Implementation of Primal Simplex Method

Ziyi Yu*

Abstract

Simplex method has been a powerful weapon against linear programming problem ever since it was raised. Here I implemented two versions of primal simplex method – classical simplex and revised simplex. I will give the pseudocode of these methods and explain some detailed implementation in the following paragraphs.

Keywords: linear programming, classical simplex, revised simplex

1 Introduction

Simplex method has been a powerful weapon against linear programming problem ever since it was raised. The idea of simplex is to jump from one feasible solution to another, continuously decreasing the objective value until it reaches a global optima, which gives us the optimal feasible solution. Many genres of simplex method have been inspired, including primal classical method, primal revised method and dual simplex method. In this project, I tried to implement primal classical method and primal revised method. Besides, to better start the simplex method, I implemented the two-stage method. A 'Bland' method is also added here to avoid degenerancy.

In general, we wish to solve the following linear programming problem in standard form:

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \boldsymbol{c}^T \boldsymbol{x} \\ s.t. \quad & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geq \boldsymbol{0} \end{aligned} \tag{1}$$

where $\boldsymbol{A} \in \mathbb{R}^{m\times n}, \boldsymbol{x} \in \mathbb{R}^{n\times 1}, \boldsymbol{b} \in \mathbb{R}^{m\times 1}, \boldsymbol{c} \in \mathbb{R}^{n\times 1}$.

*School of Information Science and Technology, ShanghaiTech University, Shanghai, China. Email: yuzy@shanghaitech.edu.cn

2 Pseudocode

In this section, the detailed pseudocode of two-stage method, primal classical simplex method and primal revised simplex method will be presented.

One should know that both primal classical simplex and primal revised simplex are carried out following the initialization of the first stage of 2-stage method.

The time complexity and space complexity is analysed here. Since classical method and the first stage of 2-stage method are really similar, I will only discuss classical method here.

1. Time complexity.

    - For each iteration of classical method, we do $O(m)$ times vector-scalar multiplication, which should cost $O(mn)$.
    - For each iteration of revised method, we do 4 matrix-vector multiplication and 1 matrix-matrix multiplication, although this may seems slow, however, with proper acceleration, this should not cost too much(Since many of the multiplications are quite simple).

2. Space complexity.

    - For each iteration of classical method, we need to update of $\boldsymbol{A}$, $\boldsymbol{b}$ and reduced cost all the time. Besides, we need $O(\max\{m,n\})$ space to store some temporal vectors. This costs $O(mn)$.
    - For each iteration of revised method, we only need to keep track of the inverse of base matrix $\boldsymbol{B}^{-1}$ and some other vectors. This costs $O(n^2)$.

Algorithm 1: 2 stage method – stage I

input : the constraint matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$, the resource constraint vector $\boldsymbol{b} \in \mathbb{R}^{m\times 1}$
output a basic feasible solution $\boldsymbol{x}_B$, inverse of base matrix $\boldsymbol{B}^{-1}$, # of pivots *num*
:

make copy of inputs in case of data pollution;
(`// check whether the primal problem has a feasible solution`);
add n auxiliary variables to $\boldsymbol{A}$*, recorded as base variables*;
calculate reduced cost;
for $i \leftarrow 0$ **to** max iteration **do**
    if reduced cost ≥ 0 **then**
        break
    end
    $MinColumnIndex \leftarrow \arg_j \min_j$ *reduced cost[j]*;
    if $\boldsymbol{A}[:, MinColumnIndex] \leq 0$ **then**
        problem has no boundary
    end
    $MinRowIndex \leftarrow$
     $\arg_j \min_j \frac{\boldsymbol{b}[j]}{\boldsymbol{A}[j,MinColumnIndex]}, \quad \boldsymbol{A}[j, MinColumnIndex] > 0;$
    normalize $\boldsymbol{A}[MinRowIndex, :]$ *and* $\boldsymbol{b}[MinRowIndex]$;
    for $rows \leftarrow 1$ **to** m **do**
        if $rows \neq MinRowIndex$ **then**
            do Gaussian elimination
        end
    end
    update reduced cost and optimal value accordingly;
    update base variables;
    $num \leftarrow num + 1$;
end
if optimal value $\neq 0$ **then**
    primal problem has no feasible solution
end
(`// remove redundant constraints and find inverse of base matrix`);
if auxiliary variables in base variables **then**
    Assume the i_{th} *base varible is auxiliary variable*;
    if $\boldsymbol{A}[i, :] = 0$ **then** *the constraint is redundant, can be removed*;
    else *continue pivoting*;
end
$\boldsymbol{x}_B \leftarrow \boldsymbol{b}$;
$\boldsymbol{B}^{-1} \leftarrow$*inverse of corresponding columns of base variables*;

Algorithm 2: primal classical simplex

input : the constraint matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$, the resource constraint vector $\boldsymbol{b} \in \mathbb{R}^{m\times 1}$ the cost vector $\boldsymbol{c} \in \mathbb{R}^{n\times 1}$

output : optimal solution $\boldsymbol{x}_O$, optimal objective value val, # of pivots num

make copy of inputs in case of data pollution;
reduced cost $\leftarrow \boldsymbol{c}$;
update reduced cost;
for $i \leftarrow 0$ **to** max iteration **do**
    if reduced cost ≥ 0 **then**
        break
    end
    (// Use Bland method to choose variables entering and leaving the base);
    $MinColumnIndex \leftarrow \min j, \quad reduced\ cost[j] < 0;$
    if $\boldsymbol{A}[:, MinColumnIndex] \leq 0$ **then**
        problem has no boundary
    end
    $MinRowIndex \leftarrow \arg_j \min_j \frac{\boldsymbol{b}[j]}{\boldsymbol{A}[j, MinColumnIndex]}, \quad \boldsymbol{A}[j, MinColumnIndex] > 0;$
    normalize $\boldsymbol{A}[MinRowIndex, :]$ *and* $\boldsymbol{b}[MinRowIndex]$;
    for $rows \leftarrow 1$ **to** m **do**
        if $rows \neq MinRowIndex$ **then**
            do Gaussian elimination
        end
    end
    update reduced cost and optimal value accordingly;
    update base variables;
    $num \leftarrow num + 1$;
end
$\boldsymbol{x}_O \leftarrow \boldsymbol{b}$ *arranged according to base variables*;
$val \leftarrow -$ *optimal value*

Algorithm 3: primal revised simplex

input : the constraint matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$, the resource constraint vector $\boldsymbol{b} \in \mathbb{R}^{m\times 1}$ the cost vector $\boldsymbol{c} \in \mathbb{R}^{n\times 1}$, inverse of base matrix $\boldsymbol{B}^{-1}$, base variables

output optimal solution $\boldsymbol{x}_O$, optimal objective value val, # of pivots num

:

make copy of inputs in case of data pollution;
get the list of non-base variables according to base variables;
for $i \leftarrow 0$ **to** max iteration **do**
  $\widetilde{\boldsymbol{b}} \leftarrow \boldsymbol{B}^{-1}\boldsymbol{b}$;
  $\boldsymbol{\lambda}^T = \boldsymbol{c}_B^T\boldsymbol{B}^{-1}$;
  $\boldsymbol{r}_N^T = \boldsymbol{c}_N^T - \boldsymbol{\lambda}^T\boldsymbol{N}$;
  if $\boldsymbol{r}_N \geq 0$ **then**
    break
  end
  (`// Use Bland method to choose variables entering and leaving the base`);
  $q \leftarrow \min j, \quad \boldsymbol{r}_N[j] < 0$;
  $\boldsymbol{y}_q = \boldsymbol{B}^{-1}\boldsymbol{A}[:, q]$;
  if $\boldsymbol{y}_q \leq 0$ **then**
    problem has no boundary
  end
  $p \leftarrow \arg_j \min_j \frac{\widetilde{\boldsymbol{b}}[j]}{\boldsymbol{y}_q[j]}, \quad \boldsymbol{y}_q[j] > 0$;
  $\boldsymbol{B}^{-1} \leftarrow \boldsymbol{E}_{pq}\boldsymbol{B}^{-1}$(`// Check slides for more details about` $\boldsymbol{E}_{pq}$);
  update optimal value;
  update base variables and non-base variables;
  $num \leftarrow num + 1$;
end
$\boldsymbol{x}_O \leftarrow \widetilde{\boldsymbol{b}}$ *arranged according to base variables*;
$val \leftarrow -$ *optimal value*

3 Numerical Results

In this section, a few numerical results is presented to show the performance of classical simplex and revised simplex. However, due to the restriction of test cases and my poor implementation, these results may not be accurate.

The first experiment, which is illustrated in fig(1), is carried out to show the performance of classical and revised simplex under different dimensions. Here, the dimension refers to the maximum of constraints and variables, i.e. $\max(m,n)$. Both simplex methods are initialized following the first stage of 2-stage method. They both employed Bland method to avoid degenerancy here. The 3 test cases are 'test5', 'test2' and 'test1' from 'data' sequently, where 'test5' is a degenerate problem. The experiment is carried 1000 times for all test cases using both methods, and we want to see how many time they takes. We can refer from the figure that the revised method, in general, are better performed than classical method, despite the falling behind in that tricky case(the degenerate one, maybe another pivoting strategy will do better).

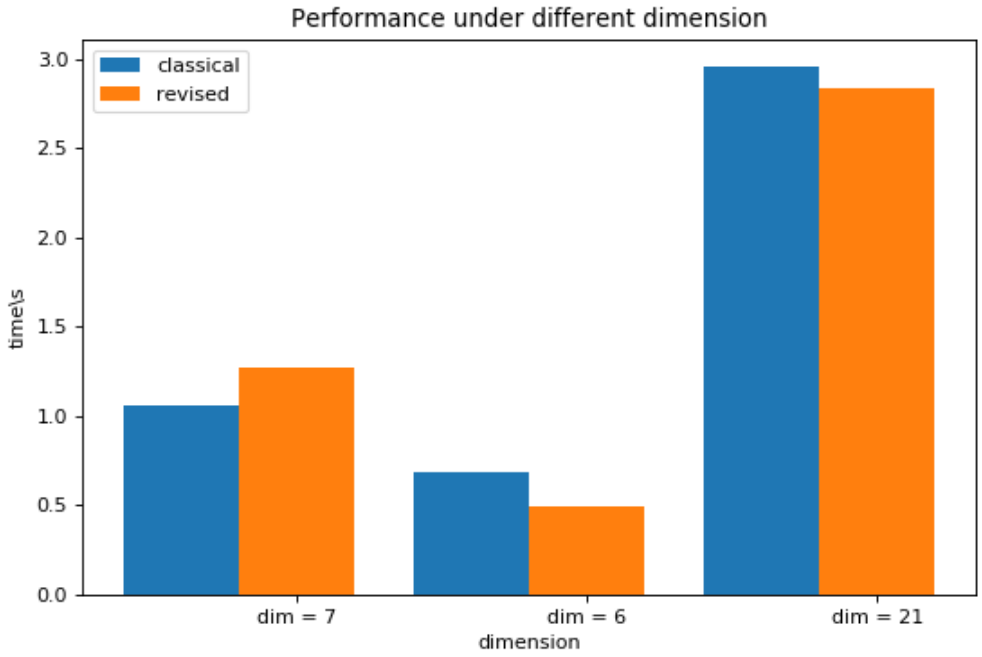

FIGURE 1: Comparison of performance under different dimensions

The following table Table(**??**) illustrated the pivots taken in different test cases using different method. Here, the figures are the pivots needed in that method. The 'i' in *2-stage Method-Stage I* is the first part of this stage: check whether primal method has a feasible solution. And 'ii' is the process removing auxiliary variables from base variables. Here, *Test1*,*Test2*,*Test3* are the given test cases. *Test5* is a degenerate problem, where *None* suggests that the default pivoting strategy failed here, while Bland method showed its

TABLE 1: num of pivots v.s. different methods

		2-stage Method-Stage I		Classical Simplex	Revised Simplex
		i	ii		
Test1	Bland	17	0	8	8
	default	17	0	6	6
Test2	Bland	3	0	0	0
	default	3	0	0	0
Test3	Bland	4	0	3	3
	default	4	0	3	3
Test5	Bland	3	0	7	7
	default	3	0	None	None

superiority. We can conclude from the table that simplex is a stable algorithm. Besides, Bland method may perform worse in some cases(*Test1*), but it outperforms default pivoting in degenerate problems.

Fig(2),Fig(3),Fig(4) are the results of the given test cases.

```
Auxiliary problem: part I -- ends!
Auxiliary problem: part I -- optimal val: 0.0
Auxiliary problem: part I -- iteration count: 17
Auxiliary var in base var: the 9th base var is the 9th aux var
The 9th constraint is redundant, removed!
Auxiliary problem: part II -- iteration count: 0
=========================================================
Classical simplex -- ends!
Classical simplex -- optimal val: -196200.0
Classical simplex -- iteration count: 6
=========================================================
Total classical iteration : 23
Total classical time : 0.01088286799999999
-----------------------------------------------------------
Revised simplex -- ends!
Revised simplex -- optimal val: 196200.0
Revised simplex -- iteration count: 6
=========================================================
Total revised iteration : 23
Total revised time : 0.011704199999999998
-----------------------------------------------------------
=========================================================
Classical calculated solution: [   0.    0.    0.    0.  300. 1100.    0.    0. 1200.  600.  400.    0.
    0.  400.  900.    0.    0.    0. 1400.    0.  600.]
Classical calculated optimal: 196200.0
Revised calculated solution: [   0.    0.    0.    0.  300. 1100.    0.    0. 1200.  600.  400.    0.
    0.  400.  900.    0.    0.    0. 1400.    0.  600.]
Revised calculated optimal: 196200.0
.........................................
True solution: [   0.    0.    0.    0.    0. 1100.  300.    0. 1200.  600.  400.    0.
    0.  400.  900.    0.    0.    0. 1700.    0.  300.]
True optimal: 196200.0
```

FIGURE 2: output of test1

```
Auxiliary problem: part I -- ends!
Auxiliary problem: part I -- optimal val: 4.440892098500626e-16
Auxiliary problem: part I -- iteration count: 3
Auxiliary problem: part II -- iteration count: 0
=========================================================
Classical simplex -- ends!
Classical simplex -- optimal val: -3.0000000000000004
Classical simplex -- iteration count: 0
=========================================================
Total classical iteration : 3
Total classical time : 0.0007318960000000096
-----------------------------------------------------------
Revised simplex -- ends!
Revised simplex -- optimal val: 3.000000000000001
Revised simplex -- iteration count: 0
=========================================================
Total revised iteration : 3
Total revised time : 0.0008061530000000316
-----------------------------------------------------------
=========================================================
Classical calculated solution: [1.         1.66666667 1.33333333 0.         0.         0.        ]
Classical calculated optimal: 3.0000000000000004
Revised calculated solution: [1.         1.66666667 1.33333333 0.         0.         0.        ]
Revised calculated optimal: 3.000000000000001
.........................................
True solution: [1.         1.66666667 1.33333333 0.         0.         0.        ]
True optimal: 3.000000002000001
```

FIGURE 3: output of test2

```
Auxiliary problem: part I -- ends!
Auxiliary problem: part I -- optimal val: 0.0
Auxiliary problem: part I -- iteration count: 4
Auxiliary problem: part II -- iteration count: 0
=========================================================
Classical simplex -- ends!
Classical simplex -- optimal val: 10000.0
Classical simplex -- iteration count: 3
=========================================================
Total classical iteration : 7
Total classical time : 0.0009481029999999779
-----------------------------------------------------------
Revised simplex -- ends!
Revised simplex -- optimal val: -10000.0
Revised simplex -- iteration count: 3
=========================================================
Total revised iteration : 7
Total revised time : 0.0012012299999999976
-----------------------------------------------------------
=========================================================
Classical calculated solution: [    0.     0. 10000.     0.     0.     0.]
Classical calculated optimal: -10000.0
Revised calculated solution: [    0.     0. 10000.     0.     0.     0.]
Revised calculated optimal: -10000.0
.........................................
True solution: [    0.     0. 10000.     0.     0.     0.]
True optimal: -10000.0
```

FIGURE 4: output of test3

4 Implementation details

In this section, I will introduce some of my implementation details for further discussion.

- Precision is important in computer computation. Since we are manipulating floating points in computer, error is unavoidable. Simplex requires so many comparisons that precision is essential here. For example, when I needed to decide whether a number is smaller than 0, I did not just simply compare them. Instead, I check whether it is distantly away from 0 to avoid cases when it is $-1e-11$, which is really close to 0. Through out this implementation, I set the precision tolerance to be $1e-10$, which may fail in larger scaled cases.
- Keeping track of base variables and non-base variables are both tricky and important. In revised simplex, we need to be aware of which variables are base variables while others are not.
- I used several python packages in this project code. Including
  - numpy
  - copy
  - time
  - re
  - matplotlib

5 Guidance on checking code

In this section, I will introduce how to check my codes.

1. codes.
   - *auxsol.py*. The codes of the first stage of 2-stage method. Do not have to run it.
   - *classicalSim.py*. The codes of primal classical simplex method. Do not have to run it.
   - *revisedSim.py*. The codes of primal revised simplex method. Do not have to run it.
   - *main.py*. Run it to see the printed result. Change the parameter of it to see differnt results in *parameter.txt*. Available parameters including:
     - *path*. Where you read the raw data.
     - *pivotMode*. Use *default* to employ default pivoting strategy; use *Bland* to employ Bland method.
     - *cleanResult*. Use *True* to set the unused variables to zero if it does not appear in objective function. Otherwise *False*.

- *method*. Use *classical* to only employ classical method. Similar for *revised*. Use *both* to employ both methods simultaneously.

After running *main.py*, you should see results similar to fig(2). For more details, please refer to *README.md*.

2. data. For each test case, there should be at least 4 pieces of data files, including:
   - *A.csv*
   - *b.csv*
   - *c.csv*
   - *x_star.csv*
3. result. After running *main.py*, the result should be printed in *output*. You can also find the result in Folder *result* where it stores the optimal solution, optimal objective value, number of pivots and running time for both classical and revised method respectively.