МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

«Национальный исследовательский Нижегородский государственный университет им. Н. И. Лобачевского»

Институт информационных технологий математики и механики

Кафедра теоретической, компьютерной и экспериментальной механики

Направление подготовки 09.03.04 Программная инженерия

*Производственная практика*

*по получению профессиональных умений и опыта профессиональной деятельности*

**Программирование базовых матричных алгоритмов**

Исполнитель: студент 381808-2 группы

Огнев Денис Вячеславович

Руководитель: доцент кафедры АГДМ,

Веселов Сергей Иванович

Нижний Новгород, 2021 г.

**Оглавление**

[**1** **Введение** 2](#_Toc73890091)

[**2** **Постановка задачи** 3](#_Toc73890092)

[**3** **Разбор алгоритмов и написанной программы** 4](#_Toc73890093)

[3.1 Определение Эрмитовой нормальной формы 4](#_Toc73890094)

[3.2 Алгоритм для матриц с полным рангом строки 4](#_Toc73890095)

[3.3 Алгоритм для общего случая 7](#_Toc73890096)

[3.4 Проверка с помощью WolframAlpha 8](#_Toc73890097)

[3.5 Решение некоторых задач 9](#_Toc73890098)

[3.6 Определение проблемы ближайшего вектора и способы ее решения 11](#_Toc73890099)

[3.7 Жадный метод: Алгоритм ближайшей плоскости Бабая 12](#_Toc73890100)

[3.8 Метод Ветвей и Границ: Алгоритм Перечисления 13](#_Toc73890101)

[**4** **Разбор основных модулей программы** 16](#_Toc73890102)

[**5** **Заключение** 16](#_Toc73890103)

[**6** **Полный исходный код** 17](#_Toc73890104)

[**7** **Ссылки на источники** 24](#_Toc73890105)

# **Введение**

В данной работе рассматриваются различные решеточные алгоритмы. Описанные алгоритмы работают за полиномиальное время позволяют решать различные задачи, например проблему эквивалентности (образуют ли два базиса одинаковую решетку) или проблему включения (включение вектора в решетку). Для решения многих таких задач применяется Эрмитова нормальная форма. Эрмитова нормальная форма также используется при решении задач общей алгебры, линейного программирования, криптографии и может быть использована для решения систем линейных диофантовых уравнений.

Также в данной работе рассматривается проблема ближайшего вектора (ПБВ), для ее решения рассматриваются такие алгоритмы, как Жадный метод: Алгоритм ближайшей плоскости Бабая, который работает за полиномиальное время и решает ПБВ за полиномиальное время, но дает только приближенные решения, и метод Ветвей и Границ: алгоритм перечисления, который точно решает ПБВ, но работает за суперэкспоненциальное время

# **Постановка задачи**

После перевода и анализа предложенной статьи необходимо запрограммировать изученные алгоритмы и решить перечисленные проблемы. Для программирования алгоритмов был выбран язык Python версии 3.9.2. Плюсом языка Python является библиотека Numpy, которую мы будем использовать для работы с матрицами и векторами. Для проверки нахождения ЭНФ можно воспользоваться сайтом WolframAlpha, а также написать собственный модуль для облегчения проверки (чтобы не вводить каждый раз матрицу на сайте), используя API WolframAlpha, который будет отсылать матрицу и принимать ее ЭНФ.

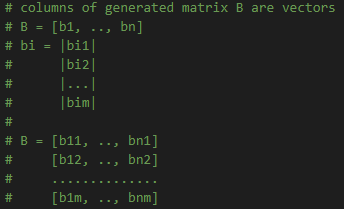
Выделим 3 основных этапа:

1. Программирование алгоритма для вычисления ЭНФ
2. Программирование двух алгоритмов для решения проблемы ближайшего вектор
3. Применение данных алгоритмов

Разберем каждый этап.

# **Разбор алгоритмов и написанной программы**

При программировании функций для ЭНФ использовалась данная структура хранения матриц:



## Определение Эрмитовой нормальной формы

*Невырожденная матрица* **B** = [**b**1*, ...,* **b***n*] ∈ *m*×*n* является Эрмитовой нормальной формой (ЭНФ), если

* Существует 1 ≤ *i*1 *< ... < ih* ≤ *m такое, что bi,j* ≠ 0 ⇒ (*j < h*) ∧ (*i* ≥ *ij*) *(строго убывающая высота столбца).*
* *Для всех , т.е. все элементы в строках ij приведены по модулю .*

## Алгоритм для матриц с полным рангом строки

Дана матрица . Предположим, что у нас есть процедура AddColumn, которая работает за полиномиальное время и принимает на вход квадратную невырожденную ЭНФ матрицы и вектор **b**, а возвращает ЭНФ матрицы [**H|b**]. ЭНФ от **B** может быть вычислена следующим образом:

1. Применить алгоритм Грама-Шмидта к столбцам **B**, чтобы найти *m* линейно независимых столбцов. Пусть **B’** – матрица размера , заданная этими столбцами.
2. Вычислить *d*=det(**B’**), используя алгоритм Грама-Шмидта или любую другую процедуру с полиномиальным временем. Пусть будет диагональной матрицей с *d* на диагонали.
3. Для *i*=1,…,*n* пусть результат применения AddColumn ко входу .
4. Вернуть .

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 x m. Для нахождения этих строк можно использовать алгоритм Грама-Шмитда: если = 0, то i-ая строка является линейной комбинацией других строк, и ее необходимо выбросить. Мы будем использовать модуль Sympy языка Python и его функцию rref (row reduced echeleon form) для получения индексов линейно независимых столбцов и составления матрицы из них. Полученная матрица будет названа **B**’
2. Для вычисления det напишем функцию det\_by\_gram\_schmidt, которая принимает на вход матрицу и вычисляет det по формуле - сумма произведений мод всех элементов, полученных после применения ортогонализации Грама-Шмидта. Матрица будет единичной матрицей размера m x m, умноженной на определитель. В результате все диагональные элементы будут равны d
3. Применяем функцию AddColumn к и первому столбцу матрицы **B -** , получаем , повторяем всеми столбцами, получаем
4. является ЭНФ(**B**)

Алгоритм AddColumn на вход принимает квадратную невырожденную ЭНФ матрицы и вектор и работает следующим образом. Если m = 0, то тут ничего не надо делать, и мы можем сразу выйти с возвращением **H**. В другом случае, пусть и и дальше:

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

Разберем подпункты:

1. Функция gcd\_extended принимает a, b, вычисляет наибольший общий делитель и целые x, y такие, что xa + yb = g
2. Составляем матрицу и умножаем ее на матрицу, составленную из первого столбца **H** и столбца **b,** чтобы получить
3. Функция reduce принимает на вход матрицуи вектор, получает необходимый вектор из решетки от матрицы на входе, чтобы сократить элементы вектора по модулю диагональных элементов из матрицы. Применяем функцию reduce к **H**’ и **b**’’
4. Рекурсивно вызываем AddColumn, на вход отправляем **H**’ и **b**’’, получаем матрицу **H’’**.
5. Вызываем функцию reduce к **H**’’ и **h**’
6. Составляем необходимую матрица и возвращаем

Полученный код на Python:

def HNF\_full\_row\_rank(matrix):  # Hermite Normal Form of full rank matrix

    m = matrix.shape[0]

    n = matrix.shape[1]

    # inds - indexes of linear independent columns

    # number of such rows is equal to m, so B\_stroke always m x m

    temp, inds = sympy.Matrix(matrix).rref()

    B\_stroke = []

    for i in inds:

        B\_stroke.append(matrix.T[i])

    B\_stroke = (np.array(B\_stroke)).T

    det = round(det\_by\_gram\_schmidt(B\_stroke))

    H = np.eye(m, dtype=int) \* det

    for i in range(n):

        H = add\_column(H, matrix.T[i])

    num\_of\_additional\_zero\_vectors = n - m

    zero\_vector = np.array([np.zeros(m, dtype=int)])

    for i in range(num\_of\_additional\_zero\_vectors):

        H = np.append(H.T, zero\_vector, axis=0).T

    return H

def add\_column(H, b\_column):

    if (H.shape[1] == 0):

        return H

    a = H[0, 0]

    h = H[1:, 0]

    H\_stroke = H[1:, 1:]

    b = b\_column[0]

    b\_stroke = b\_column[1:]

    g, x, y = gcd\_extended(a, b)

    U = np.array([[x, -b / g], [y, a / g]])

    first\_column = np.append(np.array(a), h)

    second\_column = np.append(np.array(b), b\_stroke)

    temp\_matrix = np.array([first\_column, second\_column]).T

    temp\_result = np.dot(temp\_matrix, U)

    h\_stroke = temp\_result.T[0, 1:]

    b\_double\_stroke = temp\_result.T[1, 1:]

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

    H\_double\_stroke = add\_column(H\_stroke, b\_double\_stroke)

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

    length\_of\_zeros = H\_double\_stroke.shape[0]

    result = np.array(np.append(np.array([np.append(np.array(g), np.zeros(length\_of\_zeros, dtype=int))]), np.append(

        np.array([h\_stroke]), H\_double\_stroke.T, axis=0).T, axis=0), dtype=int)

    return result

def det\_by\_gram\_schmidt(matrix):

    result = 1

    matrix = gram\_schmidt(matrix.T)

    for v in matrix:

        result \*= norm(v)

    return result

def gcd\_extended(a, b):

    if a == 0:

        return b, 0, 1

    gcd, x1, y1 = gcd\_extended(b % a, a)

    x = y1 - (b//a) \* x1

    y = x1

    return gcd, x, y

def reduce(vector, matrix):

    for i in range(vector.shape[0]):

        while(vector[i] < 0):

            vector += matrix.T[i]

        while(vector[i] >= matrix[i, i]):

            vector -= matrix.T[i]

    return vector

## Алгоритм для общего случая

1. Запустить процесс ортогонализации Грамма-Шмидта к строкам из **B,** и пусть – это множество индексов, такое, что . Определим операцию проецирования . Заметим, что строки линейно независимы и любая другая строка может быть выражена как линейная комбинация предыдущих строк Следовательно, однозначна, когда ограничена к , и ее инверсия может быть легко вычислена, используя коэффициенты Грама-Шмидта .
2. Определить новую матрицу **B’**=, которая полного ранга, и запустить алгоритм, данный в предыдущем пункте, чтобы найти ЭНФ **B**’’ от **B’.**
3. Применить функцию обратную операции проецирования, , к ЭНФ, определенной в предыдущем шаге(**B’’**), к данной матрице **H**. Легко заметить, что входят в ЭНФ. Следовательно, **H** является ЭНФ **B**.

Алгоритм прост, но вызывает вопрос функция проецирования и обратная к ней. Функция projection получает с помощью функции rref модуля Sympy индекс линейно независимых строк и на их основе строит новую матрицу, после чего к этой матрице применяется алгоритм для нахождения ЭНФ, данный в прошлом разделе (т.к. полученная матрица полного ранга). Далее нам нужно восстановить удаленные строки. Т.к. они являются линейной комбинацией линейно независимых строк, то мы можем найти коэффициенты, на которые нужно умножить строки из матрицы **B**’ и после чего сложить их, чтобы получить нужную строку, которую необходимо добавить к **B**’.

Полученный код на Python:

def HNF(matrix):

    m = matrix.shape[0]

    n = matrix.shape[1]

    B\_stroke = projection(matrix)

    B\_double\_stroke = HNF\_full\_row\_rank(B\_stroke)

    result = inverse\_projection(B\_double\_stroke, matrix)

    return result

def projection(matrix):

    # inds - indexes of linear independent rows

    temp, inds = sympy.Matrix(matrix.T).rref()

    result = []

    for i in inds:

        result.append(matrix[i])

    result = np.array(result)

    return result

def inverse\_projection(HNF, matrix):

    # inds - indexes of linear independent rows

    temp, inds = sympy.Matrix(matrix.T).rref()

    basis = HNF

    independent\_vectors = []

    for i in inds:

        independent\_vectors.append(matrix[i])

    independent\_vectors = np.array(independent\_vectors)

    for i in range(matrix.shape[0]):

        if i not in inds:

            x = np.linalg.solve(independent\_vectors.T, matrix[i])

            result = np.zeros\_like(x)

            for j in range(HNF.shape[0]):

                result += HNF[j] \* x[j]

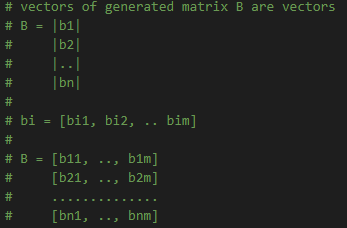
            result = np.array([result])

            basis = np.append(basis, result, axis=0)

    return basis

## Проверка с помощью WolframAlpha

WolframAlpha – мощный инструмент, в котором есть огромный набор вычислительных алгоритмов, в том числе и вычисление ЭНФ. Для языка Python существует модуль wolframalpha, который позволяет пользоваться алгоритмами в нашем коде.



WolframAlpha использует другое представление матриц, поэтому для проверки необходимо отправлять транспонированную матрицу, для этого у модуля Numpy есть свойство T. Также WolframAlpha вычисляет верхнетреугольное представление ЭНФ, поэтому для проверки нужно транспонировать ЭНФ найденное нами.

Функция проверки принимает массив и возвращает строку с найденной ЭНФ.

Полный под на Python:

import wolframalpha

def get\_HNF\_from\_wolfram\_alpha(matrix):

    app\_id = '55HY54-H7V5VVRTGY'

    client = wolframalpha.Client(app\_id)

    query = "HermiteDecomposition[{{"

    for i in range(matrix.shape[0]):

        for j in range(matrix.shape[1]):

            if (j == matrix.shape[1] - 1):

                query += str(matrix[i, j])

            else:

                query += str(matrix[i, j]) + ", "

        if (i == matrix.shape[0] - 1):

            query += "}}]"

        else:

            query += "}, {"

    res = client.query(query)

    answer = next(res.results).text

    answer = str(answer)

    index = answer.find("H =")

    result = answer[index:]

    result = result.replace("H = ", "")

    result = result.replace("(", "")

    result = result.replace(")", "")

    return result

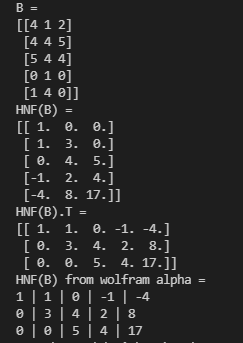
## Решение некоторых задач

Для проверки будем пользоваться верхнетреугольным определением ЭНФ.

**Основная проблема** Дано множество рациональных векторов **B**, мы хотим вычислить базис для решетки .

Эта проблема мгновенно решается (за полиномиальное время) путем вычисления ЭНФ(**B**).

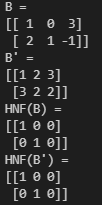
Пример: Дан массив B, найдем ЭНФ и сверим с WolframAlpha



**Проблема эквивалентности** Даны для базиса **B** и **B’**, мы хотим определить, образуют ли они одинаковую решетку

Эта проблема может решена за полиномиальное время, путем вычисления **H** = ЭНФ(**B**) и **H**’ = ЭНФ(**B**’), и проверки равенства **H** и **H**’.

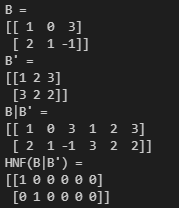
Пример:



**Объединение решеток** Даны два базиса **B** и **B**’, мы хотим определить базис для наименьшей решетки, включающей и , и .

Легко увидеть, что эта решетка сгенерирована от [**B** | **B**’], значит базис для решетки может быть легко вычислен, как ЭНФ([**B** | **B**’]).

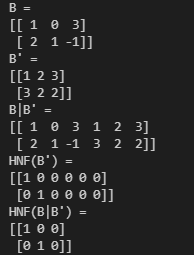
Пример:



**Проблема содержания** Даны два базиса **B** и **B**’, мы хотим определить верно ли, что подрешетка , т.е., .

Эта проблема легко сводится к проблеме объединения и равенства: тогда и только тогда, когда . Итого, чтобы проверить включение нам нужно только вычислить ЭНФ([**B | B**’]) и ЭНФ(**B**) и проверить эквивалентность базисов ЭНФ.

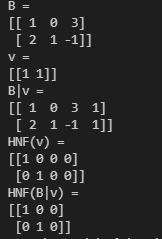
Пример:



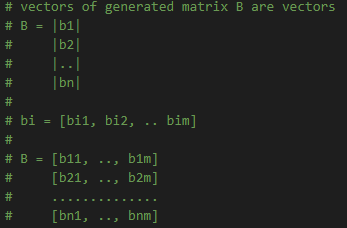
**Проблема включения** Дана решетка **B** и вектор **v**, мы хотим определить верно ли, что .

Это немедленно сводится к проблеме содержания путем проверки верно ли, что . Если нам нужно проверить включение для множества векторов тогда удобно сперва вычислить **H** = ЭНФ(**B**), и потом для каждого *i* проверять **H** = ЭНФ([**H |** ]). Заметим, что ЭНФ([**H |** ]) может быть легко вычислена намного быстрее, чем ЭНФ для матричного вычисления, потому что большинство матриц уже являются Эрмитовой Нормальной Формой.

Пример:



## Определение проблемы ближайшего вектора и способы ее решения

При программировании использовалась данная структура хранения матриц: 

Рассмотрим Проблему Ближайшего Вектора(ПБВ): Дан базис решетки и целевой вектор , найти точку решетки **Bx** такую, что минимально. Это задача оптимизации (минимизации) с допустимыми решениями, заданными всеми целочисленными векторами , и целевой функцией f(**x**) = .

Пусть **B = [B**’, **b**] и **x** = (**x**’, *x*), где , ,и .

Заметим, что если вы зафиксируете значение *x*, то задача ПБВ (**B, t**) потребует найти значение такое, что минимизировано. Это также экземпляр ПБВ (**B’, t’**) с модифицированным , и решеткой меньшего размера . В частности, пространство решений сейчас состоит из (n – 1) целочисленных переменных **x**’. Это говорит о том, что можно решить ПБВ путем установки значения **x** по одной координате за раз.

Есть несколько способов превратить этот подход к уменьшению размерности в алгоритм, используя некоторые стандартные методы алгоритмического программирования. Простейшие техники такие:

* жадный метод, который выдает приближенные значения, но работает за полиномиальное время, и
* метод ветвей и границ, который выдает точные решение за суперэкспоненциальное время.

Оба метода основаны на очень простой нижней оценке целевой функции:

## Жадный метод: Алгоритм ближайшей плоскости Бабая

Жадный метод состоит из выбора переменных по одной, определяющих пространство решений, каждый раз выбирая значение, которые выглядит более многообещающим. В нашем случае, выберем значение *x*, которое дает наименьшее возможное значение для нижней границы . Напомним, что и , и что для любого фиксированного значения *x*, ПБВ (**B, t**) сводится к ПБВ (**B’, t’**), где . Используя для нижней границы, мы хотим выбрать значение *x* такое, что

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

где компонента вектора **b**, ортогональная другим базисным векторам. Вот и все! Полный алгоритм приведен ниже:

Greedy([], **t**) = **0**

Greedy([**B** *,***b**],**t**) = *c* · **b** + Greedy(**B**, **t** − *c* · **b**)

где **b**∗ = **b**⊥**B**

*x* =

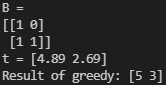
*c* =

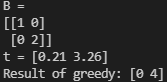
Как уже сказано выше, весь алгоритм состоит в сокращении размерности путем установки одной координаты за раз и рекурсивных вызовах.

Функция projection принимает на вход вектор и матрицу, а возвращает вектор , ортогональный другим базисным векторам.

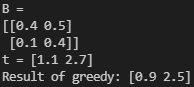
Алгоритм прост и его программирование не должно вызвать сильных затруднений.

Пример работы:





Более сложный случай:



Полный код на Python:

def projection(matrix, vector):

    projection = np.zeros(vector.shape)

    for i in range(0, matrix.shape[0]):

        projection += (scalar(vector, matrix[i]) /

                       scalar(matrix[i], matrix[i])) \* matrix[i]

    result = vector - projection

    return result

def greedy(matrix, target):

    if (matrix.shape[0] == 0):

        return 0

    b = matrix[-1]

    matrix = matrix[:-1]

    b\_star = projection(matrix, b)

    x = scalar(target, b\_star) / scalar(b\_star, b\_star)

    c = round(x)

    return c \* b + greedy(matrix, target - c \* b)

## Метод Ветвей и Границ: Алгоритм Перечисления

Структура похожа на жадный алгоритм, но вместо жадной установки на наиболее подходящее значение (то есть на то, для которого нижняя граница расстояния 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 (расстояние от цели до начала координат) в качестве верхней границы. Обычно лучше использовать жадный алгоритм, чтобы найти приближенное решение **v** = Greedy(**B, t**), и использовать в качестве верхней границы. Как только верхняя граница *u* установлена, можно ограничить переменную *x* такими значениями, что .

Окончательный алгоритм похож на жадный и описан ниже:

Branch&Bound([],**t**) = **0**

Branch&Bound([**B***,***b**],**t**) = closest(*V* ,**t**)

где **b**∗ = **b**⊥**B**

**v** = Greedy(**B**,**t**)

*X* = {}

*V* = {*x* · **b** + Branch&Bound(**B***,***t** − *x* · **b**):*x* ∈ *X*}

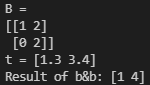
где closest(*V, t*) выбирает вектор в ближайший к цели **t.**

Как и для жадного алгоритма, производительность алгоритма Ветвей и Границ может быть произвольно плохой, если мы сперва не сократим базисы.

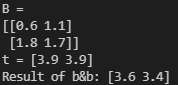
Сложность алгоритма заключается в нахождении множества X. Его можно найти, используя выражение, выведенное в прошлом алгоритме: . С помощью него мы найдем x, который точно удовлетворяет множеству, а затем будет увеличивать/уменьшать до тех пор, пока выполняется условие

Функция closest\_vector на вход принимает матрицу и вектор, и на основании расстояний между векторами возвращает ближайший к входному вектору вектор из матрицы.

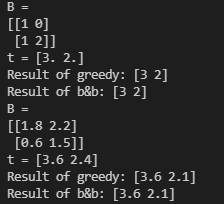
Пример работы:



Более сложный случай:



Сравнение с Greedy:



Полный под на Python:

def closest\_vector(matrix, vector):

    closest = matrix[0]

    for v in matrix:

        if (distance\_between\_two\_vectors(vector, v) < distance\_between\_two\_vectors(vector, closest)):

            closest = v

    return closest

def branch\_and\_bound(matrix, target):

    if (matrix.shape[0] == 0):

        return 0

    b = matrix[-1]

    matrix = matrix[:-1]

    v = greedy(matrix, target)

    b\_star = projection(matrix, b)

    upper\_bound = math.ceil(norm(target - v))

    x\_array = np.array([2, 3])

    v\_array = []

    x\_middle = math.floor(scalar(target, b\_star) / scalar(b\_star, b\_star))

    lower\_bound = norm(projection(matrix, target - x\_middle \* b))

    x = x\_middle

    temp\_lower\_bound = lower\_bound

    while (temp\_lower\_bound <= upper\_bound):

        x += 1

        temp\_lower\_bound = norm(projection(matrix, target - x \* b))

    x\_highest = x

    x = x\_middle

    temp\_lower\_bound = lower\_bound

    while (temp\_lower\_bound <= upper\_bound):

        x -= 1

        temp\_lower\_bound = norm(projection(matrix, target - x \* b))

    x\_lowest = x + 1

    for i in range(x\_lowest, x\_highest):

        x\_array = np.append(x\_array, i)

    for x in x\_array:

        res = x \* b + branch\_and\_bound(matrix, target - x \* b)

        v\_array.append(res)

    v\_array = np.array(v\_array)

    return closest\_vector(v\_array, target)

# **Разбор основных модулей программы**

Программа состоит из 5 модулей.

* В модуле HNF.py находятся функции для нахождения ЭНФ матрицы с полным рангом строки, функции AddColumn и функции нахождения ЭНФ в общем случае.
* В модуле CVP.py находятся функции для решения задачи ближайшего вектора – greedy, реализующая жадный алгоритм, и branch\_and\_bound, реализующая метод ветвей и границ.
* В модуле wolfram.py находится функция для проверки ЭНФ с помощью WolframAlpha
* В модуле utils.py находятся различные вспомогательные функции, такие как скалярное произведение векторов, норма вектора, функция для получения ранга матрицы, функция для проверки линейной независимости строк матрицы, алгортим ортогонализации Грама-Шмидта, алгоритм получение детерминанта через Грама-Шмидта, расширенный НОД алгоритм, функция сокращения вектора относительно диагональных элементов матрицы, функция проекции и обратной проекции, а также различные функции для создания случайных матриц.
* В модуле \_\_main\_\_.py находятся функции-примеры решения различных задач, а так же общие функции, находящие ЭНФ от случайных матриц, а также ближайший вектор для случайно заданного базиса и случайной цели.

# **Заключение**

Результатом работы является программа, позволяющая находить ЭНФ матриц, а также решать различные проблемы решеток, и решать проблему ближайшего вектора двумя способами.

# **Полный исходный код**

wolfram.py:

import wolframalpha

def get\_HNF\_from\_wolfram\_alpha(matrix):

    app\_id = '55HY54-H7V5VVRTGY'

    client = wolframalpha.Client(app\_id)

    query = "HermiteDecomposition[{{"

    for i in range(matrix.shape[0]):

        for j in range(matrix.shape[1]):

            if (j == matrix.shape[1] - 1):

                query += str(matrix[i, j])

            else:

                query += str(matrix[i, j]) + ", "

        if (i == matrix.shape[0] - 1):

            query += "}}]"

        else:

            query += "}, {"

    res = client.query(query)

    answer = next(res.results).text

    answer = str(answer)

    index = answer.find("H =")

    result = answer[index:]

    result = result.replace("H = ", "")

    result = result.replace("(", "")

    result = result.replace(")", "")

    return result

utils.py:

import numpy as np

import sympy

from sympy.matrices import matrices

from sympy.matrices.dense import Matrix

def scalar(vector1, vector2):

    s = 0

    for i in range(vector1.size):

        s += vector1[i] \* vector2[i]

    return s

def norm(vector):

    return np.sqrt(scalar(vector, vector))

def distance\_between\_two\_vectors(vector1, vector2):

    return np.sqrt(scalar(vector1 - vector2, vector1 - vector2))

def generate\_random\_linearly\_independent\_matrix(n, m, lowest, highest):

    if (n > m):

        raise("Number of vectors should be less or equal than their size")

    matrix = np.random.randint(lowest, highest, (n, m))

    while not check\_linear\_independent(matrix):

        matrix = np.random.randint(lowest, highest, (n, m))

    return matrix

def generate\_random\_linearly\_independent\_float\_matrix(n, m, lowest, highest):

    if (n > m):

        raise("Number of vectors should be less or equal than their size")

    matrix = np.random.uniform(lowest, highest, (n, m))

    for i in range(matrix.shape[0]):

        for j in range(matrix.shape[1]):

            matrix[i, j] = round(matrix[i, j], 1)

    while not check\_linear\_independent(matrix):

        matrix = np.random.uniform(lowest, highest, (n, m))

        for i in range(matrix.shape[0]):

            for j in range(matrix.shape[1]):

                matrix[i, j] = round(matrix[i, j], 1)

    return matrix

def generate\_random\_array(m, lowest, highest):

    array = np.random.uniform(lowest, highest, m)

    for i in range(array.shape[0]):

        array[i] = round(array[i], 1)

    return array

# matrix is full row rank if rank(matrix) == m

def generate\_random\_matrix\_with\_full\_row\_rank(m, n, lowest, highest):

    matrix = np.random.randint(lowest, highest, (m, n))

    rank = get\_matrix\_rank(matrix)

    while m != rank:

        matrix = np.random.randint(lowest, highest, (m, n))

        rank = get\_matrix\_rank(matrix)

    return matrix

def generate\_random\_matrix(m, n, lowest, highest):

    matrix = np.random.randint(lowest, highest, (m, n))

    return matrix

def generate\_random\_float\_matrix(m, n, lowest, highest):

    matrix = np.random.uniform(lowest, highest, (m, n))

    for i in range(matrix.shape[0]):

        for j in range(matrix.shape[1]):

            matrix[i, j] = round(matrix[i, j], 2)

    return matrix

def get\_matrix\_rank(matrix):

    return np.linalg.matrix\_rank(matrix)

def check\_linear\_independent(matrix):

    # inds - indexes of linear independent rows

    temp, inds = sympy.Matrix(matrix.T).rref()

    number\_of\_linear\_independent\_rows = len(inds)

    number\_of\_vectors = matrix.shape[0]

    if (number\_of\_linear\_independent\_rows != number\_of\_vectors):

        return False

    return True

def gram\_schmidt(matrix, normalize=False, delete\_zero\_rows=True):

    basis = []

    for vector in matrix:

        projections = np.sum(

            (scalar(vector, b) / scalar(b, b)) \* b for b in basis)

        r = vector - projections

        is\_all\_zero = np.all(r == 0)

        if (delete\_zero\_rows):

            if (not is\_all\_zero):

                if (normalize):

                    basis.append(r / scalar(r, r))

                else:

                    basis.append(r)

        else:

            if (normalize):

                basis.append(r / scalar(r, r))

            else:

                basis.append(r)

    return np.array(basis)

def det\_by\_gram\_schmidt(matrix):

    result = 1

    matrix = gram\_schmidt(matrix.T)

    for v in matrix:

        result \*= norm(v)

    return result

def gcd\_extended(a, b):

    if a == 0:

        return b, 0, 1

    gcd, x1, y1 = gcd\_extended(b % a, a)

    x = y1 - (b//a) \* x1

    y = x1

    return gcd, x, y

def sign(a):

    if np.sign(a) == 1 or np.sign(a) == 0:

        return True

    else:

        return False

def reduce(vector, matrix):

    for i in range(vector.shape[0]):

        while(vector[i] < 0):

            vector += matrix.T[i]

        while(vector[i] >= matrix[i, i]):

            vector -= matrix.T[i]

    return vector

def projection(matrix):

    # inds - indexes of linear independent rows

    temp, inds = sympy.Matrix(matrix.T).rref()

    result = []

    for i in inds:

        result.append(matrix[i])

    result = np.array(result)

    return result

def inverse\_projection(HNF, matrix):

    # inds - indexes of linear independent rows

    temp, inds = sympy.Matrix(matrix.T).rref()

    basis = HNF

    independent\_vectors = []

    for i in inds:

        independent\_vectors.append(matrix[i])

    independent\_vectors = np.array(independent\_vectors)

    for i in range(matrix.shape[0]):

        if i not in inds:

            x = np.linalg.solve(independent\_vectors.T, matrix[i])

            result = np.zeros\_like(x)

            for j in range(HNF.shape[0]):

                result += HNF[j] \* x[j]

            result = np.array([result])

            basis = np.append(basis, result, axis=0)

    return basis

def print\_matrix(matrix):

    res = str()

    for i in range(matrix.shape[0]):

        for j in range(matrix.shape[1]):

            if (j == matrix.shape[1] - 1):

                if (i != matrix.shape[0] - 1):

                    res += str(matrix[i, j]) + "\n"

                else:

                    res += str(matrix[i, j])

            else:

                res += str(matrix[i, j]) + " | "

    print(res)

CVP.py:

import math

from utils import \*

def projection(matrix, vector):

    projection = np.zeros(vector.shape)

    for i in range(0, matrix.shape[0]):

        projection += (scalar(vector, matrix[i]) /

                       scalar(matrix[i], matrix[i])) \* matrix[i]

    result = vector - projection

    return result

def closest\_vector(matrix, vector):

    closest = matrix[0]

    for v in matrix:

        if (distance\_between\_two\_vectors(vector, v) < distance\_between\_two\_vectors(vector, closest)):

            closest = v

    return closest

def greedy(matrix, target):

    if (matrix.shape[0] == 0):

        return 0

    b = matrix[-1]

    matrix = matrix[:-1]

    b\_star = projection(matrix, b)

    x = scalar(target, b\_star) / scalar(b\_star, b\_star)

    c = round(x)

    return c \* b + greedy(matrix, target - c \* b)

def branch\_and\_bound(matrix, target):

    if (matrix.shape[0] == 0):

        return 0

    b = matrix[-1]

    matrix = matrix[:-1]

    v = greedy(matrix, target)

    b\_star = projection(matrix, b)

    upper\_bound = math.ceil(norm(target - v))

    x\_array = np.array([2, 3])

    v\_array = []

    x\_middle = math.floor(scalar(target, b\_star) / scalar(b\_star, b\_star))

    lower\_bound = norm(projection(matrix, target - x\_middle \* b))

    x = x\_middle

    temp\_lower\_bound = lower\_bound

    while (temp\_lower\_bound <= upper\_bound):

        x += 1

        temp\_lower\_bound = norm(projection(matrix, target - x \* b))

    x\_highest = x

    x = x\_middle

    temp\_lower\_bound = lower\_bound

    while (temp\_lower\_bound <= upper\_bound):

        x -= 1

        temp\_lower\_bound = norm(projection(matrix, target - x \* b))

    x\_lowest = x + 1

    for i in range(x\_lowest, x\_highest):

        x\_array = np.append(x\_array, i)

    for x in x\_array:

        res = x \* b + branch\_and\_bound(matrix, target - x \* b)

        v\_array.append(res)

    v\_array = np.array(v\_array)

    return closest\_vector(v\_array, target)

HNF.py:

import numpy as np

from numpy import linalg

from utils import \*

def HNF\_full\_row\_rank(matrix):  # Hermite Normal Form of full rank matrix

    m = matrix.shape[0]

    n = matrix.shape[1]

    # inds - indexes of linear independent columns

    # number of such rows is equal to m, so B\_stroke always m x m

    temp, inds = sympy.Matrix(matrix).rref()

    B\_stroke = []

    for i in inds:

        B\_stroke.append(matrix.T[i])

    B\_stroke = (np.array(B\_stroke)).T

    det = round(det\_by\_gram\_schmidt(B\_stroke))

    H = np.eye(m, dtype=int) \* det

    for i in range(n):

        H = add\_column(H, matrix.T[i])

    num\_of\_additional\_zero\_vectors = n - m

    zero\_vector = np.array([np.zeros(m, dtype=int)])

    for i in range(num\_of\_additional\_zero\_vectors):

        H = np.append(H.T, zero\_vector, axis=0).T

    return H

def add\_column(H, b\_column):

    if (H.shape[1] == 0):

        return H

    a = H[0, 0]

    h = H[1:, 0]

    H\_stroke = H[1:, 1:]

    b = b\_column[0]

    b\_stroke = b\_column[1:]

    g, x, y = gcd\_extended(a, b)

    U = np.array([[x, -b / g], [y, a / g]])

    first\_column = np.append(np.array(a), h)

    second\_column = np.append(np.array(b), b\_stroke)

    temp\_matrix = np.array([first\_column, second\_column]).T

    temp\_result = np.dot(temp\_matrix, U)

    h\_stroke = temp\_result.T[0, 1:]

    b\_double\_stroke = temp\_result.T[1, 1:]

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

    H\_double\_stroke = add\_column(H\_stroke, b\_double\_stroke)

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

    length\_of\_zeros = H\_double\_stroke.shape[0]

    result = np.array(np.append(np.array([np.append(np.array(g), np.zeros(length\_of\_zeros, dtype=int))]), np.append(

        np.array([h\_stroke]), H\_double\_stroke.T, axis=0).T, axis=0), dtype=int)

    return result

def HNF(matrix):

    m = matrix.shape[0]

    n = matrix.shape[1]

    B\_stroke = projection(matrix)

    B\_double\_stroke = HNF\_full\_row\_rank(B\_stroke)

    result = inverse\_projection(B\_double\_stroke, matrix)

    return result

\_\_main\_\_.py:

from HNF import HNF, HNF\_full\_row\_rank

from CVP import greedy, branch\_and\_bound

from wolfram import get\_HNF\_from\_wolfram\_alpha

import numpy as np

from utils import generate\_random\_array, generate\_random\_linearly\_independent\_float\_matrix, generate\_random\_linearly\_independent\_matrix, generate\_random\_matrix, generate\_random\_matrix\_with\_full\_row\_rank, generate\_random\_float\_matrix, print\_matrix

def HNF\_problem():

    # n >= m, if matrix should be full row rank

    m = 3  # size of vector

    n = 3  # number of vectors

    lowest = 0

    highest = 6

    # generate matrix with full row rank or random matrix

    # columns of generated matrix B are vectors

    # B = [b1, .., bn]

    # bi = |bi1|

    #      |bi2|

    #      |...|

    #      |bim|

    #

    # B = [b11, .., bn1]

    #     [b12, .., bn2]

    #     ..............

    #     [b1m, .., bnm]

    B = generate\_random\_matrix\_with\_full\_row\_rank(m, n, lowest, highest)

    print("B = \n{}".format(B))

    print("B.T = \n{}".format(B.T))

    # get HNF of full ranked B

    H = HNF\_full\_row\_rank(B)

    H\_str = get\_HNF\_from\_wolfram\_alpha(B.T)

    print("HNF(B) = \n{}\n".format(H))

    print("HNF(B).T = \n{}\n".format(H.T))

    print("HNF(B) from wolfram alpha = \n{}".format(H\_str))

    m = 5  # size of vector

    n = 3  # number of vectors

    lowest = 0

    highest = 6

    B = generate\_random\_matrix(m, n, lowest, highest)

    print("B = \n{}".format(B))

    print("B.T = \n{}".format(B.T))

    H = HNF(B)

    H\_str = get\_HNF\_from\_wolfram\_alpha(B.T)

    print("HNF(B) = \n{}".format(H))

    print("HNF(B).T = \n{}".format(H.T))

    print("HNF(B) from wolfram alpha = \n{}".format(H\_str))

def CVP\_problem():

    m = 2  # vector size

    n = 2  # number of vectors

    lowest = 0

    highest = 3

    arr\_lowest = 0

    arr\_highest = 5

    # generate matrix with linear independent vectors

    # vectors of generated matrix B are vectors

    # B = |b1|

    #     |b2|

    #     |..|

    #     |bn|

    #

    # bi = [bi1, bi2, .. bim]

    #

    # B = [b11, .., b1m]

    #     [b21, .., b2m]

    #     ..............

    #     [bn1, .., bnm]

    B = generate\_random\_linearly\_independent\_matrix(

        n, m, lowest, highest)

    print("B = \n{}".format(B))

    t = generate\_random\_array(m, arr\_lowest, arr\_highest)

    print("t = {}".format(t))

    print("Result of greedy: {}".format(greedy(B, t)))

    print("Result of b&b: {}".format(branch\_and\_bound(B, t)))

    B = generate\_random\_linearly\_independent\_float\_matrix(

        n, m, lowest, highest)

    print("B = \n{}".format(B))

    t = generate\_random\_array(m, arr\_lowest, arr\_highest)

    print("t = {}".format(t))

    print("Result of greedy: {}".format(greedy(B, t)))

    print("Result of b&b: {}".format(branch\_and\_bound(B, t)))

def main\_problem\_HNF():

    m = 5  # size of vector

    n = 3  # number of vectors

    lowest = 0

    highest = 6

    B = generate\_random\_matrix(m, n, lowest, highest)

    print("B = \n{}".format(B))

    H = HNF(B)

    H\_str = get\_HNF\_from\_wolfram\_alpha(B.T)

    print("HNF(B) = \n{}".format(H))

    print("HNF(B).T = \n{}".format(H.T))

    print("HNF(B) from wolfram alpha = \n{}".format(H\_str))

def equality\_problem\_HNF():

    B = np.array([[1, 0, 3], [2, 1, -1]])

    B\_stroke = np.array([[1, 2, 3], [3, 2, 2]])

    H1 = HNF(B)

    H2 = HNF(B\_stroke)

    print("B = \n{}".format(B))

    print("B' = \n{}".format(B\_stroke))

    print("HNF(B) = \n{}".format(H1))

    print("HNF(B') = \n{}".format(H2))

def union\_of\_lattices\_problem\_HNF():

    B = np.array([[1, 0, 3], [2, 1, -1]])

    B\_stroke = np.array([[1, 2, 3], [3, 2, 2]])

    arr = np.append(B, B\_stroke, axis=1)

    H = HNF(arr)

    print("B = \n{}".format(B))

    print("B' = \n{}".format(B\_stroke))

    print("B|B' = \n{}".format(arr))

    print("HNF(B|B') = \n{}".format(H))

def containtment\_problem\_HNF():

    B = np.array([[1, 0, 3], [2, 1, -1]])

    B\_stroke = np.array([[1, 2, 3], [3, 2, 2]])

    arr = np.append(B, B\_stroke, axis=1)

    H1 = HNF(arr)

    H2 = HNF(B)

    print("B = \n{}".format(B))

    print("B' = \n{}".format(B\_stroke))

    print("B|B' = \n{}".format(arr))

    print("HNF(B') = \n{}".format(H1))

    print("HNF(B|B') = \n{}".format(H2))

def membership\_problem\_HNF():

    B = np.array([[1, 0, 3], [2, 1, -1]])

    vector = np.array([[1, 1]])

    arr = np.append(B, vector.T, axis=1)

    H1 = HNF(arr)

    H2 = HNF(B)

    print("B = \n{}".format(B))

    print("v = \n{}".format(vector))

    print("B|v = \n{}".format(arr))

    print("HNF(v) = \n{}".format(H1))

    print("HNF(B|v) = \n{}".format(H2))

def main():

    # main\_problem\_HNF()

    # equality\_problem\_HNF()

    # union\_of\_lattices\_problem\_HNF()

    # containtment\_problem\_HNF()

    # membership\_problem\_HNF()

    # HNF\_problem()

    CVP\_problem()

if \_\_name\_\_ == '\_\_main\_\_':

    main()

# **Ссылки на источники**

* <https://cseweb.ucsd.edu/classes/sp14/cse206A-a/index.html>
* <https://en.wikipedia.org/wiki/Hermite_normal_form>
* <https://amp.ww.google-info.org/8261985/1/ermitova-normalnaya-forma.html>
* <https://en.wikipedia.org/wiki/Lattice_problem>
* <https://cims.nyu.edu/~regev/teaching/lattices_fall_2004/ln/cvp.pdf>
* <https://cseweb.ucsd.edu/classes/wi12/cse206A-a/LecEnum.pdf>
* <https://www.wolframalpha.com/widgets/view.jsp?id=1fb417d31b95c3c924c112209b5c65bd>
* <https://wolframalpha.readthedocs.io/en/latest/?badge=latest>