MIE 1620H – Linear Programming and Network Flows

Instructor: Dr. Roy Kwon

Submission Date: December 17, 2023

MIE 1620 Computational Project

Prepared By: Omar Al-Hilawani

Mechanical and Industrial Engineering

Student 1008735978

Prepared By: Bader Al-Hilawani

Mechanical and Industrial Engineering

Student 1009724498

Table of Contents

1.0 Executive Summary

This report will display the objective function values and total run times of the coded simplex method and interior-point, and linprog simplex method and interior-point. It was discovered that the coded simplex method was much quicker at obtaining the objective function values for various scenarios when compared to the coded interior-point method. The opposite was true when linprog was utilized. The objective function values at certain scenarios in conjunction with specific percentage increments caused the linprog simplex to output incorrect objective function values. Additionally, it was found that when obtaining the objective function values at certain scenarios was not successful due to the code crashing, decreasing the percentage increment by singular zero was sufficient to resolve this issue. After numerous runs using different scenarios and percentage increments, it was concluded that the coded simplex method outputted correct results when the linprog simplex method started to deviate, linprog interior-point was the fastest method at obtaining the objective function values with the exception of scenarios 1 and 3 using percentage increments of 0.01 where the coded interior-point was faster.

2.0 Introduction

This project entails solving the farming stochastic programming problem problem using the following three methods: 1) The simplex method as developed in class, 2) Primal affine scaling algorithm developed in class, and 3) Industrial strength simplex solver. This project was coded using the coding language python. Each method was coded from scratch utilizing the farming problem as the basis to ensure the written code outputs the correct values.

3.0 Background & Challenges

The two-phase method is utilized when it is difficult to identify a sub-matrix in the constraint Ax = b. Artificial variables are added to create an identity sub-matrix. The main idea to create this auxiliary linear program where we try to minimize some artificial variables to drive them to zero. The relationship between a standard form of a linear program (SFLP) and an artificial form of a linear program (ALP) is as follows: 1) If the SFLP is feasible, then ALP has a feasible solution with $x_a = 0$ and 2) If ALP has a feasible solution with $x_a = 0$, then SFLP has a feasible solution. See equations below pertaining to both versions.

Standard Form of a Linear Program (SFLP) –

$$minimize\ c^T x\ subject\ to\ Ax = b\ x\ \geq 0$$

Artificial Form of a Linear Program (ALP) –

$$minimize\ c^T x\ subject\ to\ Ax + \ x_a = b\ x\ \geq 0$$

The main objective of performing the phase 1 is to minimize the sun of the artificial variables shown in vector form $1^T x_a$. $1^T x_a$ is the summation of all artificial variables. We are after x_a equaling zero and to be non-basic at optimal solution. If x_a doesn't equal zero, then the original linear program is infeasible.

Phase 1 -

$$minimize\ 1^T x\ subject\ to\ Ax + \ x_a = b\ x\ \geq 0, x_a \geq 0$$

Once phase 1 has been solved and its constraints have been satisfied, we would proceed to phase 2. Phase 2 consists of utilizing the basic feasible solution obtained in phase 1 as the identity sub-matrix. The steps that will be undertaken to solve phase 2 method are similar to the simplex method. The simplex method, developed by Dantzig back in 1948, is a well-known and famous linear programming algorithm that is currently being utilized by commercial software worldwide to solve. For the simplex method to take shape, it first requires an initial basic feasible solution (BFS). If it is optimal, then the program terminates, else, another BFS is generated which will have an objective function value that is equal if not better than the previous. This cycle keeps on going until an optimal solution is obtained. Below is the list of steps that occur when the program is running.

Step 0:

First generate an initial basic feasible solution and all other variables.

$$x^{(0)} = [x_B|\ x_N\ ]^T$$

$$c = [c_B|\ c_N\ ]^T$$

$$B\ is\ your\ basis\ matrix\ and\ N\ is\ your\ non-basis\ matrix$$

Step 1:

Perform your optimality check by computing the reduced cost for all non-basis variables.

$$r_q = c_q - c_B^T B^{-1} N_q \text{ for all } q \epsilon N$$

$$If\ r_q \geq 0\ then\ x^{(k)} is\ optimal\ so\ you\ stop. Else, select\ one\ x_q\ from\ non-basis\ such\ that\ r_q \leq 0\ and\ proceed\ to\ step\ 2.$$

Step 2:

Compute the direction vector related to the x_q that was selected in step 1.

$$d^q = \left[-B^{-1}N_q \; e_q\right]$$

$$If\ d \geq 0, then\ the\ LP\ is\ unbounded, else\ proceed\ to\ step\ 3.$$

Step 3:

Compute the step length for all negative direction vectors.

$$\alpha = \left\{-\frac{x_j^{current}}{d_j^q} | d_j^q < 0\right\}$$

Pick the smallest alpha should there be more than one.

Step 4:

Compute the improved adjacent basic feasible solution.

$$x^{(k+1)} = x^{(k)} + \alpha d^q$$

Step5:

Update your basis and non-basis matrix x_B and x_N.

Now go back to step 1 and compute the reduced cost. If $r_q \geq 0$ then the problem is optimal, else, go through the entire cycle again.

The primal affine scaling algorithm is one of many different interior point algorithms. A point is in the interior feasible region if it satisfies the blow:

$$Ax = b \;\&\; x > 0$$

Every single component in any feasible point on the inside of a feasible set when an LP is in standard form must be positive. The feasible direction is a direction that allows it to take small movement while stating to be interior feasible. To maintain feasibility, you must equal to b as shown below.

$$\underline{Ax^{k+1}} = b\ Ax^k + \alpha Ad_x^k = b\ \rightarrow Ad_x^k = 0$$

The second equation equals b because x^k is feasible which means alpha (α) is non-negative indicating Ad_x^k must equal to zero. What defines a good direction is that it lead to a new solution with a lower objective value.

$$\underline{c^T x^{k+1}} \leq c^T x^k\ c^T x^k + \alpha c^T d_x^k \leq c^T x^k\ \rightarrow c^T d_x^k \leq 0$$

The direction can be improved by finding a direction d_x^k where d_x^k is in the null space and $c^T x^k$ is non-positive. The negative of the gradient is considered as the direction to follow because that is the direction of the most rapid decrease for $c^T x$. We are unable to go beyond the feasible set as it will result in infeasibility. We can ensure feasibility by projecting the negative of the gradient to be on the null space of matrix A. After the negative gradient vector -c is projected on the null space, that new vector is going to be the new direction. Every iteration of the feasible region will be rescaled so that the current interior point will be in the middle of the rescaled feasible set. Rescaling is being performed to ensure that the current point is always in the center of the feasible domain. This provides an added benefit where we are able to take a big slide down every time without having to worry about being near a wall.

$Min \quad c^T x\ Subject\ to\ \ Ax = b\ x \geq 0$

$Min \quad c^T X_k y\ Subject\ to\ \ AX_k y = b\ y \geq 0$

Below is the list of steps that occur when the program is running.

Step 1:

Extract the relevant information to set up the A, b, c, and X_0 matrixes.

Step 2:

Compute the dual estimate (w^k) and reduced cost (r^k). If $r^k \geq 0$, and $e^T X_k r^k \leq \epsilon$ then w^k is dual feasible.

$$w^k = (AX_k^2A^T)^{-1}AX_k^2 - Dual\ Estimate$$

$$r^k = c - A^T w^k - Reduced\ Cost$$

Otherwise, proceed to step 3.

Step 3:

Compute the direction vector. If $d_y^k \geq 0$ then you are unbounded. If If $d_y^k = 0$, then you stop otherwise you proceed to step 4 and compute step length.

$$d_y^k = -X_k r^k$$

Step 4:

Compute the alpha so you can compute the new x^{k+1}. Then proceed to computing step 2 again to see if you are at optimal or if you need to go through the entire cycle again.

$$\alpha_k = min_i \left\{ \frac{\alpha}{-(d_y^k)_i} | (d_y^k)_i < 0 \right\} - Step\ Length$$

$$x^{k+1} = x^k + \alpha_k X_k d_y^k - New\ Point$$

4.0 Problem Statement

Solve the farming stochastic programming model using 1) the simplex method developed in class, 2) Primal affine scaling algorithm developed in class, and 3) an industrial strength simplex solver for various scenarios.

5.0 Objective & Tasks

The main objective is to solve the farming stochastic programming model for various scenarios using the three previously mentioned methods. The information provided below will be used as the foundation for this problem.

Table 1: Initial Data

	WHEAT	**CORN**	**SUGAR BEETS**
Average Yield (T/acre)	2.5	3	20
Planting Cost ($/acre)	150	230	260
Selling Price ($/T)	170	150	36 under 6000 T
			10 under 6000 T
Minimum Requirement (T)	200	240	-
Purchase Price ($/T)	238	210	-
Total available land: 500 acres			

Table 2: Variables

x_1	Acres of land devoted to wheat
x_2	Acres of land devoted to corn
x_3	Acres of land devoted to sugar beets
w_1	Tons of wheat sold
w_2	Tons of corn sold
w_3	Tons of sugar beets sold at favorable price
w_4	Tons of sugar beets sold at lower price
y_1	Tons of wheat purchased
y_2	Tons of corn purchased

Initial Linear Programming Problem -

$$minimize \quad 150x_1 + 230x_2 + 260x_3 + 238y_1 + 210y_2 - 170w_1 - 150w_2 - 36w_3 - 10w_4 \; subject\ to \quad x_1 + x_2 + x_3 \leq 500 \quad x_{1,2,3} \geq 0 \; 2.5x_1 + y_1 - w_1 \geq 200 \; 3x_2 + y_2 - w_2 \geq 240 \; w_3 + w_4 \leq 20x_3 \; w_3 \leq 6000 \; w_{1,2,3,4} \geq 0, \; y_{1,2,3} \geq 0$$

Table 3: Optimal Solution based on above average yields (+20%) (Scenario 1).

CULTURE	WHEAT	CORN	SUGAR BEETS
Surface (acres)	183.33	66.7	250
Yield (T)	550	240	6000
Sales (T)	350	-	6000
Purchase (T)	-	-	-
Overall Profit: $167,667			

Table 4: Optimal Solution based on expected yields (Scenario 2).

CULTURE	WHEAT	CORN	SUGAR BEETS
Surface (acres)	120	80	300
Yield (T)	300	240	6000
Sales (T)	100	-	6000
Purchase (T)	-	-	-
Overall Profit: $118,600			

Table 5: Optimal Solution based on below average yields (-20%) (Scenario 3).

CULTURE	WHEAT	CORN	SUGAR BEETS
Surface (acres)	100	25	375
Yield (T)	200	60	6000
Sales (T)	-	-	6000
Purchase (T)	-	180	-
Overall Profit: $59,950			

Revised Linear Programming Problem with all three scenarios incorporated.

$$
\begin{aligned}
\textit{minimize} \quad & 150x_1 + 230x_2 + 260x_3 \\
& + \frac{1}{3}(238y_{11} + 210y_{21} - 170w_{11} - 150w_{21} - 36w_{31} - 10w_{41}) \\
& + \frac{1}{3}(238y_{12} + 210y_{22} - 170w_{12} - 150w_{22} - 36w_{32} - 10w_{42}) \\
& + \frac{1}{3}(238y_{13} + 210y_{23} - 170w_{13} - 150w_{23} - 36w_{33} - 10w_{43})
\end{aligned}
$$

Scenario 1

Scenario 2

Scenario 3

$$
\begin{aligned}
\textit{subject to} \quad & x_1 + x_2 \\
& + x_3 \ 500 \qquad x_{1,2,3} \geq 0 \ 2.5x_1 + y_{11} - w_{11} \\
& \geq 200 \ 3x_2 + y_{21} - w_{21} \geq 240 \ w_{31} + w_{41} \leq 20x_3 \ w_{31} \leq 6000 \ w_{11,21,31,41} \geq 0, \\
& y_{11,21,31} \geq 0 \ 2.5x_1 + y_{12} - w_{12} \geq 200 \ 3x_2 + y_{22} - w_{22} \geq 240 \ w_{32} + w_{42} \leq 20x_3 \ w_{32} \\
& \leq 6000 \ w_{12,22,32,42} \geq 0, \\
& y_{12,22,32} \geq 0 \ 2.5x_1 + y_{13} - w_{13} \geq 200 \ 3x_2 + y_{23} - w_{23} \geq 240 \ w_{33} + w_{43} \leq 20x_3 \ w_{32} \\
& \leq 6000 \ w_{13,23,33,43} \geq 0, \qquad y_{13,23,33} \geq 0
\end{aligned}
$$

Table 6: Optimal Solution for all three scenarios combined.

		WHEAT	CORN	SUGAR BEETS
First Stage	Area (acres)	170	80	250
S = 1 Above	Yield Sales Purchase	510 310 -	288 48 -	6000 6000 -
S = 2 Average	Yield Sales Purchase	425 225 -	240 - -	5000 5000 -
S = 3 Below	Yield Sales Purchase	340 140 -	192 - 48	4000 4000 -
Overall Profit: $108,390				

6.0 Approach

The linear programming problem will initially be solved via two distinct methods. The first method will incorporate the use of the big M method. The second method will obtain the optimal solution via the affine scaling algorithm.

6.1 Big M method

Below is a written description of the steps detailing how the code was written. The main code along with the comments can be found under appendix 1.

We first obtain the following variables: B matrix, c_v, and π from the original linear programming problem. Then the reduced cost (r) is computed for all non-basic variables. Each answer is checked to see if it is non-negative. If at least one of the reduced costs is not non-negative, then the direction vector is calculated. The vectors are then sorted in order. If none of the direction vectors are negative, then the direction vector is unbounded. If there are negative direction vectors, then step length for each negative direction is calculated and smallest one is chosen. Finally, the new x^{k+1} is calculated after which you update the new x indices and perform the entire cycle again. Once the reduced cost is non-negative, the new x^{k+1} that was computed in the previous iteration is your optimal basic feasible solution.

6.2 Primal Affine Scaling Method

Below is a written description of the steps detailing how the code was written. The main code along with the comments can be found under appendix 2.

From the provided linear programming problem, we obtain the A matrix, b vector, and c vector. The original diagonal vector will consist of all ones as we are starting from at the center of the non-negative/first orthant. The X_k matrix will then be computed by placing the diagonal vector into a scaling matrix and everything else off the diagonal is zeros. The dual estimate (w^k), reduced cost (r^k), and duality gap (ε) were computed. We then check for optimality by ensuring that the reduced cost is non-negative, and the duality gap is less than or equal to $e^T X_k r^k$. If either is violated, the negative projected gradient of the scaled linear program $\left(direction - \ d_y^k\right)$ must then be computed and checked to see if it is greater than 0 or if it is equal to zero. If neither is violated, the step length (α) is calculated for all negative d_y^k. The lowest alpha will be obtained. Finally, the new x^{k+1} is calculated after which the x variable is updated to included the new x indices and perform the entire cycle again. Once the reduced cost is non-negative

and the duality gap is less than or equal to $e^T X_k r^k$, the solution is optimal, i.e., the new x^{k+1} that was computed in the previous iteration is the optimal basic feasible solution.

7.0 Results, Analysis, & Validation

7.1 Comparing the objective function values for various number of scenarios using different solving methods.

During the analysis part for all four solving methods (simplex method, linprog simplex method, interior point, and linprog interior-point), a trend was noticed where the objective function value for the simplex method, interior point, and linprog-simplex method stayed consistent for scenarios between 25 to 151. As the number of scenarios kept increasing, the objective function value for the linprog simplex method started to deviate. As shown in Figure 1 below, the objective function value started to deviate from scenarios 175 to 325 and the difference grew larger as the number of scenarios increased. To determine which solving method was correct (the coded simplex method and interior point or the industrial simplex method solver) a fourth solving method was used which is the linprog interior-point. After implementing the linprog interior-point solving method, the objective function value for scenarios ranging from 25 to 325 were the same as the hand built simplex method and interior point code. Therefore, using the linprog-simplex method for scenarios larger than 151 will result in an incorrect answer.

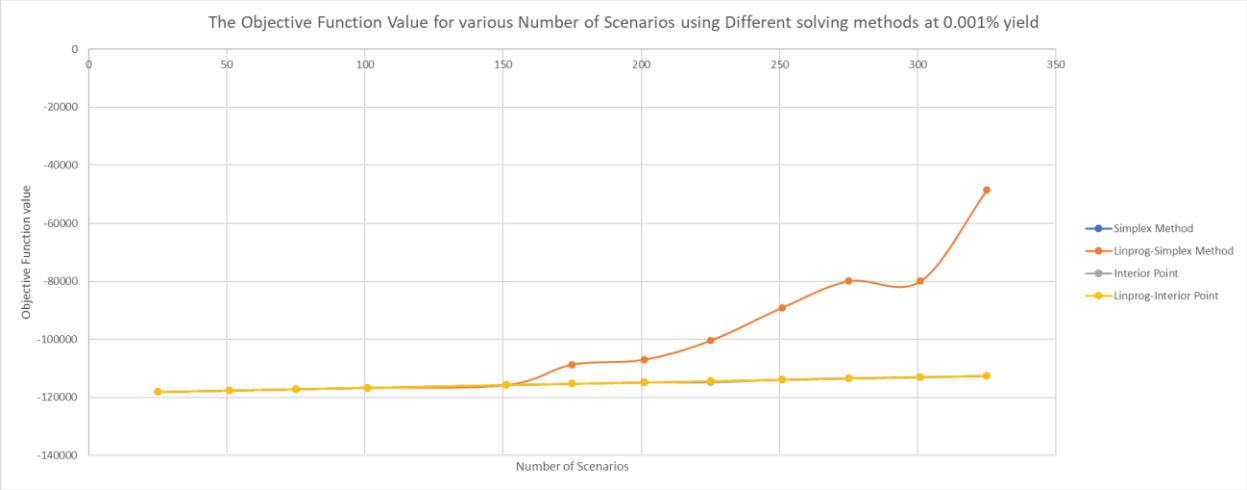

Figure 1: Objective function value for various number of scenarios ranging from 25-375 using the simplex method, linprog-simplex method, interior point, and linprog-interior point at a percentage yield of 0.001.

Furthermore, the same analysis was conducted again, however, with a different percent yield (0.001%) and using a different number of scenarios (1-152). The deviation can also be seen where the linprog simplex method outputs a different objective function value. As shown in Figure 2 below, the objective function value started to deviate from scenarios 120 to 152 and the difference grew larger as the number of scenarios increased. The linprog interior point was also graphed in comparison and the objective function value matched the simplex method and interior point method. Therefore, it can be seen that using a different percent yield and different number of scenarios resulted in objective function value of the linprog to also deviate.

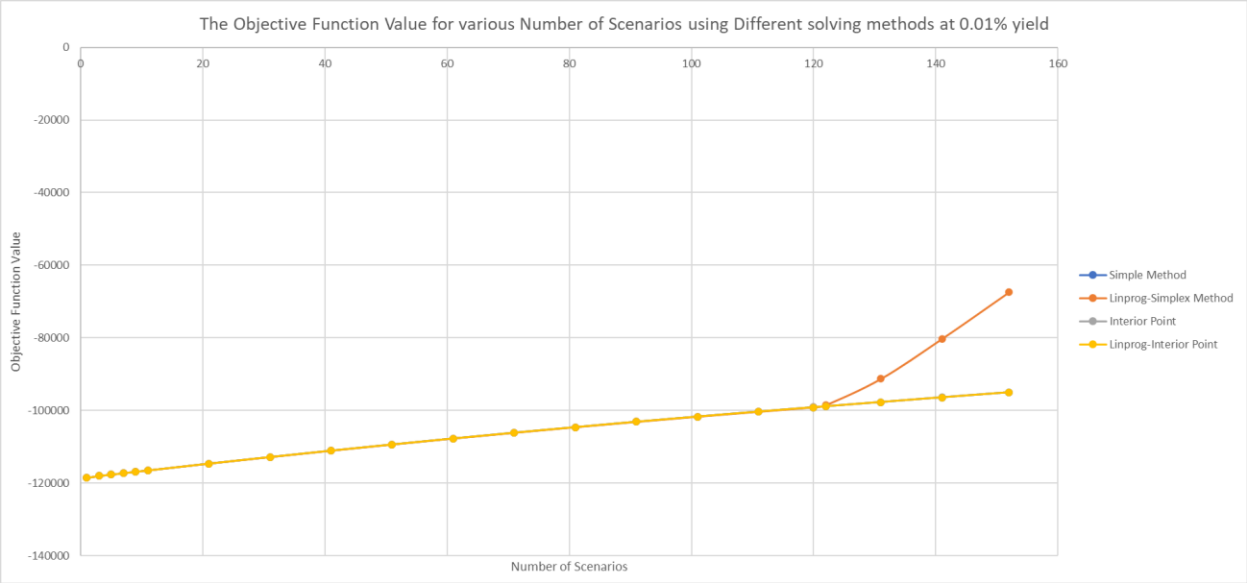

Figure 2: Objective function value for various number of scenarios ranging from 1-152 using the simplex method, linprog-simplex method, interior point, and linprog-interior point at a percentage yield of 0.001

Moreover, it can be seen that utilizing a different percent yield will cause the linprog simplex method to deviate at a different scenario number. Hence, when a larger percent yield was used (0.01%) the linprog simplex started to deviate at scenario 120 when compared to using a smaller percent yield (0.001%) the linprog simplex started to deviate at scenario 175.

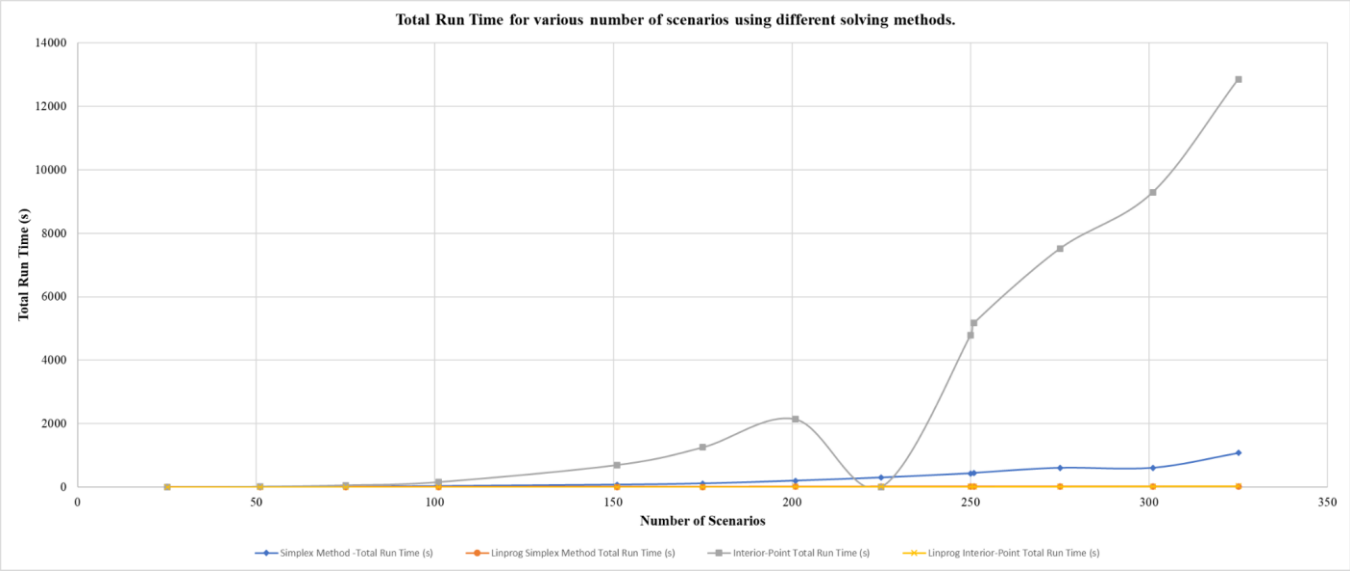

Figure 3: Total Run time in seconds for all four methods using 0.001 percentage yield.

The figure above displays the total run time for all four methods used to compute the new objective function values at various number of scenarios at a percentage increment of 0.001. It's quite evident that as the number of scenarios increases, so does the total time for all methods. Utilizing the industrial solver linprog to compute the linear programming problem via the simplex method and interior-point method displays that it consistently has lower total times when compared to the coded simplex method and interior-point. The run time between all methods is not uniform. This is evident in both the coded simplex method and more strikingly in the coded interior point. As the number of scenarios increases, the execution time goes up considerably for the coded interior-point when compared to the coded simplex method which is increasing slowly. Another observation is that the linprog simplex method's run times are relatively consistent across the different scenarios. This indicates that when the linprog is used in conjunction with the simplex method, we can expect a predictable and stable performance. Scenario 325 stands out with a significant run time for the coded interior-point. When compared to the coded simplex method, it took an additional 2 hours and 21 minutes which is a 91% increase.

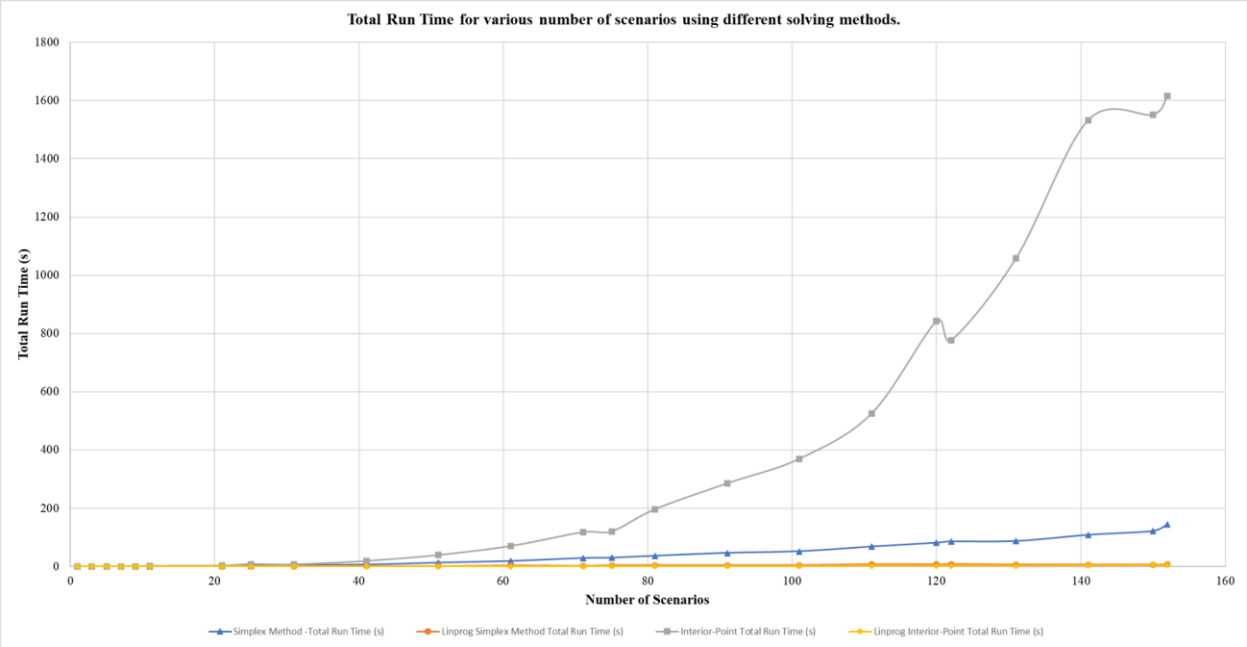

Figure 4: Total Run time in seconds for all four methods using 0.01 percentage yield.

From figure 4 shown above, it can be seen that as the number of scenarios increases, the total run time also increases. The coded simplex method and the linprog simplex method consistently exhibit lower run times for all the scenarios when compared to the coded and linprog interior-point method. This indicates that the simplex method (coded and linprog) is much more efficient at solving this problem. Another observation is that the linprog simplex method reliably outperforms the coded simplex which is expected as linprog is an industrial solver. The coded interior-point runtime increases non-linearly with the increase in the number of scenarios. The improvement gained by using linprog to solve the linear programming problem via the interior-point is more noticeable when implemented to solve via the simplex method. The erraticism in runtimes across all the different scenarios highlights the sensitivity of the optimization methods to the features of the specific problem. As the number of scenarios increases, the coded interior-point exhibits considerably longer runtimes which indicates possible challenges in converging to the optimal solution.

Table1: Percent Difference between the runtime of both the coded simplex method and the interior-point when solving for the objective function value using percent yield of 0.01.

Scenarios	Python Simplex Method Total Time (seconds)	Python Interior-point Total Time (seconds)	Time % Difference
25	2.460882	4.143407	-68.37%
51	7.055429	16.47559	-133.52%
75	13.93188	58.42358	-319.35%
101	31.20085	160.0323	-412.91%
151	74.12519	696.6313	-839.80%
175	112.8232	1251.649	-1009.39%
201	201.0178	2137.274	-963.23%
225	301.2379	3052.662	-913.37%
250	433.0572	4785.074	-1004.95%
251	446.7137	5178.465	-1059.24%

275	606.8464	7518.581	-1138.96%
301	841.0465	9294.201	-1005.08%
325	1072.902	12863.54	-1098.95%

The negative percentage values indicate that as the number of scenarios increases, the optimization problem becomes more complex to solve for the coded interior-point when compared to the coded simplex method. This indicates that the interior-point is more sensitive when the number of scenarios are increased.

Table2: Percent Difference between the runtime of both the coded simplex method and the interior-point when solving for the objective function value using percent yield of 0.01.

Scenarios	**Python Simplex Method Total Time (seconds)**	**Python Interior-point Total Time (seconds)**	**Time % Difference**
1	0.005027	0.114358	-2175.06%
3	0.023514	0.272485	-1058.84%
5	0.122975	0.296103	-140.78%
7	0.332339	0.210918	36.54%
9	0.273803	0.342926	-25.25%
11	0.616755	1.051664	-70.52%
21	1.815509	2.678546	-47.54%
25	2.153173	7.751768	-260.02%
31	4.816966	6.731457	-39.74%
41	5.97445	18.52435	-210.06%
51	12.82014	38.93869	-203.73%
61	18.40626	69.83818	-279.43%

71	28.47185	117.2973	-311.98%
75	30.0327	120.1924	-300.21%
81	36.51209	196.3785	-437.85%
91	46.28348	285.1984	-516.20%
101	52.05785	369.6723	-610.12%
111	68.06642	524.9748	-671.27%
120	81.47463	842.3157	-933.84%
122	85.70909	776.008	-805.40%
131	87.20236	1057.559	-1112.76%
141	108.3813	1532.45	-1313.94%
150	122.0187	1551.726	-1171.71%
152	145.1272	1616.114	-1013.58%

8.0 Risk Mitigation and Issues

The coded simplex method and interior-point were verified by the results provided by the farming linear programming problem. This was the quickest and most efficient method of ensuring that the code functions correctly. However, this does not mean that the code will be able to solve all different scenarios and percentages.

To determine which solving method was correct (the coded simplex and interior point or the linprog simplex) another solving method was used which was the linprog interior point to validate the results. This confirmed that the coded simplex and interior point outputted the correct objective function value compared to the linprog simplex method since the linprog interior point had the same objective function value as the coded simplex and interior point.

When utilizing a percent yield of 0.01% with scenarios starting from 201, the code would run into an error after a period of computing time. The error that would present is a singular matrix error. This error

occurs when trying to invert the B matrix which is a singular matrix thus unable to invert a singular matrix as the determinant is zero. This same error occurs when calculating scenarios 121, 151, 161, 171, and 191. In order to work around this issue, the percentage yield was increased from 0.01% to 0.001%. With a percent yield of 0.001% the objective function values were calculated for all scenarios.

Moreover, another issue encountered was coding the scenario generation. The approach used was determining if there was a pattern in the objective function, constraint matrix, and right-hand side (b) value for every scenario. A pattern was realized in which the variables were shifted to the right by 6 columns for every scenario generated therefore increasing the number of variables by 6 for every scenario generated. Additionally, only X1, X2, and X3 were repeated for every scenario (while having different coefficient depending on the percent yield) and the the first constraint was not repeated for every scenario generated. Therefore, in order to implement these requirements, if statements were used to state the changes required for every variable.

9.0 Future Improvements and Work

As discussed in the results section, the coded interior-point has an extremely long execution time when compared to the other three methods. This can be attributed to inefficient code and the approach that was taken. With additional time, the efficiency of the code can be improved by either improving the current code or by possibly coding a different approach.

Numerous more scenarios and percentages need to be tested to see where this code falls short. For instance, the code breaks after 200 scenarios utilizing percentage 0.01, but doesn't when utilizing 0.001 percentage. This can be attributed to the robustness of the simplex and interior point code. Due to the larger percent yield, at higher scenarios the code cannot handle large percent changes and the code fails while computing, however, when the percent yield was decreased the code was able to compute the objective function value at larger number of scenarios. The coded simplex and interior point could be improved to handle a larger percent yield for a larger number of scenarios.

Additionally, an improvement would be to utilize a different linprog simplex method to obtain results and compare them to the other three solving methods. This is due to the robustness of the linprog simplex method in which although the computing time is faster than the coded simplex method and interior point, it would output a different objective function value after a certain number of scenarios were generated. A message would be outputted when running the linprog simplex method which states the following "<ipython-input-51-e02b04dcbabb>:2: DeprecationWarning: `method='simplex'` is deprecated and will be removed in SciPy 1.11.0. Please use one of the HiGHS solvers (e.g. `method='highs'`) in new code. result = linprog(c, A_eq = A, b_eq = b, method = 'simplex')". Therefore, utilizing "highs" instead of "simplex" outputted the correct objective function value for all scenarios generated and therefore, no deviation was observed. Thus, the simplex method is not robust enough when considering large number of scenarios.

10.0 Conclusion

To conclude, all four solving methods (coded simplex, coded interior point, linprog simplex method and linprog interior point method) were coded and compared to each other using different metrics such as the objective function value for each solving method and computing time. While computing the objective function value for each method, a trend was noticed in which the linprog simplex method would diverge at scenario 175. In order to confirm which method was outputting the correct objective function value, another solving method was used which was the linprog interior point method. This confirmed that the coded simplex and interior point method were outputting the correct objective function value for all the scenarios generated while the objective function value generated by the linprog simplex began to deviate after a certain number of scenarios were generated. Furthermore, the percentage yield and number of scenario were changed from %0.001 to %0.01 and 25-325 scenarios to 1-152 scenarios and a new set of data for all the solving methods were obtained and compared. The same results were noticed in which the linprog simplex began to deviate, however, the deviation was at scenario 120. Therefore, depending on the percentage yield, the diversion will be at different scenarios generated. Additionally, the computing

time for all four solving methods differed as the number of scenarios increases, the efficiency of the coded interior-point deteriorates when compared to the coded simplex method. This is the reverse when utilizing the linprog solver.

11.0 References

[1] R. Kwon. (2023). MIE1620 Stochastic Programming (Farming) [PDF Document].

[2] R. Kwon. (2023). MIE1620 Lecture 3: Simplex Method [PDF Document].

[1] R. Kwon. (2023). MIE1620 Lecture 8: Affine Scaling Interior-point [PDF Document].

12.0 Appendix

For the code to function, kindly run each code cell in order.

Simplex Method:

```
import numpy as np

from itertools import count

from scipy.optimize import linprog

import time


def generate_constraint_matrix(num_scenarios,percentage): #generates the constraint matrix which takes into account the number of scenarios and the percentage of each scenario.

  constraint_matrices = [] #gets the value (from the function below) for the objective function and right hand side.

  constraints = [] #constrain value

  Slacks = [] #slack value

  variables_count = 3 + 6*num_scenarios #number variabes generated per scenario. 6 variables added per scenario while only X1,X2,X3 (the first 3 variables) stay the same.

  slacks_count = 1 + 4*num_scenarios #counts the amout of slacks that will be generated for every scenario. 1st contraint stays with the original scenario and only the next 4 constraints are repeated for every scenario

  c = [0] * (variables_count + slacks_count)

  b = [0] * (1 + 4*num_scenarios)


  for i in range(num_scenarios): #Generate constraints for each scenario

    for j in range(5):  # Number of constraints per scenario

      row = [0] * variables_count

      if i==0 and j == 0:

        row[0] = 1  # Coefficients for X1 in the first constraint

        row[1] = 1  # Coefficients for X2 in the first constraint

        row[2] = 1  # Coefficients for X3 in the first constraint

      elif i!=0 and j == 0:
```

```
        continue  # Skip the first constraint for subsequent matrices
      elif j == 1:
        row[0] = 2.5 * (1+percentage*((num_scenarios-1)/2)-percentage*i)   # Coefficients for X1 in the second constraint
        row[3+6*i] = 1  # Coefficients for X4 in the second constraint
        row[5+6*i] = -1  # Coefficients for X6 in the second constraint
      elif j == 2:
        row[1] = 3 * (1+percentage*((num_scenarios-1)/2)-percentage*i)  # Coefficients for X2 in the third constraint
        row[4+6*i] = 1  # Coefficients for X5 in the third constraint
        row[6+6*i] = -1  # Coefficients for X7 in the third constraint
      elif j == 3:
        row[2] = -20 * (1+percentage*((num_scenarios-1)/2)-percentage*i)  # Coefficients for X3 in the fourth constraint
        row[7+6*i] = 1  # Coefficients for X8 in the fourth constraint
        row[8+6*i] = 1  # Coefficients for X9 in the fourth constraint
      elif j == 4:
        row[7+6*i] = 1  # Coefficients for X8 in the fifth constraint

      constraints.append(row)

  for i in range(num_scenarios):
    # Generate constraints for each scenario
    for j in range(5):  # Number of constraints per scenario
      row = [0] * slacks_count
      if i==0 and j == 0:
        row[0] = 1  # Coefficients for s1 in the first constraint
      elif i!=0 and j == 0:
        continue  # Skip the first constraint for subsequent matrices
      elif j == 1:
        row[1+4*i] = -1  # Coefficients for s2 in the second constraint
```

```
    elif j == 2:
      row[2+4*i] = -1  # Coefficients for s3 in the third constraint
    elif j == 3:
      row[3+4*i] = 1  # Coefficients for s4 in the fourth constraint
    elif j == 4:
      row[4+4*i] = 1  # Coefficients for s5 in the fifth constraint


    Slacks.append(row)


  constraint_matrices = np.hstack((constraints,Slacks))# generates the matrix which consists of the objective function and Right hand side value


  c[0] = 150
  c[1] = 230
  c[2] = 260
  for i in range(num_scenarios):
    c[3 + (i*6)] = 238 * (1/num_scenarios)
    c[4 + (i*6)] = 210 * (1/num_scenarios)
    c[5 + (i*6)] = -170 * (1/num_scenarios)
    c[6 + (i*6)] = -150 * (1/num_scenarios)
    c[7 + (i*6)] = -36 * (1/num_scenarios)
    c[8 + (i*6)] = -10 * (1/num_scenarios)


  b[0] = 500
  for i in range(num_scenarios):
    b[1 + (i*4)] = 200
    b[2 + (i*4)] = 240
    b[3 + (i*4)] = 0
    b[4 + (i*4)] = 6000
```

```
  b = np.array(b).reshape(-1,1)

  return constraint_matrices, c, b

def generate_phase_one_lp_coefficients_from_constraints(A,b, c): #generates the matrix for the phase 1 of the simplex method.

  num_constraints, num_variables = A.shape
  x = np.ones((num_variables,1))
  artificial_matrix = np.eye(num_constraints)
  A_with_artificial = np.concatenate((A, artificial_matrix), axis=1)

  c_new = np.zeros(num_variables + num_constraints)
  c_new[:num_variables] = c
  c_new[num_variables:] = 1000000  # Coefficients for artificial variables

  coefficients = {
    'A': A_with_artificial,
    'b': b.reshape(-1,1),
    'c': c_new.reshape(-1,1)}

  return coefficients

num_scenarios = 89 #user can change the number of scenarios they would like to generate
percentage = 0.001 #user can chnage the percentage +/- for the number of scenarios generated.
A_pre,c_pre,b_pre = generate_constraint_matrix(num_scenarios,percentage)

beginning = time.time() #starts generating the time for the duration of the simplex method.
```

```
phase_one_coefficients = generate_phase_one_lp_coefficients_from_constraints(A_pre,b_pre,c_pre)

A = phase_one_coefficients['A']
b = phase_one_coefficients['b']
c = phase_one_coefficients['c']

normalX = A_pre.shape[1]
variableX = A.shape[1]
xB_Index = np.arange(normalX, variableX).reshape(1, variableX-normalX)
xN_Index = np.arange(0, normalX).reshape(1, A_pre.shape[1])

zero_X = np.zeros((A_pre.shape[1],1))
X = np.vstack((zero_X,b))

for i_count in count():
  B = A[:, xB_Index].reshape(A.shape[0],A.shape[0])
  cB = c[xB_Index, :].reshape(A.shape[0],1)
  w = np.linalg.solve(B.T, cB)

  reduced_cost_values = np.array([]) #generated the reduced cost values
  for i_N in xN_Index:
    cN_r = c[i_N,0]
    N_r = A[:, i_N]
    reduced_cost = cN_r - (w.T)@N_r
    reduced_cost_values = np.append(reduced_cost_values, reduced_cost)

  positive_r = 0 #if statement which exists the function if the reduced cost is greater than or = to 0.
Otherwise continue running the code.
  for i in reduced_cost_values:
    if i >= 0:
```

```
    positive_r = positive_r + 1
  if positive_r == reduced_cost_values.shape[0]:
    Optimal_X = X
    Optimal_value = c.T @ X
    break

  for i_r, r in enumerate(reduced_cost_values):
    if r < 0:
      enter_variable = xN_Index[0,i_r]
      break
  xB_direction = np.linalg.solve(B, -A[:, enter_variable].reshape(A.shape[0],1))# This is the direction vector in which the new variable enters
  e = np.array([])
  for num in range(xN_Index.shape[1]):
    if xN_Index[0,num] == enter_variable:
      e = np.append(e,1)
    else:
      e = np.append(e,0)
  direction = np.vstack((xB_direction, e.reshape(xN_Index.shape[1],1)))
  total_index = np.hstack((xB_Index,xN_Index))
  total_index_flat = total_index.flatten()
  sorting_index = np.argsort(total_index_flat)
  direction_sorted = direction[sorting_index]

  a = np.array([])
  a_index = np.array([])
  for i_d,d_value in enumerate(direction):
    if d_value < 0:
      a = np.append(a, -1*X[xB_Index[0,i_d],0]/d_value)
    else:
```

```
    a = np.append(a, 100000) #A reference alpha of 100,000 was used to compare the alpha values

  a_min = np.min(a)
  exit_variable = total_index[0,np.where(a == a_min)[0][0]]
  X_new = X + (a_min * direction_sorted)
  X = X_new
  xB_Index = np.append(xB_Index, enter_variable)
  mask = (xB_Index != exit_variable)
  xB_Index = xB_Index[mask]
  xB_Index = np.sort(xB_Index).reshape(1,xB_Index.shape[0])

  xN_Index = np.append(xN_Index, exit_variable)
  mask = (xN_Index != enter_variable)
  xN_Index = xN_Index[mask]
  xN_Index = np.sort(xN_Index).reshape(1,xN_Index.shape[0])

total_time = time.time() – beginning

Optimal_value

total_time

beginning2 = time.time()
result = linprog(c, A_eq = A, b_eq = b, method = 'simplex')
total_time2 = time.time() - beginning2


result.fun
total_time2
```

Interior-Point

```
#Importing libraries
import numpy as np
from itertools import count
from scipy.optimize import linprog #used to run the
import time

#Generating the constraint matrix for the number scenarios and the percentages
def generate_constraint_matrix(num_scenarios,percentage):
  #Initializing lists and counts
  constraint_matrices = []
  constraints = []
  Slacks = []
  variables_count = 3 + 6*num_scenarios
  slacks_count = 1 + 4*num_scenarios
  c = [0] * (variables_count + slacks_count)
  b = [0] * (1 + 4*num_scenarios)

#The code uses nested loops to generate the constraint matrices (constraints and Slacks), coefficient vector (c), and right-hand side vector (b).
  for i in range(num_scenarios):
    # Generate constraints for each scenario
    for j in range(5):  # Number of constraints per scenario
      row = [0] * variables_count
      if i==0 and j == 0:
        row[0] = 1  # Coefficients for X1 in the first constraint
        row[1] = 1  # Coefficients for X2 in the first constraint
```

```
        row[2] = 1  # Coefficients for X3 in the first constraint
      elif i!=0 and j == 0:
        continue  # Skip the first constraint for subsequent matrices
      elif j == 1:
        row[0] = 2.5 * (1+percentage*((num_scenarios-1)/2)-percentage*i)   # Coefficients for X1 in the second constraint
        row[3+6*i] = 1  # Coefficients for X4 in the second constraint
        row[5+6*i] = -1  # Coefficients for X6 in the second constraint
      elif j == 2:
        row[1] = 3 * (1+percentage*((num_scenarios-1)/2)-percentage*i)  # Coefficients for X2 in the third constraint
        row[4+6*i] = 1  # Coefficients for X5 in the third constraint
        row[6+6*i] = -1  # Coefficients for X7 in the third constraint
      elif j == 3:
        row[2] = -20 * (1+percentage*((num_scenarios-1)/2)-percentage*i)  # Coefficients for X3 in the fourth constraint
        row[7+6*i] = 1  # Coefficients for X8 in the fourth constraint
        row[8+6*i] = 1  # Coefficients for X9 in the fourth constraint
      elif j == 4:
        row[7+6*i] = 1  # Coefficients for X8 in the fifth constraint

      constraints.append(row)

  for i in range(num_scenarios):
    # Generate constraints for each scenario
    for j in range(5):  # Number of constraints per scenario
      row = [0] * slacks_count
      if i==0 and j == 0:
        row[0] = 1  # Coefficients for s1 in the first constraint
      elif i!=0 and j == 0:
```

```
        continue  # Skip the first constraint for subsequent matrices
    elif j == 1:
        row[1+4*i] = -1  # Coefficients for s2 in the second constraint
    elif j == 2:
        row[2+4*i] = -1  # Coefficients for s3 in the third constraint
    elif j == 3:
        row[3+4*i] = 1  # Coefficients for s4 in the fourth constraint
    elif j == 4:
        row[4+4*i] = 1  # Coefficients for s5 in the fifth constraint

    Slacks.append(row)

#This section of code constructs the final constraint matrix by concatenating the constraints and slack variables horizontally.
constraint_matrices = np.hstack((constraints,Slacks))
#Defining the original unchanged variables
c[0] = 150
c[1] = 230
c[2] = 260

#The code uses nested loops to generate the coefficient vector (c).
for i in range(num_scenarios):
    c[3 + (i*6)] = 238 * (1/num_scenarios)
    c[4 + (i*6)] = 210 * (1/num_scenarios)
    c[5 + (i*6)] = -170 * (1/num_scenarios)
    c[6 + (i*6)] = -150 * (1/num_scenarios)
    c[7 + (i*6)] = -36 * (1/num_scenarios)
    c[8 + (i*6)] = -10 * (1/num_scenarios)
```

```
  #The code uses nested loops to generate the right-hand side vector (b).
  b[0] = 500
  for i in range(num_scenarios):
    b[1 + (i*4)] = 200
    b[2 + (i*4)] = 240
    b[3 + (i*4)] = 0
    b[4 + (i*4)] = 6000
#This code converts the b list to a NumPy array and reshapes it to a column vector.
  b = np.array(b).reshape(-1,1)
#This code returns the generated constraint matrices (constraint_matrices), objective function coefficients (c), and right-hand side vector (b).
  return constraint_matrices, c, b



#This code generates the coefficients for the phase 1 simplex method


#
def generate_phase_one_lp_coefficients_from_constraints(A,b, c):
    #The code below determines the number of constraints (num_constraints) and the number of decision variables (num_variables) based on the shape of the A matrix.
    num_constraints, num_variables = A.shape
    #This code initializes a vector x of ones with a shape of (num_variables, 1).
    x = np.ones((num_variables,1))
    #Calculating the "artificial_matrix" by taking the identity matrix multiplied by the difference between the right-hand side vector b and the result of the matrix-vector multiplication A @ x.
    artificial_matrix = np.eye(num_constraints) * (b - (A@x))
    #Concatenates the original constraint matrix A and the artificial matrix horizontally to create a new matrix A_with_artificial.
    A_with_artificial = np.concatenate((A, artificial_matrix), axis=1)
```

```
    #It initializes a new coefficient vector c_new with zeros and a length of num_variables +
num_constraints.
    c_new = np.zeros(num_variables + num_constraints)
    #It assigns the first num_variables elements of c_new with the values from the original coefficient
vector c.
    c_new[:num_variables] = c
    #It assigns the remaining elements (corresponding to the artificial variables) in c_new with a large
constant value (1,000,000 in this case).
    c_new[num_variables:] = 1000000  # Coefficients for artificial variables


    #Organizing the generated coefficients
    coefficients = {
        'A': A_with_artificial,
        'b': b.reshape(-1,1),
        'c': c_new.reshape(-1,1)}


    return coefficients


#Initializing the number of scenarios to percentage
num_scenarios = 1
percentage = 0.001
e = 0.0001
A_pre,c_pre,b_pre = generate_constraint_matrix(num_scenarios,percentage)


beginning = time.time()
phase_one_coefficients = generate_phase_one_lp_coefficients_from_constraints(A_pre,b_pre,c_pre)


A = phase_one_coefficients['A']
b = phase_one_coefficients['b']
c = phase_one_coefficients['c']
```

```
zero_X = np.zeros((A_pre.shape[1],1))
x = np.ones((A.shape[1],1))

for i_count in count():
  X_k = np.diag(np.squeeze(x))
  y = np.linalg.solve(X_k, x)

  W_left = A @ (X_k @ X_k) @ A.T
  W_right = A @ (X_k @ X_k) @ c
  W_k = np.linalg.solve(W_left, W_right)
  r_k = c - A.T @ W_k

  gap = y.T @ X_k @ r_k

  if abs(gap) <= e:
    Optimal_X = x
    Optimal_value = c.T @ x
    break

  direction = -X_k @ r_k
  direction = direction.reshape(-1, 1)
  alpha = np.array([])

  for d in direction:
    if d < 0:
      a = 0.6
      a = -(a / d)
      alpha = np.append(alpha,a)
```

```
 a_min = np.min(alpha)


 x = x + a_min * (X_k @ direction)

total_time = time.time() – beginning


Optimal_value

total_time

beginning2 = time.time()
result = linprog(c, A_eq = A, b_eq = b, method = 'interior-point')
total_time2 = time.time() - beginning2

result.fun

total_time2
```

Data Set:

Scenarios	Percentage	Simplex		
		Optimal Value	Total Time (seconds)	Total Time (minutes)
25	0.001	-118116.8198	2.460882	0.041015
51	0.001	-117616.5691	7.055429	0.11759
75	0.001	-117157.7869	13.93188	0.232198
101	0.001	-116664.1413	31.20085	0.520014
151	0.001	-115724.0617	74.12519	1.23542
175	0.001	-115277.3179	112.8232	1.880386
201	0.001	-114796.6582	201.0178	3.350296
225	0.001	-114356.2236	301.2379	5.020632
250	0.001	-113900.5132	433.0572	7.217619
251	0.001	-113882.4094	446.7137	7.445228
275	0.001	-113448.2487	606.8464	10.11411
301	0.001	-112981.3641	841.0465	14.01744

325	0.001	-112553.6631	1072.902	17.8817
1	0.01	-118600	0.005027	8.38E-05
3	0.01	-118035.5556	0.023514	0.000392
5	0.01	-117668.0488	0.122975	0.00205
7	0.01	-117281.761	0.332339	0.005539
9	0.01	-116882.0166	0.273803	0.004563
11	0.01	-116507.5958	0.616755	0.010279
21	0.01	-114632.0513	1.815509	0.030258
25	0.01	-113906.0513	2.153173	0.035886
31	0.01	-112821.6453	4.816966	0.080283
41	0.01	-111067.7507	5.97445	0.099574
51	0.01	-109367.2992	12.82014	0.213669
61	0.01	-107720.6365	18.40626	0.306771
71	0.01	-106132.7006	28.47185	0.474531
75	0.01	-105512.3238	30.0327	0.500545
81	0.01	-104601.684	36.51209	0.608535
91	0.01	-103125.6913	46.28348	0.771391
101	0.01	-101704.5246	52.05785	0.867631

111	0.01	-100337.5593	68.06642	1.13444
120	0.01	-99141.66046	81.47463	1.35791
122	0.01	-98879.55408	85.70909	1.428485
131	0.01	-97708.06027	87.20236	1.453373
141	0.01	-96427.15488	108.3813	1.806355
150	0.01	-95291.20227	122.0187	2.033645
152	0.01	-95040.54308	145.1272	2.418787

Scenarios	Percentage	Linprog-Simplex		
		Optimal Value	Total Time (seconds)	Total Time (minutes)
25	0.001	-118116.8198	0.234834	0.003914
51	0.001	-117616.5691	0.294863	0.004914
75	0.001	-117157.7869	1.15612	0.019269
101	0.001	-116664.1413	2.402996	0.04005
151	0.001	-115724.0617	6.857419	0.11429
175	0.001	-108672.3415	7.176424	0.119607

201	0.001	-106967.5514	11.61287	0.193548
225	0.001	-100404.9806	14.11624	0.235271
250	0.001	-89745.43142	14.50647	0.241775
251	0.001	-89049.51798	14.64416	0.244069
275	0.001	-79922.29627	17.82487	0.297081
301	0.001	-65663.615	20.59362	0.343227
325	0.001	-48577.46716	17.82487	0.297081
1	0.01	-118600	0.011201	0.000187
3	0.01	-118035.5556	0.015155	0.000253
5	0.01	-117668.0488	0.047825	0.000797
7	0.01	-117281.761	0.024129	0.000402
9	0.01	-116882.0166	0.054894	0.000915
11	0.01	-116507.5958	0.036396	0.000607
21	0.01	-114632.0513	0.21049	0.003508
25	0.01	-113906.0513	0.274165	0.004569
31	0.01	-112821.6453	0.329469	0.005491
41	0.01	-111067.7507	0.918255	0.015304
51	0.01	-109367.2992	0.745488	0.012425

61	0.01	-107720.6365	3.151678	0.052528
71	0.01	-106132.7006	1.863664	0.031061
75	0.01	-105512.3238	4.075658	0.067928
81	0.01	-104601.684	4.084986	0.068083
91	0.01	-103125.6913	4.430833	0.073847
101	0.01	-101704.5246	4.327394	0.072123
111	0.01	-100337.5593	7.201981	0.120033
120	0.01	-99110.86722	6.715329	0.111922
122	0.01	-98515.95259	7.885529	0.131425
131	0.01	-91308.92571	6.190152	0.103169
141	0.01	-80286.19291	5.957397	0.09929
150	0.01	-17383.33333	6.189131	0.103152
152	0.01	-67437.81718	7.23875	0.120646

Scenarios	Percentage	Interior-point		
		Optimal Value	Total Time (seconds)	Total Time (minutes)
25	0.001	-118114.737	4.143407	0.069057

51	0.001	-117618.6443	16.47559	0.274593
75	0.001	-117152.2151	58.42358	0.973726
101	0.001	-116679.3169	160.0323	2.667205
151	0.001	-115707.0009	696.6313	11.61052
175	0.001	-115294.1283	1251.649	20.86082
201	0.001	-114803.489	2137.274	35.62124
225	0.001	-114796.6578	3052.662	50.87769
250	0.001	-113835.6938	4785.074	79.75123
251	0.001	-113868.3514	5178.465	86.30774
275	0.001	-113367.3596	7518.581	125.3097
301	0.001	-112978.9422	9294.201	154.9034
325	0.001	-112641.3691	12863.54	214.3923
1	0.01	-118599.9903	0.114358	0.001906
3	0.01	-118035.4963	0.272485	0.004541
5	0.01	-117667.6339	0.296103	0.004935
7	0.01	-117281.5931	0.210918	0.003515
9	0.01	-116882.7831	0.342926	0.005715

11	0.01	-116507.1793	1.051664	0.017528
21	0.01	-114631.5141	2.678546	0.044642
25	0.01	-113905.1917	7.751768	0.129196
31	0.01	-112819.2336	6.731457	0.112191
41	0.01	-111063.3452	18.52435	0.308739
51	0.01	-109372.4517	38.93869	0.648978
61	0.01	-107717.2217	69.83818	1.16397
71	0.01	-106108.8629	117.2973	1.954954
75	0.01	-105515.3386	120.1924	2.003207
81	0.01	-104601.6459	196.3785	3.272975
91	0.01	-103122.159	285.1984	4.753307
101	0.01	-101695.9582	369.6723	6.161205
111	0.01	-100305.4865	524.9748	8.749579
120	0.01	-99133.55219	842.3157	14.03859
122	0.01	-98868.96896	776.008	-98879.55
131	0.01	-97725.85961	1057.559	-98879.55
141	0.01	-96389.90297	1532.45	-98879.55

150	0.01	-95249.48334	1551.726	25.86209
152	0.01	-95036.75058	1616.114	26.93523

Scenarios	Percentage	Linprog-Interior-point		
		Optimal Value	Total Time (seconds)	Total Time (minutes)
25	0.001	-118116.8198	0.06469	0.0010782
51	0.001	-117616.5691	0.214672	0.0035779
75	0.001	-117157.7869	0.622619	0.010377
101	0.001	-116664.1413	1.602331	0.0267055
151	0.001	-115724.0617	3.356411	0.0559402
175	0.001	-115277.3179	5.09559	0.0849265
201	0.001	-114796.6573	9.384196	0.1564033
225	0.001	-114356.2236	11.38904	0.1898173
250	0.001	-113900.5131	13.4567	0.2242784
251	0.001	-113882.4094	13.31947	0.2219912
275	0.001	-113448.2487	17.22058	0.2870097
301	0.001	-112981.3641	23.81131	0.3968552
325	0.001	-112553.6628	28.07649	0.4679415
1	0.01	-118600	0.079131	0.0013188
3	0.01	-118035.5553	0.031607	0.0005268
5	0.01	-117668.0488	0.022104	0.0003684
7	0.01	-117281.761	0.0196	0.0003267
9	0.01	-116882.0166	0.021362	0.000356
11	0.01	-116507.5958	0.030975	0.0005163
21	0.01	-114632.0512	0.087274	0.0014546
25	0.01	-113906.0513	0.141424	0.0023571

31	0.01	-112821.6452	0.089818	0.001497
41	0.01	-111067.7507	0.380148	0.0063358
51	0.01	-109367.2992	0.350243	0.0058374
61	0.01	-107720.6365	0.527704	0.0087951
71	0.01	-106132.7005	0.660085	0.0110014
75	0.01	-105512.3236	1.000148	0.0166691
81	0.01	-104601.684	0.702748	0.0117125
91	0.01	-103125.6913	1.161235	0.0193539
101	0.01	-101704.5245	1.200758	0.0200126
111	0.01	-100337.5588	2.468973	0.0411495
120	0.01	-99141.66046	2.281243	0.0380207
122	0.01	-98879.55408	2.200068	0.0366678
131	0.01	-97708.05999	2.2817	0.0380283
141	0.01	-96427.15487	3.915458	0.0652576
150	0.01	-95291.20225	4.9454	0.0824233
152	0.01	-95040.54307	4.80391	0.0800652

Scenarios	Percentage	O.F. % Difference (Python Simplex Method vs Linprog Simplex)	Time % Difference (Python Simplex Method vs Linprog Simplex)	O.F. % Difference (Python Interior-point vs Linprog Interior-point)
25	0.001	0.00%	90.46%	0.00%
51	0.001	0.00%	95.82%	0.00%
75	0.001	0.00%	91.70%	0.00%
101	0.001	0.00%	92.30%	0.01%
151	0.001	0.00%	90.75%	-0.01%
175	0.001	5.73%	93.64%	0.01%
201	0.001	6.82%	94.22%	0.01%

225	0.001	12.20%	95.31%	0.38%
250	0.001	21.21%	96.65%	-0.06%
251	0.001	21.81%	96.72%	-0.01%
275	0.001	29.55%	97.06%	-0.07%
301	0.001	41.88%	97.55%	0.00%
325	0.001	56.84%	98.34%	0.08%
1	0.01	0.00%	-122.84%	0.00%
3	0.01	0.00%	35.55%	0.00%
5	0.01	0.00%	61.11%	0.00%
7	0.01	0.00%	92.74%	0.00%
9	0.01	0.00%	79.95%	0.00%
11	0.01	0.00%	94.10%	0.00%
21	0.01	0.00%	88.41%	0.00%
25	0.01	0.00%	87.27%	0.00%
31	0.01	0.00%	93.16%	0.00%
41	0.01	0.00%	84.63%	0.00%
51	0.01	0.00%	94.19%	0.00%
61	0.01	0.00%	82.88%	0.00%
71	0.01	0.00%	93.45%	-0.02%
75	0.01	0.00%	86.43%	0.00%
81	0.01	0.00%	88.81%	0.00%
91	0.01	0.00%	90.43%	0.00%
101	0.01	0.00%	91.69%	-0.01%
111	0.01	0.00%	89.42%	-0.03%
120	0.01	0.03%	91.76%	-0.01%
122	0.01	0.37%	90.80%	-0.01%
131	0.01	6.55%	92.90%	0.02%
141	0.01	16.74%	94.50%	-0.04%
150	0.01	81.76%	94.93%	-0.04%
152	0.01	29.04%	95.01%	0.00%

Scenarios	Percentage	Time % Difference (Python Interior-point vs Linprog Interior-point)	Time % Difference (Python Simplex Method vs Python Interior-point)	Time % Difference (Lingprog Simplex vs Lingprog Interior-point)
25	0.001	98.44%	-68.37%	72.45%
51	0.001	98.70%	-133.52%	27.20%
75	0.001	98.93%	-319.35%	46.15%
101	0.001	99.00%	-412.91%	33.32%
151	0.001	99.52%	-839.80%	51.05%
175	0.001	99.59%	-1009.39%	29.00%
201	0.001	99.56%	-963.23%	19.19%
225	0.001	99.63%	-913.37%	19.32%
250	0.001	99.72%	-1004.95%	7.24%
251	0.001	99.74%	-1059.24%	9.05%
275	0.001	99.77%	-1138.96%	3.39%
301	0.001	99.74%	-1005.08%	-15.62%
325	0.001	99.78%	-1098.95%	-57.51%
1	0.01	30.80%	-2175.06%	-606.45%
3	0.01	88.40%	-1058.84%	-108.56%
5	0.01	92.54%	-140.78%	53.78%
7	0.01	90.71%	36.54%	18.77%
9	0.01	93.77%	-25.25%	61.09%
11	0.01	97.05%	-70.52%	14.89%
21	0.01	96.74%	-47.54%	58.54%
25	0.01	98.18%	-260.02%	48.42%
31	0.01	98.67%	-39.74%	72.74%
41	0.01	97.95%	-210.06%	58.60%
51	0.01	99.10%	-203.73%	53.02%
61	0.01	99.24%	-279.43%	83.26%
71	0.01	99.44%	-311.98%	64.58%

75	0.01	99.17%	-300.21%	75.46%
81	0.01	99.64%	-437.85%	82.80%
91	0.01	99.59%	-516.20%	73.79%
101	0.01	99.68%	-610.12%	72.25%
111	0.01	99.53%	-671.27%	65.72%
120	0.01	99.73%	-933.84%	66.03%
122	0.01	99.72%	-805.40%	72.10%
131	0.01	99.78%	-1112.76%	63.14%
141	0.01	99.74%	-1313.94%	34.28%
150	0.01	99.68%	-1171.71%	20.10%
152	0.01	99.70%	-1013.58%	33.64%