Solving linear program with Chubanov queries and bisection moves.

Adrien Chan-Hon-Tong

Solving linear program with Chubanov queries and bisection moves.

Adrien CHAN-HON-TONG
ONERA - université paris sud

January 2019

Abstract

This short article focus on the link between linear feasibility and generic linear program. An algorithm is presented to solve generic linear program using linear feasibility queries and working at constraint level instead of raw values level. Even if the number of required linear feasibility queries is not established, this algorithm may be especially interesting, since, thank to Chubanov algorithm, there is a strongly polynomial time algorithm to solve linear feasibility problem.

1 Introduction

Interior point algorithms (e.g. [3]) are mainly the state of the art to solve linear program. Yet, interior point algorithms have two drawbacks. First, these algorithms work directly at raw values level. This way, the moves of the current point can not directly be matched with a set of constraints. In addition, these algorithms can not take advantage of, for example, Chubanov algorithm [2] which provides a strongly polynomial time algorithm for linear feasibility problem.

In this short paper, the focus is given to an algorithm which has not these two drawbacks. The algorithm explicitly works at constraint level and explicitly solves generic linear program by solving a set of linear feasibility problems. This can be done for example with Chubanov algorithm.

This way, this algorithm is interesting to link linear feasibility to generic linear program. And, it may also be an interesting way to look for a strongly polynomial time algorithm to solve generic linear program (especially, seeing that [1] shows that major interior point family is not). Indeed, such property would directly result from a number of Chubanov queries bounded by a polynomial in the size of the problem. Such bound is absolutely not established in this short paper. But, maybe, this kind of algorithm may be updated to reach such property.

2 Some useful lemmas

2.1 Linear feasibility and linear separability

First of all, [2] allows 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 can not be used directly. This is why a minor but required lemma is introduced here.

Chubanov algorithm can also be used to solve linear separability.

Indeed, with a simple trick, Chubanov algorithm can be used to solve (at least, under the assumption that some solution exists):

$$\exists ?x \in \mathbb{Q}^N / \mathcal{A}x > \mathbf{0}$$

with any $\mathcal{A} \in \mathcal{M}_{M,N}(\mathbb{Q})$.

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 identity matrix. Applying the Chubanov algorithm 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 $\mathcal{A}x = Ix_3 = x_3 > \mathbf{0}$.

Let note that the rank is obviously M as there is a identity bloc. This is this routine i.e. solving $\exists ?x \in \mathbb{Q}^N / \mathcal{A}x > \mathbf{0}$ that why be required in the offered algorithm. (It will never be called without be sure that a solution exist.)

2.2 Dealing with all possibles linear programs

The offered algorithm assumes that the input linear program meet some requirements. Yet, these requirements do not restrict generality as from any linear program, one can form a derived linear program meeting the requirement and whose solution contains solution of the original one. This will be presented here.

In this short article, transposition is omitted in scalar product: if p, q are 2 vectors pq corresponds to $p^T q = \sum_n p_n \times q_n$.

2.2.1 The working form

Definition of the working form: in this short paper, the working form of a linear program will be:

- $\min_{x/Ax \geq b} cx$ with $A \in \mathcal{M}_{M,N}(\mathbb{Q})$, $b \in \mathbb{Q}^M$ and $c \in \mathbb{Q}^N$
- both c and row of the matrix A are normalized i.e. $cc = 1$ and for all m, $A_m A_m = 1$
- $\exists \gamma > 0$ such with $Ac > \gamma \mathbf{1}$
  - it implies that from any point x belonging to the admissible space, moving along c increases equally the distance to all constraints
  - it implies that there is a non empty admissible space and a trivial solution (x = λc for $\lambda \gg 1$)
- and with x being an optimal solution iif $Ax \geq b$ and $cx = 0$

2.2.2 overview of the processing to meet requirement

for any linear program, it is possible to form an equivalent linear program in working form.

The generation of a derived linear program from a standard is a combination of classical tricks but this combination is absolutely not trivial due to the specificity of the requirements of the offered algorithm. In one sentence, it is about combining primal and dual twice plus using trick to normalize row and c.

Let review this derived linear program quickly than I will come back in each part more precisely.

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, $y \geq \mathbf{0}$.

This problem can be folded into a A_{big} matrix and a b_{big} as $A_{big}x_{big} \geq b_{big}$.

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

Now, let consider $\min_{A_{big}x_{big} + z\mathbf{1} \geq b_{big}, z \geq 0} z$, with z being just a scalar. This last problem verifies $A_{big}c_{big} > \mathbf{0}$. But, there is a solution (x, y, z) with $z = 0$ iif there exists a solution x, y to the original problem. This is not sufficient for the algorithm which needs to be sure that a solution with $z = 0$ exists.

Yet, there is a simple solution: this process is done ones again (this is not classical) leading to

$$\min_{A_{double}x_{double} + z_{double}\mathbf{1} \geq b_{double}, z_{double} \geq 0} z_{double}$$

Here A_{double} is almost for A_{big} what A_{big} is for A_{raw}.

Eventually, this new linear program is equivalent to the raw input (solving the derived leads to the solution of the original) but this new linear program is in working form. Also, it is sufficient to add little normalization to get the other property.

Now, let review precisely each part of this processing !

2.2.3 Reaching normalized linear program

For any linear program $Ax \geq b$ in $\mathbb{Q}$, one can form an equivalent $\Gamma\chi \geq \beta$ such that for any m $\Gamma_m\Gamma_m = 1$ in $\mathbb{Q}$. Let notice that no square root are needed !

It is sufficient to consider

$$\Gamma = \begin{pmatrix} \frac{1}{\frac{A_1A_1}{2}+1}A_1 & \frac{1}{(\frac{A_1A_1}{2}+1)}\frac{A_1A_1}{2} & \frac{1}{\frac{A_1A_1}{2}+1} \\ \dots & \dots & \dots \\ \frac{1}{\frac{A_MA_M}{2}+1}A_M & \frac{1}{(\frac{A_MA_M}{2}+1)}\frac{A_MA_M}{2} & \frac{1}{\frac{A_MA_M}{2}+1} \\ \mathbf{0} & 1 & 0 \\ \mathbf{0} & -1 & 0 \\ \mathbf{0} & 0 & 1 \\ \mathbf{0} & 0 & -1 \end{pmatrix},$$

and $\beta = \begin{pmatrix} \frac{1}{\frac{A_1A_1}{2}+1}b_1 \\ \dots \\ \frac{1}{\frac{A_MA_M}{2}+1}b_M \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$.

Indeed, the two new variables added in χ are forced to be 0 by the 4 new constraints. So, everything goes like if all added value was null. Yet, now all rows of the matrix are normalized !

Indeed, norm of the row m is $(\frac{1}{\frac{A_mA_m}{2}+1})^2A_mA_m + (\frac{1}{\frac{A_mA_m}{2}+1})^2\frac{(A_mA_m)^2}{4} + (\frac{1}{\frac{A_1A_1}{2}+1})^2$ which is $(\frac{1}{\frac{A_mA_m}{2}+1})^2 \times (A_mA_m + \frac{(A_mA_m)^2}{4} + 1)$ which is $(\frac{1}{\frac{A_mA_m}{2}+1})^2 \times (\frac{A_mA_m}{2} + 1)^2$ which is 1 !

2.2.4 Keeping normalization when adding z variables

For any normalized linear program $Ax \geq b$ in $\mathbb{Q}$, it is sufficient to form $\Gamma = \begin{pmatrix} \frac{4}{5}A_1 & \frac{3}{5} & 0 \\ \dots & \dots & \dots \\ \frac{4}{5}A_M & \frac{3}{5} & 0 \\ \mathbf{0} & \frac{3}{5} & \frac{4}{5} \\ \mathbf{0} & \frac{3}{5} & -\frac{4}{5} \end{pmatrix}$, and $\beta = \begin{pmatrix} \frac{4}{5}b_1 \\ \dots \\ \frac{4}{5}b_M \\ 0 \\ 0 \end{pmatrix}$ to get a new linear program which is

- still normalized

- with ($\mathbf{0}$ 1 0) being normalized and with a common scalar product with each row
- equivalent to the first linear program when the first added variable is 0
- this first added variable being forced to be greater than 0

Thus, if $Ax \geq b$ has a solution then $\min_{\Gamma\chi \geq \beta}$ ($\mathbf{0}$ 1 0)χ will find this solution.

If $Ax \geq b$ has no solution, then optimal value of the added variable will not be 0, but, let notice that this new linear program is at least bounded and resolvable.

2.2.5 Primal dual

The primal dual trick to convert a maximization/minimization into a set of linear inequality is quite classical, and, will not be presented more than in 2.2.2.

Let notice than, since, 2.2.4 allows to form a bounded and resolvable linear program, the idea of applying twice the primal dual is straightforward in some way.

2.2.6 Summary of all these derivations

In 2.2.3, 2.2.4 and 2.2.5, some tricks are presented allowing to normalize linear program, add a row with a common scalar product, and, finally form an equivalent problem with property to have solution. Using these 3 tricks, one is always able to form an equivalent linear program meeting the requirement of 2.2.1 from any raw linear program.

Hence, there is no restriction to focus only on linear program meeting the requirement of 2.2.1. For these ones, an algorithm based on Chubanov one is presented in next section.

3 Algorithm

Consistently with previous section, N is the number of variables and M the number of constraint (i.e. rows of matrix A), and, the problem is $\min_{Ax \geq b} cx$ with assumptions described before. Transposition is still omitted in scalar product: if p, q are 2 vectors pq corresponds to $p^T q = \sum_n p_n \times q_n$.

Constraints, rows of matrix A and planes will be 3 ways to speak about the same objects. If A is matrix (or sub matrix), $Ker(A)$ is the sub space of all vectors h such that $Ah = 0$ i.e. $Ker(A) = \{h / Ah = 0\}$.

3.1 Key points

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

- The algorithm works on the interior of the admissible space like interior point method. But, this algorithm remains close to the geometric structure of the problem. Indeed, all moves of the current point are matched with a sets of constraints.
- At any step of the algorithm only the *closest* constraints of the current point have an influence.
- Precisely, the *closest* constraints of the current point push the point away in order both to maintain the distance with the constraints and to decrease cost function. Eventually, this leads to meet an other constraint that enter the set of *closest* constraints.
- When no such move exists, the routine based on Chubanov algorithm is called leading to a reboot of the set of *closest* constraints. ($Ax \geq 1$ could have a solution while $Ax = 1$ no.)
- The algorithm terminates because the set of closest constraint can not be twice the same

3.2 Pseudo code

The pseudo code the algorithm which should start from a trivial interior point x 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. check trivial termination
   (a) if x is a solution ($Ax \geq b$, $cx = 0$), return x
   (b) if z is a solution, return z
6. let v be the projection of $-c$ on $Ker(A_D)$ (may result in $cv < 0$, $A_D v = \mathbf{0}$)
7. 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
8. call Chubanov routine to find θ such that $\begin{pmatrix} A_D \\ -c \end{pmatrix} \theta > \mathbf{0}$
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

From a point interior point, the coarse algorithm generates a set of interior points plus one optimal point.
well defined :

First, all steps and moves are well defined.

Step 6 is just the projection of a vector on a sub space.

In step 7.a, moving along v decreases the cost cx of x (because $v \neq \mathbf{0}$) which is obviously bounded as $cx \geq 0$. So this move can not be done for ever, and, so, necessarily some plane with meet the ball of center x and radius d - it will be added to D. These moves are generalization of bisector move.

Step 8 should returns something if there is a possible solution. And, there is at least one: let consider $\theta = x^* + \frac{cz}{2}c - z$. Indeed, x^* the solution (there is one see 2.2) verifies $Ax^* \geq b$ so let define $\rho = x^* + \frac{cz}{2}c$. Then, ρ verifies $A\rho > b$ but still $c\rho = \frac{cz}{2} < cz$ (because $cx^* = 0$ and $cz > 0$ otherwise z should have been a solution in 5.b). So, $c(\rho - z) > 0$ and $A_S(\rho - z) = A_S\rho > b_S$ (since $A_S z = 0$ and $A\rho > b$). 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, a solution exists so Chubanov algorithm will return one.

Then, step 9 is obviously possible seeing the definition of S: for all $m \notin S$, $A_m z > b_m$ and for all $m \in S$, $A_m z = b_m$ 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 (assuming that all points are still in the admissible space i.e. $\{x/Ax \geq b\}$.

Interior moves: All moves are designed to never go outside the interior of the admissible space (i.e. $\{x/Ax > b\}$) - except to reach an optimal point. Indeed, step 7.b keeps constant the distance to the constraint satisfaction of constraint. And, step 11 starts from a corner and uses θ to increases satisfaction of the all constraints forming the corner before meeting constraints outside the corner.

Decreasing cost:

Then, trivially cx is decreasing during the algorithm (as 7.b and 10 decrease cx by construction of v and θ).

breaking the loop:

Then, the set D can not have twice the same value in step 9. Indeed, if algorithm reaches two time in step 9 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 $A_D z_2 = A_D z_1 = 0$. 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 bellow 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 10. And, step 7.b strictly increases D which is bounded by $\{1, ..., M\}$. So, the algorithm can not perform more than $M2^M$ loops. Each loop taking less than a call of the Chubanov algorithm.

Is the algorithm strongly polynomial ?

The central question is the number of call of the Chubanov algorithm from step 9.

In numerical experimentation on toy example, it seems that any call of the Chubanov algorithm leads to the definitive rejection of at least one constraint.

Even, if this statement is probably wrong in large dimension and/or complex case, the underlying idea is that during step 11, the set of closest constraints form a linear cone with a single vertex z and with the property that each linear plane forming the cone is tangent to the ball of center x and radius d. Then, the cone is cut by the plane $\{u/cu = cz\}$. And, the question is: could the point x can meet all planes forming the cone without crossing neither other planes nor $\{u/cu = cz\}$.

This question is interesting because complexity of the algorithm is more or less the complexity of Chubanov call (which is strongly polynomial) times the number of call of this routine which could be bounded by M if such statement was true. So, this (probably wrong) statement is related to an hypothetical strong polynomial time property of the algorithm running in $O(M^2)$ Chubanov call plus $O(NM^2)$ elementary operations in $\mathbb{Q}$.

More precisely, the source code of the toy experiment is provided in appendix. The generated linear program may be too simple, and implementation of Chubanov as $x^* + \frac{cz}{2}c - z$ helps the algorithm too much. Yet, more than 40 runs of the algorithm on none trivial size ($N = 15$ and $M = 30$), this rejection effect has been always observed (even if surely false).

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.

```python
from   future   import print function

from fractions import Fraction

def checkSquare(Q):
    if len(Q)==0:
        print("len(Q)==0")
        quit()
    if len(Q[0])==0:
        print("len(Q[0])==0")
        quit()
    for i in range(len(Q)):
        if len(Q[i])!=len(Q[0]):
            print("len(Q["+str(i)+"])!=len(Q[0])")
            quit()

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

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

def projection(u, BOG):
    #checkSquare([u]+BOG)
    pu = []
    for i in range(len(u)):
        pu.append(Fraction())

    for v in BOG:
        pu=combinaisonlineaire(pu,produitscalaire(u,v)/produitscalaire(v,v),v)

    return pu

def gramschimdBOG(H ):
    #checkSquare(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


print("a,b,optimal should be set by hand")
A = []
b = []
c = []
optimal = []

x = []
vectorzero = []
moinsc = []

def checkSize():
    checkSquare(A)
    checkSquare([optimal,c])
    if len(A)!=len(b) or len(c) != len(A[0]):
        print("len(A)!=len(b) or len(c) != len(A[0])")
        quit()

```

```
71          global vectorzero
72          for n in range(len(c)):
73              vectorzero.append(Fraction())
74
75          global moinsc
76          moinsc = combinaisonlineaire(vectorzero,Fraction(-1),c)
77
78          print("size of input data are consistant")
79
80      def computed(y):
81          d = produitscalaire(A[0],y)-b[0]
82          for m in range(len(A)):
83               if produitscalaire(A[m],y)-b[m]<d:
84                   d = produitscalaire(A[m],y)-b[m]
85          return d
86
87      def computeD(y):
88          D = []
89          d = computed(y)
90          for m in range(len(A)):
91               if produitscalaire(A[m],y)-b[m]==d:
92                   D.append(m)
93          return D
94
95      def checkAdmissible(y):
96          return computed(y)>=Fraction()
97      def checkStrinctAdmissible(y):
98          return computed(y)>Fraction()
99      def checkOptimal(y):
100         return checkAdmissible(y) and produitscalaire(c,y)==Fraction()
101
102     def checkProperties():
103         if not checkOptimal(optimal):
104             print("not checkOptimal(optimal)")
105             quit()
106
107         for m in range(len(A)):
108             if produitscalaire(A[m],A[m]) != Fraction(1):
109                 print("produitscalaire(A["+str(m)+"],A["+str(m)+"]) != Fraction(1) :
                    "+str(A[m])+" "+str(produitscalaire(A[m],A[m])))
110                 quit()
111             if produitscalaire(A[m],c) != produitscalaire(A[0],c):
112                 print("produitscalaire(A["+str(m)+"],c) != produitscalaire(A[0],c)")
113                 quit()
114
115         if produitscalaire(A[0],c) <= Fraction():
116             print("produitscalaire(A[0],c) <= Fraction()")
117             quit()
118
119         maxb = max(b)
120         global x
121         x=combinaisonlineaire(vectorzero,maxb/produitscalaire(A[0],c)+Fraction(1),c)
122         checkStrinctAdmissible(x)
123
124         print("input data seems to meet requirement")
125
126     def computez(y):
127         z =combinaisonlineaire(y, -computed(y)/produitscalaire(A[0],c) ,c)
128         if not checkAdmissible(z) or checkStrinctAdmissible(z):
129             print("not checkAdmissible(z) or checkStrinctAdmissible(z) "+str(z))
130             quit()
131         return combinaisonlineaire(y, -computed(y)/produitscalaire(A[0],c) ,c)
132
133     def simulatechubanov(z):
134         z09 = combinaisonlineaire(vectorzero,Fraction(9,10),z)
135         optANDc = combinaisonlineaire(optimal,produitscalaire(z,c)/Fraction(2),c)
136         out = combinaisonlineaire(z09,Fraction(1,10),optANDc)
137
138         if not checkStrinctAdmissible(out) or
            produitscalaire(c,out)>=produitscalaire(c,z):
```

```
139             print("not checkStrinctAdmissible(out) or produitscalaire(c,out)>=produitscalaire(c,z) "+str(z) + " " + str(out))
140             quit()
141         return out
142 
143 
144     def computeBisector(y):
145         D = computeD(y)
146         projectionOnVect = projection(moinsc, gramschimdBOG([A[m].copy() for m in D]))
147         projectionOnKer = combinaisonlineaire(moinsc,Fraction(-1),projectionOnVect)
148 
149         for m in D:
150             if produitscalaire(A[m],projectionOnKer)!=Fraction():
151                 print("produitscalaire(A["+str(m)+"],projectionOnKer)!=Fraction() "+str(projectionOnVect)+" "+str(projectionOnKer))
152                 quit()
153         if produitscalaire(moinsc,projectionOnKer)<Fraction():
154             print("produitscalaire(moinsc,projectionOnKer)<Fraction()")
155             quit()
156 
157         return projectionOnKer
158 
159 
160     def bisectormove(y):
161         d = computed(y)
162         v = computeBisector(y)
163 
164         if produitscalaire(v,v)==Fraction():
165             return False,v,Fraction(),y
166 
167         candidate = []
168         for m in range(len(A)):
169             if produitscalaire(v,A[m])<Fraction():
170                 currentdir = produitscalaire(A[m],x)-b[m]
171                 desireddir = d
172                 move = -(currentdir-desireddir)/produitscalaire(v,A[m])
173                 candidate.append(move)
174 
175         if candidate == []:
176             print("candidate == []")
177             quit()
178 
179         move = min(candidate)
180         if move == Fraction():
181             print("move == Fraction()")
182             quit()
183 
184         nexty = combinaisonlineaire(y,move,v)
185         if not checkStrinctAdmissible(nexty):
186             print("not checkStrinctAdmissible(nexty) "+str(y)+" "+str(v)+" "+str(move)+" "+str(nexty))
187             quit()
188 
189         return True,v,move,nexty
190 
191 
192     def iteration():
193         print("x=" + str(x))
194 
195         if checkOptimal(x):
196             return True,x
197 
198         z = computez(x)
199         if checkOptimal(z):
200             return True,z
201 
202         d = computed(x)
203         print(d)
204 
205         D = computeD(x)
```

```
206         #print(produitscalaire(A[D[0]],x))
207         #print(A[D[0]])
208         #print(b[D[0]])
209         print(D)
210 
211         flag,v,move,y = bisectormove(x)
212         if flag:
213             print("bisector move "+str(v)+" "+str(move))
214             return False, y
215         else:
216             print("chubanov move")
217             return False, simulatechubanov(z)
218 
219 
220     def algorithm():
221         global x
222 
223         nbiter = 0
224         while True:
225             print("########################################## "+str(nbiter))
226             nbiter+=1
227             flag,y = iteration()
228             if flag:
229                 print("YEAH")
230                 return y
231             else:
232                 x = y.copy()
233 
234     import numpy as np
235 
236     def randomproblem(N,M):
237         npoptimal = np.random.randint(-100,101,size=(N))
238         print(npoptimal)
239 
240         Aequal = np.random.randint(-100,101,size=(M,N))
241         bequal = np.dot(Aequal,npoptimal)
242 
243         Agreater = np.random.randint(-100,101,size=(M,N))
244         bgreater = np.dot(Agreater,npoptimal) - np.random.randint(0,101,size=(M))
245 
246 
247         global optimal
248         global c
249         global A
250         global b
251         for n in range(N):
252             optimal.append(Fraction(int(npoptimal[n])))
253         optimal.append(Fraction())
254         optimal.append(Fraction())
255         optimal.append(Fraction())
256 
257         vectorzero = []
258         for n in range(len(optimal)):
259             vectorzero.append(Fraction())
260 
261         c = vectorzero.copy()
262         c[-1] = Fraction(1)
263 
264         for m in range(M):
265             tmp = []
266             for n in range(N):
267                 tmp.append(Fraction(int(Aequal[m][n])))
268 
269             norm = produitscalaire(tmp,tmp)
270             for n in range(N):
271                 tmp[n] *= Fraction(4,5) / (norm/Fraction(2)+Fraction(1))
272 
273             tmp.append(Fraction(4,5) * norm/Fraction(2) / (norm/Fraction(2)+Fraction(1)))
274             tmp.append(Fraction(4,5) / (norm/Fraction(2)+Fraction(1)))
275             tmp.append(Fraction(3,5))
```

```
276
277             A.append(tmp.copy())
278             b.append(Fraction(int(bequal[m])) * Fraction(4,5) /
                (norm/Fraction(2)+Fraction(1)))
279
280             tmp = combinaisonlineaire(vectorzero,Fraction(-1),tmp)
281             tmp[-1]*=Fraction(-1)
282             tmp[-2]*=Fraction(-1)
283             tmp[-3]*=Fraction(-1)
284             A.append(tmp)
285             b.append(Fraction(-int(bequal[m])) * Fraction(4,5) /
                (norm/Fraction(2)+Fraction(1)))
286
287         for m in range(M):
288             tmp = []
289             for n in range(N):
290                 tmp.append(Fraction(int(Agreater[m][n])))
291
292             norm = produitscalaire(tmp,tmp)
293             for n in range(N):
294                 tmp[n] *= Fraction(4,5) / (norm/Fraction(2)+Fraction(1))
295
296             tmp.append(Fraction(4,5) * norm/Fraction(2) / (norm/Fraction(2)+Fraction(1)))
297             tmp.append(Fraction(4,5) / (norm/Fraction(2)+Fraction(1)))
298             tmp.append(Fraction(3,5))
299
300             A.append(tmp.copy())
301             b.append(Fraction(int(bgreater[m])) * Fraction(4,5) /
                (norm/Fraction(2)+Fraction(1)))
302
303
304         A.append(vectorzero.copy())
305         A.append(vectorzero.copy())
306         A.append(vectorzero.copy())
307         A.append(vectorzero)
308
309         b.append(Fraction())
310         b.append(Fraction())
311         b.append(Fraction())
312         b.append(Fraction())
313
314         A[-1][-1] =Fraction(3,5)
315         A[-1][-2] =Fraction(4,5)
316         A[-2][-1] =Fraction(3,5)
317         A[-2][-2] =-Fraction(4,5)
318
319         A[-3][-1] =Fraction(3,5)
320         A[-3][-3] =Fraction(4,5)
321         A[-4][-1] =Fraction(3,5)
322         A[-4][-3] =-Fraction(4,5)
323
324
325     randomproblem(15,30)
326     checkSize()
327
328     print(A)
329     print(b)
330     print(c)
331     print(optimal)
332
333     checkProperties()
334
335     print("GO")
336     x = algorithm()
337     print(x)
338
339
```