



Solving generic linear program using Chubanov algorithm and simple projections.

Adrien Chan-Hon-Tong

► To cite this version:

Adrien Chan-Hon-Tong. Solving generic linear program using Chubanov algorithm and simple projections.. 2019. hal-00722920v11

HAL Id: hal-00722920

<https://hal.science/hal-00722920v11>

Preprint submitted on 15 Mar 2019 (v11), last revised 16 Jan 2023 (v38)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Solving generic linear program using Chubanov algorithm and simple projections.

Adrien CHAN-HON-TONG
ONERA - université paris sud

March 15, 2019

Abstract

This short article presents an algorithm which solves generic linear program ($Ax \geq b$ or $\min_{Ax \geq b} cx$) using simple projections and call to a sub solver dedicated to linear feasibility ($Ax = \mathbf{0}$, $x > \mathbf{0}$). Such link could be interesting since linear feasibility can be solved in strong polynomial time thank to Chubanov algorithm.

1 Introduction

Interior point algorithms (e.g. [3]) are mainly the state of the art of linear program solver.

Yet, interior point algorithms may not allow to solve linear program in strong polynomial time (indeed, [1] shows that major interior point family can not). This drawback is, may be, linked to the fact that these algorithms work directly at raw numerical level, and, hence scaling the linear program may result in different move of the interior point.

This short article presents an algorithm which prevents sensibility to raw numerical level by forcing interior point moves to be matched with a set of constraints. In addition, this algorithm links generic linear program and linear feasibility. This last point could be interesting since linear feasibility can be solved in strong polynomial time thank to Chubanov algorithm [2].

Indeed, the offered algorithm only performs two kind of operations: computing simple projections or calling a sub solver on a linear feasibility problem. Thus, thank to Chubanov algorithm [2], his complexity is directly linked to the number of call of the sub solver: a strong polynomial bound on the number of linear feasibility queries will lead to a strong polynomial algorithm. Such bound is unfortunately not established in this short paper. But, this kind of algorithm could be interesting as one of them may reach such property.

Next section recalls some useful lemmas, and, presents a pre processing required for the main algorithm presented in section 3. Termination of the algorithm is proven in section 4 before perspectives.

Notation

In this paper, \mathbb{Q} is the set of rational number, \mathbb{Q}^I the set of I dimensional vector of \mathbb{Q} , \cdot_i will design the component i of the vector. $\mathcal{M}_{J,I}(\mathbb{Q})$ is the set of matrix with J rows and I columns, with values in \mathbb{Q} .

Less classically, if p and q are two vectors of \mathbb{Q}^I , then pq will be $p^T q = \sum_n p_n \times q_n$ (transposition is omitted in scalar product). Also, $\mathbb{U}(\mathbb{Q}^I)$ will be the set of normalized vector from \mathbb{Q}^I i.e. $v \in \mathbb{U}(\mathbb{Q}^I) \Leftrightarrow vv = 1$. $\mathbb{U}(\mathcal{M}_{J,I}(\mathbb{Q}))$ will be the set of matrix from $\mathcal{M}_{J,I}(\mathbb{Q})$ whose rows are in $\mathbb{U}(\mathbb{Q}^I)$.

2 Some useful lemmas

2.1 Linear feasibility and linear separability

[2] presents an algorithm to solve in strongly polynomial time the following problem:

$$\exists ?x \in \mathbb{Q}^N / Ax = \mathbf{0}, x > \mathbf{0}$$

under the assumption that $A \in \mathcal{M}_{M,N}(\mathbb{Q})$ has a rank of M .

In this short paper, raw linear feasibility solver can not be used directly. This is why a minor but required lemma is introduced here.

Chubanov algorithm can be applied on a derived problem to solve linear separability $\exists ?x \in \mathbb{Q}^N / Ax > \mathbf{0}$ under no assumption.

Indeed, let $\mathcal{A} \in \mathcal{M}_{M,N}(\mathbb{Q})$ (without requiring anything about the rank). The goal is to solve $\exists ?x \in \mathbb{Q}^N / Ax > \mathbf{0}$.

It is sufficient, for that, to consider the matrix $A = \begin{pmatrix} \mathcal{A} & -\mathcal{A} & -I \end{pmatrix}$ formed with \mathcal{A} concat with $-\mathcal{A}$ concat with $-I$ the opposite of identity matrix.

First, this matrix has full rank M due to the identity block.

Then if a solution exists, applying the Chubanov algorithm (or any linear feasibility solver) to this matrix A will lead to x_1, x_2, x_3 such that

$$\begin{pmatrix} \mathcal{A} & -\mathcal{A} & -I \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \mathbf{0}$$

and $\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} > \mathbf{0}$ So, $x = x_1 - x_2$ and $Ax = Ix_3 = x_3 > \mathbf{0}$. And, if no solution exists, then Chubanov algorithm will provide a certificate.

2.2 Normalizing linear program

If the linear program given as input is $\min_{Ax \geq b} cx$ and verifies $A \in \mathbb{U}(\mathcal{M}_{J,I}(\mathbb{Q}))$, $b \in \mathbb{Q}^M$, $c \in \mathbb{U}(\mathbb{Q}^N)$ and $Ac = \gamma \mathbf{1}$ with $\gamma > 0$, and, cx being bounded by 0, then the offered algorithm can be directly used.

Otherwise, the linear program has to be normalized with the following scheme:

1. If the linear program is as an optimisation problem (e.g. $\max_{Ax \leq b, x \geq \mathbf{0}} cx$), it should first be converted into a inequality system $A'x \geq b'$. This could be done by combining primal and dual.
2. After that (or directly if input was an inequality system), an other normalisation is performed to reach required property (here with $\gamma = \frac{3}{5}$)
3. the important point is that from any linear program, pre processing can form an equivalent linear program meeting these requirements

2.2.1 Primal dual

The conversion of an optimisation linear program into a linear inequality system is quite classical. A brief recall is provided below.

Let assume original goal is to solve $\max_{A_{raw}x \leq b_{raw}, x \geq \mathbf{0}} c_{raw}x$. It is well known that the dual problem is $\min_{A_{raw}^T y \geq c_{raw}, y \geq \mathbf{0}} b_{raw}y$. Now, the primal dual is formed by combining all constraints: $A_{raw}x \leq b_{raw}$, and, $x \geq \mathbf{0}$, and, $A_{raw}^T y \geq c_{raw}$, and $c_{raw}x = b_{raw}y$, and finally, $y \geq \mathbf{0}$.

So, the problem $\max_{A_{raw}x \leq b_{raw}, x \geq \mathbf{0}} c_{raw}x$ can be folded into $A_{big}x_{big} \geq b_{big}$ with

$$A_{big} = \begin{pmatrix} -A_{raw} & 0 \\ I & 0 \\ 0 & A_{raw}^T \\ 0 & I \\ c_{raw} & -b_{raw} \\ -c_{raw} & b_{raw} \end{pmatrix} \text{ and } b_{big} = \begin{pmatrix} -b_{raw} \\ 0 \\ c_{raw} \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

2.2.2 Normalized primal dual error minimization

This normalisation step takes a linear program $\Gamma\chi \geq \beta$ as input, and, produces an equivalent linear program $\min_{Ax \geq b} cx$ with $A \in \mathbb{U}(\mathcal{M}_{J,I}(\mathbb{Q}))$, $b \in \mathbb{Q}^M$, $c \in \mathbb{U}(\mathbb{Q}^N)$, and, $Ac = \frac{3}{5}\mathbf{1}$, and, cx being bounded by 0.

All this normalization is entirely done in \mathbb{Q} i.e. no square root are needed.

It is sufficient to consider:

$$A = \begin{pmatrix} \frac{4}{5(\frac{\Gamma_1\Gamma_1}{2}+1)}\Gamma_1 & \frac{4}{5(\frac{\Gamma_1\Gamma_1}{2}+1)}\frac{\Gamma_1\Gamma_1}{2} & \frac{4}{5(\frac{\Gamma_1\Gamma_1}{2}+1)} & \frac{3}{5} \\ \dots & \dots & \dots & \dots \\ \frac{4}{5(\frac{\Gamma_M\Gamma_M}{2}+1)}\Gamma_M & \frac{4}{5(\frac{\Gamma_M\Gamma_M}{2}+1)}\frac{\Gamma_M\Gamma_M}{2} & \frac{4}{5(\frac{\Gamma_M\Gamma_M}{2}+1)} & \frac{3}{5} \\ \mathbf{0} & \frac{4}{5} & 0 & \frac{4}{5} \\ \mathbf{0} & -\frac{4}{5} & 0 & \frac{4}{5} \\ \mathbf{0} & 0 & \frac{4}{5} & \frac{4}{5} \\ \mathbf{0} & 0 & -\frac{4}{5} & \frac{4}{5} \end{pmatrix}$$

and

$$b = \begin{pmatrix} \frac{4}{5(\frac{\Gamma_1\Gamma_1}{2}+1)}\beta_1 \\ \dots \\ \frac{4}{5(\frac{\Gamma_M\Gamma_M}{2}+1)}\beta_M \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad c = \begin{pmatrix} 0 \\ \dots \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

First, the produced linear program is in the desired form: $\min_{Ax \geq b} cx$ with $A \in \mathbb{U}(\mathcal{M}_{J,I}(\mathbb{Q}))$, $b \in \mathbb{Q}^M$, $c \in \mathbb{U}(\mathbb{Q}^N)$, and, $Ac = \frac{3}{5}\mathbf{1}$.

Trivially, $Ac = \frac{3}{5}\mathbf{1}$ by construction, and, all rows of A are normalized either directly because $(\frac{4}{5})^2 + (\frac{3}{5})^2 = 1$, or, because of that, and the fact that, $(\frac{1}{\frac{\Gamma_m\Gamma_m}{2}+1})^2\Gamma_m\Gamma_m + (\frac{1}{\frac{\Gamma_m\Gamma_m}{2}+1})^2\frac{(\Gamma_m\Gamma_m)^2}{4} + (\frac{1}{\frac{\Gamma_m\Gamma_m}{2}+1})^2$ is $(\frac{1}{\frac{\Gamma_m\Gamma_m}{2}+1})^2 \times (\Gamma_m\Gamma_m + \frac{(\Gamma_m\Gamma_m)^2}{4} + 1)$ which is $(\frac{1}{\frac{\Gamma_m\Gamma_m}{2}+1})^2 \times (\frac{\Gamma_m\Gamma_m}{2} + 1)^2$ which is 1 !

Then, the 4 last constraint prevent x_{N+3} to be negative so cx is well bounded by 0. Indeed, if $x_{N+2} + x_{N+3} \geq 0$ and $-x_{N+2} + x_{N+3} \geq 0$, then $x_{N+3} \geq 0$, and, when $x_{N+3} = 0$ these constraints force $x_{N+1} = x_{N+2} = 0$ because the three constraints $x_{N+2} + x_{N+3} \geq 0$, $-x_{N+2} + x_{N+3} \geq 0$, and $x_{N+3} = 0$ can be reduced to $x_{N+2} \geq 0$ and $-x_{N+2} \geq 0$ which force $x_{N+2} = 0$.

Now, the goal is to minimize $cx = x_{N+3}$. So, either the minimum is $x_{N+3} = 0$ or either there is no such solution. In the case $Ax \geq b, x_{N+3} = 0$, it holds $x_{N+1} = x_{N+2} = x_{N+3} = 0$, and, $x_1, \dots, x_N = \chi$ with $\frac{4}{5(\frac{\Gamma_m\Gamma_m}{2}+1)}\Gamma_m\chi \geq \frac{4}{5(\frac{\Gamma_m\Gamma_m}{2}+1)}\beta_m$. But this last inequality can be reduced to $\Gamma_m\chi \geq \beta_m$. So, if the solution of the derived linear program is x with $Ax \geq b, x_{N+3} = 0$, then $x_1, \dots, x_N = \chi$ is a solution of the original set of inequality.

And, inversely, if there is a solution χ , then, $x = \chi, 0, 0, 0$ is a solution of the optimisation problem (because x_{N+3} is bounded by 0).

So, this derived linear program is equivalent to the inequality set.

3 Algorithm

3.1 Key points

Before introducing the algorithm, I present here some key points.

- The most interesting point is that this algorithm links generic linear program to linear feasibility. Indeed, the algorithm is very simple, all difficulties being forwarded into the linear feasibility solver (e.g. Chubanov algorithm [2]).
- To delegate difficulties to the linear feasibility solver, the algorithm keeps the current point on the interior of the admissible space ($Ax > b$). The

current point performs trivial greedy move when such trivial move exists, or, call the sub solver to find a jump move. All moves keep the current point into the inner space ($Ax > b$). Currently, working in the inner space may not be that required because one could then project on the border of the admissible space ($Ax = b$). Yet, working on the inner space is even easier.

- The fact to work in the inner space is made possible because $Ac > \mathbf{0}$. Indeed, $Ac > \mathbf{0}$ allows to build a trivial admissible point λc for some large λ , and, allows to build an inner point for any admissible point (if $Ax = b$, $A(x + \varepsilon c) > b$). More precisely, if x^* is optimal and x an admissible non optimal point then there exist an inner point better than x (see section 4). The algorithm even assumes $Ac = \frac{3}{5}\mathbf{1}$ for convenience.
- Each call to the sub solver is matched with a specific vertex/corner of the admissible space. Yet, very differently from the simplex algorithm, moves of the current point are not linked to moves along edges. Precisely, between two consecutive calls to the sub solver, the sets of saturated constraint can be completely disjoint (most probably reduced to a single point depending in some implementation detail). So the main idea of the algorithm is not to explore vertex/corner like in simplex, but, mostly to explore faces. And the key point is that there is an exponential number of vertex but a linear number of faces. Currently, in numerical toy experiments, the algorithm definitely rejects one constraint between two call of the sub solver.
- Unfortunately, in this short paper, no bounds are proven on the number of explored vertex/corner, neither on the rejection of a face between two consecutive calls. So this algorithm despite being very different from the simplex may have similar drawback of exploring an exponential number of vertex/corner. I only prove that the algorithm terminates (this is easy as the set of constraints during a sub solver call can not be twice the same). But, maybe such kind of algorithm that links generic linear program to linear feasibility can be updated to have a polynomial number of calls to the sub solver (then such algorithm may be strongly polynomial by using Chubanov algorithm as sub solver).

3.2 Pseudo code

N and M are the numbers of variables and constraints. The input linear program is $\min_{Ax \geq b} cx$ with $A \in \mathbb{U}(\mathcal{M}_{J,I}(\mathbb{Q}))$, $b \in \mathbb{Q}^M$, $c \in \mathbb{U}(\mathbb{Q}^N)$, and, $Ac = \frac{3}{5}\mathbf{1}$, and, cx being bounded by 0.

The pseudo code the algorithm (which starts from a trivial interior point x e.g. $(1 + \frac{5}{3}\max_m b_m)c$) is:

1. compute $d = \min_m A_m x - b_m$
2. compute D the set of planes at distance d
3. compute $p = \max_{A_m(x-\lambda c) \geq b_m, c(x-\lambda c) \geq 0} \lambda$
4. $z = x - pc$
5. let v be the projection of $-c$ on $\text{Ker}(A_D)$ (may result in $cv < 0$, $A_D v = \mathbf{0}$)
6. if $v \neq \mathbf{0}$
 - (a) compute λ : $\min_{m \notin D / A_m v < 0} \frac{A_m x - b_m - d}{-A_m v}$
 - (b) $x \leftarrow x + \lambda v$
 - (c) GO TO 1
7. call Chubanov algorithm to find θ such that $\begin{pmatrix} A_D \\ -c \end{pmatrix} \theta > \mathbf{0}$
8. if no such θ exists, then return x and the certificate
9. find $\delta > 0$ such that $A(z + \delta\theta) > b$ (by construction $c(z + \delta\theta) < cz$)
10. $x \leftarrow z + \delta\theta$
11. GO TO 1

4 Termination of the algorithm

First, let stress that all moves (steps 6.b or 10) are either not defined or maintain the current point in the inner space $(\{x / Ax > b\})$. Also, trivially cx is decreasing during the algorithm (as 6.b and 10 decrease cx by construction of v and θ if defined).

4.1 The algorithm is well defined

Steps 1, 2, 3, 4, 5, 8, 10, 11 are trivially well defined assuming that all points are still in the admissible space i.e. $\{x/Ax \geq b\}$.

6.a the min is well defined because moving along v decreases the cost cx of x (because $v \neq \mathbf{0}$) which is bounded by 0. So this move can not be done for ever, and, so, necessarily some plane will meet the ball of center x and radius d - it will be added to D . These moves are generalization of bisector moves, but, are very simple projections.

Step 7 is the call to the sub solver (e.g. Chubanov algorithm). The set of saturated constraint on z is exactly D because $Ac = \frac{3}{5}\mathbf{1}$ (moving along c decreases equally the distance to each constraint - remember that every row and c are normalized !).

If z is not the solution then step 7 should return something. Indeed, let consider $\theta = x^* + \frac{cz}{2}c - z$ with x^* being the optimal. As, x^* the solution verifies $Ax^* \geq b$, let define $\rho = x^* + \frac{cz}{2}c$. Then, ρ verifies $A\rho > b$ but still $c\rho = \frac{cz}{2} < cz$ (because $cz > cx^*$ otherwise it would have been a solution). So, $c(\rho - z) > 0$ and $A_D(\rho - z) = A_D\rho > b_D$ (since $A_Dz = 0$ and $A_D\rho > b$). So, I have proven that $\theta = x^* + \frac{cz}{2}c - z$ is a possible solution. Of course, there is NO reason that the solution returned by the call to the Chubanov algorithm will be $\theta = x^* + \frac{cz}{2}c - z$ (if so, one call will be enough). But, still, if a solution exists, Chubanov algorithm should return one.

So either there is no solution and 8 leads to termination, either 9 will be possible seeing the definition of θ : for all $m \notin D$, $A_mz > b_m$ and for all $m \in D$, $A_mz = b_m$ but $A_m\theta > 0$. So finding δ is possible (in $O(NM)$ just by looping over all constraints).

So the algorithm is well defined and all moves are in the inner space ($\{x/Ax > b\}$).

4.2 Breaking the loop

The set D can not have twice the same value in step 7. Indeed, if algorithm reaches two time in step 7 in point x_1 and x_2 with a common value for D , then, let consider the point z_1 and z_2 from step 4. Then, it holds both $cz_2 < cz_1$ and $A_Dz_2 = A_Dz_1 = \mathbf{0}$. So, the algorithm should not have pass the step 6 when meeting x_1 .

In other words, algorithm explores the $\ker(A_D)$ to increase D , then when D is big enough, the algorithm moves below a corner of A_D . As D was big enough at this step, it means that the $\ker(A_D)$ is nul and that D will never be reached again.

So, it can not have more than 2^M step 7.

And, step 6.b strictly increases D which is bounded by $\{1, \dots, M\}$. So, the algorithm can not perform more than $M2^M$ loops. Each loop taking less than a call of the Chubanov algorithm (each of them being strongly polynomial).

5 Perspectives

This short paper presents an algorithm which solves generic linear program ($\max_{Ax \leq b, x \geq \mathbf{0}} cx$) by solving a sequence of linear feasibility ones ($Ax = \mathbf{0}, x \geq \mathbf{0}$). Unfortunately, this short paper does not prove any bound on the size of this sequence.

So, at this point, this short paper is not very interesting. At most, this kind of algorithm could be an interesting way toward strongly polynomial algorithm for linear program and/or a way to link linear feasibility and generic linear program.

Yet, the offered algorithm exhibits interesting behaviour on toy numerical experiments, and, does not rely on vertex exploration like the simplex (but rather in face exploration).

This idea of face exploration is interesting. Currently, here, I prove D can not be twice the same when reaching step 7. But there is a lot of D ... But, the underlying idea behind this idea of face exploration is that if the set D corresponding to the x is two time the same **just after** a step 10, then there is chance that all the following steps 6 should be the same. This would lead to the same D in the next step 7 which has been proven impossible. So, the **not proven** idea is that D can not be twice the same just after a step 10. Yet, just after the step 10, there is large chance that D is a singleton. Indeed, in step 7, the sub solver looks for $A_D\theta > 0$ and not for $A_D\theta = \mu\mathbf{1}$. So, $A_m\theta$ have no reason to be equal, and thus, step 10 should lead to a singleton.

So, if all these not proven statements were true, it would mean that only $M + 1$ steps 10 are possible...

Unfortunately, step 7 can select θ such that lot of $m \in D$ have same $A_m\theta$ even if just required to produce $A_D\theta > 0$ (this way the number of D after step 10 is still large).

References

- [1] Xavier Allamigeon, Pascal Benchimol, Stéphane Gaubert, and Michael Joswig. Log-barrier interior point methods are not strongly polynomial. *SIAM Journal on Applied Algebra and Geometry*, 2(1):140–178, 2018.
- [2] Sergei Chubanov. A polynomial projection algorithm for linear feasibility problems. *Mathematical Programming*, 153(2):687–713, 2015.
- [3] Yurii Nesterov and Arkadii Nemirovskii. *Interior-point polynomial algorithms in convex programming*, volume 13. Siam, 1994.

```

from __future__ import print_function

#####
##### MAIN ALGORITHM
#####

from fractions import Fraction

##### REQUIRED BASIC LINEAR ALGEBRA FUNCTIONS #####

def combinaisonlineaire(u,l,v):
    w = []
    for i in range(len(u)):
        w.append(u[i]+l*v[i])
    return w

def allocateVector(N):
    output = []
    for n in range(N):
        output.append(Fraction())
    return output

def produitscalairevecteur(l,v):
    return combinaisonlineaire(allocateVector(len(v)),l,v)

def produitscalaire(u,v):
    w = Fraction()
    for i in range(len(u)):
        w+=u[i]*v[i]
    return w

def allocateVector(N):
    output = []
    for n in range(N):
        output.append(Fraction())
    return output

def projection(u, BOG):
    pu = allocateVector(len(BOG[0]))
    for v in BOG:
        pu=combinaisonlineaire(pu,produitscalaire(u,v)/produitscalaire(v,v),v)
    return pu

def gramschimdBOG(H_):
    H = [h.copy() for h in H_ if produitscalaire(h,h) != Fraction()]
    BOG = []
    while len(H)>0:
        BOG.append(H.pop())
        for i in range(len(H)):
            H[i] = combinaisonlineaire(H[i],Fraction(-1),projection(H[i],BOG))
        H = [h for h in H if produitscalaire(h,h) != Fraction()]
    return BOG

##### ALGORITHM #####

def solvenormalized(A,b,c,xoptimal,x):
    M = len(A)
    N = len(A[0])

    while True:
        print(x)

        d = produitscalaire(A[0],x)-b[0]
        for m in range(M):
            if produitscalaire(A[m],x)-b[m]<d:

```

```

        d = produitscalaire(A[m],x)-b[m]

    if d<Fraction():
        print("d<Fraction()")
        quit()
    if d==Fraction() and produitscalaire(x,c)!=produitscalaire(xoptimal,c):
        print("d==Fraction() and produitscalaire(x,c)!
=produitscalaire(xoptimal,c)")
        quit()
    print(d)

    D = []
    for m in range(len(A)):
        if produitscalaire(A[m],x)-b[m]==d:
            D.append(m)
    print(D)

    if produitscalaire(x,c)==produitscalaire(xoptimal,c):
        quit()

    projectionOnVect = projection(produitscalairevecteur(Fraction(-1),c),
gramschimdB0G([A[m].copy() for m in D]))
    projectionOnKer =
combinaisonlineaire(produitscalairevecteur(Fraction(-1),c),Fraction(-1),projectionOnVect)

    for m in D:
        if produitscalaire(A[m],projectionOnKer)!=Fraction():
            print("produitscalaire(A[m],projectionOnKer)!=Fraction()")
            quit()
        if produitscalaire(c,projectionOnKer)>Fraction():
            print("produitscalaire(c,projectionOnKer)>Fraction()")
            quit()
        if produitscalaire(c,projectionOnKer)==Fraction() and
produitscalaire(projectionOnKer,projectionOnKer)!=Fraction():
            print("produitscalaire(c,projectionOnKer)==Fraction() and
produitscalaire(projectionOnKer,projectionOnKer)!=Fraction()")
            quit()

    v = projectionOnKer
    print(v)

    if produitscalaire(v,v)!=Fraction():
        candidate = []
        for m in range(M):
            if produitscalaire(v,A[m])<Fraction():
                currentdir = produitscalaire(A[m],x)-b[m]
                desiredir = d
                move = -(currentdir-desiredir)/produitscalaire(v,A[m])
                candidate.append(move)

        if candidate == []:
            print("candidate == []")
            quit()

        move = min(candidate)
        if move == Fraction():
            print("move == Fraction()")
            quit()

        x = combinaisonlineaire(x,move,v)
        print("### bisector move ###")
        continue

    y =combinaisonlineaire(x, -d/produitscalaire(A[0],c) ,c)

    if produitscalaire(y,c)==Fraction():

```

```

        x = y
        print("done")
        continue

    optANDc = combinaisonlineaire(xoptimal,produitscalaire(y,c)/Fraction(2),c)
    y09 = produitscalairevecteur(Fraction(999,1000),y)
    simulateChubanov = combinaisonlineaire(y09,Fraction(1,1000),optANDc)
    x = optANDc
    print("### chubanov jump ###")
    continue

#####
##### PRE PROCESSING
#####

def allocateMatrix(M,N):
    mymatrix = []
    for m in range(M):
        mymatrix.append(allocateVector(N))
    return mymatrix

def primaldual(rawA,rawb,rawc):
    #primal: max {rawc rawx / rawA rawx<= rawb, rawx>=0}
    #dual: min {rawb rawy / transpose(rawA) rawy>= rawc, rawy>=0}
    #primal dual: {rawx / rawA rawx<=rawb, rawx>=0, transpose(rawA) rawy >=rawc,
    rawy>=0, rawc rawx=rawb rawy} unfolded into A x >= b
    M = len(rawA)
    N = len(rawA[0])
    A = allocateMatrix(M+N+N+M+2,N+M)
    b = allocateVector(M+N+N+M+2)

    offsetY = N
    offset = 0
    for m in range(M):
        for n in range(N):
            A[m+offset][n] = -rawA[m][n]
            b[m+offset] = -rawb[m]

    offset += M
    for n in range(N):
        A[n+offset][n] = 1

    offset += N
    for n in range(N):
        for m in range(M):
            A[n+offset][m+offsetY] = rawA[m][n]
            b[n+offset] = rawc[n]

    offset += N
    for m in range(M):
        A[m+offset][m+offsetY] = 1

    for n in range(N):
        A[-2][n] = rawc[n]
    for m in range(m):
        A[-2][m+offsetY] = -rawb[m]

    for n in range(N):
        A[-1][n] = -rawc[n]
    for m in range(m):
        A[-1][m+offsetY] = rawb[m]

    return A,b

def normalize(rawA, rawb, rawxoptimal):

```

```

#input rawA rawx >= rawb
#return A,b,c such that min{cx / Ax>=b} if equivalent
#+ A is normalized, c is normalized, cx is 0 bounded, Ac = 3/4 vector(1)
M = len(rawA)
N = len(rawA[0])
A = allocateMatrix(M+4,N+3)
b = allocateVector(M+4)
c = allocateVector(N+3)
c[-1] = Fraction(1)

normRawA = allocateVector(M)
normRawAtrick = allocateVector(M)
for m in range(M):
    normRawA[m] = produitscaire(A[m],A[m])
    normRawAtrick[m] = normRawA[m]/Fraction(2)+Fraction(1)

for m in range(M):
    for n in range(N):
        A[m][n] = rawA[m][n]*Fraction(4,5)/normRawAtrick[m]
    A[m][-3] = normRawA[m]*Fraction(4,5*2)/normRawAtrick[m]
    A[m][-2] = Fraction(4,5*2)/normRawAtrick[m]
    A[m][-1] = Fraction(3,5)
    b[m] = rawb[m]*Fraction(4,5)/normRawAtrick[m]

A[-4][-3] = Fraction(4,5)
A[-4][-1] = Fraction(3,5)
A[-3][-3] = -Fraction(4,5)
A[-3][-1] = Fraction(3,5)
A[-2][-2] = Fraction(4,5)
A[-2][-1] = Fraction(3,5)
A[-1][-2] = -Fraction(4,5)
A[-1][-1] = Fraction(3,5)

xoptimal = allocateVector(N+3)
for n in range(N):
    xoptimal[n] = rawxoptimal[n]

#check x solution
if produitscaire(c,xoptimal)!=Fraction():
    print("produitscaire(c,xoptimal)!=Fraction()")
    quit()
for m in range(M):
    if produitscaire(A[m],xoptimal)<b[m]:
        print("produitscaire(A[m],xoptimal)<b[m]")
        quit()

x = allocateVector(N+3)
x[-1] = max(b)+1
x[-1] *= Fraction(5,3)
return A,b,c,xoptimal,x

#####
##### TOY EXPERIMENT
#####

import random

def randomVector(N):
    output = []
    for n in range(N):
        output.append(Fraction(random.randint(-100,100)))
    return output

def randomMatrix(M,N):
    mymatrix = []

```

```

    for m in range(M):
        mymatrix.append(randomVector(N))
    return mymatrix

def randomproblem(N,M):
    xoptimal = randomVector(N)

    Aequal = randomMatrix(M,N)
    bequal = allocateVector(M)
    for m in range(M):
        bequal[m] = produitscalaire(Aequal[m],xoptimal)

    Agreater = randomMatrix(M,N)
    bgreater = allocateVector(M)
    for m in range(M):
        bgreater[m] = produitscalaire(Agreater[m],xoptimal)-
Fraction(random.randint(1,100))

    return Aequal+Agreater,bequal+bgreater,xoptimal

#def cubeproblem(N):
#    twopower = allocateVector(N)
#    twopower[0]=2
#    for n in range(1,N):
#        twopower[n] = Fraction(2)*twopower[n-1]
#
#    b = allocateVector(N)
#    b[0] = Fraction(5)
#    for n in range(1,N):
#        b[n] = Fraction(5) * b[n-1]
#
#    c = twopower[::-1]
#
#    A = allocateMatrix(N,N)
#    for n in range(2,N):
#        for k in range(n):
#            A[n][k] = twopower[n-k]
#        A[n][n] = 1
#
#    return A,b,c

rawA,rawb,rawxoptimal = randomproblem(10,10)
print(rawA,rawb,rawxoptimal)

A,b,c,xoptimal,x = normalize(rawA,rawb,rawxoptimal)
print(A,b,c,xoptimal,x)

solvenormalized(A,b,c,xoptimal,x)

```