22 марта 2019 г.

Содержание

Введение

Ортогональная проекция на выпуклый полиэдр в эвклидовом пространстве — это довольно типичная операция для многих вычислительных алгоритмов. Здесь рассматривается случай, когда полиэдр задан в виде выпуклой оболочки набором точек $X = \{\hat{x}^1, \ldots, \hat{x}^m\}$. Такое описание полиэдра менее распространено по сравнению с описанием с помощью системы линейных неравенств и позволяет представить только ограниченные многогранники. Однако оно находит применение в недифференциальной оптимизации и других областях. Более того, у этого описания есть определённые достоинства, которые позволяют разработать более простые версии алгоритмов, основанных на множествах.

1 Работа с программой

Для запуска алгоритма требуется подготовить координаты вершин многогранника и записать их в матрицу по столбцам. Затем выбрать максимальное число итераций (`maxiter`), точность решения (`eps`) и подробный или сокращённый вывод (`verbose`). И передать выбранные параметры в функцию `ptp` (можно также задать начальный базис `kvec0` и разложение `R0`, если необходимо улучшить уже полученное ранее решение). Рассмотрим простой пример.

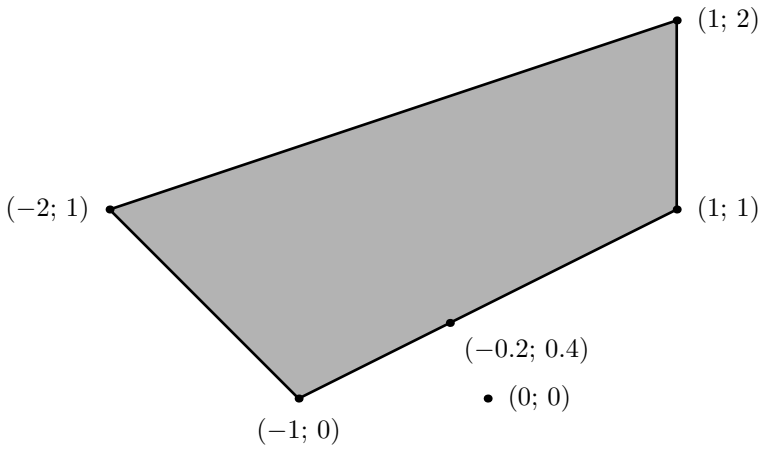

Рис. 1: Выпуклая оболочка из четырёх точек

Пусть заданы четыре точки: $(-1;\,0)$, $(1;\,1)$, $(1;\,2)$, $(-2;\,1)$. Выпуклая оболочка этих точек показана на рисунке 1 (чёрные границы и всё, что выделено серым). Чтобы найти проекцию на эту оболочку, используя `ptp`, запишем все точки в виде матрицы:

$$X = \begin{pmatrix} -1 & 1 & 1 & -2 \\ 0 & 1 & 2 & 1 \end{pmatrix}$$

Запись матрицы `X` на языке Python будет выглядеть как:

⟨ *Polyhedron Vertices* 2a ⟩ ≡

```
X = np.array([[-1, 1, 1, -2], [0, 1, 2, 1]])
```

◇

Fragment referenced in 2b.

Осталось только выбрать остальные параметры. Возьмём максимальное число итераций равное 100, точность $1e-8$ и выберем подробный вывод (1 на месте последнего аргумента функции).

`"example.py"` 2b≡

```
import numpy as np

⟨ Polyhedron Vertices 2a ⟩
z, reps, iter, lmb, kvec, R, info = ptp(X, 100, 1e-8, 1)

print(z)
print(iter)
print(lmb)
print(kvec)
```

◇

Вывод будет следующим:

```
 ++ iter    0(+)    0(-) len    1 zx   0.0000e+00 in      0 zz   1.00000000e+00
 ++ iter    1(+)    0(-) len    2 zx  -2.0000e+00 in      1 zz   2.00000000e-01

XXXX Solved to optimality

 ++ iter    1(+)    0(-) len    2 zx  -1.1102e-16 in     -1 zz   2.00000000e-01
[-0.2  0.4]
[1 0]
[0.6 0.4]
[0 1]
```

Переменная `z` представляет собой проекцию точки (0; 0) на выпуклую оболочку X, что и требовалось найти. Переменная `iter` состоит из двух целых чисел, первое число — количество итераций алгоритма, второе — количество векторов, удалённых из базиса в процессе работы алгоритма. Барицентрические координаты проекции представлены переменной `lmb`, а `kvec` — номера векторов (по столбцам), взятых в базис. Для данного примера в базисе оказались два вектора: $(-1; 0)$, $(1; 1)$.

Вся работа с программой заключается в описании фигуры с помощью точек. Грубо говоря, выпуклая оболочка соединяет все точки лежащие на границе и образует множество, которое соответствует внутренности заданной фигуры. Поэтому не имеет смысла давать на вход ещё и внутренние точки, например, для описания всех внутренних точек треугольника выпуклой оболочке достаточно трёх точек на углах.

2 ptp

Постановка задачи следующая: имеется множество точек $X = \{\hat{x}^1, \dots, \hat{x}^m\}$, необходимо найти вектор z^* с наименьшей нормой из выпуклой оболочки $Z = \text{co}\{X\}$. Что можно записать как

$$\min ||z||^2$$

$$z = X\lambda$$

$$\lambda \in \Delta_m$$

где X определяет матрицу, столбцы которой — векторы из множества X, $\Delta_m = \{\lambda = (\lambda_1, \dots, \lambda_m), \sum\limits_{i=1}^{m} \lambda_i = 1, \lambda_i \geq 0, i = \overline{1, m}\}$ - стандартный m-размерный симплекс. Алгоритм представлен функцией `ptp`, которая принимает на вход:

- `X` — вершины полиэдра;
- `maxit` — число итераций;
- `eps` — точность;
- `verbose` — включение/отключение вывода информации об итерации;
- `kvec0` — начальный базис;
- `R0` — верхнее треугольное разложение Холецкого.

И возвращает:

- `z` — искомый вектор z^*;
- `reps` — точность решения;
- `iter` — количество итераций, которая выполнила программа;
- `lmb` — вектор $\lambda^* = \underset{\lambda}{\text{argmin}}\{X\lambda\}$;
- `kvec` — базис;
- `R` — верхнее треугольное разложение Холецкого, полученное на последней итерации;

- `info` — информация о работе программы (`0` в случае нахождения оптимального решения).

Необходимо отметить, что векторы, которые подаются на вход, должны иметь размерность `(n,)`, а не `(n,1)` и не `(1,n)`. Это достигается инициализацией вектора как одномерного массива.

`"ptp.py"` 4a≡

```
⟨ Import 4b ⟩
⟨ Global variables 4c ⟩
⟨ ptp body 5a ⟩
⟨ Subfunctions 6c ⟩
◇
```

⟨ *Import* 4b ⟩ ≡

```
import numpy as np

from numpy.linalg import norm
from numpy.linalg import cholesky as chol
from scipy.linalg import solve_triangular
◇
```

Fragment referenced in 4a.

⟨ *Global variables* 4c ⟩ ≡

```
OPTIMAL                 =   0
OVERSIZED_BASIS         = -19
LINDEP_BASIS            = -20
INIT_ERROR              = -21
NO_WAY_TOI              = -22
CHOLIN_CRASH            = -23
BAD_OLD_L               = -24
MULTI_IN                = -25
DBLE_VTX                = -26
EXCESS_ITER             = -27
RUNNING                 = -28
NEGGRAMM                = -29

epsmach = np.finfo(float).eps
◇
```

Fragment referenced in 4a.

⟨ *ptp body* 5a ⟩ ≡

```
def ptp(X, maxit, eps, verbose, kvec0=np.array([]), R0=np.array([])):
    curr = dict()
    info = RUNNING
    ifac = 0

        ⟨ Initialize basis 5b ⟩
        curr['z'] = X[:, curr["kvec"]].dot(curr["lmb"])

    report = {'zz': sumsq(curr['z']), 'zx': 0, 'in': ifac, 'out': 0,
                        'iter': np.zeros(2, dtype=int), 'lenbas': len(curr['kvec'])}

    if verbose >= 0:
                report_iter(report)

        ⟨ Iteration 6b ⟩
        report['zz'] = sumsq(curr['z'])

    if verbose >= 0:
        if info == OPTIMAL:
            print("\nXXXX Solved to optimality\n")
        report_iter(report)
    return curr['z'], eps, report['iter'], curr['lmb'], curr['kvec'], curr['R'], info
```

◇

Fragment referenced in 4a.

Если `kvec0` пустой, то выполняется холодный старт.

⟨ *Initialize basis* 5b ⟩ ≡

```
if len(kvec0) == 0:
        ifac, curr = cold_start(X, curr)
else:
        ⟨ Warm start 5c ⟩
```

◇

Fragment referenced in 5a.

иначе при одинаковых размерностях `kvec0` и `R0` выполняется тёплый старт, если их размерности отличаются, то выполняется холодный старт.

⟨ *Warm start* 5c ⟩ ≡

```
if R0.shape[0] != len(kvec0):
        ⟨ Generate error 6a ⟩
        ifac, curr = cold_start(X, curr)
else:
        curr['kvec'] = kvec0
        curr['R'] = R0
        lmb = np.linalg.solve(R0, np.linalg.solve(R0.T, np.ones(kvec0.shape[0])))
        curr['lmb'] = lmb / sum(lmb)
```

◇

Fragment referenced in 5b.

⟨ *Generate error* 6a ⟩ ≡

```
info = INIT_ERROR
print("XXXX INIT_ERROR: nonmatching sizes of kvec0 {}, {}".format(kvec0.shape[0], 1))
print(" and R0 {}, {}.".format(*R0.shape))
print(" Reverting to the cold start.")
```

◇

Fragment referenced in 5c.

⟨ *Iteration* 6b ⟩ ≡

```
while (report['iter'] <= maxit).all():
        vmin, ifac = get_ifac(X, curr['z'], eps)
        report['zx'] = vmin
        report['in'] = ifac
        if ifac == -1:
                info = OPTIMAL
            break

        curr['kvec'], curr['lmb'], curr['R'], del_iter = \
newbas(curr['kvec'], curr['lmb'], ifac, curr['R'], X)
        curr['z'] = X[:, curr['kvec']].dot(curr['lmb'])

        report['iter'][0] += 1
        report['iter'][1] += del_iter
        report['zz'] = sumsq(curr['z'])
        report['lenbas'] = len(curr['kvec'])
        if verbose > 0 and (report['iter'][0] % verbose == 0):
                report_iter(report)
```

◇

Fragment referenced in 5a.

3 Вспомогательные функции

⟨ *Subfunctions* 6c ⟩ ≡

⟨ *proplus* 8 ⟩
⟨ *newbas* 9b ⟩
⟨ *Get improvement factor* 10b ⟩
⟨ *Cholesky last insert* 10c ⟩
⟨ *Cold start* 11a ⟩
⟨ *Report generator* 11b ⟩
⟨ *baric* 11c ⟩
⟨ *midlambda* 11d ⟩
⟨ *Other functions* 12a ⟩

◇

Fragment referenced in 4a.

3.1 proplus

Данная функция — это основной вычислительный блок всей программы. Функция выполняет проектирование на аффинную оболочку подмножетсва точек из X:

$$\min ||x||^2$$
$$x = \sum_{i \in I} \mu_i \hat{x}^i \tag{1}$$

$$\sum_{i \in I} \mu_i = 1,$$

где $I \subset \{1, 2, \ldots, m\}$, а m – число точек в X. На вход подаётся:

- `kvec` — базис;
- `ifac` — номер вектора из X, который будет добавлен в базис;
- `R` — верхнее треугольное разложение Холецкого;
- `X` — вершины полиэдра.

Для начала перепишем задачу (1) в следующем виде:

$$\min \frac{1}{2}||z||^2$$

$$z = G\lambda + \xi g \tag{2}$$

$$e^T\lambda + \xi = 1$$

где λ и ξ - барицентрические координаты проекции вектора z по отношению к старому базису и вектору g, e – вектор единиц. Под векторами подразумеваются векторы–столбцы.

Оптимальные условия для задачи (2) следующие:

$$H\lambda + \xi G^T g + \theta e = 0, \tag{3}$$

$$g^T G\lambda + \xi||g||^2 + \theta = 0, \tag{4}$$

$$e^T\lambda + \xi = 1. \tag{5}$$

где θ - множитель Лагранжа в условии нормировки. В целях упрощения примем, что $r = G^T g$, тогда системы (3)-(5) можно переписать в виде:

$$H\lambda + \xi r + \theta e = 0, \tag{6}$$

$$r^T\lambda + \xi||g||^2 + \theta = 0, \tag{7}$$

$$e^T\lambda + \xi = 1. \tag{8}$$

Данная система может быть легко решена при помощи следующих шагов.

3.1.1 Вычисление λ

Вычислить λ можно, решив (3):

$$\lambda = -\theta H^{-1}e - \xi H^{-1}r = -(\xi p + \theta q), \tag{9}$$

где $p = H^{-1}r$, $r = G^T g$, $q = H^{-1}e$.

Уравнение (9) может быть записано более компактно:

$$\lambda = -Zx_\theta \tag{10}$$

где Z - матрица $m \times 2$, столбцы которой являются векторы q и p, то есть:

$$Z = ||pq|| = ||H^{-1}rH^{-1}e|| = H^{-1}||re|| = H^{-1}R_e.$$

И столбцы матрицы R_e – векторы r, e. Вектор x_θ состоит из двух компонентов ξ, θ (именно в таком порядке).

3.1.2 Вычисление x_θ

Проведя подстановку (9) в (4) и (5), получаем следущие уравнения:

$$-r^T Z x_\theta + \xi ||g||^2 + \theta = (h - r^T Z) x_\theta = a_1 x_\theta = 0,$$

где $h = (||g||^2, 1)^T$ и

$$-e^T Z x_\theta + \theta = -e^T Z x_\theta + f^T x_\theta = a_2 x_\theta = 1,$$

где $f = (0, 1)^T$.

Эта система может быть записана как:

$$Ax_\theta = b,$$

$$A = \begin{Vmatrix} a_1 \\ a_2 \end{Vmatrix} = \begin{Vmatrix} h - r^T Z \\ -e^T Z + f \end{Vmatrix},$$

$$b = (0, 1)^T.$$

Решение x_θ данной системы 2×2 определяет λ, исходя из (10).

3.1.3 Реализация

Решение для этой системы реализовано в функции `proplus`.

На вход подаётся:

- `kvec` — текущий базис;
- `ifac` — индекс нового вектора, добавленного в базис;
- `R` — разложение Холецкого;
- `X` — вершины многогранника.

Функция возвращает барицентрические координаты λ проекции вектора на аффинную оболочку расширенного базиса `[kvec, ifac]`.

⟨ *proplus* 8 ⟩ ≡

```
def proplus(kvec, ifac, R, X):
    r = X[:, ifac] @ X[:, kvec]
    if len(kvec) == X.shape[0]:
                ⟨ Shortcut 9a ⟩
    Re = np.c_[r, np.ones(r.shape)]
    Z = solve_chol(R, Re)
    A = np.c_[[sumsq(X[:, ifac]), 1], [1, 0]] - Re.T.dot(Z)
    xit = np.linalg.inv(A)[:, 1]
    return np.r_[-Z @ xit, xit[0]]
```

◇

Fragment referenced in 6c.

Во фрагменте ниже мы пытаемся упростить вычисления, когда число векторов в текущем базисе `kvec` достигает размерности пространства. Конечно в этом случае аффинная оболочка базиса и дополнительный вектор (вектор, который предлагается добавить в базис) совпадают со всем пространством. Автоматически самый короткий вектор становится нулевым и двойтсвенная переменная для условия нормировки также становится нулевой.

Соответственно оптимальная λ может быть вычислена напрямую из уравнения $G\lambda + \xi g = 0$ или

$$G^T G\lambda + \xi G^T g = G^T G\lambda + \xi r = R^T R\lambda + \xi r = 0$$

чтобы сделать возможным использование предварительно–вычисленных множителей $G^T G$. Разделив последнее выражение на ξ, получим $R^T Rz = -r$, где $z = \lambda/\xi$.

Решение z^* этой системы позволяет сделать замену в условии нормировки:

$$1 = e^T\lambda + \xi = \xi e^T z^* + \xi$$

что даёт барицентрический вес ξ вектора g и следовательно определяют остальные барицентрические веса $\lambda = \xi z^*$. Это реализовано всего лишь в три строчки кода:

⟨ *Shortcut* 9a ⟩ ≡

```
lmb = solve_chol(R, r)
return 1 / (1 + sum(lmb)) * np.r_[lmb, 1]
```

◇

Fragment referenced in 8.

3.2 newbas — генерация нового базиса

⟨ *newbas* 9b ⟩ ≡

```
def newbas(kvec, lmb_old, ifac, R, X, eps):
        iter = 0

    lmb_new = proplus(kvec, ifac, R, X)

    if all(lmb_new >= -epsmach):
            ⟨ Suitable new basis – return right away 9c ⟩

    lmb, izero = mid_lambda(np.r_[lmb_old, 0], lmb_new)
    if izero == -1:
        print(" ST-OPT !!!")
        exit(-1)

        ⟨ Delete – add 9d ⟩

    while any(lmb_new < -epsmach):
            ⟨ Delete neg 10a ⟩
        iter += 1
    return kvec, lmb_new, R, iter
```

◇

Fragment referenced in 6c.

⟨ *Suitable new basis – return right away* 9c ⟩ ≡

```
kvec = np.r_[kvec, ifac]
R = lastadd(X[:, kvec], R)
return kvec, lmb_new, R, iter
```

◇

Fragment referenced in 9b.

⟨ *Delete – add* 9d ⟩ ≡

```
kvec = np.delete(kvec, izero)
lmb = np.delete(lmb, izero)
R = choldelete(R, izero)
kvec = np.r_[kvec, ifac]
R = lastadd(X[:, kvec], R)
lmb_new = baric(R)
```

◇

Fragment referenced in 9b.

⟨ *Delete neg* 10a ⟩ ≡

```
lmb, izero = mid_lambda(lmb, lmb_new)
kvec = np.delete(kvec, izero)
lmb = np.delete(lmb, izero)
R = choldelete(R, izero)
lmb_new = baric(R)
```
◇

Fragment referenced in 9b.

3.3 get_ifac

На каждой итерации происходит добавление нового вектора в базис, но для этого необходимо правильно выбрать вектор, чем и занимается функция ниже.

⟨ *Get improvement factor* 10b ⟩ ≡

```
def get_ifac(X, z, epstol):
    v = z.dot(X) - sumsq(z)
    ifac = np.argmin(v)
    vmin = v[ifac]
    reps = epstol * norm(X[:, ifac])
    if vmin > -reps:
        ifac = -1
    return vmin, ifac
```
◇

Fragment referenced in 6c.

3.4 lastadd — добавление вектора в разложение Холецкого

Добавление нового вектора в базис всегда осуществляется в конец базиса. Этот факт позволяет намного упростить алгоритм, поэтому была написана специальная функция, которая заново вычисляет разложение Холецкого при добавлении вектора, но делает это быстрее функции cholesky из NumPy.

Предположим, есть две матрицы X и $\overline{X}$, где $\overline{X} = |Xz|$, то есть $\overline{X}$ была получена добавление вектора столбца z в конец матрицы X. Известно, что $X^TX = R^TR$, где R - верхнее треугольное разложение Холецкого для X^TX. Какое же тогда будет верхнее разложение Холецкого $\overline{R}$ такое, что $\overline{X}^T\overline{X} = \overline{R}^T\overline{R}$ и как можно найти $\overline{R}$ с наименее возможными вычислительными затратами, используя R?

Функция принимает на вход:

- `X` — вершины полиэдра;
- `R` — верхнее треугольное разложение X^TX без последней строки и столбца.

Функция возвращает верхнее треугольное разложение для X^TX.

⟨ *Cholesky last insert* 10c ⟩ ≡

```
def lastadd(X, R):
    u = X[:, X.shape[1]-1] @ X
    q = solve_triangular(R.T, u[0:len(u) - 1].T, lower=True, check_finite=False)
    zz = np.sqrt(abs(u[-1] - sumsq(q)))
    RU = np.r_[R, np.zeros((1, R.shape[1]))]
    return np.c_[RU, np.r_[q, zz]]
```
◇

Fragment referenced in 6c.

3.5 cold_start

Для холодного старта выбирается базис с минимальной длиной.

⟨ *Cold start* 11a ⟩ ≡

```
def cold_start(X, curr):
    ifac = np.argmin(sumsq(X))
    curr['R'] = norm(X[:, ifac])
    curr['kvec'] = np.array([ifac])
    curr['lmb'] = np.array([1])
    return ifac, curr
```

◇

Fragment referenced in 6c.

3.6 report_iter

Вывод данных на итерации.

⟨ *Report generator* 11b ⟩ ≡

```
def report_iter(report):
    print(" ++", end='')
    print(" iter {iter[0]:4d}(+) {iter[1]:4d}(-)".format(iter=report["iter"]), end='')
    print(" len {:4d}".format(report["lenbas"]), end='')
    print(" zx {:12.4e}".format(report["zx"]), end='')
    print(" in {:6d}".format(report["in"]), end='')
    print(" zz {:16.8e}".format(report["zz"]))
```

◇

Fragment referenced in 6c.

3.7 baric

⟨ *baric* 11c ⟩ ≡

```
def baric(R):
    lmb = solve_chol(R, np.ones(R.shape[0]))
    return lmb / sum(lmb)
```

◇

Fragment referenced in 6c.

3.8 mid_lambda

⟨ *midlambda* 11d ⟩ ≡

```
def mid_lambda(lmb_old, lmb_new):
    if all(lmb_new < 0):
        return lmb_new, -1
    lmb = (lmb_old / (lmb_old - lmb_new))[lmb_new < -epsmach]
    imin = np.argmin(lmb)
    izero = np.array([i for i in range(lmb_new.shape[0])])[lmb_new < -epsmach][imin]
    return lmb[imin] * lmb_new + (1 - lmb[imin]) * lmb_old, izero
```

◇

Fragment referenced in 6c.

3.9 Остальные функции

⟨ *Other functions* 12a ⟩ ≡

⟨ *sumsq* 12b ⟩
⟨ *Cholesky delete* 12d ⟩
⟨ *Solve Cholesky* 12c ⟩
⟨ *Cone Projection* 13 ⟩
◇

Fragment referenced in 6c.

Сумма квадратов.

⟨ *sumsq* 12b ⟩ ≡

```
def sumsq(A):
    return sum(A ** 2)
```
◇

Fragment referenced in 12a.

Для решения системы с помощью разложения Холецкого необходимо решить вспомогательную систему:

$$\begin{cases} R^T y = b \\ Rx = y \end{cases}$$

⟨ *Solve Cholesky* 12c ⟩ ≡

```
def solve_chol(R, b):
    return solve_triangular(R, solve_triangular( \\
                R.T, b, lower=True, check_finite=False), check_finite=False)
```
◇

Fragment referenced in 12a.

3.9.1 choldelete — удаление вектора в разложении Холецкого

При удалении вектора разложение Холецкого изменяется в зависимости от положения удалённого вектора, поэтому для вычисления нового разложения с помощью старого можно пересчитывать только некоторую часть, что сокращает время работы программы.

⟨ *Cholesky delete* 12d ⟩ ≡

```
def choldelete(R, i_del):
    rows = R.shape[0]
    S0 = np.array([R[i_del, (i_del+1):]])
    S1 = R[(i_del+1):, (i_del+1):]
    R = np.delete(R, i_del, 1)
    R = np.delete(R, [range(i_del, rows)], 0)
    S = np.linalg.cholesky(S1.T.dot(S1) + S0.T.dot(S0)).T
    return np.r_[R, np.c_[np.zeros((rows-i_del-1, i_del)), S]]
```
◇

Fragment referenced in 12a.

3.10 Проекция на конус

Если требуется найти проекцию точки на конус, то можно воспользоваться следующей функцией:

⟨ *Cone Projection* 13 ⟩ ≡

```
def cone_project(A, b, m, mstep, eps, maxiterptp):
    while True:
        z, _, _, lmb, kvec, _, info = ptp(np.c_[m * A - b.reshape(len(b), 1), -b],
                                          maxiterptp, eps, -1)
        if A.shape[1] in kvec:
            return z + b, m, kvec[kvec != A.shape[1]], lmb[kvec != A.shape[1]]
        m *= mstep
```

◇

Fragment referenced in 12a.

Конус задаётся набором вектором `A`, которые указывают его направление. Конус бесконечно распространён по этим направлением в пространстве, а алгоритм подходящих аффинных подпространств работает с выпуклой оболочкой, которая дискретна. Для нахождения проекции точки b на конус можно задать некоторое малое число m, которое будет растягивать или стягивать конус, то есть, домножив все направления на m, получим некоторую часть конуса и на эту часть будем проектировать. После этого необходимо убедиться, что найдена проекция точки на весь конус. Это достигается проверкой наличия точки b в базисе. Сформулируем данную идею более строго.

Если конус задан набором векторов $a^1, \ldots, a^N$, то точка z является проекцией точки b на конус, если при нахождении проекции алгоритмом подходящих аффинных пространств на выпуклую оболочку $\text{co}\{ma^1 - b, \ldots, ma^N - b\}$, где $m > 0$, в базисе оказалась точка $-b$.

Функция `cone_project` проектирует на часть конуса, ограниченного величиной m. Если сказанное выше условие не выполняется, то m увеличивается в δ раз.

Функция принимает на вход:

- `A` — набор векторов, задающий конус;
- `b` — точка, проекцию которой требуется найти;
- `m` — начальное значение m;
- `mstep` — величина шага δ;
- `eps` — точность для `ptp`;
- `maxiterptp` — максимальное количество итераций для `ptp`.

И возвращает:

- `z` — проекция точки b на конус;
- `m` — величина m;
- `kvec` — базис;
- `lmb` — барицентрические координаты точки z в базисе.