# Linking linear feasibility and linear program.

Adrien Chan-Hon-Tong

# Linking linear feasibility and linear program.

Adrien CHAN-HON-TONG

July 5, 2019

**Abstract**

This paper presents an algorithm based on linear feasibility queries ($Ax = \mathbf{0},\ x > \mathbf{0}$) which solves linear programming ($\min_{x\ /\ Ax \geq b} cx$). This paper presents also some ideas that seem interesting to bound the complexity of the algorithm.

## 1 Introduction

Linear programming is the very studied task of solving $\min_{x\ /\ Ax \geq b} c^T x$ with $A$ a matrix, $b$ and $c$ some vectors. This problem can be solved in polynomial times since [7, 4, 6] i.e. in a number of binary operations bounded by a $L^\gamma$ where $L$ is binary size required to write the matrix $A$ and $\gamma$ a constant. Today, state of the art algorithm to solve linear program are interior point algorithms (e.g. [8]). Yet, [1] shows that major interior point algorithms do not solve linear program in strong polynomial time i.e. in a number of rational operations bounded by $\max(M, N)^\gamma$ where $M, N$ are the sizes of $A$ (independently from the binary size of values of $A$). Only, some families of linear program can be solved in strongly polynomial times today:

- combinatorial linear program [10] (by specific algorithm)
- linear program with at most two variables per inequality [5] (by specific algorithm)
- markov chain [9] (by simplex)
- system having binary solution [2] (by specific algorithm)

Independently, [3] recently shows that linear feasibility i.e. $Ax = \mathbf{0},\ x > \mathbf{0}$ can be solved in strongly polynomial time. This is a very interesting result seeing [1]. So, the question of solving linear program as a strong polynomial sequence of linear feasibility queries is relevant.

This paper focus on this last point, and, presents algorithms which solve linear programming by solving a sequence of linear feasibility problems.

## Notations

$\mathbb{N}, \mathbb{Q}$ are the sets of integer and rational numbers. $\setminus$ is the ensemble subtraction. For all integers $i, j$, $[i, j]$ will symbolize the **integer** range i.e. $\{i, ..., j\}$ which is empty if $i > j$ (there will be no ambiguity with the interval in $\mathbb{R}$ as there is no real range in this paper).

For all integers $i, j, I, J$, $\mathbb{Q}^I$ is the set of $I$ dimensional vectors on $\mathbb{Q}$, and, $\mathbb{Q}^{I \times J}$ is the set of matrix with $I$ rows and $J$ columns, with values in $\mathbb{Q}$, and, $\cdot_i$ designs the $i$ component: a row for a matrix and a rational for vector or a row. $\mathbb{Q}^I$ would be matched with $\mathbb{Q}^{I \times 1}$ i.e. vectors are seen as columns, and, row of a matrix are matched with $\mathbb{Q}^{1,J}$. For all sets $S \subset \mathbb{N}$, $A_S, b_S$ is the submatrix or subvector obtained when keeping only components indexed by $s \in S$. $^T$ is the transposition operation i.e. $A^T_{j,i} = A_{i,j}$. $\mathbf{0}$ and $\mathbf{1}$ are the 0 and 1 vector i.e. vector contains only 0 or only 1, and $\mathbf{I}$ is the identity matrix.

$\mathbb{U}_I$ is the set of normalized vectors from $\mathbb{Q}^I$ i.e. $\mathbb{U}_I = \{v \in \mathbb{Q}^I, v^T v = 1\}$. $\mathbb{U}_{I,J}$ is the set of matrix from $\mathbb{Q}^{I \times J}$ whose rows are in $\mathbb{U}_J$ (rows not necessarily columns).

If $A \in \mathbb{Q}^{I \times J}$, the null vector space of $A$ (i.e. the kernel) is written $Ker(A) = \{v \in \mathbb{Q}^J / Av = \mathbf{0}\}$, with the convention that $Ker$ of empty $A$ is all space.

All notations are quite classical except $\mathbb{U}$ to indicate normalization of vectors and/or matrix rows.

# 2 First version of slide and jump algorithm

## 2.1 Key idea

The starting point of this paper is [3] which offers an algorithm to solve linear feasibility ($\exists ? x \in \mathbb{Q}^N \ / \ Ax = \mathbf{0},\ x > \mathbf{0}$ when $A \in \mathbb{Q}^{M \times N}$ has full rank) in strongly polynomial time.

First, with this algorithm, one can solve with the same complexity linear separability problem ($\exists ? y \in \mathbb{Q}^N \mathcal{A}y > \mathbf{0}$ for any $\mathcal{A} \in \mathbb{Q}^{M \times N}$) in strongly polynomial time (see proof just above).

Then, it is easy to get a small improvement from a not optimal admissible solution for linear program by solving a derived linear separability problem. Indeed, if $x$ verifying $Ax \geq b$ has not minimal $cx$ under assumption that $Ac = \frac{3}{5}\mathbf{1}$, then it is possible (see proof in subsection 2.3) to find $u$ such that $\begin{pmatrix} A_D \\ -c^T \end{pmatrix} u > \mathbf{0}$ with $D$ the current saturated constraints i.e. $D$ such that $A_D w = b_D$.

Now, it could take infinite time to reach the optimal solution by computing such $u$ and updating $x = x + \varepsilon u$. These moves can be see as *jump* because its always lead to some improvement independently from current saturated constraint. But jumping is not enough. So, the algorithm also considers *sliding* moves. It considers moves which aim to keep the structure between $x$ and the constraints. Such moves can be computed by solving the easy optimization

problem derived by freezing this structure, for example, solving $\min_{w \ / \ A_D v = \mathbf{0}} c^T v$ keeps $D$ unchanged. This last problem can be trivially solved because it is just a projection on a vectorial space (can be done with gram schmidt basis transform).

Before presenting the algorithms, let first consider the lemma linking linear feasibility and linear separability:

**Lemma 2.1.** *If it is possible to solve* $\exists ? x \in \mathbb{Q}^I \ / \ Ax = \mathbf{0},\ x > \mathbf{0}$ *when* $A \in \mathbb{Q}^{J,I}$ *has rank* $J$ *with less than* $O(\max(I,J)^\gamma)$ *operations on* $\mathbb{Q}$*, then it is possible to solve* $\exists ? y \in \mathbb{N} \ / \ \mathcal{A}y > \mathbf{0}$ *for any* $\mathcal{A} \in \mathbb{Q}^{M \times N}$ *with less than* $O(\max(N,M)^\gamma)$ *operations on* $\mathbb{Q}$*.*

*Proof.* Let $\mathcal{A} \in \mathbb{Q}^{M \times N}$ a matrix without any assumption. Let consider the matrix $A = \begin{pmatrix} \mathcal{A} & -\mathcal{A} & -\mathbf{I} \end{pmatrix} \in \mathbb{Q}^{M \times 2N+M}$ i.e. formed with $\mathcal{A}$ concat with $-\mathcal{A}$ concat with $-\mathbf{I}$ the opposite of identity matrix.

First, this matrix has rank $M$ due to the identity block. So by assumption, one can apply the solver to know $\exists ? x \in \mathbb{Q}^{2N+M} \ / \ Ax = \mathbf{0},\ x > \mathbf{0}$ which will produce an output in less than $O((2N+M)^\gamma) \leq O(3^\gamma \max(N,M)^\gamma) = O(\max(N,M)^\gamma)$ on $\mathbb{Q}$. This output is either a solution or a certificate that no solution exists.

If solver finds a solution $x \in \mathbb{Q}^{2N+M}$. Let split the solution: $x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$ with $(x_1, x_2, x_3) \in \mathbb{Q}^N \times \mathbb{Q}^N \times \mathbb{Q}^N$. By assumption, $A \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \mathbf{0}$ and $x_1, x_2, x_3 > \mathbf{0}$. Yet, as $A = \begin{pmatrix} \mathcal{A} & -\mathcal{A} & -\mathbf{I} \end{pmatrix}$, it means that $\mathcal{A}x_1 - \mathcal{A}x_2 - \mathbf{I}x_3 = \mathbf{0}$. As $\mathbf{I}x_3 = x_3$ and $x_3 > \mathbf{0}$, it leads that $\mathcal{A}(x_1 - x_2) = x_3 > \mathbf{0}$. So $x_1 - x_2$ is a solution of the desired problem found in less than $O(\max(N,M)^\gamma)$ operation on $\mathbb{Q}$.

Then, let assume that there is $y$ such that $\mathcal{A}y > \mathbf{0}$, and, let consider $x_1 = \max(y, \mathbf{0}) + 1$, $x_2 = \max(-y, \mathbf{0}) + 1$, and, $x_3 = \mathcal{A}y$. Then, $x_1, x_2, x_3 > \mathbf{0}$, and $x_1 - x_2 = y$. So, $A \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \mathcal{A}y - \mathcal{A}y = 0$. So, there is a solution of the derived problem, solver will find one. And, so, if solver does not find solution, then, there is not. □

Also, a quite trivial but required lemma:

**Lemma 2.2.** *Solving* $\exists ? x \mathbb{Q}^N \ / \ Ax > \mathbf{0}$ *or* $\exists ? x \mathbb{Q}^N \ / \ Ax \geq \mathbf{1}$ *is equivalent.*

*Proof.* Any solution of the second problem are solution of the first, and any solution $w$ of the first can be normalised in $(\frac{1}{\min_m A_m w})w$ to form a solution of the second. □

## 2.2 First version of the algorithm

In the first version of the algorithm, only saturated constraints $D$ are considered during sliding move. The point is that *sliding* moves allow the algorithm to explore completely a particular combinatorial situation ($D$) while *jumping* moves allow to exit such structure. The termination will be guarantee by the impossibly to see $D$ twice, and, the correctness, by capacity to jump to next $D$ until optimal solution is reached.

**Algorithm 1** Slide and jump for linear program

**Require:** $c \in \mathbb{Q}^N$, $b \in \mathbb{Q}^M$, $A \in \mathbb{Q}^{M \times N}$ with $Ac > \mathbf{0}$, $Ax \geq b \Rightarrow c^T x \geq 0$, and $Ax_{start} \geq b$ (e.g. $x_{start} = \lambda c$) *this does not restrict generality*
**Ensure:** return x: $Ax \geq b$, $cx$ is minimal

1: **while** True **do**
2: $D = \{m \in [1, M] \ / \ A_m x = b_m\}$
3: compute $u$ the orthogonal projection of $-c$ on $Ker(A_D)$
4: **if** $u = \mathbf{0}$ **then**
5: call subsolver $\exists ? v \ / \ \begin{pmatrix} A_D \\ -c^T \end{pmatrix} v \geq \mathbf{1}$
6: **if** $v$ not exists **then return** $x$
7: chose $\delta$ with $0 < \delta < h = \underset{m \in [1,M] / A_m v < 0}{min} \frac{A_m x - b_m}{-A_m v}$ (e.g. $\delta = \frac{h}{4}$)
8: $x = x + \delta v$
9: **else**
10: $g = \underset{m \in [1,M] \ / \ A_m u < 0}{min} \frac{A_m x - b_m}{-A_m u}$
11: $x = x + gu$

Pseudo code for linear program is presented in algorithm 1: algorithm assumes the input linear program is $c \in \mathbb{Q}^N$, $b \in \mathbb{Q}^M$, $A \in \mathbb{Q}^{M \times N}$ with $Ac > \mathbf{0}$, $Ax \geq b \Rightarrow c^T x \geq 0$, and $Ax_{start} \geq b$ (e.g. $x_{start} = \lambda c$).

**These assumptions do not restrict generality because all linear programs can be pushed in this shape using classical primal dual trick (see appendix).**

## 2.3 Correctness and termination

This subsection presents a proof that this slide and jump algorithm is well defined, terminates, and produces an exact optimal solution for both linear program.

**Lemma 2.3.** *Algorithm is well defined*

*Proof.* Problematic steps are step 3 (non standard operation) and step 10 (set should not be empty). All other steps are standard operations.

**step 10:**

By assumption $cx$ is bounded by 0 i.e. $\forall x \in \mathbb{Q}^N$, $Ax \geq b \Rightarrow cx \geq 0$. If there was $\omega$ such that $c\omega < 0$ and $A\omega \geq \mathbf{0}$, then, one could produce an unbounded

admissible point $x + \lambda\omega$ as $A(x + \lambda\omega) \geq Ax \geq b$ and $c(x + \lambda\omega) \underset{\lambda\to\infty}{\to} -\infty$. So, if $c\omega < 0$ then $\exists m \in [1, M]$ / $A_m\omega < 0$.

**step 3:**

Step 3 is the projection on a vectorial space i.e. it consists to solve $\arg\min_{A_D u=0}(c+u)^T(c+u)$. This step can be done by gram schmidt orthogonalisation, and, is more detailed in appendix. This procedure can not fail (returns $-c$ on empty input), and, always returns a vector with a strictly positive scalar product with $-c$ or $\mathbf{0}$.

□

**Lemma 2.4.** *Algorithm keeps the current point in the admissible space*

*Proof.* Steps 8 and 11 move the current point.

**Step 8:**

Seeing the test in step 6, $A_D v > \mathbf{0}$ so for all $m \in D$, $A_m(x + \lambda v) > b_m$. Now, for all $m \notin D$, $A_m x > b_m$, so $\exists\delta > 0$ such that $A(x + \delta v) > b$ (the offered $\delta$ works but anyway it could be less).

**Step 11:**

First, $\forall m$ / $A_m u \geq 0$, $A_m(x + \lambda u) \geq b$.

Then, seeing the definition of $u$ from step 3, if $m \in D$, $A_m u = 0$ and if $m \notin D$, $A_m x > b_m$. So $\forall m \in [1, M]$, $A_m u < 0 \Rightarrow m \notin D \Rightarrow A_m x > b_m$, and so $g > 0$. Now, for $m$ / $A_m u < 0$, $g$ is a minimum, so $\frac{A_m x - b_m}{-A_m u} \geq g$. When multiplying by a negative: $\frac{A_m x - b_m}{-A_m u} A_m u \leq g A_m u$. And, so $\forall m \in [1, M]$, / $A_m u < 0$: $A_m x + g A_m u - b_m \geq A_m x + \frac{A_m x - b_m}{-A_m u} A_m u - b_m = 0$. □

**Lemma 2.5.** *algorithm outputs optimal exact solution*

*Proof.* The proof consists to assume the algorithm returns non optimal admissible point $x$, while optimal solution was $x^*$ ($cx > cx^*$). Then, it is possible to built $\hat{x}$ (by adding $\varepsilon c$) which are in the interior of the admissible space and still better than $x$. But, it means that $\hat{x} - x$ is both better for the objective and for saturated constraints. So, the subsolver should have returned something, and so, there is a contradiction.

Precisely, let consider $\phi = x^* - x + \frac{c^T(x-x^*)}{2\sqrt{c^T c}} c$, then, $A_D\phi = A_D(x^* - x + \frac{c^T(x-x^*)}{2} c) = A_D(x^* + \frac{c^T(x-x^*)}{2} c) \geq \frac{c^T(x-x^*)}{2} A_D c > \mathbf{0}$ and $c(\phi - x) = -\frac{c^T(x-x^*)}{2} < 0$. □

**Lemma 2.6.** *algorithm can not loop more then $M+1$ times without calling the subsolver*

*Proof.* $g$ is exactly build such that the $k$ corresponding to the minimum enters in $D$. Let consider $k \notin D$ such that $\frac{A_k x - b_k}{-A_k u} = g$. So, $A_k(x + gv) - b_k = A_k x + \frac{A_k x - b_k}{-A_k v} A_k v - b_k = 0$.

So, $D$ strictly increases each time the algorithm reaches step 11, but, $D$ is bounded by $[1, M]$, so, test 4 can not return true more than $M$ consecutive times. □

**Lemma 2.7.** *All moves strictly decrease the objective function.*

*Proof.* Steps 8 and 11 move the current point.

**step 11:** By construction $c^T u < 0$, and, $g > 0$. So, cost decreases when moving along $u$.

**step 8:** $c^T v < 0$ so moving a little along $v$ decreases the cost. □

**Lemma 2.8.** *A value for set $D$ observed in step 5 can not be observed again in step 2.*

*Proof.* This lemma is proven by contradiction: if $D$ is seen again, one can prove that exploration of $D$ should have been continued (i.e. the test step 3 should have been false) instead of calling the subsolver on step 5.

Let consider $x_2 - x_1$ with $x_1$ corresponding to the observation of $D$ in step 5 and $x_2$ to any ulterior observation. As all moves strictly decreases $c^T x$, it means $c^T(x_2 - x_1) < 0$, but, by definition of $D$, $A_D x_2 = A_D x_1 = b_D$ so $A_D(x_2 - x_1) = \mathbf{0}$. So, projection of $-c$ on $Ker(A_D)$ is not null (at least it could be $x_2 - x_1$), so, algorithm should not have passed the test step 3.

*More precisely, an underlying lemma is that the projection of a vector on a kernel **should** have a strictly positive scalar product with this vector if possible. This is a linear algebra result recalled in appendix.* □

From all previous lemmas, observing that $D$ is bounded (so looping without call to subsolver is impossible), and that, number of subsets from $[1, M]$ is finite (so number of call to subsolver is bounded because subsolver is never called twice on the same value), algorithm terminates. And, independently, algorithm is well defined, works in admissible space and can not output something else than an optimal solution. So both correction and termination are proven.

**Theorem 2.9.** *The slide and jump algorithm solves linear program.*

# 3 Extended version of slide and jump algorithm

## 3.1 complexity of first slide and jump algorithm

First version of the slide and jump algorithm solves linear program because $D$ the set of saturated constraint can not be seen after a call of chubanov algorithm on it. Yet, there is $2^M - 1$ possible sets.

Of course, not all the sets are possible (some sets are not feasible). And, of course, not all sets will be explored.

For example, in numerical experiment, slide and jump algorithm solves the Klee Minty cube (see appendix) by exploring a very low number of vertex. For example, it explores only 4 vertices for the cube with $N = M = 20$, whereas, simplex would explore 1048576 vertices on this particular instance. Obviously, it is unfair to compare number of explorated vertices between two algorithms on the worse case of the second (even if it has not being designed for this particular

instance). Yet, it still shows that the slide and jump algorithm can explore a low number of vertices even if there is a lot.

But, this algorithm may still not be polynomial in worse case. This is why some extensions presented now try to decrease it complexity.

## 3.2 Additional requirements

From now, all algorithms will assume that the linear program taken as input $\min_{x\in\mathbb{Q}^N,\ /\ Ax\geq b} c^T x$ has the following property: $A \in \mathbb{U}_{M,N}$, $b \in \mathbb{Q}^M$, $c \in \mathbb{U}_N$, $Ac = \frac{3}{5}\mathbf{1}$, $x_{start}$ being a trivial admissible point e.g. $(\frac{5}{3}\max_{m\in[1,M]} b_m)c$, and, $c^T x$ being bounded by 0.

**Again, these assumptions do not restrict generality because all linear programs can be pushed in this shape using classical primal dual trick (see appendix).**

The interest of having normalized row is to remove ambiguity due to the scaling, and, so to have a proper geometric view of the constraints. In particular, from any admissible point $x$, one view $x$ as a ball with radius $d$ the distance to the closest constraints. Then, there is an equivalence between the closest constraints and the constraints that will become saturated is one makes $x$ moving along $-c$ (precisely $x - \frac{5d}{3}c$). But, most importantly, from any admissible point $x$ saturating $D$, and any $m \in D$, one can derive a point $y$ such that $A_m y = b_m$ and $A_{[1,M]\setminus\{m\}} y > b_{[1,M]\setminus\{m\}}$. Indeed, it is sufficient to consider: $x + \frac{5}{3}c - A_d^T$ (see a more formal proof bellow).

## 3.3 Checking constraint rejection

A good case for algorithm convergence is if at some time, the current point goes bellow the minimal point attached with one constraint. Indeed, in this case, this constraint will never be seen again.

As the number of slide move is bounded by $M$ between two linear separability queries, the focus to improve complexity of slide and jump is to bound the number of calls of linear separability queries. These calls are matched with $D$. Each time a constraint is rejected all possible combinations for a set $D$ which included this constraints are also rejected. Typically, if one constraint is rejected at each query, the number of query is bounded by $M$. Obviously this is a very strong assumption, but, in this case, the algorithm is strongly polynomial !

So computing the *still possible combination of* $D$ can be an interesting feature for such solver. Basically, let $x$ be point such that $Ax \geq b$, and $m \in [1, M]$, if there is no $y$ such that $A_m y = b_m$, $A_{[1,M]\setminus\{m\}} y \geq b_{[1,M]\setminus\{m\}}$ and $cy < cx$, then constraint $m$ will never be included in the set of saturated constraint $D$ after this point $x$ in the algorithm. However, checking if such $y$ exists is as hard as solving the original problem.

Yet, one can perform weaker check.

**Lemma 3.1.** *If $m \in D$ such that $u = 0$ in algorithm 1 and there is no $\omega$ such that $A_m w = 0, A_{D\setminus\{m\}}\omega \geq 1$, then $m$ will never be seen again in $D$.*

*Proof.* First, it is easy to check if exists $\omega$ such that $A_m w = 0, A_{D\setminus\{m\}}\omega \geq 1$. Indeed, this can be done by combining linear feasibility and linear separability problem. Let consider the matrix $\begin{pmatrix} A_m & -A_m & 0 \\ A_{D\setminus\{m\}} & -A_{D\setminus\{m\}} & -\mathbf{I} \end{pmatrix}$, and, let call Chubanov algorithm on it (rank is full as soon as $A_m$ is not null due to the first row + the identity bloc). It will lead to $\omega$ verifying the desired property if one exists.

Now, if there is $m$ such that no such $\omega$ exists, then, it means that $m$ will never be included in the set of saturated constraint $D$ after this point in the algorithm. If it was, let $y$ the next point with $m \in D$, let consider $\omega = y - x + \frac{5}{3}c - A_m^T$. By definition of $D$, $A_m y = A_m x = b_m$ so $A_m\omega = 0$ (because $A_m c = \frac{3}{5}$ and $A_m A_m^T = 1$), but, for all $i \in D\setminus\{m\}$ both $A_i(y - x) \geq 0$ and $A_i(\frac{5}{3}c - A_m^T) > 0$. This would be a contradiction. □

Of course, there can exist one $\omega$ for all $m$, and, independently, there can exist a lower point for all constraints in $D$ (it may exist $\omega$ but not point bellow, and, it could exist $\omega$ because there is point bellow).

So, this check is not very useful, but, at least, it highlights some property of the algorithm, and, it is strongly polynomial.

## 3.4 Example of constraint rejection

Also, this situation may not be that unrealistic. Let consider $x$ associated with $D$ and $u \neq \mathbf{0}$, let consider $m$ the constraint being hit by $x$ when moving along $u$, and, let assume that $Ker(A_{D\cup\{m\}}) = \mathbf{0}$.

First, one can remark that moving straightforwardly on $m$ is not possible: projection of $-c$ on $m$ is $\omega = -c + \frac{3}{5}A_m^T$ but $A_i\omega < 0$ for $i \neq m$. This highlights the fact that du to the assumption (normalized $c$, row, and $Ac = \frac{3}{5}\mathbf{1}$) constraints tend to forbid to move along each other.

For example, if $N = 3$ with these assumptions (meaning that the underlying problem is a simple 2D problem), there is always a constraint that it rejected (this will not be more formally proven here). Let consider 3 planes $(\frac{2\sqrt{2}}{5}, \frac{2\sqrt{2}}{5}, \frac{3}{5}), (\frac{2\sqrt{2}}{5}, -\frac{2\sqrt{2}}{5}, \frac{3}{5}), (\frac{4}{3}, 0, \frac{3}{5})$, there is not point attached to the last one bellow $\mathbf{0}$.

Of course, this is not true in higher dimension, and, if multiple constraints are added in the same time:
$(\frac{2\sqrt{2}}{5}, \frac{2\sqrt{2}}{5}, 0, \frac{3}{5}), (\frac{2\sqrt{2}}{5}, -\frac{2\sqrt{2}}{5}, 0, \frac{3}{5}), (\frac{12}{25}, 0, \frac{16}{25}, \frac{3}{5}), (0, 0, \frac{4}{5}, \frac{3}{5}), (0, 0, -\frac{4}{5}, \frac{3}{5})$
moving along $x$ is still possible as the 3rd dimension allows the 3rd constraint to have lower ratio $x$,$c$, this way, having all constraints saturated is not possible, but, having the 1st and the 3rd or the 2nd and the 3rd is possible.

### 3.5 Sliding with maximal set

This last observation invites to consider an extension of the slide and jump algorithm where instead of just computing the projection of $-c$ on $Ker(A_D)$, one compute instead the projection of $-c$ on a maximal sets of constraints sorted by the distance to $x$, and, in particular in order to try to keep the symmetry between constraints.

To unfold this idea in geometry, with normalization, slide and jump consists to make a ball following the gravity path to the lower point, and, this extension is about to do the same with a kind of ellipse.

The idea is first to sort all constraints according to their distance to $x$. This result in a partition $D_0, D_1, ..., D_K$ of $[1, M]$ with: $\forall i, j \in D_k, A_i x - b_i = A_j x - b_j$, and, $\forall I < J, i, j \in D_I, D_J, A_i x - b_i < A_j x - b_j$ with $D$ being $D_0 \cup D_1$ ($D_0$ being the last added constraints in $D$).

Now, instead of considering $u$ to be the projection of $-c$ on $Ker(A_D)$, the idea is to consider the projection of $-c$ on matrix $\phi_k$ and $\psi_k$ with:

$$\phi_k = \begin{pmatrix} \psi_{k-1} \\ A^T_{i_{k,2}} - A^T_{i_{k,1}} \\ ... \\ A^T_{i_{k,r}} - A^T_{i_{k,1}} \end{pmatrix} \qquad \psi_k = \begin{pmatrix} \phi_k \\ A^T_{i_{k,1}} \end{pmatrix}$$

So first priority is to keep distance within $D_0$ equal ($\phi_0$), and, then unchanged ($\psi_1$). Then, priority is to keep distance within $D_1$ equal ($\phi_1$) and then unchanged ($\psi_1$). Then, the priority is to keep distance within $D_2$ equal... When it is not possible to handle more constraint, $u$ is returned and used to do a sliding move, or, if $\neg A_D u \geq \mathbf{0}$ a jumping move is done instead.

Currently, the move along $u$ can be done either until some plan enter in $D$ or until the partition change. Code offered in appendix use the first, but, second seems best.

Currently, computing $u$ each time partition change seems a more straightforward implementation of the idea. Yet, just changing the computation of $u$ but keeping all other instruction unchanged allows to state the this updated version also solves linear program as the complete proof also stand here.

Experimental behaviour is bad as computing $u$ becomes much less efficient. Yet, this does not really matter: computing $u$ is still polynomial, so, only matter the bound on the number of call. Unfortunately, even with this update, the algorithm is not proven polynomial. The main difference with the original version is that if $i, j, k$ were in $D$, then only $k$ then $i$ but not $j$, and, then $j$ but not $i$, it raises the question about why algorithm does not keep $i, j$ as a block.

## 4 Perspective

This short paper offers an algorithm which solves linear program by using Chubanov linear feasibility algorithm. The offered algorithm is unfortunately

not proven polynomial. But, some ideas are presented as possible way to make this algorithm a strongly polynomial solver for linear programming.

# 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 strongly polynomial algorithm for linear systems having a binary solution. *Mathematical programming*, 134(2):533–570, 2012.

[3] Sergei Chubanov. A polynomial projection algorithm for linear feasibility problems. *Mathematical Programming*, 153(2):687–713, 2015.

[4] Martin Grötschel, László Lovász, and Alexander Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1(2):169–197, 1981.

[5] Dorit S Hochbaum and Joseph Naor. Simple and fast algorithms for linear and integer programs with two variables per inequality. *SIAM Journal on Computing*, 23(6):1179–1192, 1994.

[6] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311. ACM, 1984.

[7] Leonid Khachiyan. A polynomial algorithm for linear programming. *Doklady Akademii Nauk SSSR*, 1979.

[8] Yurii Nesterov and Arkadii Nemirovskii. *Interior-point polynomial algorithms in convex programming*, volume 13. Siam, 1994.

[9] Ian Post and Yinyu Ye. The simplex method is strongly polynomial for deterministic markov decision processes. *Mathematics of Operations Research*, 40(4):859–868, 2015.

[10] Eva Tardos. A strongly polynomial algorithm to solve combinatorial linear programs. *Operations Research*, 34(2):250–256, 1986.


# Appendix

## Normalizing linear program

If the linear program given as input is $\min_{Ax \geq b} cx$ and verifies $A \in \mathbb{U}_{M,N}$, $b \in \mathbb{Q}^M$, $c \in \mathbb{U}_N$ and $Ac = \gamma \mathbf{1}$ with $\gamma > 0$, and, $cx$ being bounded by 0, then the offered algorithm of section 2 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 is 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

**Primal dual**

The conversion of an linear program to be optimized into a linear inequality system is quite classical. A brief recall is provided bellow.

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

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

**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 $A \in \mathbb{U}_{M,N}$, $b \in \mathbb{Q}^M$, $c \in \mathbb{U}_N$, and, $Ac = \frac{3}{5}\mathbf{1}$, and, $cx$ being bounded by 0.

It is sufficient to consider:

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

and

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

First, the produced linear program is in the desired form: $\min_{Ax \geq b} cx$ with $A \in \mathbb{U}_{J,I}$, $b \in \mathbb{Q}^M$, $c \in \mathbb{U}_N$, 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, $x_1, ..., x_N = \chi$ 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 $x_1, ..., x_N = \chi$ is a solution of the original set of inequality.

And, inversely, if there is a solution $\chi$, 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.

**For any linear program, it is possible to create a derived form meeting the requirement of the offered algorithm.**
**All this normalization is entirely done in $\mathbb{Q}$ i.e. no square root are needed.**

## Projection on vectorial space

Let $c \in \mathbb{Q}^N$ and $A \in \mathbb{Q}^{M \times N}$, let consider the problem: $\min_{p \in \mathbb{Q}^N, Ap = \mathbf{0}} (c-p)^T(c-p)$.

Let $\nu_1, ...\nu_K$ be a basis of $\{p \in \mathbb{R}^N, Ap = \mathbf{0}\}$, completed by $\nu_{K+1}, ..., \nu_N$ into a basis $\mathbb{R}^N$. By applying Gram Shimd, one forms $e_1, ..., e_K, ..., e_N$ a orthonormal basis of $\mathbb{R}^N$ such that $e_1, ..., e_K$ is a basis of $\{p \in \mathbb{R}^N, Ap = \mathbf{0}\}$.

Now, as $e_1, ..., e_N$ is an orthonormal basis of $\mathbb{R}^N$, $c = \sum_{k\in[1,N]} e_k^T c e_k$. And, as $e_1, ..., e_N$ is an orthonormal basis of $\{p \in \mathbb{R}^N, Ap = \mathbf{0}\}$, it means that $\forall p, Ap = \mathbf{0} \Rightarrow p = \sum_{k\in[1,K]} e_k^T p e_k$. In particular, $\forall p, Ap = \mathbf{0} \Rightarrow (c-p)^T(c-p) \geq \sum_{k\in[K+1,N]} (e_k^T c)^2$. Independently, $q = \sum_{k\in[1,K]} e_k^T c e_k$ verifies both that $Aq = \mathbf{0}$ and that $(c-q)^T(c-q) = \sum_{k\in[1,N]\setminus[1,K]} (e_k^T c)^2$.

So $q$ is the solution of $\min_{p\in\mathbb{Q}^N, Ap=\mathbf{0}} (c-p)^T(c-p)$. Thus, solving $\min_{p\in\mathbb{Q}^N, Ap=\mathbf{0}} (c-p)^T(c-p)$ is easy, and, can be used in algorithm 1.

## Some source code

```
from __future__ import print_function

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

from fractions import Fraction

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

def combinaisonlineaire(u,l,v):
    return [u[n]+l*v[n] for n in range(len(u))]

def produitscalairevecteur(l,v):
    return [l*e for e in v]

def opposedvector(v):
    return [-e for e in v]

def produitscalaire(u,v):
    return sum([u[n]*v[n] for n in range(len(u))])

def saturatedconstraints(A,b,x):
    return [m for m in range(len(A)) if produitscalaire(A[m],x)==b[m]]

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

def resteprojection(u,BOG):
    return combinaisonlineaire(u,Fraction(-1),projection(u,BOG))

def gramschimdBOG(H):
    BOG = []
    while True :
        H = [resteprojection(h,BOG) for h in H]
        H = [h for h in H if produitscalaire(h,h) != Fraction()]
        if H!=[]:
            BOG.append(H.pop())
        else:
            return BOG

################ LINEAR ALGEBRA MODULES ################

#returns v such that Av=0 and (v-c)(v-c) is minimal
def projection_on_ker(A,c):
    BOG = gramschimdBOG(A)
    if len(BOG)==0:
        return c
    else:
        return resteprojection(c,BOG)

#returns x such that Ax=b
#OR RETURNS NONE IF IMPOSSIBLE
def solve_linear_equality(A,b):
    A = [a.copy() for a in A]
    for m in range(len(A)):
        A[m].append(-b[m])

    c = [Fraction(0)]*len(A[0])
```

```
    c[-1] = Fraction(1)

    v = projection_on_ker(A,c)
    if v[-1]==Fraction():
        return None
    else:
        return produitscalairevecteur(Fraction(1)/v[-1],v[0:-1])

#returns l>0 such that forall e>0, not A (x+(l+e)v) >= b
#MAY FAIL ON UNBOUNDED OR UNEXISTING SOLUTION
def maximalmoves(A,b,x,v):
    S = [m for m in range(len(A)) if produitscalaire(v,A[m])<Fraction()]
    if S==[]:
        print("maximalmoves: S==[]")
        quit()

    alll = [-(produitscalaire(A[m],x)-b[m])/produitscalaire(v,A[m]) for m in S]
    l = min(alll)
    if l==Fraction():
        print("maximalmoves: l==0")
        quit()
    return l


#takes x as input with Ax>=b
#returns y such that cy<cx, Ay>=b and exists D such that A_Dy=b_D
#                and projection_on_ker(A_D,opposedvector(c))=0
#ASSUME CX IS BOUNDED
def slide1(A,b,c,x):
    while True:
        print("$", end="", flush=True)
        D = saturatedconstraints(A,b,x)

        v = projection_on_ker([A[m] for m in D],opposedvector(c))
        if produitscalaire(v,v)==Fraction():
            print(" ")
            return x
        l = maximalmoves(A,b,x,v)
        x = combinaisonlineaire(x,l,v)


def gramschimdBOG_hotstart(H,previousBOG):
    BOG = previousBOG
    while True :
        H = [resteprojection(h,BOG) for h in H]
        H = [h for h in H if produitscalaire(h,h) != Fraction()]
        if H!=[]:
            BOG.append(H.pop())
        else:
            return BOG
def projection_on_ker_hotstart(A,c,previousBOG):
    BOG = gramschimdBOG_hotstart(A,previousBOG)
    if BOG==[]:
        return c,BOG
    else:
        return resteprojection(c,BOG),BOG

def slide2(A,b,c,x):
    lastAdded = []
    while True:
        print("$", end="", flush=True)

        #compute ranking of all constraints
        alldist = set([produitscalaire(A[m],x)-b[m] for m in range(len(A))])
        D = saturatedconstraints(A,b,x)
        R = [lastAdded,[m for m in D if m not in lastAdded]]
        for d in alldist:
            if d!=Fraction():#d==0 already processed with lastAdded and D
                tmp = [m for m in range(len(A)) if produitscalaire(A[m],x)-b[m]==d]
                R.append(tmp)


        #check a vector which satisfies a maximum number of constraint
        v = [Fraction(0)]*len(x)
        block = []
        previousBOG = []
        for l in R:
            print("*", end="", flush=True)
            block += [combinaisonlineaire(A[m],Fraction(-1),A[l[0]]) for m in l]
            w,tmp = projection_on_ker_hotstart(block,opposedvector(c),previousBOG)
            previousBOG = tmp.copy()
            if produitscalaire(w,w)!=Fraction():
                v = w.copy()
            else:
                break

            block = block+[A[m] for m in l]
            w,tmp = projection_on_ker_hotstart(block,opposedvector(c),previousBOG)
            previousBOG = tmp.copy()
            if produitscalaire(w,w)!=Fraction():
                v = w.copy()
```

```
            else:
                break


        if produitscalaire(v,v)==Fraction():
            print(" ")
            return x
        if not (all([produitscalaire(A[m],v)>=Fraction() for m in D])):
            print("~")
            return x
        l = maximalmoves(A,b,x,v)
        x = combinaisonlineaire(x,l,v)
        Dafter = saturatedconstraints(A,b,x)
        lastAdded = [d for d in Dafter if d not in D]


############### ALGORITHM FOR LP ###############

KNOWN_OPTIMAL_SOLUTION = None

#return v such that A_Dv>0, cv<0
#with D index such that A_Dx=b_D
#OR RETURNS NONE IF IMPOSSIBLE
def BAD_chubanov_implementation(A,b,c,x):
    if KNOWN_OPTIMAL_SOLUTION is None:
        D = saturatedconstraints(A,b,x)

        allsubset = [[]]
        for d in D:
            allsubsetbis = [S.copy() for S in allsubset]
            for S in allsubsetbis:
                allsubset.append(S+[d])
        allsubset = allsubset[::-1]
        allsubset.pop()

        for S in allsubset:
            print(".",end="")
            subA = [A[m].copy() for m in S if m>=0]+[opposedvector(c)]
            v = solve_linear_equality(subA,[Fraction(1)]*(len(S)+1))
            if v is None:
                continue
            if all([produitscalaire(A[m],v) > Fraction() for m in D]):
                print(" ")
                return v
        print(" ")
        return None


    print("using known optimal solution instead of a real query")
    v = combinaisonlineaire(KNOWN_OPTIMAL_SOLUTION,Fraction(-1),x)
    alpha = produitscalaire(v, c)
    w = combinaisonlineaire(v,-Fraction(99)*alpha/Fraction(100),c)
    return w


#return x such that Ax>=b and cx minimal
#under assumption that Ac = 3/5 1 - cx is bounded by 0
def slideandjump1(A,b,c):
    x = [Fraction()]*len(A[0])
    x[-1] = Fraction(max(b)+1) * Fraction(5,3)

    while True:
        print("sliding move")
        x = slide1(A,b,c,x)
        D = saturatedconstraints(A,b,x)
        print(D)

        if produitscalaire(x,c)==Fraction():
            print("found optimal")
            return x

        print("chubanov jump")
        v = BAD_chubanov_implementation(A,b,c,x)
        if v==None:
            print("found optimal")
            return x
        l = maximalmoves(A,b,x,v)
        x = combinaisonlineaire(x,l/Fraction(4),v)


def slideandjump2(A,b,c):
    x = [Fraction()]*len(A[0])
    x[-1] = Fraction(max(b)+1) * Fraction(5,3)

    while True:
        print("sliding move")
        x = slide2(A,b,c,x)
        D = saturatedconstraints(A,b,x)
        print(D)

        if produitscalaire(x,c)==Fraction():
```

```
            print("found optimal")
            return x

        print("chubanov jump")
        v = BAD_chubanov_implementation(A,b,c,x)
        if v==None:
            print("found optimal")
            return x
        l = maximalmoves(A,b,x,v)
        x = combinaisonlineaire(x,l/Fraction(4),v)


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

def allocatematrix(M,N):
    emptyrow = [Fraction() for n in range(N)]
    return [emptyrow.copy() for m in range(M)]

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 = [Fraction()]*(M+4)
    c = [Fraction()]*(N+3)
    c[-1] = Fraction(1)

    normRawA = [Fraction()]*M
    normRawAtrick = [Fraction()]*M
    for m in range(M):
        normRawA[m] = produitscalaire(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 = [Fraction()]*(N+3)
    for n in range(N):
        xoptimal[n] = rawxoptimal[n]

    return A,b,c,xoptimal

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 = [Fraction()]*(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] = Fraction(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] = Fraction(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

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

def cubeproblemPrimal(N):
    twopower =[Fraction()]*N
    twopower[0]=Fraction(2)
    for n in range(1,N):
        twopower[n] = Fraction(2)*twopower[n-1]

    b = [Fraction()]*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(N):
        for k in range(n):
            A[n][k] = twopower[n-k]
        A[n][n] = Fraction(1)

    return A,b,c

def cubeproblem(N):
    Araw,braw,craw = cubeproblemPrimal(N)
    A,b = primaldual(Araw,braw,craw)

    xoptimal = [Fraction()]*(2*N)
    xoptimal[-1]=Fraction(2)
    xoptimal[N-1] = braw[-1]

    return A, b, xoptimal

import random

def randomVector(N):
    return [Fraction(random.randint(-100,100)) for n in range(N)]
def randomNegVector(N):
    return [Fraction(random.randint(-100,-1)) for n in range(N)]

def randomMatrix(M,N):
    return [randomVector(N) for m in range(M)]

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

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

    Agreater = randomMatrix(M,N)
    left = randomNegVector(M)
    bgreatertmp = [produitscalaire(Agreater[m],xoptimal) for m in range(M)]
    bgreater = [bgreatertmp[m] + left[m] for m in range(M)]

    return Aequal+Agreater,bequal+bgreater,xoptimal

##############################################################################
############################## MAIN ##########################################
##############################################################################

print("########### random ###########")
rawA,rawb,rawxoptimal = randomproblem(10,30)

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


KNOWN_OPTIMAL_SOLUTION = xoptimal
x = slideandjump1(A,b,c)
print(all([produitscalaire(A[m],x)>=b[m] for m in range(len(A))]))
print(produitscalaire(c,x))
x = slideandjump2(A,b,c)
print(all([produitscalaire(A[m],x)>=b[m] for m in range(len(A))]))
print(produitscalaire(c,x))
```

```
print("########### cube ###########")
rawA,rawb,rawxoptimal = cubeproblem(20)

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

KNOWN_OPTIMAL_SOLUTION = xoptimal
x = slideandjump1(A,b,c)
print(all([produitscalaire(A[m],x)>=b[m] for m in range(len(A))]))
print(produitscalaire(c,x))
x = slideandjump2(A,b,c)
print(all([produitscalaire(A[m],x)>=b[m] for m in range(len(A))]))
print(produitscalaire(c,x))
```