1 Аннотация

Целью данной работы является изучение конструкции T-многообразий сложности 1 в терминах CDP (комбинаторных дивизориальных многогранников) и компьютерная реализация этих многообразий и некоторых их свойств. В данной работе был разработан и реализован алгоритм проверки CDP на эквивалентность, а также была реализована генерация CDP из многогранников. Весь код написан на python3 с использованием библиотеки sagemath.

Содержание

2 Введение

Библиотека sagemath предоставляет функционал для работы с нормальными торическими многообразиями, однако торические многообразия покрывают лишь малую часть многообразий с эффективным действием тора на них. Большинство этих многообразий сложны для реализации, однако существует подкласс таких многообразий - T-многообразия сложности 1 - имеющий удобное для хранения представление.

Для понимания теории, связанной с T-многообразиями, необходимо понимание теории торических многообразий. Торические многообразия — это алгебраические многообразия, возникающие из элементарных геометрических и комбинаторных объектов, таких как выпуклые многогранники в евклидовом пространстве с вершинами в точках решетки. Поскольку многие понятия алгебраической геометрии, такие как особенности, бирациональные отображения, циклы, гомологии, переводятся в простые факты о многогранниках, торические многообразия представляют собой хороший источник примеров в алгебраической геометрии. С другой стороны, общие факты из алгебраической геометрии имеют значение для таких многогранников, например, для проблемы количества содержащихся в них узлов решетки. Несмотря на то, что торические многообразия занимают особое место в спектре всех алгебраических многообразий, они обеспечивают чрезвычайно полезную испытательную площадку для общих теорий. ([2])

Все определения взяты из [1].

2.1 Торические многообразия

Пусть $N \simeq \mathbb{Z}^n$ - решетка размерности n, $M \simeq \mathbb{Z}^n$ - двойственная к ней решетка. Определим $N_{\mathbb{Q}} = N \otimes_{\mathbb{Z}} \mathbb{Q} \simeq \mathbb{Q}^n$, $M_{\mathbb{Q}} = M \otimes_{\mathbb{Z}} \mathbb{Q} \simeq \mathbb{Q}^n$, аналогично определим $N_{\mathbb{R}} \simeq \mathbb{R}^n$, $M_{\mathbb{R}} \simeq \mathbb{R}^n$

Определение 2.1 (Рациональный выпуклый полиэдральный конус). Пусть $S \subset N$ - конечное подмножество. Рациональным выпуклым полиэдральным конусом называется $\sigma = Cone(S) = \{\sum_{u \in S} \lambda_u u | \lambda_u \geq 0\} \subseteq N_{\mathbb{R}}$

Определение 2.2 (Двойственный конус). Пусть $\sigma \subset N_{\mathbb{R}}$ - полиэдральный конус. Тогда двойственным конусом называется $\sigma^\vee = \{m | \langle u, m \rangle \geq 0 \forall u \in$

$\sigma\} \subseteq M_{\mathbb{R}}$

Пусть $m \in M$, определим гиперплоскость $H_m = \{b \in N_{\mathbb{R}} | \langle v, m \rangle = 0\}$ и замкнутое полупространство $H_m^+ = \{v \in N_{\mathbb{R}} | \langle v, m \rangle \geq 0\}$

Определение 2.3 (Опорная гиперплоскость). Гиперплоскость H_m называется опорной для σ, если H_m^+ содержит σ.

Определение 2.4 (Грань конуса). Гранью конуса σ называется его пересечение с опорной гиперплоскостью.

Пусть $\mathbb{C}^n$ - афинное пространство, $\mathbb{C}[z_1, \ldots, z_n] = \mathbb{C}[z]$ - кольцо регулярных функций на $\mathbb{C}^n$.

Определение 2.5 (Афинное алгебраическое множество). Пусть $I \subset \mathbb{C}[z]$ - идеал, тогда множество общих нулей всех его многочленов $V(I) = \{z \in \mathbb{C}^n | f(z) = 0\} \forall f \in I$ называется афинным алгебраическим множеством.

Определение 2.6 (Афинное многообразие). Афинное алгебраическое множество V неприводимо, если его нельзя представить в виде объединения двух афинных алгебраических множеств, являющихся строгими подмножествами V. Неприводимое афинное алгебраическое множество называется афинным многообразием.

Определение 2.7. Пусть $V \subset \mathbb{V}^n$ — некоторое множество точек. Тогда идеалом, порождённым V , называется $I(V) = \{f \in \mathbb{C}[z] | f(z) = 0 \forall z \in V\}$.

Определение 2.8 (Координатное кольцо). Пусть $V \subset \mathbb{C}^n$ — некоторое афинное многообразие. Тогда назовём координатным кольцом V кольцо $\mathbb{C}[V] = \mathbb{C}[z]/I(V)$.

Определение 2.9 (Нормальное афинное многообразие). Афинное многообразие $V \subset \mathbb{C}^n$ называется нормальным, если его координатное кольцо является областью целостности.

Определение 2.10 (Афинная полугруппа). Конечно-порожденная полугруппа S называется афинной, если она изоморфна некоторой полугруппе $\mathbb{Z}^n$.

Определение 2.11 (Афинное торическое многообразие). Пусть дан конус $\sigma \subset N_{\mathbb{R}}$ и его двойственный конус $\sigma^{\vee} \subset M_{\mathbb{R}}$. Тогда $S_\sigma = \sigma^{\vee} \cap M$ - конечно-порожденная афинная полугруппа, $V_\sigma = Spec(\mathbb{C}[S_\sigma])$ называется афинным торическим многообразием, порожденным конусом σ ($Spec$ - спектр кольца $\mathbb{C}[S_\sigma]$, то есть множество максимальных идеалов).

Определение 2.12 (Тор). Множество $\mathbb{T} = (\mathbb{C}^*)^n$ называется алгебраическим тором.

Пусть $\sigma \subset N_{\mathbb{R}}$ - полиэдральный конус. Афинное торическое многообразие X_σ содержит тор $\mathbb{T} = (\mathbb{C}^*)^n$ в качестве открытого плотного подмножества в топологии Зарисского ([1]).

Определение 2.13 (Веер). Веером Σ на $N_{\mathbb{R}}$ называется объединение полиэдральных выпуклых рациональных конусов в $N_{\mathbb{R}}$ таких что:

1. Грань конуса в Σ является конусом в Σ
2. Пересечение любых двух конусов в Σ является гранью обоих

Определение 2.14 (Торическое многообразие). Любой конус по определению является веером. Каждый конус веера Σ задает афинное торическое многообразие. Пересечение любых двух из них также является конусом в Σ. Можно построить склейку данных многообразий по их подмногообразиям и получить многообразие веера V_Σ, называемое торическим многообразием.

Каждое торическое многообразие размерности n содержит тор $\mathbb{T} = (\mathbb{C}^*)^n$ в качестве открытого плотного подмножества в торологии Зарисского ([1]).

Определение 2.15 (Простой дивизор). Простым дивизором D многообразия X называется подмногообразие коразмерности 1, то есть $dimD = dimX - 1$

Каждый простой дивизор порождает подкольцо. Множество всех простых дивизоров порождает свободную абелеву группу, обозначим ее $Div(X)$. Элементы этой группы называются дивизорами Вейля, они могут быть записаны в виде формальной суммы $D = \sum_i n_i A_i - \sum_j m_j B_j, n_i, m_j > 0$, где A_i, B_j - простые дивизоры.

2.2 CDP и T-многообразия

Определения взяты из [3], [5], [4].

Пусть задана решетка N, $\square$ - многогранник с вершинами в узлах решетки, $\square^\circ$ - внутренность многогранника, $\partial\square$ - его граница.

Определение 2.16 (Дивизориальный многогранник). Дивизориальный многогранник на гладкой проективной кривой Y относительно решетки N - это кусочно-линейная вогнутая функция $\Psi = \sum_{P\in Y} \psi_P \cdot P : \square \to Div_{\mathbb{Q}} Y$, область определения которой - многогранник $\square$ на решетке N, область значений - подмножество группы $\mathbb{Q}$-дивизоров на Y, такая что

1. $deg\Psi(u) > 0$ для $u \in \square^\circ$
2. $deg\Psi(u) > 0$ или $\Psi(u) \sim 0$ для $u \in \square$
3. График ψ_P имеет целочисленные вершины для всех $P \in Y$

, где $deg\Psi(u) = \sum_{P\in Y} \psi_P$

Определение 2.17 (T-многообразие). T-многообразием сложности k называется нормальное многообразие X над $\mathbb{C}_n$ эффективным действием тора $T \simeq (\mathbb{C}^*)^{n-k}$

Как видно из определения, торические многообразия являются T-многообразиями сложности 0.

Дивизориальные многогранники соответствуют рациональным поляризованным T-многообразиям сложности 1 ([5]).

Определение 2.18 (CDP). Комбинаторный дивизориальный многогранник (combinatorial divisorial polytope) относительно решетки M состоит из многогранника $\square \subset M \otimes \mathbb{R}$ и кортежа $(\psi_1, \ldots, \psi_n)$ кусочно-линейных вогнутых функций $\psi_i : \square \to \mathbb{R}$ таких что

1 Для всех i граф ψ_i - полиэдральный комплекс с целочисленными вершинами

2 $\forall u \in \square^\circ \sum_{i=1}^n \psi_i(u) > 0$

$\square$ называется базой CDP

Если прикрепить каждую из функций ψ_i к точке P_i на кривой $\mathbb{P}^1$ (проективная прямая) - получится дивизориальный многогранник на $\mathbb{P}^1$ ([4]), что соответствует рациональным поляризованным T-многообразиям сложности 1.

2.3 Преобразования CDP

Пусть дан исходный CDP с набором функций $(\psi_1, \ldots, \psi_n)$ и базой $\square$

Определение 2.19 (Трансформация базы (transformation of the base)). Пусть ϕ - линейное обратимое преобразование решетки N, тогда CDP с базой $\phi(\square)$ и набором функций $\psi_i \circ \phi^{-1}$ получен из исходного с помощью трансформации базы.

Определение 2.20 (Скашивание (shearing)). Пусть $v \in M^*, \beta_1, \ldots, \beta_n \in \mathbb{Z}$, тогда CDP с базой $\square$ и набором функций $u \mapsto \psi_i(u) + \beta_i \langle u, v \rangle, \forall u \in \square$, $\sum_{i=1}^{n} \beta_i = 0$, получен из исходного с помощью скашивания.

Определение 2.21 (Перенос (translation)). Пусть $\alpha_1, \ldots, \alpha_n \in \mathbb{Z}$, $\sum_{i=1}^{n} \alpha_i = 0$, тогда CDP с базой $\square$ и набором функций $\psi_i + \alpha_i$ получен из исходного с помощью переноса.

2.4 Эквивалентность

Определение 2.22 (Эквивалентные CDP). CDP, полученный из исходного с помощью преобразований скашивания, переноса и трансформации базы, примененных в любой последовательности и в любых количествах, эквивалентен исходному.

Определение 2.23 (Эквивариантный изоморфизм). Пусть дана группа, X и Y - действия этой группы на множестве S, тогда функция $f : X \to Y$ - эквивариантная, если $f(g\dot{x}) = g\dot{f}(x)$ для всех $g \in G$ и $x \in X$. Если, кроме того, f биективно, то f - эквивариантный изоморфизм.

Геометрический смысл определения эквивалентности заключается в следующем: если для CDP ψ прикрепить точки $P_i \in \mathbb{P}^1$ - получится дивизориальный многогранник на $\mathbb{P}^1$, который соответствует рациональному

поляризованному T-многообразию сложности 1. Эквивалентные CDP задают эквивариантно изоморфные T-многообразия сложности 1 при условии "правильного"выбора точек P_i.

Определение 2.24 (Грань высоты 1). Грань F многогранника $P \in (M \times \mathbb{Z}) \otimes \mathbb{R}$ является гранью высоты 1, если существует вектор $u \in M^* \times \mathbb{Z}$ такой что $\langle v, u \rangle = 1$ для всех $v \in F$

Определение 2.25 (CDP со свойством Фано). CDP обладает свойством Фано, если он эквивалентен какому-то CDP с базой $\square$ и функциями $\psi_1, \dots \psi_n$ для которого существуют коэффициенты $a_1, \dots, a_n$ такие что

1. $0 \in \square^\circ$
2. $\sum_{i=1}^n a_i = -2$
3. Для всех i $\psi_i(0) + a_i + 1 > 0$, и каждая грань $\Gamma(\psi_i + a_i + 1)$ - грань высоты 1
4. Для каждой грани F многогранника $\square$ не являющейся гранью высоты 1 верно $\sum_{i=1}^n \psi_i \equiv 0$ на всей F

Все свойства определения Фано сохраняются для эквивалентных CDP при условии что трансформация базы оставляет начало координат внутри базы ([4]). Внутри класса рациональных поляризованных T-многообразий сложности 1, CDP, удовлетворяющие свойству Фано, соответствуют каноническим T-многообразиям Горенштейна-Фано с антиканонической поляризацией.

2.5 Генерация CDP из многогранников

Рассмотрим многогранник $P \in (M \times \mathbb{Z}) \otimes \mathbb{R}$ с вершинами на решетке $N \times \mathbb{Z}$. Из него можно получить CDP. Пусть π_1 - проекция на $M \otimes \mathbb{R}$, π_2 - проекция на $\mathbb{R}$. Возьмем $\square = \pi_1(P)$ и $\psi_1, psi_2 : \square \to \mathbb{R}$: $\psi_1(u) = max(\pi_2(\pi_1^{-1}(u) \cap P))$, $\psi_2(u) = -min(\pi_2(\pi_1^{-1}(u) \cap P))$. Таким образом, из любого многогранника на решетке можно получить CDP с двумя функциями. И наоборот, из любого CDP с двумя функциями можно получить многогранник на решетке.

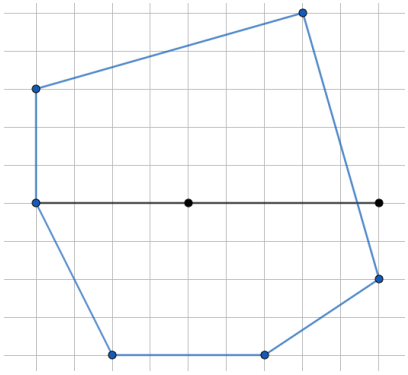

Рис. 1: Многогранник

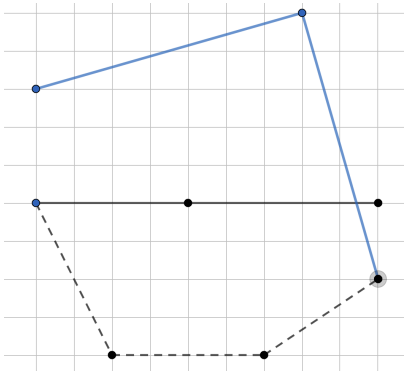

Рис. 2: $\square = \pi_1(P)$

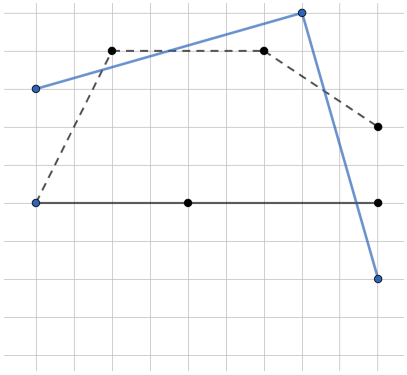

Рис. 3: CDP

Рис. 4: Пример (На втором и третьем рисунках пунктирная линия - грани, которые вошли в график ψ_1, голубая линия - грани, которые вошли в график ψ_2)

Определение 2.26 (Торический CDP). CDP называется торическим, если он эквивалентен CDP с ровно двумя функциями

Любой торический CDP можно получить из многогранника с помощью алгоритма, описанного выше.

Торический CDP является Фано тогда и только тогда, когда соответствующий ему многогранник изоморфен рефлексивному.

3 Компьютерная реализация CDP

Весь код реализован на Python3 и представлен в Приложении, а также доступен по ссылке . В реализации используется класс Polyhedron из библиотеки sagemath.

Класс CDP имеет два поля - base (класс Polyhedron из sage) и psi_list - список объектов класса PiecewiseAffineFunction. Класс PiecewiseAffineFunction - реализация кусочно-линейной функции, класс имеет одно поле - affine_pieces, список объектов класса AffineFunction. Класс AffineFunction - реализация линейной функции, содержит два поля - domain (класс Polyhedron из sage, это область, на которой задана функция) и coefs - список коэффициентов в формате $[a_0, a_1, \ldots, a_{k-1}]$, тогда функция имеет вид $x_k = a_0 + a_1 x_1 + \cdots + a_{k-1} x_{k-1}$

CDP - сложный объект, нужно соблюсти множество условий, чтобы сконструировать правильный CDP, поэтому в конструкторе проверяется

корректность поданных на вход данных.

Пример корректного CDP:

Пример взят из статьи [4]

```
base = Polyhedron(vertices=[[-1], [1]])
# y = 1 + x, x \in [-1, 0]
f_11 = AffineFunction([1, 1], Polyhedron(vertices=[[-1], [0]]))
# y = 1 - x, x \in [0, 1]
f_12 = AffineFunction([1, -1], Polyhedron(vertices=[[0], [1]]))
f_1 = PiecewiseAffineFunction([f_11, f_12])
# y = 1/2x + 1/2, x \in [-1, 1]
f_2 = PiecewiseAffineFunction([AffineFunction([1 / 2, 1 / 2], Polyhedron(
    vertices=[[-1], [1]]))])
cdp = CDP([f_1, f_2], base)
print(cdp)
```

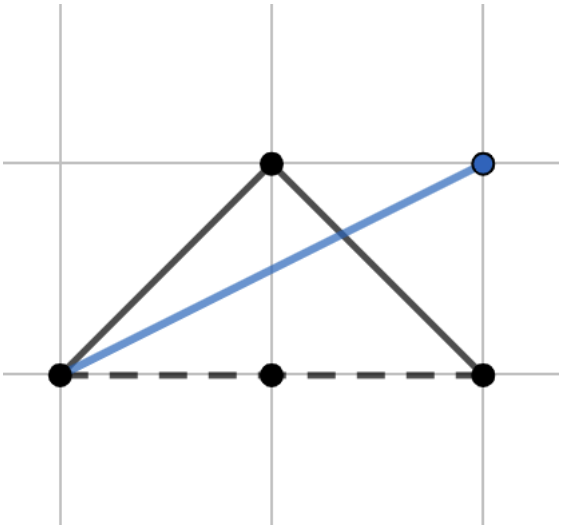

Рис. 5: Корректный CDP

Программа выведет:

```
CDP object, psi list:
Piecewise affine function:
Affine function 1 + x_1 with domain [(-1), (0)]
Affine function 1 - x_1 with domain [(0), (1)]

Piecewise affine function:
Affine function 0.5 + 0.5x_1 with domain [(-1), (1)],

base: (A vertex at (-1), A vertex at (1))
```

Пример некорректного CDP (сумма ψ_i отрицательная в одной из точек базы):

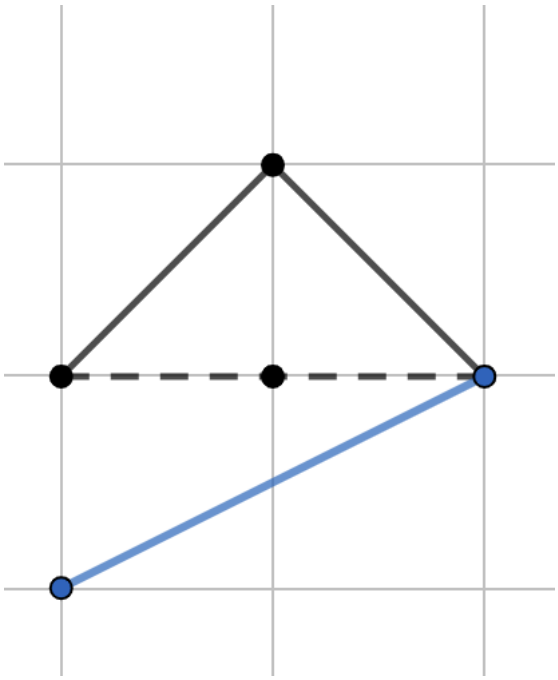

Рис. 6: Некорректный CDP

```
base = Polyhedron(vertices=[[-1], [1]])
# y = x, x \in [-1, 0]
f_11 = AffineFunction([0, 1], Polyhedron(vertices=[[-1], [0]]))
# y = -x, x \in [0, 1]
f_12 = AffineFunction([0, -1], Polyhedron(vertices=[[0], [1]]))
f_1 = PiecewiseAffineFunction([f_11, f_12])
# y = -1/2 + 1/2x, x \in [-1, 1]
f_2 = PiecewiseAffineFunction([AffineFunction([-1 / 2, 1 / 2], Polyhedron(
    vertices=[[-1], [1]]))])
cdp = CDP([f_1, f_2], base)
```

Программа выведет:

```
Traceback (most recent call last):
  File "test_cdp_validity.py", line 65, in <module>
    test.test_not_valid_cdp()
  File "test_cdp_validity.py", line 30, in
     test_not_valid_cdp
    cdp = CDP([f_1, f_2], base)
  File "cdp.py", line 27, in __init__
    raise ValueError(
ValueError: Not a valid CDP - sum of psi is -2.0 on A
   vertex at (-1)
```

3.1 Преобразования CDP

Применим скашивание к CDP из примера 3: с коэффициентом -1 для первой функции и 1 для второй, с вектором (-1).

```
cdp = CDP([f_1, f_2], base)
cdp.shear([-1, 1], [-1])
print(cdp)
```

Программа выведет:

```
CDP object, psi list:
Piecewise affine function:
Affine function 1 + 2x_1 with domain [(-1), (0)]
Affine function 1 with domain [(0), (1)]

Piecewise affine function:
Affine function 0.5 - 0.5x_1 with domain [(-1), (1)],

base: (A vertex at (-1), A vertex at (1))
```

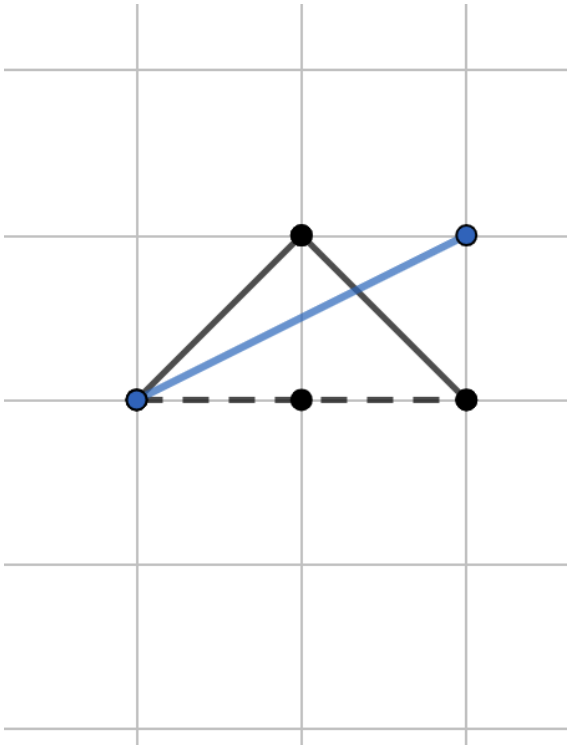

Рис. 7: Исходный CDP

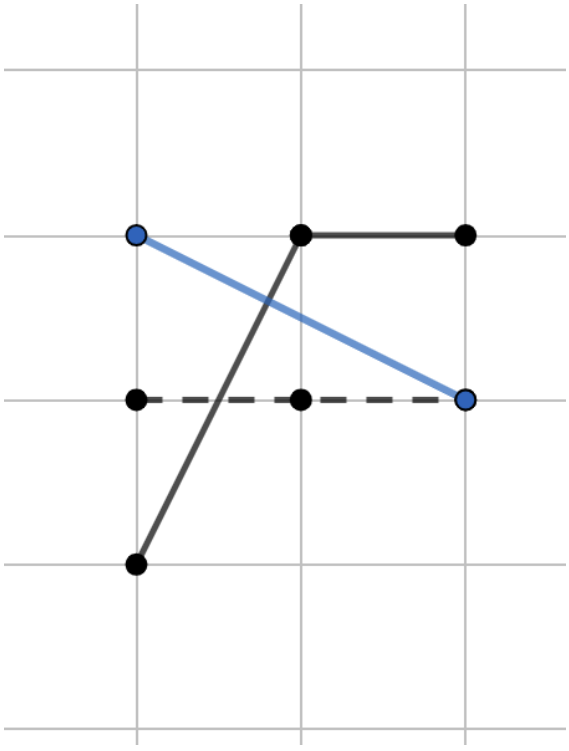

Рис. 8: После скашивания

Применим сдвиг с коэффициентами 1, -1 к результату предыдущего преобразования:

```
cdp.translate([1, -1])
print(cdp)
```

Результат:

```
CDP object, psi list:
Piecewise affine function:
Affine function 2 + 2x_1 with domain [(-1), (0)]
Affine function 2 with domain [(0), (1)]

Piecewise affine function:
Affine function -0.5 - 0.5x_1 with domain [(-1), (1)],

base: (A vertex at (-1), A vertex at (1))
```

Применим трансформацию базы - зеркальное отображение относительно начала координат - к результату предыдущего преобразования:

```
A = matrix(ZZ, [[-1]])
phi = linear_transformation(A)
cdp.transform_base(phi)
print(cdp)
```

Результат:

```
CDP object, psi list:
Piecewise affine function:
Affine function 2 - 2x_1 with domain [(0), (1)]
Affine function 2 with domain [(-1), (0)]

Piecewise affine function:
Affine function -0.5 + 0.5x_1 with domain [(-1), (1)],

base: (A vertex at (-1), A vertex at (1))
```

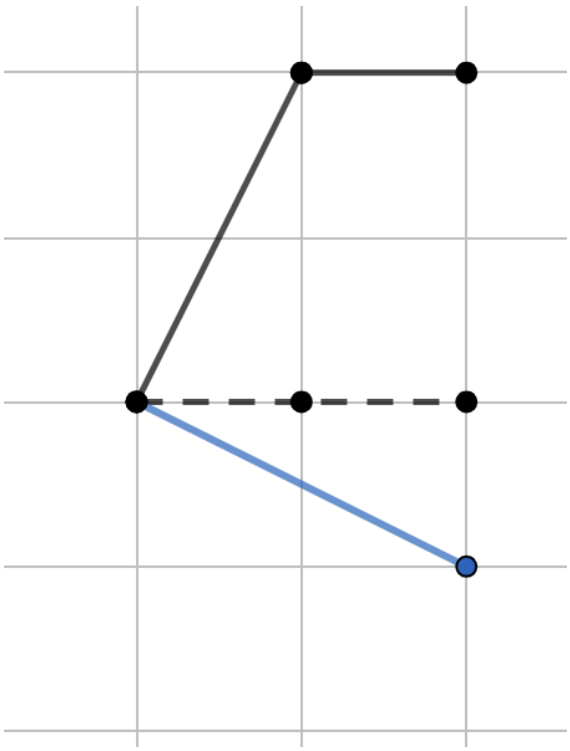

Рис. 9: После переноса

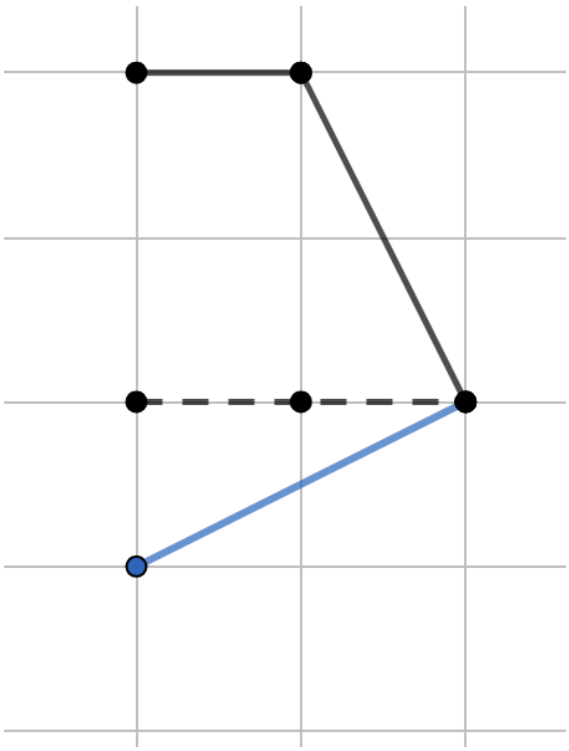

Рис. 10: После трансформации

3.2 Алгоритм проверки эквивалентности

Считаем, что трансформация базы всегда происходит только 1 раз, и это первая операция в цепочке преобразований (ниже будет доказано, почему такое предположение корректно).
Введем операцию обобщенного скашивания:

Определение 3.1 (Обобщенное скашивание). пусть $\sum_{i=1}^{n} v_i = 0$, тогда CDP с базой $\square$ и набором функций $u \mapsto \psi_i(u) + \langle u, v_i \rangle$ получен из CDP $(\square, \psi_1, \ldots, \psi_n)$ с помощью обобщенного скашивания.

3.2.1 Описание

Полный код алгоритма находится в Приложении в разделе 28 Пусть $CDP_1 = (\square, (\psi_1, \ldots, \psi_n))$, $CDP_2 = (\square', (\psi_1', \ldots, \psi_n'))$, проверим их на эквивалентность:

1 Выберем нумерацию вершин в $\square'$. Если уже перебрали все возможные нумерации - завершаем алгоритм, CDP не эквивалентны. Этому шагу соотвеутствуют строки 201-203 в коде алгоритма (перебираем все возможные перестановки вершин, проверяем каждую из них на то, что она сохраняет ребра - функция _vert_permutation_is_valid)

2 Найдем такую матрицу A, которая переводит $\square$ в $\square'$ с учетом нумерации вершин (поиск матрицы преобразования описан в разделе 3.2.2), найдем матрицу обратного преобразования - A^{-1}. Если не удалось найти A или A^{-1} - переходим к шагу 1 (выбираем другую нумерацию вершин). Применим A^{-1} к $\psi_1, ..., \psi_n$, получим $\chi_1 := \psi_1\phi^{-1}, ..., \chi_n := \psi_n\phi^{-1}$. Этому шагу соответствуют строки 204-213 (находим матрицу A, пробуем применить трансформацию базы с такой матрией к исходному CDP)

3 Для каждой χ_i найдем множество $S_i = \{j : \chi_i \sim \psi'_j\}$ - множество ииндексов функций ψ'_j, имеющих такие же области линейности (иллюстрация 11). С помощью переноса или скашивания нельзя из линейной функции получить кусочно-линейную и наоборот, поэтому если для каких-то χ_i или ψ'_j не нашлось соответствия - возвращаемся на шаг 1, выбираем другую нумерацию вершин базы. Иллюстрация к этому шагу: 11. Этому шагу соответствует функция _get_equivalence_classes в коде алгоритма.

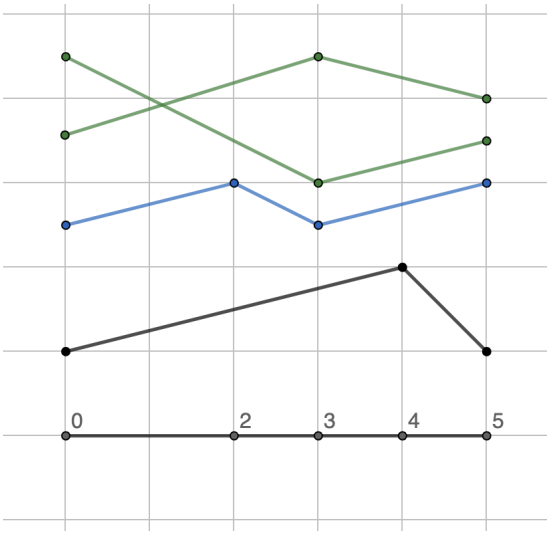

Рис. 11: Зеленые функции имеют области линейности $[[0, 3], [3, 5]]$, голубая - $[[0, 2], [2, 3], [3, 5]]$, черная - $[[0, 4], [4, 5]]$. Зеленые функции имеют одинаковые области линейности.

4 Каждая функция представляется в виде набора коэффициентов $a_0, a_1, \ldots, a_m$, где a_0 - свободный коэффициент (константа), а $a_1, \ldots, a_m$ - коэффициенты перед переменными $x_1, \ldots, x_m$. Выберем σ - перестановку функций χ_i с учетом классов эквивалентности (то

есть рассматриваем только такие σ, что $\sigma(i) \in S_i$ - $\chi_{\sigma(i)}$ и $\psi_i^{'}$ имеют одинаковые области линейности). Если все перестановки уже перебрали - переходим к шагу 1. Этим действиям соответствуют строки 218-219 в коде алгоритма. Функция _list_mappings возвращает все возможные перестановки с учетом классов эквивалентности.

Пусть у $\chi_{\sigma(i)}$ коэффициенты $a_{i0}, a_{i1}, \ldots, a_{im}$, у $\psi_i^{'}$ - $b_{i0}, b_{i1}, \ldots, b_{im}$, тогда если $\sum_{i=1}^{n}(b_{i0} - a_{i0}) = 0$, то существует перенос, позволяющий коэффициенты a_{i0} перевести в b_{i0} (перенос по определению меняет только свободные коэффициенты). Если $\sum_{i=1}^{n}(b_{i0} - a_{i0}) = 0$ - обозначаем функции с коэффициентами $b_{i0}, a_{i1}, \ldots, a_{im}$ как $\chi_i^{'}$ и переходим к шагу 5, если нет - повторяем шаг 4 для другой перестановки. Этому действию соответствует функция _can_be_translated в коде алгоритма.

5 Для каждой пары функций $\chi_i^{'}$, $\psi_i^{'}$ пытаемся найти вектор v_i, такой что $\chi_i^{'}(u) + \langle u, v_i \rangle = \psi_i^{'}(u)$, при этом для всех областей линейности должен получиться одинаковый вектор. Если для всех пар функций получилось найти такой вектор и $\sum_{i=1}^{n} v_i = 0$ - CDP эквивалентны (поскольку удалось найти последовательность из трансформации базы, переноса и обобщенного скашивания, переводящую CDP1 в CDP2), если нет - возвращаемся к пункту 4, выбирая другую перестановку. Этому шагу соответствует функция _can_be_sheared в коде алгоритма.

3.2.2 Поиск матрицы преобразования

Пусть n - размерность пространства, в котором находятся многогранники $\square$ и $\square'$, k - число вершин в этих многогранниках. Составим матрицу V из вершин многогранника $\square$ и матрицу W из вершин многогранника $\square'$. Получатся матрицы размерности $n \times k$. Нужно найти матрицу A такую что $AV = W$, домножим обе стороны справа на V^T: $AVV^T = WV^T$. Если V невырожденная матрица (то есть многогранник $\square$ действительно имеет размерность n, а не меньшую), то VV^T - обратимая матрица. Тогда $A = WV^T(VV^T)^{-1}$.

3.2.3 Примеры работы:

Пример из статьи [4]. В предыдущем разделе (преобразования CDP) CDP2 был получен из CDP1 с помощью цепочки преобразований.

```
base1 = Polyhedron(vertices=[[-1], [1]])
# y = 1 + x, x \in [-1, 0]
f_11 = AffineFunction([1, 1], Polyhedron(vertices=[[-1], [0]]))
# y = 1 - x, x \in [0, 1]
f_12 = AffineFunction([1, -1], Polyhedron(vertices=[[0], [1]]))
f_1 = PiecewiseAffineFunction([f_11, f_12])
# y = 1/2 + 1/2x, x \in [-1, 1]
f_2 = PiecewiseAffineFunction([AffineFunction([1 / 2, 1 / 2], Polyhedron(
    vertices=[[-1], [1]]))])
cdp1 = CDP([f_1, f_2], base1)
base2 = Polyhedron(vertices=[[-1], [1]])
# y = 2, x \in [-1, 0]
y_11 = AffineFunction([2, 0], Polyhedron(vertices=[[-1], [0]]))
# y = 2 - 2x, x \in [0, 1]
y_12 = AffineFunction([2, -2], Polyhedron(vertices=[[0], [1]]))
y_1 = PiecewiseAffineFunction([y_11, y_12])
# y = -1/2 + 1/2x, x \in [-1, 1]
y_2 = PiecewiseAffineFunction([AffineFunction([-1 / 2, 1 / 2], Polyhedron(
    vertices=[[-1], [1]]))])
cdp2 = CDP([y_1, y_2], base2)
print(cdp1.equal(cdp2))
print(cdp2.equal(cdp1))
```

```
True
True
```

Пример для большей размерности:

```
base1 = Polyhedron(vertices=[[1, 0], [0, -1], [-1, 0], [0, 1]])
f_11 = AffineFunction([1, -1, -1], Polyhedron(vertices=[[0, 0], [1, 0], [0,
    1]]))
f_12 = AffineFunction([1, -1, 1], Polyhedron(vertices=[[0, 0], [1, 0], [0,
    -1]]))
f_13 = AffineFunction([1, 1, -1], Polyhedron(vertices=[[0, 0], [-1, 0], [0,
    1]]))
f_14 = AffineFunction([1, 1, 1], Polyhedron(vertices=[[0, 0], [-1, 0], [0,
    -1]]))
f_1 = PiecewiseAffineFunction([f_11, f_12, f_13, f_14])
f_21 = AffineFunction([1, 0, 0], Polyhedron(vertices=[[0, 1], [1, 0], [0,
    -1]]))
f_22 = AffineFunction([1, 1, 1], Polyhedron(vertices=[[0, 0], [0, -1], [-1,
    0]]))
f_23 = AffineFunction([1, 1, -1], Polyhedron(vertices=[[0, 0], [-1, 0], [0,
    1]]))
```

```
f_2 = PiecewiseAffineFunction([f_21, f_22, f_23])
cdp1 = CDP([f_1, f_2], base1)
cdp2 = deepcopy(cdp1)
cdp2.shear([1, -1], [1, 1])
phi = linear_transformation(matrix(ZZ, [[-1, 0], [0, -1]]))
cdp2.transform_base(phi)
print(cdp1.equal(cdp2))
print(cdp2.equal(cdp1))
```

```
True
True
```

Пример не эквивалентных CDP:

```
base1 = Polyhedron(vertices=[[-1], [1]])
# y = 1 + x, x \in [-1, 0]
f_11 = AffineFunction([1, 1], Polyhedron(vertices=[[-1], [0]]))
# y = 1 - x, x \in [0, 1]
f_12 = AffineFunction([1, -1], Polyhedron(vertices=[[0], [1]]))
f_1 = PiecewiseAffineFunction([f_11, f_12])
# y = 1/2 + 1/2x, x \in [-1, 1]
f_2 = PiecewiseAffineFunction([AffineFunction([1 / 2, 1 / 2], Polyhedron(
    vertices=[[-1], [1]]))])
cdp1 = CDP([f_1, f_2], base1)
base2 = Polyhedron(vertices=[[-1], [1]])
# y = 2, x \in [-1, 0]
y_11 = AffineFunction([2, 0], Polyhedron(vertices=[[-1], [0]]))
# y = 2 - x, x \in [0, 1]
y_12 = AffineFunction([2, -1], Polyhedron(vertices=[[0], [1]]))
y_1 = PiecewiseAffineFunction([y_11, y_12])
# y = -1/2 + 1/2x, x \in [-1, 1]
y_2 = PiecewiseAffineFunction([AffineFunction([-1 / 2, 1 / 2], Polyhedron(
    vertices=[[-1], [1]]))])
cdp2 = CDP([y_1, y_2], base2)
print(cdp1.equal(cdp2))
print(cdp2.equal(cdp1))
```

```
False
False
```

3.2.4 Доказательство корректности

Теорема 3.1. Любую цепочку преобразований CDP можно представить в виде трех последовательных преобразований: трансформации базы, переноса и обобщенного скашивания.

Доказательство. Все трансформации базы, присутствующие в цепочке преобразований, можно перенести в ее начало по Лемме 3.3. После этого все трансформации можно объединить в одну по Лемме 3.4. Далее можно поменять местами переносы и скашивания так, чтобы все переносы происходили последовательно следом за трансформацией базы по Лемме 3.5. Оставшиеся операции скашивания можно объединить в одно обобщенное скашивание по Лемме 3.7. □

Теорема 3.2. Если удалось найти цепочку преобразований, переводящую CDP1 в CDP2 и состоящую из трех последовательных преобразований - трансформации базы, переноса и обобщенного скашивания - то эти CDP эквивалентны.

Доказательство. Нужно доказать, что существует цепочка, состоящая из трансформаций базы, переносов и обычных скашиваний, переводящая CDP1 в CDP2, то есть нужно доказать, что обощенное скашивание можно представить в виде композиции обычных. Это доказывается в лемме 3.8. □

Лемма 3.3. Трансформацию базы можно всегда считать первой в цепочке преобразований

Доказательство. Пусть CDP_1 $(\square, \psi_1, ..., \psi_n)$ переводится в CDP_2 $(\square', \psi_1', ..., \psi_n')$ с помощью цепочки преобразований BC, где B - операция скашивания, C - трансформация базы. Покажем, что тогда существует другая цепочка преобразований - $C'B'$, где C' - трансформация базы, B' - операция скашивания, - которая также переводит CDP_1 в CDP_2.
$\psi_i(u) \xrightarrow{C'B'} \psi_i'(u_{new}) = \psi_i(\phi^{-1}(u_{new})) + \beta_i\langle u_{new}, v\rangle$
$\psi_i(u) \xrightarrow{B} \psi_i(u) + \beta_i\langle u, v_1\rangle$, положим $v_1 = \phi(v)$, тогда $\psi_i(u) \xrightarrow{B}$ $\psi_i(u) + \beta_i\langle u, \phi(v)\rangle$, теперь применим трансформацию C: $\psi_i(u) \xrightarrow{BC}$ $\psi_i(\phi^{-1}(u_{new})) + \beta_i\langle\phi^{-1}(u_{new}), \phi(v)\rangle = \psi_i(\phi^{-1}(u_{new})) + \beta_i\langle u_{new}, v\rangle = \psi_i'(u_{new})$, то есть цепочки преобразований BC и $C'B'$ дают одинаковый результат

Аналогично для операции переноса (translation) □

Лемма 3.4. Можно считать, что трансформация базы происходит только 1 раз в цепочке преобразований

Доказательство. Все трансформации базы можно перенести в начало цепочки по предыдущему пункту, а композиция линейных обратимых преобразований также является линейным обратимым преобразованием, поэтому можно заменить композицию трансформаций на единственную трансформацию базы.

□

Лемма 3.5. Перенос и скашивание можно менять местами

Доказательство. $\psi_i(u) + \beta_i\langle u, v\rangle + \alpha_i = \psi_i(u) + \alpha_i + \beta_i\langle u, v\rangle$ □

Лемма 3.6. Два переноса можно объединить в один

Доказательство. $\psi_i + \alpha_i + \alpha_i^{'}$, $\sum_{i=1}^{n} \alpha_i = 0$, $\sum_{i=1}^{n} \alpha_i^{'} = 0$ - значит и $\sum_{i=1}^{n}(\alpha_i + \alpha_i^{'}) = 0$, тогда $\psi_i + \alpha_i^{''}$, $\alpha_i^{''} = \alpha_i + \alpha_i^{'}$ - корректный перенос. □

Лемма 3.7. Два скашивания можно объединить в одно обобщенное скашивание

Доказательство. Пусть $\phi_i(u) = \psi_i(u) + \beta_i\langle u, v\rangle$, $\chi_i(u) = \phi_i(u) + \beta_i^{'}\langle u, v^{'}\rangle$, тогда $\chi_i(u) = \psi_i(u) + \beta_i\langle u, v\rangle + \beta_i^{'}\langle u, v^{'}\rangle = \psi_i(u) + \langle u, w\rangle$ $\sum_{i=1}^{n} \beta_i = 0$, $\sum_{i=1}^{n} \beta_i^{'} = 0$, тогда $\sum_{i=1}^{n} \beta_i\langle u, v\rangle = 0$ и $\sum_{i=1}^{n} \beta_i^{'}\langle u, v^{'}\rangle = 0$, а значит $\sum_{i=1}^{n} w = 0$, то есть χ_i можно получить из ϕ_i с помощью операции обобщенного скашивания. □

Лемма 3.8. Любое обобщенное скашивание можно представить в виде суммы обычных

Доказательство. Пусть $v_1, \ldots, v_n$ - обобщенное скашивание. Покажем, что v_i можно представить в виде $\beta_{i1}v_1^{'} + \cdots + \beta_{ik}v_k^{'}$, $\forall j \sum_{i=1}^{n} \beta_{ij} = 0$

$$v_1 = \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix}, \ldots, v_n = \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix} \tag{1}$$

$$\begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} = a_{11} \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + a_{21} \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + \cdots + a_{m1} \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix} = a_{12} \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + a_{22} \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + \cdots + a_{m2} \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

$$\vdots$$

$$\begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix} = a_{1n} \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + a_{2n} \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} + \cdots + a_{mn} \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

$\forall i \sum_{j=1}^{n} a_{ij} = 0$ (суммы коэффициентов в столбцах), поскольку $\sum_{i=1}^{n} v_i = 0$.

Значит, $a_{1j} \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, a_{2j} \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \ldots, a_{mj} \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$ - корректные операции скашивания (в качестве вектора v для каждого преобразования берем базисный вектор, а в качестве коэффициентов β_i - координаты исходных векторов).

Таким образом, если в пункте 4 нашли обобщенное скашивание, которое переводит CDP_1 в CDP_2 - найдется и последовательность обычных, которые делают то же самое. □

3.3 Алгоритм генерации CDP из многогранников

Полный код алгоритма находится в Приложении в разделе 34 Алгоритм принимает на вход многогранник P размерности k.

1. Для каждой грани находим вектор нормали, внешний по отношению к многограннику. Если проекция вектора нормали на последнюю ось координат (x_k) положительная - относим грань к списку $psi1\_list$, если отрицательная - к $psi2\_list$, а если равна нулю - пропускаем грань, не относим ее ни к какому списку. В коде алгоритма - строки 50-54, находим нормальный конус к грани с помощью функции normal_cone из библиотеки sagemath.

2. Находим проекцию многогранника на подпространство $(x_1, \ldots, x_{k-1})$, для этого достаточно избавиться от последней координаты каждой из вершин и взять выпуклую оболочку полученного множества точек. Полученная выпуклая оболочка - это база CDP. В коде алгоритма - строки 57-58.

3. Каждая грань соответствует области линейности одной из функций ψ_1, ψ_2 (класс AffineFunction). Для каждой грани из списка $psi1\_list$ выполняем следующее: по точкам, соответствующим вершинам грани, строим уравнение гиперплоскости (функция plane_from_points, строки 13-24). Таким образом заполняем поле coefs класса AffineFunction. Берем выпуклую оболочку проекций вершин грани на подпространство $(x_1, \ldots, x_{k-1})$ - получается многогранник, на котором функция ψ_1 линейна (строки 42-43). Это соответствует полю domain класса AffineFunction. Аналогично для списка $psi2\_list$ (для ψ_2 берем все коэффициенты с минусом, поскольку нужно получить зеркальное отображение относительно $(x_1, \ldots, x_{k-1})$).

3.3.1 Примеры работы алгоритма

Генерация CDP из пирамиды:

```
poly = Polyhedron(vertices=[[1, 0, 0], [0, 1, 0], [0, 0, 1], [-1, 0, 0],
                            [0, 0, -1]])
cdp = generate_cdp_from_polytope(poly)
```

```
print(cdp)
```

Результат:

```
CDP object, psi list:
Piecewise affine function:
Affine function 1 - x_1 - x_2 with domain [(0, 0), (0, 1)
    , (1, 0)]
Affine function 1 + x_1 - x_2 with domain [(-1, 0), (0,
    0), (0, 1)]

Piecewise affine function:
Affine function 1 - x_1 - x_2 with domain [(0, 0), (0, 1)
    , (1, 0)]
Affine function 1 + x_1 - x_2 with domain [(-1, 0), (0,
    0), (0, 1)],

base: (A vertex at (-1, 0), A vertex at (1, 0), A vertex
    at (0, 1))
```

Генерация CDP для двумерного случая:

```
poly = Polyhedron(vertices=[[-2, 0], [0, 2], [1, 2], [2, 1], [2, -2], [-2,
    -2]])
cdp = generate_cdp_from_polytope(poly)
print(cdp)
```

Результат:

```
CDP object, psi list:
Piecewise affine function:
Affine function 3 - x_1 with domain [(1), (2)]
Affine function 2 with domain [(0), (1)]
Affine function 2 + x_1 with domain [(-2), (0)]

Piecewise affine function:
Affine function 2 with domain [(-2), (2)],

base: (A vertex at (-2), A vertex at (2))
```

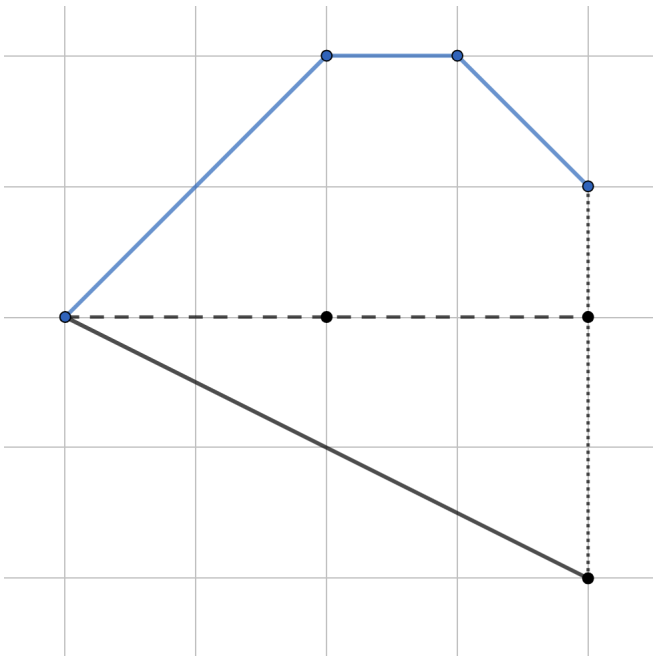

Рис. 12: Пример генерации CDP из многогранника: голубым обозначены ребра, которые войдут в ψ_1, черным (сплошным) - ребра, которые войдут в ψ_2

4 Приложение

4.1 piecewise_affine_function.py

```
#!/usr/bin/env sage
from typing import List
from sage.all import *
import numpy as np


class AffineFunction:
    def __init__(self, coefficients: List[float], domain: Polyhedron):
        dim = len(domain.vertices()[0].vector())
        if len(coefficients) != dim + 1:
            raise ValueError(f'Domain dimension {len(coefficients) - 1} '
                             f'and coefficients list dimension {dim} do not match')
        self.coefs = list(coefficients)
        self.domain = domain
        self.dim = dim

    def __eq__(self, other):
        return self.coefs == other.coefs and self.domain == other.domain

    def value(self, x: List[int]):
        if x not in self.domain:
            raise ValueError(
                f'{x} is not in function domain {self.domain.vertices()}')
        if len(x) != self.dim:
            raise ValueError(f'Wrong dimension of x: {len(x)}')
        return sum([a * b for a, b in zip(self.coefs[1:], x)]) + self.coefs[0]

    def _coef_str(self, i):
        if self.coefs[i] == 0:
            return ''
        if self.coefs[i] == 1:
            return f' + x_{i}'
        elif self.coefs[i] == -1:
            return f' - x_{i}'
        elif self.coefs[i] > 0:
            return f' + {self.coefs[i]}x_{i}'
        return f' - {-self.coefs[i]}x_{i}'

    def __str__(self):
        coefs_repr = []
        if self.coefs[0] != 0 or len(self.coefs) == 2 and self.coefs[1] == 0:
```

```
            coefs_repr.append(str(self.coefs[0]))
        for i in range(1, len(self.coefs)):
            coefs_repr.append(self._coef_str(i))
        coefs_repr = "".join(coefs_repr)
        return f'Affine function {coefs_repr} with domain ' \
               f'{[vert.vector() for vert in self.domain.vertices()]}'


class PiecewiseAffineFunction:
    def __init__(self, affine_pieces: List[AffineFunction]):
        self.affine_pieces = affine_pieces

    def __eq__(self, other):
        if not len(self.affine_pieces) == len(other.affine_pieces):
            return False
        for piece in self.affine_pieces:
            if not piece in other.affine_pieces:
                return False
        return True

    def value(self, x: List[int]):
        for piece in self.affine_pieces:
            if x in piece.domain:
                return piece.value(x)
        raise ValueError(f'{x} is not in function domain')

    def transform(self, phi, phi_inverse):
        for j in range(len(self.affine_pieces)):
            res = np.matmul(self.affine_pieces[j].coefs[1:], phi_inverse)
            self.affine_pieces[j].coefs = [self.affine_pieces[j].coefs[0]]
            self.affine_pieces[j].coefs.extend(res)
            vertices = []
            for vert in self.affine_pieces[j].domain.vertices():
                vertices.append(phi(vert.vector()))
            self.affine_pieces[j].domain = Polyhedron(vertices=vertices)

    def _domains_mapping(self, other_psi):
        if len(self.affine_pieces) != len(other_psi.affine_pieces):
            return False, 0
        domains_mapping = [0 for _ in range(len(other_psi.affine_pieces))]
        for i, piece in enumerate(self.affine_pieces):
            for j, other_piece in enumerate(other_psi.affine_pieces):
                if piece.domain == other_piece.domain:
                    domains_mapping[i] = j
        return domains_mapping
```

```
87
88      def can_be_translated(self, other_psi):
89          domains_mapping = self._domains_mapping(other_psi)
90          alpha = other_psi.affine_pieces[domains_mapping[0]
91                                          ].coefs[0] - self.affine_pieces[0].coefs[0]
92          for i in range(1, len(domains_mapping)):
93              a = other_psi.affine_pieces[domains_mapping[i]
94                                          ].coefs[0] - self.affine_pieces[i].coefs[0]
95              if a != alpha:
96                  return False, 0
97          return True, alpha
98
99      def cat_be_sheared(self, other_psi):
100         domains_mapping = self._domains_mapping(other_psi)
101         n = len(other_psi.affine_pieces[domains_mapping[0]].coefs)
102         v = [other_psi.affine_pieces[domains_mapping[0]].coefs[i] -
103              self.affine_pieces[0].coefs[i]
104              for i in range(1, n)]
105         for j in range(1, len(domains_mapping)):
106             other_v = [other_psi.affine_pieces[domains_mapping[j]].coefs[i] -
107                        self.affine_pieces[j].coefs[i]
108                        for i in range(1, n)]
109             if other_v != v:
110                 return False
111         return True
112
113     def __str__(self):
114         resp = '\n'.join([str(piece) for piece in self.affine_pieces])
115         return 'Piecewise affine function:\n' + resp
```

4.2 cdp.py

```
#!/usr/bin/env sage
from sage.all import *
from typing import List
from piecewise_affine_function import PiecewiseAffineFunction
from collections import defaultdict
from itertools import permutations
import numpy as np


class CDP:
    def __init__(
            self, psi_list: List[PiecewiseAffineFunction], base: Polyhedron):
        # Check that sum of psi is non negative on borders of domains
        # (check polytop vertices of domains laying inside of function scope)
        for psi in psi_list:
            for piece in psi.affine_pieces:
                for vert in piece.domain.vertices():
                    if vert in base:
                        try:
                            s = sum([ps.value(vert.vector())
                                     for ps in psi_list])
                        except ValueError:
                            continue
                        else:
                            if s < 0:
                                raise ValueError(
                                    f'Not a valid CDP - sum of psi is {s} on {vert}
    ')
        # Check that sum of psi is non negative on base vertices
        for vert in base.vertices():
            try:
                s = sum([ps.value(vert) for ps in psi_list])
            except ValueError:
                raise ValueError('Not a valid CDP: psi is not defined on base')
            else:
                if s < 0:
                    raise ValueError(
                        f'Not a valid CDP - sum of psi is {s} on {vert}')
        self.psi_list = psi_list
        self.base = base
        self.n = len(self.base.vertices()[0].vector())
        self.k = len(self.base.vertices())
        self.base_adjacency_map = defaultdict(set)
```

```
43
44      def __str__(self):
45          psi_str = "\n\n".join([str(psi) for psi in self.psi_list])
46          return f'CDP object, psi list:\n{psi_str},\n\nbase: {self.base.vertices()}\
    n'
47
48      def __eq__(self, other):
49          """
50          Note: the order of the functions in psi_list is not important for this
    implementation.
51          """
52          if not self.base == other.base:
53              return False
54          if not len(self.psi_list) == len(other.psi_list):
55              return False
56          for psi in self.psi_list:
57              if psi not in other.psi_list:
58                  return False
59          return True
60
61      def _build_adjacency_map(self):
62          for i in range(self.k):
63              for j in range(i + 1, self.k):
64                  if self.base.vertices()[i].is_incident(
65                          self.base.vertices()[j]):
66                      self.base_adjacency_map[i].add(j)
67                      self.base_adjacency_map[j].add(i)
68
69      def transform_base(self, phi: linear_transformation):
70          vertices = []
71          for vert in self.base.vertices():
72              vertices.append(phi(vert.vector()))
73          self.base = Polyhedron(vertices=vertices)
74          try:
75              inv = phi.inverse().matrix()
76          except ZeroDivisionError:
77              raise ValueError(f'phi is not invertible')
78          inv = np.array([np.array(row) for row in inv])
79          for i in range(len(self.psi_list)):
80              self.psi_list[i].transform(phi, inv)
81
82      def translate(self, alpha_list: List[int]):
83          if len(alpha_list) != len(self.psi_list):
84              raise ValueError(f'Length of alpha_list is {len(alpha_list)}, '
85                               f'should be {len(self.psi_list)}')
```

```
        if sum(alpha_list) != 0:
            raise ValueError('Sum of coefficients should be 0')
        for idx, psi in enumerate(self.psi_list):
            for piece in psi.affine_pieces:
                piece.coefs[0] += alpha_list[idx]

    def shear(self, beta_list: List[int], v: List[int]):
        if len(beta_list) != len(self.psi_list):
            raise ValueError(f'Length of beta_list is {len(beta_list)}, '
                             f'should be {len(self.psi_list)}')
        if sum(beta_list) != 0:
            raise ValueError('Sum of coefficients should be 0')
        m = len(self.psi_list[0].affine_pieces[0].coefs) - 1
        if len(v) != m:
            raise ValueError(f'Wrong dimension of v: {len(v)}, should be {m}')
        for idx, psi in enumerate(self.psi_list):
            for piece in psi.affine_pieces:
                for j, coef in enumerate(v):
                    piece.coefs[j + 1] += coef * beta_list[idx]

    def _vert_permutation_is_valid(self, perm):
        """Check that vertices permutation saves incidence
        """
        for vert, inc_list in self.base_adjacency_map.items():
            for other_vert in inc_list:
                if not perm[other_vert] in self.base_adjacency_map[perm[vert]]:
                    return False
        return True

    def _get_transform_matrix(self, points, point_images):
        left = np.matmul(points.T, points)
        try:
            inverse = np.linalg.inv(left)
        except np.linalg.LinAlgError:
            raise ValueError(f'Base of the CDP should be non-degenerate '
                             f'(dimension {self.n} is provided, but '
                             'the real dimension is lower)')
        A = np.matmul(np.matmul(point_images, points), inverse)
        return A

    def _domains_match(self, psi1: PiecewiseAffineFunction,
                       psi2: PiecewiseAffineFunction) -> bool:
        if len(psi1.affine_pieces) != len(psi2.affine_pieces):
            return False
        return set([p.domain for p in psi1.affine_pieces]) == set(
```

```
            [p.domain for p in psi2.affine_pieces])

    def _get_equivalence_classes(self, other_psi_list):
        used = [False for _ in range(len(other_psi_list))]
        classes = [[] for _ in range(len(self.psi_list))]
        for i, psi1 in enumerate(self.psi_list):
            for j, psi2 in enumerate(other_psi_list):
                if self._domains_match(psi1, psi2):
                    used[j] = True
                    classes[i].append(j)
        if len([c for c in classes if len(c) == 0]) > 0 or len(
                [u for u in used if not u]) > 0:
            return False, []
        return True, classes

    def _list_mappings(self, classes):
        dicts_res = self._list_mappings_recursive(classes, dict(), idx=0)
        res = []
        for small_res in dicts_res:
            res.append([0] * len(classes))
            for k, v in small_res.items():
                res[-1][v] = k
        return res

    def _list_mappings_recursive(self, classes, current_dict, idx=0):
        res = []
        for elem in classes[idx]:
            if elem in current_dict:
                continue
            if idx + 1 == len(classes):
                d = deepcopy(current_dict)
                d[elem] = idx
                return [d]
            d = deepcopy(current_dict)
            d[elem] = idx
            res += self._list_mappings_recursive(classes, d, idx + 1)
        return res

    def _can_be_translated(self, mapping, other_psi_list):
        alpha_list = []
        for i, j in enumerate(mapping):
            can, alpha = self.psi_list[i].can_be_translated(other_psi_list[j])
            if not can:
                return False
            alpha_list.append(alpha)
```

```
        if sum(alpha_list) != 0:
            return False
        return True

    def _can_be_sheared(self, mapping, other_psi_list):
        for i, j in enumerate(mapping):
            if not self.psi_list[i].cat_be_sheared(other_psi_list[j]):
                return False
        return True

    def equal(self, other_cdp):
        """
        Suppose that zero functions are already removed
        """
        # Check that self.base is convertible to other_cdp.base with some phi
        # Check that psi lists split into matching equivalence classes
        # Check that a constant for translation exists
        # Check that a vector for shearing exists
        if len(self.base.vertices()) != len(other_cdp.base.vertices()):
            return False
        if len(self.psi_list) != len(other_cdp.psi_list):
            return False
        G = np.array([np.array(vert.vector())
                      for vert in other_cdp.base.vertices()])
        G = G.T
        for perm in permutations([i for i in range(self.k)]):
            if not self._vert_permutation_is_valid(perm):
                continue
            V = np.array([np.array(self.base.vertices()[i].vector())
                          for i in perm])
            # A transforms base of one CDP to the base of another
            A = self._get_transform_matrix(V, G)
            A = linear_transformation(matrix(QQ, A))
            cdp_after_base_transform = deepcopy(self)
            try:
                cdp_after_base_transform.transform_base(A)
            except ValueError:
                continue
            splits, classes = cdp_after_base_transform._get_equivalence_classes(
                other_cdp.psi_list)
            if not splits:
                continue
            mappings = cdp_after_base_transform._list_mappings(classes)
            for m in mappings:
                can = cdp_after_base_transform._can_be_translated(
```

```
                    m, other_cdp.psi_list)
                if not can:
                    continue
                can = cdp_after_base_transform._can_be_sheared(
                    m, other_cdp.psi_list)
                if not can:
                    continue
                return True
        return False
```

4.3 generate_cdp.py

```
#!/usr/bin/env sage
from sage.all import *
from sympy.matrices import Matrix
from sympy import Rational, lcm
from piecewise_affine_function import PiecewiseAffineFunction, AffineFunction
from cdp import CDP


def generate_cdp_from_polytope(poly: Polyhedron):
    # Walk over all facets, if facet's normal projection to x_n is positive -
    # this facet if a part of psi_1 graph, otherwise - part of -psi_2 grapg

    def plane_from_points(points):
        # (p_11, ..., p_1n), ..., (p_n1, ..., p_nn)
        # p_11, p_21, ..., p_n1
        #         ...             = P
        # p_1n, p_2n, ..., p_nn
        # AP = b, b = (1, ..., 1) => A = bP^-1
        # 1 - a_1*x_1 - ... - a_n*x_n = 0, return [1, -a1, ..., -a_n]
        points = points[:len(points[0])]
        b = Matrix([[1 for i in range(len(points[0]))]])
        P = Matrix(points).T
        A = b * P.inv()
        return [Rational(1)] + [-a for a in list(A.row(0))]

    def integer_coefs_from_rational(coefs):
        qs = [c.q for c in coefs]
        l = lcm(qs)
        coefs = [c * l for c in coefs]
        return [c / coefs[-1] for c in coefs]

    def piecewise_from_facets(facets, invert_coefs=False):
        pieces = []
        for facet in facets:
            # Find a plane equation with rational coefficients from facet's
    vertices
            # and multiply by least common multiple of denominators to obtain
    integer
            # coefficients.
            r = integer_coefs_from_rational(
                plane_from_points([p.vector() for p in facet]))
            if invert_coefs:
                r = [-c for c in r]
```

```
            pieces.append(AffineFunction(coefficients=r[:-1], domain=Polyhedron(
                vertices=[p.vector()[:-1] for p in facet])))
        return PiecewiseAffineFunction(affine_pieces=pieces)

    psi1_facets = []
    psi2_facets = []
    for facet in poly.facets():
        # Find out whether a facet belongs to psi1 or psi2
        coef = facet.normal_cone(direction='outer').rays_list()[0][-1]
        if coef > 0:
            psi1_facets.append(facet.vertices())
        elif coef < 0:
            psi2_facets.append(facet.vertices())
    psi1 = piecewise_from_facets(psi1_facets, invert_coefs=True)
    psi2 = piecewise_from_facets(psi2_facets)
    return CDP(psi_list=[psi1, psi2], base=Polyhedron(
        vertices=[v.vector()[:-1] for v in poly.vertices()]))
```

Список литературы

[1] Jean-Paul Brasselet. Introduction to Toric Varieties. 2008.

[2] William Fulton. Introduction to Toric Varieties. 1993.

[3] Nathan Ilten и Hendrik Suss. “K-stability for Fano manifolds with torus action of complexity one”. В: (2017). DOI: https://arxiv.org/pdf/1507.04442.pdf.

[4] Marni Mishna Nathan Ilten и Charlotte Trainor. “Classyfying Fano complexity-one T-varieties via Divisorial Polytopes”. В: (2018). DOI: https://arxiv.org/pdf/1710.04146.pdf.

[5] Nathan Owen Ilten Hendrik S¨uß. “Polarized Complexity-One T-Varieties”. В: (2011). DOI: https://arxiv.org/pdf/0910.5919.pdf.