Saint Petersburg National Research University of Information Technologies, Mechanics and Optics (ITMO University)

Linear Programming Algorithms

Authors:

Egor Turukhanov (C4134)

Anastasiia Kiseleva (C4133)

Juan Camilo Diosa Echeverri (C4134)

Date:27/11/2019

Saint-Petersburg, 2019

Linear Programming algorithms

A linear programming algorithm is defined as an algorithm that allows solving a problem of optimization (maximization or minimization) of a linear function subject to linear constraints by applying several processes or steps. These constraints may be equalities or designations.[1]

Simplex Algorithm

This is the standard technique in "*Linear Programming*" to solve an optimization problem, in which you have an *"objective function"* and several functions expressed as inequalities called *"restrictions".*

In this way the vertices are tested as possible solutions.

Algebraically, this procedure consists in finding a possible basic solution, then with it, generating new possible basic solutions, so that with each of them, the value of the objective function increases.[2]

In 1947, George B. Dantzig developed a technique to solve linear programs,[3] inspired by the input output Matrix created by the nobel prize in economics Wassily Leontief.[4]

Algorithm:

$$max(min)z = \ c^T x$$

Subject to:

$$Ax = b$$

$$x \geq 0$$

To solve a linear programming model using the Simplex method the following steps are necessary:

The simplex algorithm is an iterative process that starts from the origin of the n-D vector space $x_1 = \cdots = x_n = 0$, and goes through a sequence of vertices of the polytope to eventually arrive at the optimal vertex at which the objective function is maximized. To do so, we first convert the standard form of the problem into a tableau, a table of $n + m + 1$ columns and $m + 1$ rows:

[1] https://www.math.ucla.edu/~tom/LP.pdf

[2] http://repobib.ubiobio.cl/jspui/bitstream/123456789/282/3/Chavez_Abello_Rodrigo.pdf

[3] https://math.mit.edu/~goemans/18310S15/lpnotes310.pdf

[4] http://repobib.ubiobio.cl/jspui/bitstream/123456789/282/3/Chavez_Abello_Rodrigo.pdf

Basic variables	X_1	X_2	...	Xn	s_1	s_2	...	S_m	*Basic Solution*
s1	a_{11}	a_{12}	...	a_{1n}	1	0	...	0	b_1
s2	a_{21}	a_{22}	...	a_{2n}	0	1	...	0	b_2
...	...	...	...	...	...	...	...	...	...
Sm	a_{m1}	a_{m2}	...	a_{mn}	0	0	...	1	b_m
Z	$-C_1$	$-C_2$	...	$-C_m$	0	0	...	0	0

We select x_j in the "j" column of the tableau if it is most heavily weighted by ($Cj = max\{C_1, \ldots, C_n\}$ (- Cj is most negative), as this x_j will increase $z = \sum_{j=1}^{n} c_j x_j$ more than any other x_k $(k \neq j)$.

Choose the constraint with least value $\frac{b_k}{a_{kj}}$.

Divide all elements in the "i" row of A by a_{kj}, so that a_{kj}=1.

Subtract a_{kj}, times the "i" row from the "k" row so that $a_{kj} = 0$, $(k = 1, \ldots, m, k \neq j)$.

So a_{kj} becomes 1 while all other elements in the "j" column become zero. This is Gaussian elimination based on pivoting.

Repeat the steps above until all elements in the last row are non-negative.[5] In this way the optimal feasible basic solution is found.

When there is no initial basic solution, the methods of the Big- M or the two-phase method are usually used[6],through the use of artificial variables.

Code (Python):

Consider the following linear programming system of a company that wants to maximize the production of 2 goods, $x_1$and x_2 and has the function Z defined:

$$Z = 4x_1 + 3x_2$$

and is subject to some restrictions of space, time and financial resources and defines its restrictions in the form of equations linear:

$$\leq 6$$

$$\leq 3$$

$$\leq 5$$

$$\leq 4$$

$$\geq 0$$

[5] http://fourier.eng.hmc.edu/e176/lectures/NM/node32.html

[6] http://repobib.ubiobio.cl/jspui/bitstream/123456789/282/3/Chavez_Abello_Rodrigo.pdf

Using the scipy.optimize library in python:

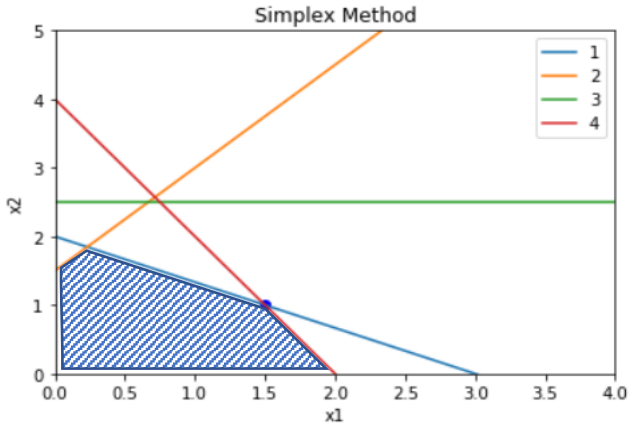
```
from scipy.optimize import linprog

c=[-4,-3]

A_ub=[[2,3],[-3,2],[0,2],[2,1]]

b_ub=[6,3,5,4]

res=linprog(c,A_ub,b_ub,bounds=(0,None))

print(res)

print("The optimal value is= ",(res.fun)*-1,"\nX:", "Where x1 = ",np.around(res.x[0],1), "and x2 = ",res.x[1])
```

Result:

```
con: array([], dtype=float64)

fun: -9.0 message: 'Optimization terminated successfully.'

nit: 5

slack: array([0. , 5.5, 3. , 0. ])

status: 0

success: True x: array([1.5, 1. ])

The optimal value is= 9.0 X: Where x1 = 1.5 and x2 = 1.0
```

Graph:

```
x_vals = np.arange(0,11,1)
v=[0,4,0,5]
y1 = (((6-2*(x_vals))/3))
y2 = ((3+(3*(x_vals)))/2)
y3 = 5/2+(0*x_vals)
y4 = (4-2*(x_vals))
plt.axis(v)
plt.plot(x_vals, y1,label="1")
plt.plot(x_vals, y2,label="2")
plt.plot(x_vals, y3,label="3")
plt.plot(x_vals, y4,label="4")
plt.scatter(1.5,1,c="b")


plt.title("Simplex Method")
plt.legend()
plt.show()
```



*The highlighted blue area represents the set of basic feasible solutions, while the blue dot represents the point where the z function takes its maximum value.

Ellipsoid algorithm

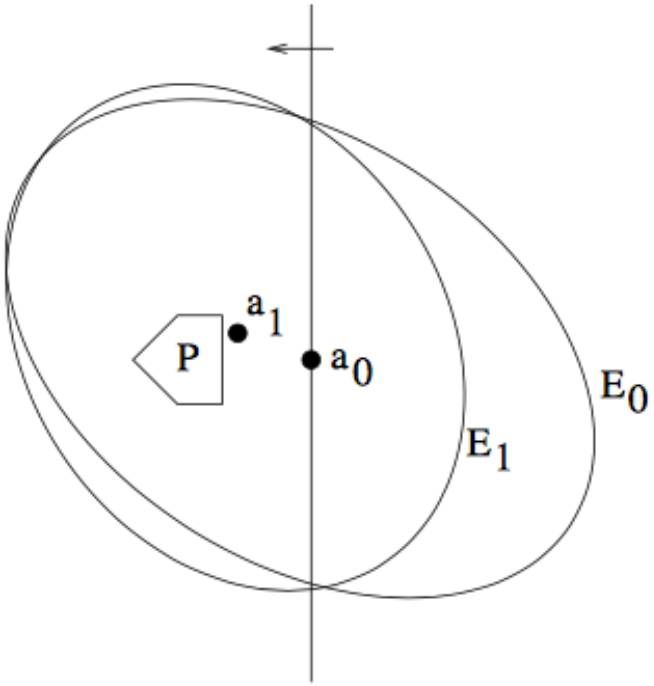
The Ellipsoid algorithm is the first polynomial-time algorithm discovered for linear programming. The Ellipsoid algorithm was proposed by the Russian mathematician Shor in 1977 for general convex optimization problems and applied to linear programming by Khachyan in 1979. Contrary to the simplex algorithm, the ellipsoid algorithm is not very fast in practice; however, its theoretical polynomiality has important consequences for combinatorial optimization problems.

The problem being consider by the ellipsoid algorithm is:

Given a bounded convex set P∈Rn find x∈P

We will see that we can reduce linear programing to finding an x in P = {x∈Rn : Cx≤ d}.

The ellipsoid algorithm works as follows. We start with a big ellipsoid E that is guar- anteed to contain P. We then check if the center of the ellipsoid is in P. If it is, we are done, we found a, it is explicitly given in the description of P) which is not satisfied by our center. One iteration of the ellipsoid algorithm is illustrated in next Figure. The ellipsoid algorithm is the following:



Algorithm

1. Let E0 be an ellipsoid containing P
2. While center ak of Ek is not in P do:

    a. Let cTx <= cTak be such that {x : cTx <= cTak} ⊇P
    b. Let Ek+1 be the minimum volume ellipsoid containing Ek ⋂{x : cTx <= cTak}
    c. k <- k+1

The ellipsoid algorithm has the important property that the ellipsoids constructed shrink in volume as the algorithm proceeds; this is stated precisely in the next lemma. This means that if the set P has positive volume, we will eventually find a point in P. We will need to deal with the

case when P has no volume (i.e. P has just a single point), and also discuss when we can stop and be guaranteed that either we have a point in P or we know that P is empty.

Definition of ellipsoid

Given a center a, and a positive definite matrix A, the ellipsoid E(a, A) is defined as $\{x \in R^n (x-a)^T A^{-1} (x-a) <= 1 \}$

Also, we can use ellipsoid method when it is necessary define: does the system of inequality Cx <= d belong any solution and find at least one of solution.

Ellipsoid algorithm for linear programming

If in the linear programming problem, it was possible to construct a ball containing the desired solution (it describes in Ellipsoid method), then it can be solved be the ellipsoid method. To do this, first find some point u inside the ball, satisfying the constraints of the problem. Draw a hyperplane through it f(x) < f(u), where f – is our objective function, and then find point at the intersection of the original and new hyperplanes (starting from the current ellipsoid). Now, with the new found point, we do the same. Process converges to optimal solution with exponential speed (because value of ellipsoid decrease with exponential speed).

Projective algorithm

Karmarkar's algorithm is an algorithm introduced by Narendra Karmarkar in 1984 for solving linear programming problems. It was the first reasonably efficient algorithm that solves these problems in polynomial time. The ellipsoid method is also polynomial time but proved to be inefficient in practice.

Denoting n as the number of variables and L as the number of bits of input to the algorithm, Karmarkar's algorithm requires $O(n^{3,5}L)$ operations on $O(L)$digit numbers, as compared to such operations for the ellipsoid algorithm. The runtime of Karmarkar's algorithm is $O(n^{3,5}L^2)$
Karmarkar's algorithm falls within the class of interior point methods: the current guess for the solution does not follow the boundary of the feasible set as in the simplex method, but it moves through the interior of the feasible region, improving the approximation of the optimal solution by a definite fraction with every iteration, and converging to an optimal solution with rational data.

Linear programming problem in matrix form

Max $c^T x$
Subject to $Ax \leq b$.

Karmarkar's algorithm determines the next feasible direction toward optimality and scales back by a factor $0 < \gamma \leq 1$. Karmarkar also has extended the method to solve problems with integer constraints and non-convex problems.

Since the actual algorithm is rather complicated, researchers looked for a more intuitive version of it, and in 1985 developed affine scaling, a version of Karmarkar's algorithm that uses affine transformations where Karmarkar used projective ones, only to realize four years later that they had rediscovered an algorithm published by Soviet mathematician I. I. Dikin in 1967. The affine-scaling method can be described succinctly as follows. The affine-scaling algorithm, while applicable to small scale problems, is not a polynomial time algorithm.

Affine-Scaling Algorithm[7]

Input: A, b, c, x^0, ***stopping criterion***, γ.

Algorithm in pseudo code:

$$k \leftarrow 0$$

*Do while **stopping criterion** not satisfied*

$$v^k \leftarrow b \leftarrow Ax^k$$

$$D_v \leftarrow diag\left(v_1^k, \dots, v_m^k\right)$$

$$h_x \leftarrow (A^T D_v^{-2} A)^{-1} c$$

$$h_v \leftarrow Ah_x$$

If $h_v \geq 0$ *then*
return unbounded
End of $\alpha \leftarrow \gamma * \{(h_v)_i < 0, i = 1, \dots, m\}$

$$x^{k+1} \leftarrow x^k + \alpha h_x$$

$$k \leftarrow k + 1$$

End do.

[7] Lagarias, J. C., and R. J. Vanderbei. "II Dikin's convergence result for the affine scaling algorithm." *Contemporary Math* 114 (1990): 109-119.

Original projective algorithm

However, original projective Karmarkar's algorithm paper[8] was:

Input: $A \in Z^{m\,x\,n}, c \in Z^n$.

Let $\Omega = \{Ax = 0\}, \Delta = \{x | x \geq 0, \sum \quad x_i = 1\}$.

Assumptions: $c^T x \geq 0$ for every $x \in \Omega \cap \Delta$; a_0 is a feasible starting point, i.e. $Aa_0 = 0$; $c^T a_0 \neq 0$.

Output: either $x_0 \in \Omega \cap \Delta$ such that $c^T x_0 = 0$ or a proof that $\{ e^T x | x \in \Omega \cap \Delta \} > 0$.

Step 0. Initialize

$$x^0 = center\ of\ the\ simplex$$

Step 1. Compute the next point in the sequence

$$x^{k+1} = \varphi(x^{(k)})$$

The function $b = \varphi(a)$ is defined by the following sequences of operators.

Let $D = diag\{a_1, a_2, \dots, a_n\}$ be diagonal matrix whose i^{th} diagonal entry is a_i.

1. Let $B = \left[\frac{AD}{e^T}\right]$

   i.e., augment the matrix AD with a row of all 1's. This guarantees that Ker $B \subseteq \sum$.
2. Compute the orthogonal projection of D_c into the null space of B.

$$c_p = [I - B^T(BB^T)^{-1}B]Dc.$$

3. $\hat{c} = \frac{c_p}{|c_p|}$, $i.e.\ \hat{c}$ is the unit vector in the direction c_p.
4. $b\acute{} = a_0 - \alpha r \hat{c}$ i.e take a step of length αr in the direction $\hat{c}$, where r is the radius of the largest inscribed sphere

$$r = \frac{1}{\sqrt{n(n-1)}}$$

And $\alpha \in (0, 1)$ is a parameter which can be set equal ¼.

5. Apply inverse projective transformation to $b\acute{}$

$$b = \frac{Db\acute{}}{e^T Db\acute{}}$$

Return b.

Step 2. Check for infeasibility

We define a "potential" function by

$$f(x) = \sum_i \quad ln\frac{c^T x}{x_i}$$

We expect certain improvement δ in the potential function at each step. The value of δ depends on the choice of parameter α in Step 1.4. For example, if $\alpha = 1/4$ then δ = 1/8. If we don't observe the expected improvement i.e. if $f\left(x^{(k+1)}\right) > f\left(x^{(k)}\right) - \delta$, then we stop and conclude that the minimum value of the objective function must be strictly positive. When the canonical form of the problem was obtained by transformation on the standard linear program, then this situation corresponds to the case that the original problem does not have a finite optimum i.e. it is either infeasible or unbounded.

[8] Karmarkar, Narendra. "A new polynomial-time algorithm for linear programming." Proceedings of the sixteenth annual ACM symposium on Theory of computing. ACM, 1984..

Step 3. Check for optimality.

This check is carried out periodically. It involves going from the current interior point to an extreme point without increasing the value of the objective function and testing the extreme point for optimality. This is done only when the time spent since the last check exceeds the time required for checking.

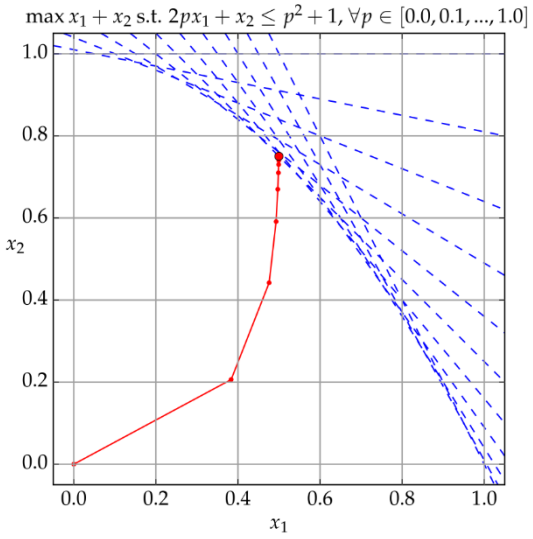
Go to Step 1.

However, this algorithm works by moving across the interior of the feasible region but it is far more complicated than the ellipsoid method. Luckily, within a year, Gill had shown that the path traced by the projective algorithm was the same as the path traced by a simpler algorithm in the family of barrier methods.

The primal **Newton barrier method** is equivalent to Karmarkar's original projective algorithm, but it captures the form of modern interior point methods well.

Graph of projective algorithm implementation for linear problem:

$$\begin{aligned} \text{maximize} \quad & x_1 + x_2 \\ \text{subject to} \quad & 2px_1 + x_2 \leq p^2 + 1, \quad p = 0.0, 0.1, 0.2, \ldots, 0.9, 1.0. \end{aligned}$$



II. Practice:

Problem formulation:

Whiskas cat food, shown above, is manufactured by Uncle Ben's. Uncle Ben's want to produce their cat food products as cheaply as possible while ensuring they meet the stated nutritional analysis requirements shown on the cans. Thus, they want to vary the quantities of each ingredient used (the main ingredients being chicken, beef, mutton, rice, wheat and gel) while still meeting their nutritional standards.

The costs of the chicken, beef, and mutton are \$0.013, \$0.008 and \$0.010 respectively, while the costs of the rice, wheat and gel are \$0.002, \$0.005 and \$0.001 respectively. (All costs are per gram.) For this exercise we will ignore the vitamin and mineral ingredients. (Any costs for these are likely to be very small anyway.)

Each ingredient contributes to the total weight of protein, fat, fibre and salt in the final product. The contributions (in grams) per gram of ingredient are given in the table below.

Stuff	Protein	Fat	Fibre	Salt
Chicken	0.100	0.080	0.001	0.002
Beef	0.200	0.100	0.005	0.005
Rice	0.000	0.010	0.100	0.002
Wheat bran	0.040	0.010	0.150	0.008

Identify the Decision Variables For the Whiskas Cat Food Problem the decision variables are the percentages of the different ingredients we include in the can. Since the can is 100g, these percentages also represent the amount in g of each ingredient included. We must formally define our decision variables, being sure to state the units we are using.

Assume Whiskas want to make their cat food out of just two ingredients: Chicken and Beef. We will first define our decision variables:

$$x_1 = \ percentage\ of\ chicken\ meat\ in\ can\ of\ cat\ food$$

$$x_2 = \ percentage\ of\ beef\ used\ in\ can\ of\ cat\ food$$

The limitations on these variables (greater than zero) must be noted but for the Python implementation, they are not entered or listed separately or with the other constraints.

The objective function becomes:

$$min\ 0.013x_1 + 0.008x_2$$

The constraints on the variables are that they must sum to 100 and that the nutritional requirements are met:

$$1.000x_1 + 1.000x_2 = 100.0$$

$$0.100x_1 + 0.200x_2 \geq 8.0$$

$$0.080x_1 + 0.100x_2 \geq 6.0$$

$$0.001x_1 + 0.005x_2 \leq 2.0$$

$$0.002x_1 + 0.005x_2 \leq 0.4$$

For each algorithm we will run 10 times and measure average time. All algorithm was implemented on one PC.

Solutions and methods comparing.

Simplex algorithm

```
c = [0.013,0.008]
A_ub = [[0.001, 0.005],[0.002,0.005],[-0.080,-0.1],[-0.1,-0.2]]
b_ub = [2,0.4,-6,-8]
A_eq = [[1,1]]
b_eq = [100]
total_time = []
vector_mean = pd.DataFrame(columns=['Time'])
for i in range(0,10,1):
  t = timeit.default_timer()
  res=linprog(c,A_ub,b_ub,A_eq,b_eq, bounds=(0,None))

  #Total_time = time() - start_time
  elapsed_time = timeit.default_timer() - t
  total_time.append(elapsed_time)
vector_mean = vector_mean.append({'Time': np.mean(total_time)}, ignore_index=True)
```

We make 10 run of our algorithm and average time is 0.012601 seconds with 5 iterations to find optimal solution, which is $x_1 = 33.333$, $x_2 = 66.667$ and objective function result is 0.9667

```
In [40]: vector_mean
Out[40]:
       Time
0  0.012601
```

Ellipsoid algorithm

```
C = [-1 0; 0 -1; -0.1 -0.2; -0.08 -0.1; 0.001 0.005; 0.002 0.005; -1 -1;1 1];
d = [0; 0; -8 ; -6;2; 0.4; -99.8; 100.0];
o = [0.013 0.008];


% C = [1 1 1 1 1 1;-1 0 0 0 0 0;0 -1 0 0 0 0;0 0 -1 0 0 0; 0 0 0 -1 0 0; 0 0 0 0 -1 0;0 0 0 0 0 -1;
% d = [100;0; 0; 0; 0; 0;0;-8;-6;2;0.4; -99];
% o = [0.013 0.008 0.010 0.002 0.005 0.001];

[a,iter] = ellipsoid_method(C,d,o,'central','optimize',1,'eps',2, 'numiter', 50000);

%[a,iter] = ellipsoid_method(C,d,'radius',7,'numiter',8,'plot_fig',1,'plot_iter',[0:7],
 'plot_title',title,'plot_separating',0,'plot_gradient',0,'ignore_blowup',1);
format longG
fprintf('Best feasible point after iteration %i:\n', iter)
disp(a)
fprintf('Objective Value: %d\n', o*a)
```

For ellipsoid algorithm after 10 run average time is 0.0654 seconds. Each run takes 324 iterations to find optimal solution, which is x1 = 40.57 and x2 = 59.29 and objective function is 1.0019345

Projective algorithm

Firstly, we make our objective function and constrains in matrix form. Code for this operation, you can be below:

```
"""A -- An n x m numpy matrix of constraint coefficients
b -- A 1 x m numpy row vector of constraint RHS values
c -- A 1 x n numpy row vector of objective function coefficients"""


A = np.matrix([[1, 0.1,0.08,0.001,0.002], [1, 0.2,0.1,0.005,0.005]])
b = np.array([[100,8,6,2,0.4],])
c = np.array([[-0.013, -0.008],])


A = A.reshape(5,2)
```

Next, we implemented our algorithm:

```
def karmarkar(start_point):
    D = np.diagflat(start_point)
    c_tilde = np.matmul(c, D)
    A_tilde = np.matmul(A, D)
    A_tildeT = A_tilde.transpose()
    AAT_inverse = np.linalg.pinv(np.matmul(A_tilde, A_tildeT))
    # matrix multiplication is associative
    P = np.identity(m) - np.matmul(np.matmul(A_tildeT, AAT_inverse), A_tilde)
    cp_tilde = np.matmul(c_tilde, P)
    k = -0.5 / np.amin(cp_tilde)
    x_tilde_new = np.ones((1, m), order='F') + k * cp_tilde
    return np.matmul(x_tilde_new, D)

```

After that, we created algorithm, which will search till required precision will be reached. In our case, precision=0.001

```
def solve(start_point, tolerance=1e-5, max_iterations=50, verbose=True):


    """Uses Karmarkar's Algorithm to solve a Linear Program.
    start_point     -- A starting point for Karmarkar's Algorithm. Must be a row vector.
    tolerance       -- The stopping tolerance of Karmarkar's Algorithm.
    max_iterations  -- The maximum number of iterations to run Karmarkar's Algorithm.
    verbose         -- List all intermediate values.
    """
    x = start_point
    #solution = LPSolution()
    for i in range(max_iterations):
        x_new = karmarkar(x)
        #if verbose:
            #print(x_new)

        dist = np.linalg.norm(x - x_new)
        x = x_new
        solution.append([x,i,dist])
        print(np.float64(np.tensordot(c, solution[-1][0], axes=((0,1),(0,1)))))
        #solution.append(x)
        if dist < tolerance:
            break

    solution.append([x,i,dist])
```

```
for i in range(0,10,1):
    t = timeit.default_timer()
    solve(start_point)
    elapsed_time = timeit.default_timer() - t
    total_time.append(elapsed_time)

vector_mean = vector_mean.append({'Time': np.mean(total_time)}, ignore_index=True)
```

So, projective algorithm with 10 runs average time is 0.008662 seconds and 17 iterations to find optimal solution, which is x1= 38.147 and x2=61.9888 and objective function result is 0.9918214.

```
In [65]: vector_mean
Out[65]:
       Time
0  0.008662
```

Analysis of result.

Algorithm	Average time (10 runs), sec	Iterations, num	Optimal Objective Function
Simplex	0.012	5	0.9667
Ellipsoid	0.0654	324	1.0019
Projective	0.008	17	0.9918

We have solved our linear programming problem with 3 different algorithms. The fastest algorithm was Projective method, the second fastest was Simplex algorithm. However, for simplex method it takes only 5 iterations to find optimal solution to our problem.

Literature

1. https://www.math.ucla.edu/~tom/LP.pdf
2. http://repobib.ubiobio.cl/jspui/bitstream/123456789/282/3/Chavez_Abello_Rodrigo.pdf
3. https://math.mit.edu/~goemans/18310S15/lpnotes310.pdf
4. http://fourier.eng.hmc.edu/e176/lectures/NM/node32.html
5. https://personal.utdallas.edu/~metin/Or6201/simplex_s.pdf
6. Lagarias, J. C., and R. J. Vanderbei. "II Dikin's convergence result for the affine scaling algorithm." *Contemporary Math* 114 (1990): 109-119.
7. Karmarkar, Narendra. "A new polynomial-time algorithm for linear programming." *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. ACM, 1984.
8. Rebennack S. (2008) Ellipsoid Method. In: Floudas C., Pardalos P. (eds) Encyclopedia of Optimization. Springer, Boston
9. http://www-math.mit.edu/~goemans/18433S09/ellipsoid.pdf
10. https://www.cs.princeton.edu/courses/archive/fall05/cos521/ellipsoid.pdf
11. https://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15859-f11/www/notes/lecture08.pdf
12. https://www.coursera.org/lecture/advanced-algorithms-and-complexity/optional-the-ellipsoid-algorithm-N9rzA