Hands on Approach to Linear Programming Problem Solving via Simplex Method

Aldo Terán Espinoza
aldot@kth.se

November 24, 2019

In this document, we present a hands-on tutorial on how to solve linear programming problems using the Simplex method. We wrote a small library in Python with all the tools necessary to run use simplex to solve a linear program (LP). This report is divided into two main sections: (1) a quick description of the simplex method and how it is implemented in python, and (2) we use an example application to demonstrate how the simplex algorithm solves an LP with our python code.

1 The Simplex Method in Python

There are several ways to find an optimal solution for a linear programming problem. In this document we will explain the Simplex method. The modus operandi of the simplex method is to go from one basic feasible solution to another, decreasing the value of the objective function every iteration. The algorithm will stop once it has found an optimal solution. Algorithm 1 details the Simplex method. This represents only one iteration, it normally requires more than one iteration to solve an LP, and solving them by hand quickly becomes a hassle. As a consequence, software solvers are available both open source[1] or through software suites such as MATLAB.

To give a hands-on demonstration of the simplex method, we will implement Algorithm 1 using Python. To start, we will use the ubuquitous Python library `numpy` to handle matrices and vectors. Given a linear problem in standard from (A, b, c) and an initial basic feasible solution $(beta, nu)$, we compute lines 2–5:

```
A_beta = A_mat[:, beta]
A_nu = A_mat[:, nu]
c_beta = c_vec[beta]
c_nu = c_vec[nu]
```

[1]https://online-optimizer.appspot.com

Algorithm 1 Simplex Method (one iteration)

```
1:  procedure SIMPLEX($\beta, \nu, A, b, c$)                ▷ We start with an initial BFS
2:      $A_\beta \leftarrow$ cols of $A$ in $\beta$
3:      $A_\nu \leftarrow$ cols of $A$ in $\nu$
4:      $c_\beta \leftarrow$ indices of $c$ in $\beta$
5:      $c_\nu \leftarrow$ indices of $c$ in $\nu$
6:      $\bar{b} = A_\beta^{-1} b$                           ▷ Calculate $\bar{b}, y, r_\nu$
7:      $y = A_\beta^{-T} c_\beta$
8:      $r_\nu = c_\nu - A_\nu^T y$
9:      if $r \geq 0$ then                                  ▷ If true we found optimal solution
10:         return $x_\beta = \bar{b}, x_\nu = 0$
11:     $q = argmin(r_\nu)$                ▷ Otherwise, find index $q$ that enters the basis
12:     $\bar{a}_{\nu_q} = A_\beta^{-1} a_q$                ▷ Where $a_q$ is the $q$th column of $A$
13:     if $\bar{a}_{\nu_q} \leq 0$ then                    ▷ If true there is no solution
14:         exit
15:     $t = \{\frac{\bar{b}_i}{\bar{a}_{i,\nu_q}} : \bar{a}_{i,\nu_q} > 0\}$   ▷ Otherwise, get index $p$ that enters the basis
16:     $p = argmin(t)$
17:     interchange $\beta_p$ for $\nu_q$
18:     return updated $\beta$ and $\nu$
```

Next, we calculate $\bar{b}$, y, and r_ν from lines 6–8:

```
A_beta_inv = inv(A_beta)
b_bar = np.dot(A_beta_inv, b_vec)
y = np.dot(A_beta_inv.transpose(), c_beta)
r_nu = c_nu - np.dot(A_nu.transpose(), y)
```

If the condition on line 9 is fulfilled, we have found an optimal solution. We rearrange the x vector and calculate the optimal value $c^T x$ and return this information:

```
if min(r_nu) >= 0:
    x = np.vstack((b_bar, np.zeros((m,1))))
    min_val = np.dot(c_vec.transpose(), x)
    x = np.zeros(n)
    x[beta] = b_bar
    return [x, min_val]
```

We use the `numpy.array.argmin` method for line 11 and we use the previously computed A_β^{-1} for line 12:

```
q = r_nu.argmin()
a_bar = np.asarray(np.dot(A_beta_inv, A_mat[:, [q]]))
```

For the `if` statement on line 13 we use the `numpy.array.all()` method to check all elements of the array individually:

```
if (a_bar <= 0).all():
    exit()
```

To make the steps 15 and 16 more explicit, we used a simple search loop that keeps track of the index and the minimum value of t:

```
t_max = INF
i = 0
for val in a_bar:
    if val > 0:
        temp = b_bar[i] / val
        if temp < t_max:
            t_max = temp
            p = i
    i += 1
```

Finally, line 17 and 18 are straightforward:

```
new_basis = nu[q]
nu[q] = beta[p]
beta[p] = new_basis
return [beta, nu]
```

Now we are done! We can use the new basis β and ν for the next iteration of the simplex.

We have detailed only one iteration of the `run_simplex` method from the `SimplexSolver` class we have written in Python. We have implemented as well a method `find_initial_bfs` that will solve an artificial linear program (**P'**) with simplex to find a BFS for the initial program **P**. The method `find_solution` will run the simplex method iteratively until an optimal solution (or no solution) is found. If the python script is ran on an interpreter, it will solve the pre-programmed Example 5.4 from the book (pag. 44-47) which can be used as a . The code is avialable online here.

2 Example Application

To illustrate the application of the Simplex method for solving linear programming problems we have the following example.

In a robotic manufacturing company, four different models of automated service robots are sold: Sweep X, Sweep Y, Mower Z, and Mower XYZ. The *Sweep* models are indoor autonomous vacuuming robots, while the *Mower* models are autonomous lawnmowers. The manufacturing team made a big order with several different types of sensors needed for building these robots;

the quantity for each sensor was clearly specified since the factory has a constant production rate. Unfortunately, the postal service lost a big part of the shipment, wreaking havoc on the manufacturing team. Panicked and out of breath, the manufacturing team came to us for help in order to come up with the best production plan with the sensors that managed to reach the factory–with the obvious objective of maximizing the profits we will get out of it. They gave us the following information:

	Lidar	**Camera**	**GPS**	**Wheel Encoders**	**Selling Price (SEK)**
Sweep X	1	0	0	4	4000
Sweep Y	1	2	0	4	6000
Mower Z	0	2	1	4	10000
Mower XYZ	2	2	1	4	12000
Inventory	16	18	3	40	

We instantaneously noticed that the problem could be modeled as a linear program:

$$\begin{array}{ll} \max & 4000x_1 + 6000x_2 + 10000x_3 + 12000x_4 \\ \text{s.t.} & x_1 + x_2 + 2x_4 \leq 16 \\ & 2x_2 + 2x_3 + 2x_4 \leq 18 \\ & x_3 + x_4 \leq 3 \\ & 4x_1 + 4x_2 + 4x_3 + 4x_4 \leq 40 \\ & x_i \geq 0 \qquad\qquad \forall i \in 1...n \end{array} \tag{1}$$

Where x_1 is the number of Sweep X robots, x_2 the number of Sweep Ys, x_3 the number of Mower Zs, and x_4 the number of Mower XYZs. Lucky for us, we just programmed a library to solve linear programming problems with the Simplex method. First step is to change the problem into standard form using slack variables:

$$\begin{array}{ll} \min & -4000x_1 - 6000x_2 - 10000x_3 - 12000x_4 \\ \text{s.t.} & x_1 + x_2 + 2x_4 + x_5 = 16 \\ & 2x_2 + 2x_3 + 2x_4 + x_6 = 18 \\ & x_3 + x_4 + x_7 = 3 \\ & 4x_1 + 4x_2 + 4x_3 + 4x_4 + x_8 = 40 \\ & x_i \geq 0 \qquad\qquad \forall i \in 1...n \end{array} \tag{2}$$

Once in standard from, we just have to build the A matrix and the vectors b and c:

$$A = \begin{bmatrix} 1 & 1 & 0 & 2 & 1 & 0 & 0 & 0 \\ 0 & 2 & 2 & 2 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 4 & 4 & 4 & 4 & 0 & 0 & 0 & 1 \end{bmatrix} \quad b = \begin{bmatrix} 16 \\ 18 \\ 3 \\ 76 \end{bmatrix} \quad c = \begin{bmatrix} 4000 \\ 6000 \\ 10000 \\ 12000 \\ \mathbf{0} \end{bmatrix}$$

Where 0 is a vector of zeros of size $m = 4$. We go ahead and plug in the values into our solver. We use the `iPython`[2] in the Linux terminal as a Python interpreter and import our solver class. We use numpy arrays for the matrices and vectors and instantiate our solver:

```
In: from SimplexSolver import SimplexSolver
In: import numpy as np
In: A = np.array([[1,1,0,2,1,0,0,0],
                  [0,2,2,2,0,1,0,0],
                  [0,0,1,1,0,0,1,0],
                  [4,4,4,4,0,0,0,1]])
In: b = np.array([[16],[18],[3],[76]])
In: c = np.array([[-4000],[-6000],[-10000],[-12000],[0],[0],[0],[0]])
In: solver = SimplexSolver()
```

Once the solver is instantiated, we go ahead and use the slack variables as our initial basic feasible solution, i.e., $\beta = [4, 5, 6, 7]$ and $\nu = [0, 1, 2, 3]$:

```
In: beta = [4,5,6,7]
In: nu = [0,1,2,3]
In: x_opt = solver.find_solution(beta, nu, A, b, c)
```

When instantiating the solver, we can use `SimplexSolver(verbose=True)` to print the whole procedure the simplex method goes through for every iteration. We added the full output of the solver on the appendix (Section 4.2).

After 5 short iterations, the simplex solver found an optimal solution with:

$$\hat{x} = \begin{bmatrix} 10 & 6 & 3 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T \quad Min = -106000$$

Since the slack variables are all equal to zero, we can appreciate that the postal service delivered an even amount of sensors so we will not have any leftovers. To our consumer's disdain, however, the Mower XYZ will be out of stock until the other batch of sensors arrives. We rush this information to the manufacturing team:

[2]Any Python interpreter of your preference will also work.

	Sweep X	**Sweep Y**	**Mower Z**	**Mower XYZ**	**Total Earnings (SEK)**
No.	10	6	3	0	106,000

They were extremely grateful for our impeccable work so they bought fika for the whole company.

3 References

1. Amol Sasane and Krister Svanberg, *Optimization* course book, Department of Mathematics, KTH.

4 Appendix

4.1 Python Script

```python
#!usr/bin/env python

"""
Class script for a linear programming problem solver
with the Simplex method.

Author: Aldo Teran <aldot@kth.se>
"""

import numpy as np
from numpy.linalg import inv
import matplotlib.pyplot as plt
import pdb

class SimplexSolver:

    def __init__(self, A_mat, b_vec, c_vec, is_min=True, verbose=True):
        """
        Initialize the object with the 'A' matrix,
        the vector 'b' with the contraints, vector
        'c' with the associated costs. It is assumed
        that the problem is already in standard form.
        """

        # Assign variables to class attributes
        self.A_mat = A_mat
        self.b_vec = b_vec
        self.c_vec = c_vec
        self.is_min = is_min
        self.iteration = 0
        self.verbose = verbose

    def plot_2D_problem(self):
        # Will plot the constraints of the problem
        self.fig = plt.figure()
        self.ax = plt.axes()

        [m, n] = self.A_mat.shape

        # Create a dictionary with variables
    variables = {}
    for i in range(n):
        variables["x_{0}".format(i)] = np.linspace(0, max(self.b_vec)*1.5, 10)

    # Plot constraints with x_2 on the vertical axis
    for i in range(m):
        x_2 = (self.b_vec[i] - variables['x_0'] * self.A_mat[i, 0]) / self.A_mat[i ,1]
        plt.plot(variables['x_0'], x_2, label=r'$f_{0}$'.format(i))

    plt.xlim((0, max(self.b_vec)))
    plt.ylim((0, max(self.b_vec)))
    plt.xlabel(r'$x_1$')
    plt.ylabel(r'$x_2$')
    plt.legend()

def run_simplex(self, beta, nu, A_mat, b_vec, c_vec):
    """
    Runs one iteration of the simplex algorithm on
    the given problem.
    """
    if self.verbose:
        print("----------------------------------------------------")
        print("Starting iteration number {0}".format(self.iteration))

    # First off, get the dimensions of the problem
    [m, n] = A_mat.shape

    # Separate A_beta and A_nu
    A_beta = A_mat[:, beta]
    A_nu = A_mat[:, nu]

    # Separate costs
    c_beta = c_vec[beta]
    c_nu = c_vec[nu]

    # Calculate the value (b_bar) for our basic variables (A_beta * b_bar = b)
    A_beta_inv = inv(A_beta)
    b_bar = np.dot(A_beta_inv, b_vec)

    if self.verbose:
        print("A_beta:")
        print(A_beta)
        print("A_nu:")
        print(A_nu)
        print("c_beta:")
        print(c_beta)
        print("c_nu:")
        print(c_nu)
        print("b_bar:")
        print(b_bar)

    # Calculate simplex multipliers (y) with (A_beta^T * y = c_beta)
    y = np.dot(A_beta_inv.transpose(), c_beta)

    # Compute reduce cost for non basic variables (r_nu)
    # with (r_nu = c_nu - A_nu^T * y)
    r_nu = c_nu - np.dot(A_nu.transpose(), y)

    if self.verbose:
        print("Simplex multipliers y:")
        print(y)
        print("Reduced cost for iteration {0}".format(self.iteration))
        print(r_nu)

    # Done if elements in r_nu nonegative
    if min(r_nu) >= 0:
        print("Found optimal solution! The optimal x is:")
        x = np.vstack((b_bar, np.zeros((m,1))))
        print(x)
        print("Optimal value is:")
        print(np.dot(c_vec.transpose(), x))

        # Create an empty vector for x and fill it with b_bar values
        x = np.zeros(n)
        x[beta] = b_bar

        return [x, beta, nu, True]

    # Get min of r_nu to determine which index (q) enters the basis
    q = r_nu.argmin()

    # Determine vector (a_bar) with (A_beta * a_bar = a_q)
    a_bar = np.asarray(np.dot(A_beta_inv, A_mat[:, [q]]))

    if self.verbose:
        print("Index {0} enters the basis".format(nu[q]))
        print("a_bar:")
        print(a_bar)
    # Check if values are negative or 0
    if (a_bar <= 0).all():
        print("Solution does not exits! Sorry!")
        print("exiting...")
        return -1

    # Find the index that leaves the basis
    t_max = 9999999
    i = 0
    for val in a_bar:
        if val > 0:
            temp = b_bar[i] / val
            if temp < t_max:
                t_max = temp
                p = i
        i += 1

    if self.verbose:
        print("Index {0} leaves the basis".format(beta[p]))

    # Create an empty vector for x and fill it with b_bar values
    x = np.zeros(n)
    x[beta] = b_bar

    # We now know that index q replaces idx p in the basis
    new_basis = nu[q]
    nu[q] = beta[p]
    beta[p] = new_basis

    if self.verbose:
        print("Solution for iteration {0} is:".format(self.iteration))
        print(x)
        print("The new basis beta is:")
        print(beta)

    self.iteration += 1

    return [x, beta, nu, False]

def find_solution(self, beta, nu, A_mat, b_vec, c_vec):
    """
    Runs the run_simplex method iteratively until
    a solution is found, given an initial solution.
    Returns the
    """
    finished = False
    while not finished:
        # The solver will return True when optimal solution is found
        [x, beta, nu, finished] = self.run_simplex(beta, nu, A_mat, b_vec, c_vec)

    return [x, beta, nu]

def find_initial_bfs(self, A_mat, b_vec, c_vec):
    """
    Finds an intial Basic Feasible Solution using
    slack variables to solve an artificial linear
    program (P').
    """

    # Add slack vars to A
    [rows, cols] = np.shape(A_mat)
    A_mat = np.hstack((A_mat, np.eye(rows)))
    c_vec = np.vstack((np.zeros((cols, 1)), np.ones((rows, 1))))
    [rows, cols] = np.shape(A_mat)

    # Create index list for nu and beta values
    idx_list = [i for i in range(cols)]

    # Beta indices will correspond to the slack variables
    beta_idx = idx_list[-rows:]

    # Nu indices will be the leftovers
    nu_idx = idx_list[:-rows]

    if self.verbose:
        print("Attempting to find initial Basic Feasible Solution with slack variables")
        print("Artificial linear program P':")
        print("A:")
        print(A_mat)
        print("b:")
        print(b_vec)
        print("c:")
        print(c_vec)

        print("Starting basis with slack variables")
        print("beta:")
        print(beta_idx)
        print("nu:")
        print(nu_idx)

    # Run simplex with new matrices and initial basic feasible solution
    [x_opt, beta, nu] = self.find_solution(beta_idx, nu_idx, A_mat, b_vec, c_vec)
    # Case no. 1: optimal solution for P' is positive
    if x_opt.all() > 0:
        print("The linear program (P) has no basic feasible solution!")
        print("exiting program...")
        return -1
    # Case no. 2: optimal solution for P' is 0
    elif x_opt.all() == 0:
        print("Found optimal basic feasible solution for P', the solution is:")
        print(x_opt)

    print("Updating basic feasible solution for linear problem P...")

    self.iteration = 0

    # Return indices corresponding to the BFS of primal problem P
    return [beta, nu]

def main():
"""
Main function to solve the linear programming example 5.4
in the book using the SimplexSolver class above.
"""

# Let's try solving example 5.4 from the book
A = np.array([[1,1,1,0],
              [2,1,0,1]])
b = np.array([[200],
              [300]])
c = np.array([[-400],
              [-300],
              [0],
              [0]])

# We instantiate the simplex solver class with the arrays above
simplex = SimplexSolver(A, b, c)
simplex.plot_2D_problem()
plt.show()

# The last two variables are our slack variables so we start
```

```python
    # with them as out initial basic variables.
    beta = [2,3] # Python starts indexing at 0
    nu = [0,1]

    # Now that we have eveything set up, we start running the solver
    finished = False
    while not finished:
        # The solver will return True when optimal solution is found
        [x, beta, nu, finished] = simplex.run_simplex(beta, nu, A, b, c)

    # x = np.expand_dims(x,1)
    print("Resulting x is:")
    print(x)
    print("Min cost is:")
    print(np.dot(c.transpose(), x))

if __name__ == "__main__":
    main()
```

4.2 Output from example application

```
----------------------------------------------------
Starting iteration number 0
A_beta:
[[1 0 0 0]
 [0 1 0 0]
 [0 0 1 0]
 [0 0 0 1]]
A_nu:
[[1 1 0 2]
 [0 2 2 2]
 [0 0 1 1]
 [4 4 4 4]]
c_beta:
[[0]
 [0]
 [0]
 [0]]
c_nu:
[[ -4000]
 [ -6000]
 [-10000]
 [-12000]]
b_bar:
```

```
[[ 16.]
 [ 18.]
 [  3.]
 [ 76.]]
Simplex multipliers y:
[[ 0.]
 [ 0.]
 [ 0.]
 [ 0.]]
Reduced cost for iteration 0
[[ -4000.]
 [ -6000.]
 [-10000.]
 [-12000.]]
Index 3 enters the basis
a_bar:
[[ 2.]
 [ 2.]
 [ 1.]
 [ 4.]]
Index 6 leaves the basis
Solution for iteration 0 is:
[  0.   0.   0.   0.  16.  18.   3.  76.]
The new basis beta is:
[4, 5, 3, 7]
----------------------------------------------------
Starting iteration number 1
A_beta:
[[1 0 2 0]
 [0 1 2 0]
 [0 0 1 0]
 [0 0 4 1]]
A_nu:
[[1 1 0 0]
 [0 2 2 0]
 [0 0 1 1]
 [4 4 4 0]]
c_beta:
[[     0]
 [     0]
 [-12000]
 [     0]]
c_nu:
[[ -4000]
```

```
 [ -6000]
 [-10000]
 [     0]]
b_bar:
[[ 10.]
 [ 12.]
 [  3.]
 [ 64.]]
Simplex multipliers y:
[[     0.]
 [     0.]
 [-12000.]
 [     0.]]
Reduced cost for iteration 1
[[ -4000.]
 [ -6000.]
 [  2000.]
 [ 12000.]]
Index 1 enters the basis
a_bar:
[[ 1.]
 [ 2.]
 [ 0.]
 [ 4.]]
Index 5 leaves the basis
Solution for iteration 1 is:
[  0.   0.   0.   3.  10.  12.   0.  64.]
The new basis beta is:
[4, 1, 3, 7]
----------------------------------------------------
Starting iteration number 2
A_beta:
[[1 1 2 0]
 [0 2 2 0]
 [0 0 1 0]
 [0 4 4 1]]
A_nu:
[[1 0 0 0]
 [0 1 2 0]
 [0 0 1 1]
 [4 0 4 0]]
c_beta:
[[     0]
 [ -6000]
 [-12000]
 [     0]]
c_nu:
[[ -4000]
 [     0]
 [-10000]
 [     0]]
b_bar:
[[  4.]
 [  6.]
 [  3.]
 [ 40.]]
Simplex multipliers y:
[[    0.]
 [-3000.]
 [-6000.]
 [    0.]]
Reduced cost for iteration 2
[[-4000.]
 [ 3000.]
 [ 2000.]
 [ 6000.]]
Index 0 enters the basis
a_bar:
[[ 1.]
 [ 0.]
 [ 0.]
 [ 4.]]
Index 4 leaves the basis
Solution for iteration 2 is:
[  0.   6.   0.   3.   4.   0.   0.  40.]
The new basis beta is:
[0, 1, 3, 7]
---------------------------------------------------
Starting iteration number 3
A_beta:
[[1 1 2 0]
 [0 2 2 0]
 [0 0 1 0]
 [4 4 4 1]]
A_nu:
[[1 0 0 0]
 [0 1 2 0]
 [0 0 1 1]
 [0 0 4 0]]
c_beta:
[[ -4000]
 [ -6000]
 [-12000]
 [     0]]
c_nu:
[[     0]
 [     0]
 [-10000]
 [     0]]
b_bar:
[[  4.]
 [  6.]
 [  3.]
 [ 24.]]
Simplex multipliers y:
[[-4000.]
 [-1000.]
 [-2000.]
 [    0.]]
Reduced cost for iteration 3
[[ 4000.]
 [ 1000.]
 [-6000.]
 [ 2000.]]
Index 2 enters the basis
a_bar:
[[-2.]
 [ 0.]
 [ 1.]
 [ 8.]]
Index 3 leaves the basis
Solution for iteration 3 is:
[  4.   6.   0.   3.   0.   0.   0.  24.]
The new basis beta is:
[0, 1, 2, 7]
----------------------------------------------------
Starting iteration number 4
A_beta:
[[1 1 0 0]
 [0 2 2 0]
 [0 0 1 0]
 [4 4 4 1]]
```

```
A_nu:
[[1 0 2 0]
 [0 1 2 0]
 [0 0 1 1]
 [0 0 4 0]]
c_beta:
[[ -4000]
 [ -6000]
 [-10000]
 [     0]]
c_nu:
[[     0]
 [     0]
 [-12000]
 [     0]]
b_bar:
[[ 10.]
 [  6.]
 [  3.]
 [  0.]]
Simplex multipliers y:
[[-4000.]
 [-1000.]
 [-8000.]
 [    0.]]
Reduced cost for iteration 4
[[ 4000.]
 [ 1000.]
 [ 6000.]
 [ 8000.]]
Found optimal solution! The optimal x is:
[[ 10.]
 [  6.]
 [  3.]
 [  0.]
 [  0.]
 [  0.]
 [  0.]
 [  0.]]
Optimal value is:
[[-106000.]]
```