result = vec -
                    projections;
199
200             bool is_all_zero = result.isZero(1e-3);
```

```
201             if (!is_all_zero)
202             {
203                 basis.push_back(result);
204                 indexes.push_back(counter);
205             }
206             counter++;
207         }
208 
209         Eigen::Matrix<mp::cpp_int, -1, -1> result(matrix.rows(),
                indexes.size());
210         Eigen::Matrix<mp::cpp_rational, -1, -1>
                gram_schmidt(matrix.rows(), basis.size());
211 
212         for (int i = 0; i < indexes.size(); i++)
213         {
214             result.col(i) = matrix.col(indexes[i]);
215             gram_schmidt.col(i) = basis[i];
216         }
217         return std::make_tuple(result, gram_schmidt);
218     }
219 
220     // 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, cobsisting of Gram Schmidt coefficients
221     // @param matrix input matrix
222     // @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>>
223     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)
224     {
225         std::vector<Eigen::Vector<mp::cpp_rational, -1>> basis;
226         std::vector<int> indicies;
227        std::vector<int> deleted_indicies;
228         Eigen::Matrix<mp::cpp_rational, -1, -1> T =
                Eigen::Matrix<mp::cpp_rational, -1,
                -1>::Identity(matrix.rows(), matrix.rows());
229 
230         int counter = 0;
231         for (const Eigen::Vector<mp::cpp_rational, -1> &vec :
                matrix.cast<mp::cpp_rational>().rowwise())
232         {
233             Eigen::Vector<mp::cpp_rational, -1> projections =
                    Eigen::Vector<mp::cpp_rational, -1>::Zero(vec.size());
234             for (int i = 0; i < basis.size(); i++)
235             {
236                 Eigen::Vector<mp::cpp_rational, -1> basis_vector =
                        basis[i];
237                 mp::cpp_rational inner1;
238                 mp::cpp_rational inner2;
239                 #pragma omp parallel sections
240                 {
241                     #pragma omp section
242                     {
243                         inner1 = std::inner_product(vec.data(),
                                vec.data() + vec.size(), basis_vector.data(),
                                mp::cpp_rational(0.0));
244                     }
245                     #pragma omp section
246                     {
247                         inner2 = std::inner_product(basis_vector.data(),
                                basis_vector.data() + basis_vector.size(),
                                basis_vector.data(), mp::cpp_rational(0.0));
248                     }
249                 }
```

```
250                 mp::cpp_rational u_ij = 0;
251                 if (!inner1.is_zero())
252                 {
253                     u_ij = inner1 / inner2;
254                     projections += u_ij * basis_vector;
255                     T(counter, i) = u_ij;
256                 }
257             }
258 
259             Eigen::Vector<mp::cpp_rational, -1> result = vec -
                    projections;
260 
261             bool is_all_zero = result.isZero(1e-3);
262             if (!is_all_zero)
263             {
264                 indicies.push_back(counter);
265             }
266             else
267             {
268                 deleted_indicies.push_back(counter);
269             }
270             basis.push_back(result);
271             counter++;
272         }
273 
274         Eigen::Matrix<mp::cpp_int, -1, -1> result(indicies.size(),
                matrix.cols());
275         for (int i = 0; i < indicies.size(); i++)
276         {
277             result.row(i) = matrix.row(indicies[i]);
278         }
279         return std::make_tuple(result, indicies, deleted_indicies, T);
280     }
281 
282     // Extended GCD algorithm, returns tuple of g, x, y such that xa +
            yb = g
283     // @return std::tuple<boost::multiprecision::cpp_int,
            boost::multiprecision::cpp_int, boost::multiprecision::cpp_int>
284     // @param a first number
285     // @param b second number
286     std::tuple<mp::cpp_int, mp::cpp_int, mp::cpp_int>
            gcd_extended(mp::cpp_int a, mp::cpp_int b)
287     {
288         if (a == 0)
289         {
290             return std::make_tuple(b, 0, 1);
291         }
292         mp::cpp_int gcd, x1, y1;
293         std::tie(gcd, x1, y1) = gcd_extended(b % a, a);
294 
295         mp::cpp_int x = y1 - (b / a) * x1;
296         mp::cpp_int y = x1;
297 
298         return std::make_tuple(gcd, x, y);
299     }
300 
301     // Generates random matrix with full column rank (all columns are
            linearly independent)
302     // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
303     // @param m number of rows, must be greater than one and greater
            than or equal to the parameter n
304     // @param n number of columns, must be greater than one and lower
            than or equal to the parameter m
305     // @param lowest lowest generated number, must be lower than lowest
            parameter by at least one
306     // @param highest highest generated number, must be greater than
            lowest parameter by at least one
```

```
307     Eigen::MatrixXd generate_random_matrix_with_full_column_rank(const
            int m, const int n, int lowest, int highest)
308     {
309         if (m < n)
310         {
311             throw std::invalid_argument("m must be less than or equal
                    n");
312         }
313         if (m < 1 || n < 1)
314         {
315             throw std::invalid_argument("Number of rows or columns
                    should be greater than one");
316         }
317         if (highest - lowest < 1)
318         {
319             throw std::invalid_argument("highest parameter must be
                    greater than lowest parameter by at least one");
320         }
321         std::random_device rd;
322         std::mt19937 gen(rd());
323         std::uniform_int<int> dis (lowest, highest);
324 
325         Eigen::MatrixXd matrix = Eigen::MatrixXd::NullaryExpr(m, n, [&]()
326                                                                 { return
                                                                    dis(gen);
                                                                    });
327 
328         Eigen::FullPivLU<Eigen::MatrixXd> lu_decomp(matrix);
329         auto rank = lu_decomp.rank();
330 
331         while (rank != n)
332         {
333             matrix = Eigen::MatrixXd::NullaryExpr(m, n, [&]()
334                                                   { return dis(gen); });
335 
336             lu_decomp.compute(matrix);
337             rank = lu_decomp.rank();
338         }
339 
340         return matrix;
341     }
342 
343     // Generates random array
344     // @return Eigen::VectorXd
345     // @param m number of rows, must be greater than one
346     // @param lowest lowest generated number, must be lower than lowest
            parameter by at least one
347     // @param highest highest generated number, must be greater than
            lowest parameter by at least one
348     Eigen::VectorXd generate_random_vector(const int m, double lowest,
            double highest)
349     {
350         if (m < 1)
351         {
352             throw std::invalid_argument("Number of rows or columns
                    should be greater than one");
353         }
354         if (highest - lowest < 1)
355         {
356             throw std::invalid_argument("highest parameter must be
                    greater than lowest parameter by at least one");
357         }
358         std::random_device rd;
359         std::mt19937 gen(rd());
360         std::uniform_real_distribution<double> dis(lowest, highest);
361 
362         Eigen::VectorXd array = Eigen::VectorXd::NullaryExpr(m, [&]()
```

```
363                                                                 { return
                                                                       dis(gen);
                                                                       });
364
365         return array;
366     }
367
368     // Computes projection of a vector onto a matrix using equations
        from Gram Schmidt computing
369     // @return Eigen::VectorXd
370     // @param matrix input matrix
371     // @param vector input vector
372     Eigen::VectorXd projection(const Eigen::MatrixXd &matrix, const
        Eigen::VectorXd &vector)
373     {
374         Eigen::MatrixXd t_matrix(matrix.rows(), matrix.cols() + 1);
375         t_matrix << matrix, vector;
376         std::vector<Eigen::VectorXd> basis;
377
378         for (const Eigen::VectorXd &vec : t_matrix.colwise())
379         {
380             Eigen::VectorXd projections =
                    Eigen::VectorXd::Zero(vec.size());
381
382             for (int i = 0; i < basis.size(); i++)
383             {
384                 Eigen::VectorXd basis_vector = basis[i];
385                 double inner1;
386                 double inner2;
387                 #pragma omp parallel sections
388                 {
389                     #pragma omp section
390                     {
391                         inner1 = std::inner_product(vec.data(),
                            vec.data() + vec.size(), basis_vector.data(),
                            0.0);
392                     }
393                     #pragma omp section
394                     {
395                         inner2 = std::inner_product(basis_vector.data(),
                            basis_vector.data() + basis_vector.size(),
                            basis_vector.data(), 0.0);
396                     }
397                 }
398                 double coef = inner1 / inner2;
399                 projections += basis_vector * coef;
400             }
401
402             Eigen::VectorXd t_result = vec - projections;
403
404             basis.push_back(t_result);
405         }
406
407         Eigen::VectorXd result = basis[basis.size() - 1];
408
409         return result;
410     }
411
412     // Finds vector that is closest to other vectors in matrix
413     // @return Eigen::VectorXd
414     // @param matrix input matrix
415     // @param vector input vector
416     Eigen::VectorXd closest_vector(const std::vector<Eigen::VectorXd>
        &matrix, const Eigen::VectorXd &vector)
417     {
418         Eigen::VectorXd closest = matrix[0];
419         for (const auto &v : matrix)
```

```
420             {
421                 if ((vector - v).norm() <= (vector - closest).norm())
422                 {
423                     closest = v;
424                 }
425             }
426             return closest;
427         }
428     }
```