Stochastic Farming Problem

University of Toronto

Fall 2023

Layth Al-Nemri **Neil Sanjiv Punjani**

Contents

1. Initial Problem Description and Motivation.

1.1 Introduction to the goal and the problem.

This project is targeted towards solving a Stochastic Linear Programming problem where the goal is to compare the computational speed and ability of (1) The simplex method, (2), the Primal Affine Scaling Interior Point Method and (3) An Industrial Linear programming solver under an increasing number of scenarios. The problem structure is as follows:

The three methods will be analyzed on a farming problem which involves assigning acres of land towards the production of three different plants (Wheat, Corn and Sugar Beets) in order to maximize the profit (or minimize the cost). In the initial problem, we are given the following data:

	Information Given:					**Variables to solve for:**		
Plant	**Average Yield Tons/Acre**	**Planting Cost/$**	**Selling Price/$**	**Minimum Requirement/ Units**	**Purchase Price from market /$**	**Acres Assigned to Plant/ units**	**Amount Purchased from the market**	**Tons of plants sold**
Wheat	2.5	150	170	200	238	x_1	y_1	w_1
Corn	3	230	150	240	210	x_2	y_2	w_2
Sugar Beets	20	260	36 - Under and equal to 6000 Tons sold.	-	-	x_3	-	w_3
			10 - Above 6000 Tons sold					w_4

Table 1: Initial Stochastic Farming Problem – Given Data.

The problem has some stated requirements that will translate into constraints:

1. The maximum number of available acres for planting are 500 acres.

2. The production of wheat and corn must at least satisfy the minimum requirements of availability.
3. If the amount produced of wheat and corn do not at least meet the minimum requirements of each plant then the remaining quantity must be purchased from the market.
4. If the minimum requirements are met, additional plants yielded may be sold at the selling price.

1.2 Initial Problem formulation

As talked about in the previous section, the objective function becomes related to maximizing the profit of the production or in standard form, the minimization of the cost of production. The two objective functions are formulated based on the data given above in **Table 1** are highlighted below:

$$maximize \;\; 170\; w_1 + 150\, w_2 + 36\, w_3 \; + 10\, w_4 - 150\, x_1 - 230\, x_2 - 260\, x_3 - 238\, y_1 - 210\; y_2$$

Or

$$minimize \; 150\, x_1 + 230\, x_2 + 260\, x_3 + 238\, y_1 + 210\; y_2 - 170\, w_1 - 150\, w_2 - 36\, w_3 \; - 10\, w_4$$

Our constraints for this formulation will be based on the stated requirements elaborated upon in the previous section. Thus, the constraints for this formulation become

1. $x_1 + x_2 + x_3 \leq 500$ (Satisfying the availability of acres of land)

2. $2.5x_1 + y_1 - w_1 \geq 200$ (Satisfying the minimum requirement of wheat)

3. $3x_2 + y_2 - w_2 \geq 240$ (Satisfying the minimum requirement of Corn)

4. $w_3 + w_4 - 20\, x_3 \leq 0$ (The amount of sugar beets that you sell cannot be more than you produce)

5. $w_3 \leq 6000$ (The amount of sugar beets sold associated with the variable w_3 must be less than 6000.)

6. $x_1, x_2, x_3, y_1, y_2, w_1, w_2, w_3, w_4 \geq 0$ (Non-Negativity Constraints)

1.3 Final problem formulation to address uncertainty

The goal of this reformulation is to address the fact that in relation to the farming problem, we could have uncertainty in the yield of production from each acre of plant depending on different external factors. In order to determine the number of acres to assign to each plant, we must protect against this uncertainty ensure that the solution we obtain can be applicable to a variety of different uncertainties.

To highlight this, we are given the information regarding the probability of a different scenario and yield of each plant in that scenario from each acre of land under three different scenarios. This information is highlighted in the table below:

	Scenario 1	**Scenario 2**	**Scenario 3**
Probability of Scenario Occurrence	1/3	1/3	1/3
Yield Change for Each Plant Under Each Scenario	20%	0	-20%

Table 2: Initial conditions for reformulation

Under the Stochastic Linear Programming problem, in order to account for uncertainty, we must reformulate the objective function and add new constraints that account for this additional change in yield:

- Constraint (1) in **section 1.2** associated with the availability of the farmland will not change under the new scenarios because the availability will always remain 500 acres as stated in the problem.

- Constraint (2), (3) must change under the new scenarios and new variables must be introduced in order to account for this change in yield under the scenarios.

- Constraints (4) and (5) in **section 1.2** will also change in order to account for the fact that the amount of sugar beets sold will vary under these new scenarios.

- The objective function must also account for the change in cost associated with the change in scenarios.

Therefore, we add additional constraints that introduce new variables to account for the change that would occur in amount of each plant purchased from the market, the amount of each plant sold to satisfy the requirements and the impact of these new scenarios on the final assigned acres of land to each plant.

We change the objective function by adding these new variables associated with each scenario by multiplying it with the probability of that scenario occurring to account for the change that would occur in our cost function if a scenario were to occur.

Therefore, the standard form reformulation of the stochastic linear programming model based on the information given in **table 2** is highlighted below:

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

Unchanging Constraint

$$st.\ x_1 + x_2 + x_3 \leq 500$$

$$x_1, x_2, x_3 \geq 0$$

Scenario 1 Constraints:

$3x_1 + y_{11} - w_{11} \geq 200$

$3.6x_2 + y_{21} - w_{21} \geq 240$

$w_{31} + w_{41} - 24\,x_3 \leq 0$

$w_{31} \leq 6000$

$y_{11}, y_{21}, w_{11}, w_{21,} w_{31}, w_{41} \geq 0$

Scenario 2 Constraints:

$2.5x_1 + y_{12} - w_{12} \geq 200$

$3x_2 + y_{22} - w_{22} \geq 240$

$w_{32} + w_{42} - 20\,x_3 \leq 0$

$w_{32} \leq 6000$

$y_{12}, y_{22}, w_{12}, w_{22,} w_{32}, w_{42} \geq 0$

Scenario 3 Constraints:

$2x_1 + y_{13} - w_{13} \geq 200$

$2.4x_2 + y_{23} - w_{23} \geq 240$

$w_{33} + w_{43} - 16\,x_3 \leq 0$

$w_{33} \leq 6000$

$y_{13}, y_{23}, w_{13}, w_{23,} w_{33}, w_{43} \geq 0$

1.4 Scenario Generation approach for comparison of different methods

As we have set up the foundation of our problem, we can now discuss in detail the objective and the methods of this project. The goal of this assignment is to compare a hand-coded simplex method, a hand-coded affine scaling interior point method and an industrial linear programming solver to compare the performance of each method in solving the farming stochastic problem with an increasing number of scenarios.

In the previous section, we have demonstrated how the Linear program will change with 3 different scenarios where each additional scenario adds 4 different constraints (excluding non-negativity constraints) and each additional scenario adds 6 different variables (excluding slacks). Therefore, for the previous example, where had a total of 3 different scenarios we had a total of 13 different constraints (4 for each scenario and 1 constant constraint that will not change) and 21 variables (3 constant variables and 6 variables each for each scenario.)

For the cost function, the number of variables will directly correspond to the number of scenarios. We see in the previous example, that our cost function had a total of 21 variables (3 constant variables and 6 variables each for each scenario). Each variable will have the same coefficient values which will be generated by multiplying the original cost coefficients stated in the objective function in **section 1.3** (except cost coefficients associated with x1, x2 and x3) to the new probabilities that will depend upon the number of scenarios we wish to solve the linear program for.

In order to generate the new scenarios, the following method will be used:

- The probability of each scenario will be determined by $\frac{1}{Number\ of\ Scenarios}$, and used to compute the new cost coefficient values of the new variables for each scenario as described in paragraph 3 of **this section**.

- The % yield change will be determined initially for each different time we change how many scenarios we wish to have our linear program for. This yield % will be added and subtracted to all respective x variables' coefficient in each constraint to create two individual scenarios. For example, as we see in scenario 2 constraints our original coefficients for (x1, x2, x3) were (2.5, 3, 20) and these values were updated with 20% yield added and 20% yield subtracted to get scenario 1 and scenario 3 respectively.

- Therefore, if we have 5 different scenarios we wish to code, and we have a 20% yield increase. Then the first two scenarios apart from the original will add 20% and subtract 20% to the coefficients of x values and generate individual constraints and for the remaining two scenarios, 40% will be added and subtracted from each original coefficient value of x, to get the 2 other scenarios.

- We will keep our scenario generation for odd number of values. The reason is that we wish to increase our yield around our nominal values of the x coefficients as they are in scenario 2 above. Also, we wish to keep the percentage change as we describe it in our scenario generation. By using an odd number of scenarios, we will be able to keep our initial scenario centered and generate scenarios with the yield exactly as we have specified.

2. Code Logic and Explanations

2.1 Scenario generation

The actual code is shown in the Appendix, under section 6.1

The implementation of the code is based on the logic that is described in the previous section.

1. We started the code by finding the percentage increments of each scenario as described in the previous section.

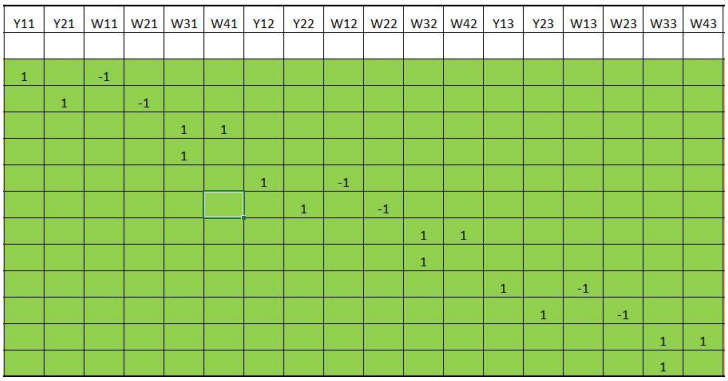
2. In the pictures 1, 2 and 3, we saw that the first row will always be fixed and it will not be subject to change. Therefore, the first row will be with the first 4 coefficients as 1, and the rest of the variables will be 0. This will not change no matter how many scenarios we generate. This row will be concatenated at the end.

3. The generation of the A matrix for each scenario would follow a certain pattern. We used this pattern to generate our A matrixes for our new constraints. For the coefficients of our x values:

X1	X2	X3	S1
1	1	1	1
2			
	2.4		
		-16	
2.5			
	3		
		-20	
3			
	3.6		
		-24	

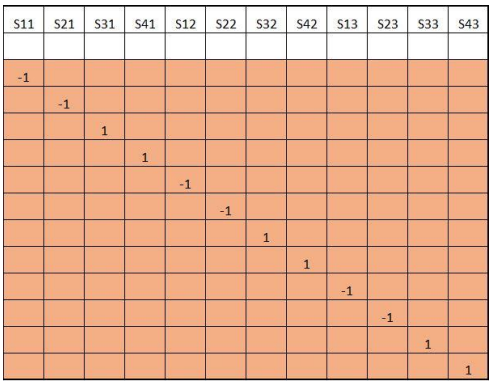
Picture 1: Example displaying the pattern of the x coefficients.

a.) The yellow section in picture 1, highlights that for each of the coefficients of the x values, we simply get the coefficients of the x1, x2 and x3 values for the new scenarios with the percentage increments on the 2.5, 3 and 20. Therefore, we use the percentage increments on each scenario and multiply the original A matrix by the percentage yield change as per the new scenarios to get a new A. The only values that would change would be the coefficients of x while the rest of the A matrix coefficients will remain the same. (The blank spaces are all zeroes). After getting the new A for each scenario based on these methods, we simply vertically stack it.



Y11	Y21	W11	W21	W31	W41	Y12	Y22	W12	W22	W32	W42	Y13	Y23	W13	W23	W33	W43
1		-1															
	1		-1														
				1	1												
				1													
						1		-1									
							1		-1								
										1	1						
										1							
												1		-1			
													1		-1		
																1	1
																1	

Picture 2: Diagonal matrix for the y and w constraints



S11	S21	S31	S41	S12	S22	S32	S42	S13	S23	S33	S43
-1											
	-1										
		1									
			1								
				-1							
					-1						
						1					
							1				
								-1			
									-1		
										1	
											1

Picture 3: Diagonal Matrix for the slack variables

b.) In the Picture 2, we see that that for each scenario, the 6 variables for each scenario represented by y and w repeat in structure in terms of their coefficients. (The blank spaces are all zeroes). Therefore, in order to implement this in our generation of the A matrix, we used the kron function in python to generate a repeating pattern of these coefficients where the pattern of the y and w variables in our original constraints in our case where we had only one scenario and was extended towards the increasing number of scenarios that we wished to calculate our linear program for.

c.) The same was the case for the slack variables for each scenario. Each scenario had 4 slack variables (4 constraints) which would repeat their coefficients in the same pattern for every single scenario. (Blank spaces are all zeroes).We used this fact to generate an update A matrix for these slack variables as they would follow the same repeating coefficients. We did this in the same way we achieved it for the y and w variables which was by using the kron function to repeat this structure of coefficients.

4. After following steps 1, 2, 3 a), b) and c) each time we increased our scenarios, we made our final concatenated A matrix with all of these sub-A matrixes which would be used as the basis for solving the linear program. Therefore, we then horizontally stacked the matrixes generated in steps 3. a), b) and c) and then vertically stacked the row that was generated in step 2 to get out final A matrix.

4. For the b matrix, the steps were more straightforward. We knew that the pattern of the b coefficients would keep repeating themselves for each of the 4 constraints that would be generated for each scenario. We therefore then repeated this pattern for the number of scenarios we wished to implement them for, made a column array with these b coefficients and then vertically stacked this new column of b with the original single element array of the coefficient of the unchanging constraint that was 500. This gave our updated b matrix to be used when solving the linear program.

5. For the c matrix, the steps were also relatively straightforward. We demonstrated in section 1.3 that the c values for the coefficients of x associated with the first constraint would not change. The slack for this constraint will also always have a zero.. Therefore, In order to get the new c values for the y and w coefficients, we used the probability depending on the number of scenarios we wanted and multiplied this to the coefficients of the original values of y and w for 1 scenario and repeated this pattern for all scenarios to get an array. For the new slack variables associated with each constraint under the new scenarios, their coefficients would always be zero. Therefore, we used this fact to generated an array with all 0's for the slack variables. Finally, we vertically stacked all of these arrays to get our final array for our c to be used in the linear program.

2.2 The simplex method implementation

The simplex method code that we computed with an explanation of each step of the code is presented in **Appendix 6.2** that breaks down the steps. It is not included in the main body of this report as the explanation is very long and is a corollary to the results. In this section we would like to highlight some of the main points of our code, observations and improvisations we did to code our simplex method.

- Our very first attempt at generating a Basic Feasible Solution was using the two-phase method. After coding it, our phase 1 method outputted a BFS for up to the 3 scenarios. The simplex method we coded was able to successfully solve the problem. However, after generating any number of scenarios more than 3, our phase 1 method was not able to

compute the answers and kept giving us an unbounded answer. However, we knew that the problem was still feasible as we got an optimal solution from linprog. The pictures below displays the code for phase 1 and the error that we got when feeding this into the simplex method code.

```
def phase_1(A,b):

  m,n = A.shape

  cb_1 = np.ones((m, 1))
  cn_1 = np.zeros(((n), 1))
  c_1 = np.vstack((cn_1, cb_1))
  XB_1 = b
  XN_1 = np.zeros(((n), 1))

  IB_1 = np.zeros((1, m))
  for i in range(m):
    IB_1[:, i] = i+n+1

  IN_1 = np.zeros((1, n))
  for i in range(n):
    IN_1[:, i] = i+1

  identity = np.eye(m)
  A_1 = np.hstack((A, identity))
```

Picture 4: Code for Phase 1

```
LP Unbounded <class 'str'>
LP Unbounded <class 'str'>
```

Picture 5: Error message when running this code in simplex

- We hence decided to use the big M method and this implementation was able to successfully generate the correct solution. This may be the case because by doing the big M we are essentially forcing the program to compute the optimal solution while the phase 1 method was a separate linear program on its own. We selected our M values based on a trial-and-error approach. The highest positive coefficient value for our c was 260. Hence, we initially decided a M value of 100,000 was significantly high and would drive the artificial variables to 0. When we solved the problems with a M value of 100,000, the linear program was outputting the correct answer. For additionally safety we decided to use 1,000,000 rather than 100,000 as our final M.

- Another part of the linear program that we wish to talk about is the use of the linalg.solve which is the function of solving a system of linear equations instead of calculating the inverse of the matrix. This helped our linear program run quicker for the simplex method.

- Lastly, in order to prevent cycling that could occur due to degeneracy we implemented Blands' rule in our simplex method implementation. We did this by reorganizing the indices of the basic variables and the non-basic variables each time we did an iteration along with the basic variable values and non-basic variable values. Therefore, the entering non basic variable would always be the one with a lower index as we only computed the reduced cost for the first non-basic variable that had a negative reduced cost. Lastly, the variable that exited would always be the one with the lower index if the alpha is the same because we do not update the index of alpha if we get the same for any of the next index of basic variables.

2.3 Interior point method implementation.

Implementing the affine scaling interior point method code was more straightforward then the simplex method. The interior point method code is displayed in **appendix 6.4** with a complete set of steps describing the code. The same as the simplex, we do not want to include it in the main body of the report. There are some observations that we wish to discuss in this section when we coded the interior point method.

- We used the big M method to generate an initial BFS that uses a M. This implementation of M was based on the same trial and error approach as we did for the simplex method. The highest coefficient value of the original variables remain the same. The M that was trial and errored was 100,000 and it gave the correct solution. Same as the phase 1, we implemented a 1,000,000 M value to be on the safer side.

- Secondly, we decided on an epsilon value of 0.01 which would used as our tolerance. This value was chosen on the fact that when we increased the value of epsilon our computational

time increased as well as the accuracy of our solutions compared to the linprog was worse. The picture below demonstrates the difference in the time and objective function value. [1]

	Scenarios_no	Objective_function_value	Time_needed	Largest_percentage_change
0	101.0	-111876.298118	70.325313	0.18

Picture 6: Computational time and Objective function value for 0.0001 epsilon

Our interior point method, when running for an epsilon value of 0.01 converged and gave us the solution of -111872.5344 for our objective function at 46.5527 seconds. The output of the linprog function was -111872.4657. Therefore, not only did the time of computation improve but the accuracy also worsened and hence we kept the epsilon value as 0.01 for our interior point method.

- The last observation we made was regards to using the value of alpha as 0.99 to compute our alpha(k). When using such a high value of alpha the computational time improved as the algorithm was more aggressive but it fed out an objective function value of the problem to be incorrect. The picture below shows this issue.

	Scenarios_no	Objective_function_value	Time_needed	Largest_percentage_change
0	101.0	-111091.141906	30.167509	0.18

Picture 7: Computational time and alpha value for 0.99 alpha value.

In the above picture, we see that the computational time improved from 46.5527 seconds to 30.1675 seconds but the objective function value was inaccurate and verified by linprog. We thought this alpha value was way to aggressive for the algorithm to run correctly and so we used an alpha value of 0.5 which would allow the process to converge accurately and moderately quickly. If we chose an alpha value which was any lower than this, our algorithm would be too slow.

[1] All of the results demonstrated in this section are for the percentage increment set as 0.36%.

3. Results

3.1 General Observations and Summary

Before beginning our results section, we would like to mention that all of our algorithms were run on the Google Colab interface where we coded as well as ran our algorithms. As this runs on google servers, we expected the time of our algorithms running to be slower compared to if we ran it on our desktop.

We ran all three of our algorithms where the scenario generation described in **section 2.1** and **appendix 6.1** used a percentage increment of 0.36%. The linprog function was used from ScyPy library available for python. The time function was used in python to calculate the time each algorithm took to run. The table below is a result of the concatenation of the data outputted from our code into a pandas dataframe and then to excel. This data is a summarization of the iterations we performed. The complete set of scenarios we ran for each algorithm and their data is in **appendix section 6.5**

	Simplex Method		**Interior Point Method**		**Linprog**	
No. of Scenarios	*Objective Function value*	*Time for convergence /Seconds*	*Objective Function value*	*Time for convergence /Seconds*	*Objective Function value /Seconds*	*Time for convergence /Seconds*
1	-118600	0.003801	-118599.9945	0.006969	-118600	0.004734
9	-117987.5446	0.05475	-117978.5439	0.08931	-117978.5446	0.02489
27	-116875.1444	0.4361	-116875.1447	0.6013	-116875.1444	0.1560
77	-113408.7316	7.3069	-113408.4257	18.9673	-113408.7316	1.3189
125	-110377.8582	29.8404	-110375.7499	98.7157	-109804.1651	4.7360
177	-107282.0498	85.0439	-107277.1868	351.5186	-59656.4991	6.6096
205	-105697.1464	117.0731	-105695.5542	620.7654	-84927.5796	8.4976
253	-103112.8449	210.9246	-103109.6371	1359.1187	-90235.1258	14.0562
301	-100692.7284	364.1195	-100687.8992	2626.4620	-75982.9252	15.7063
353	-98214.7894	593.7707	-	-	-27959.6702	23.3160
401	-96000.4927	889.9443	-	-	-53864.6565	25.5869
453	-93665.6459	1457.9322	-	-	-54476.1269	32.7612
501	-91558.3078	1963.7981	-	-	-55887.2698	40.0682
601	-58072.2942	3496.2862			-60889.8000	76.8369

Table 3: Summarized results of objective function and time of computation for all 3 algorithms.

- We were able to run the simplex method algorithm up to 600 scenarios which was computed in approximately 58 minutes.

- The interior point method was able to run upto 301 scenarios. However, when we tried to compute the value for 353 scenarios for the interior point algorithm ran for over one and a half hour and did not converge to a solution. We decided to stop the iterations for this method then.

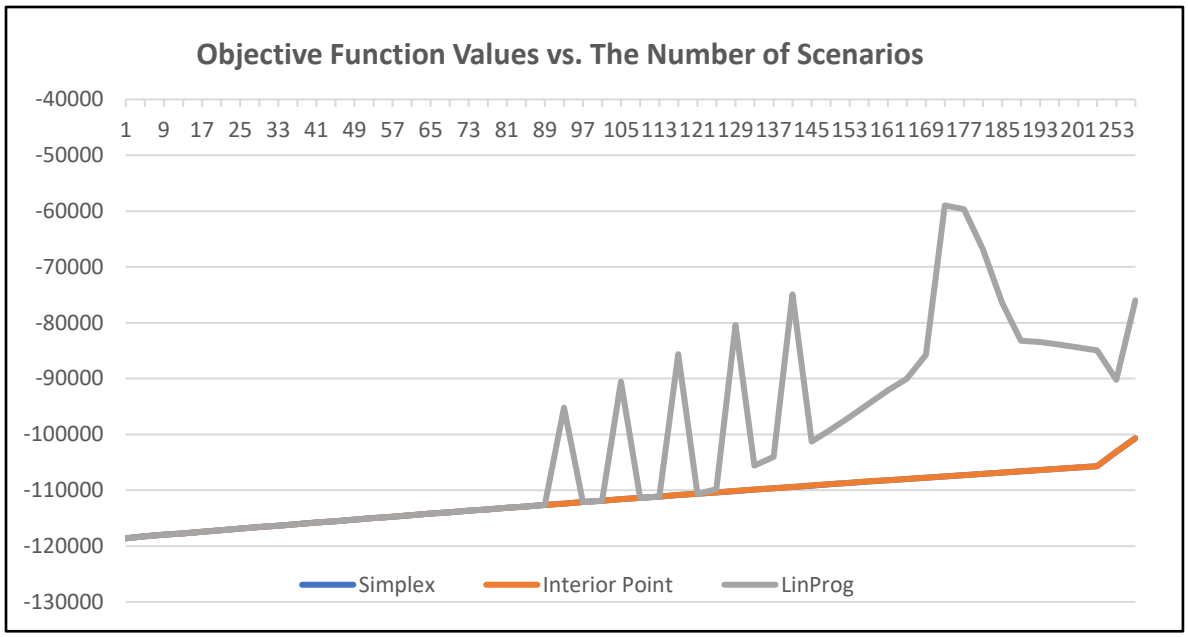
- The simplex method was consistently faster than the interior point method and always slower than the linprog function. This is discussed in **section 3.3.**

- The objective function values for simplex method and the interior point remained the same until the linear program with 89 scenarios after which the linprog function kept diverging and fluctuating in values. This is expanded in **section 3.2.**

- The objective function value for the interior point method remained very close to the simplex objective function values. When the number of scenarios increased, the interior point methods answers started diverging more from the simplex method algorithms values. This is described in section **3.2** .

3.2 Accuracy of the three algorithms.

The graph below shows the objective function values for all 3 algorithms up to 301 scenarios. We do not plot anything above this value because our interior point method was not able to converge to a solution for a value of more than 301 scenarios. Therefore, for comparison of all 3 algorithms we only graph the objective function values for 301 scenarios.



Graph 1: Objective Function Values for all 3 algorithms up to 301 scenarios.

- In the above graph, we see that the simplex method and the interior point objective function values are very close to each other such that they overlap in this graph. However, due to the scale of the graph we cannot see the differences between them but are more clear **in appendix 6.5** where the complete table of data is shown. The simplex method gives the exact answer of the objective function matched by linprog until the 89 scenarios, however the interior point is close to the answer but not exact. This was expected as we are in the interior of our feasible region and hence never truly reaching the Basic Feasible Solution that gives us the exact optimal value. In conclusion the simplex method is more accurate than the interior point method.

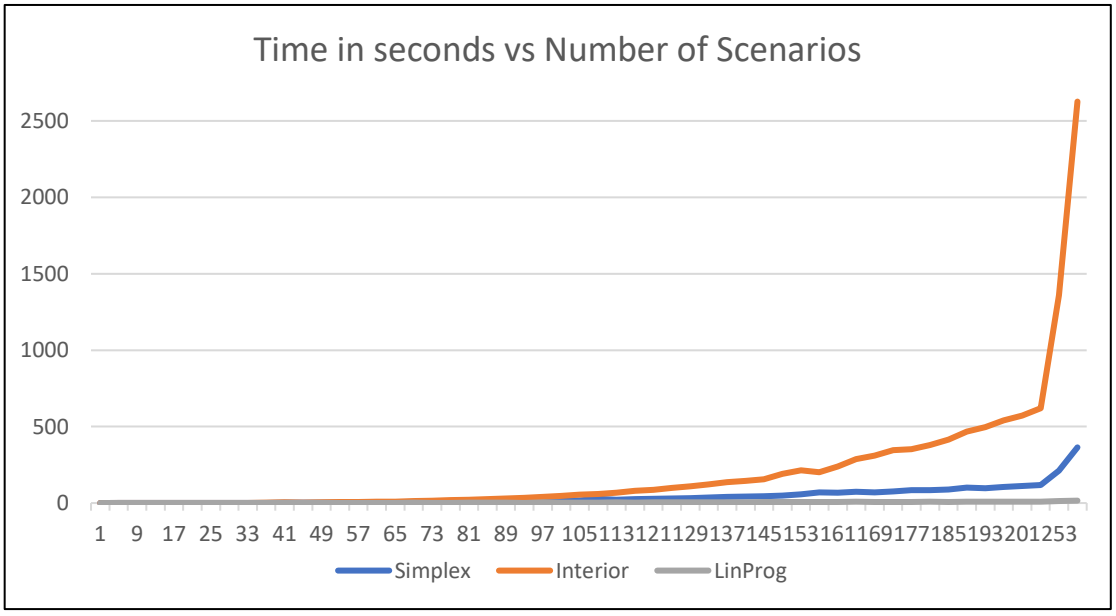
- Interior point method also has a particular risk of tuning the parameters to get the correct answer. As demonstrated in **picture 7**, when alpha was set to 0.99, we received an inaccurate answer for our objective function value. The accuracy was also affected by changing the epsilon value as shown in **picture 6**. Therefore, it is not feasible to check the parameters that give the highest accuracy for each scenario number, making it an inferior method in terms of accuracy compared to the simplex.

- The linprog function gives us the same values as the simplex method up to scenario 89. However, after this 89th scenario the objective function values started to fluctuate with them intersecting with our values of simplex and getting very close to the interior solution under a particular number of scenarios. We strongly believe that our custom simplex is giving us the most accurate answer as the interior point also converges to a value close to the simplex methods' algorithm and follows a predictable and expected pattern of returns.

- However, as linprog is an industrial solver, it is not possible for it to be incorrect while our custom simplex is correct. The problem may be that it's interpreting our scenario generation A and c matrix differently that was implemented in **section 2.1** and **appendix 6.1** and there could be logic error in the way that we generate our scenarios for linprog.

3.3 Speed of the three algorithms

The graph below portrays the computational time for each of these three algorithms. We only plot it till 301 scenarios as the interior point method did not converge in 1 and a half hours for the next scenario of 353 scenarios. The computational time for simplex and the linprog functions for more than 301 scenarios have been shown in table 3.



Graph 2: Time for convergence for all 3 algorithms upto 301 scenarios.

- We notice in graph 2, that upto the 73rd scenario the computational times for simplex and the interior point method are close even though the simplex always performed better than the interior point method.

- After the 73rd scenario the times of computation between the simplex and the interior (as demonstrated in **appendix 6.5**) start to diverge rapidly with the times of computation for 301 scenarios to be 364.1195 seconds 2626.462 seconds respectively. Even more so, the simplex was able to compute for 601 scenarios in approximately 58 minutes while even in 90 minutes the interior point method was not able to compute 353 scenarios.

  This result was surprising as we thought that the interior point method would converge quicker due to theoretically requiring lesser iterations. It might be the case that it may be getting stuck near a vertex of the feasible region that is not the optimal Basic Feasible solution.

- Another reason for the slow interior point convergence might be the alpha value that we chose to compute alpha(k). In **picture 7**, we demonstrated that by increasing alpha our computational time reduced to 30.1675 seconds compared to the 46.5572 seconds that it took when alpha was 0.5. The answer was incorrect however, and even when we chose the alpha value of 0.99 it was still slower than our simplex method for 101 scenarios, which computed the optimal solution in 16.2533 seconds. Therefore, we see that the implemented interior point method will always remain inferior in comparison to the simplex method we implemented.

- Lastly, we see that the difference between computation time between linprog and simplex is relatively close upto around 69 scenarios but the linprog still performs better. After, that however, the linprog solver computes the optimal solution much quicker than our simplex. After 89 scenarios the linprog outputs different objective function values as discussed in **section 3.2.** Even if these values were incorrect, we still expect linprog to always be significantly quicker than the simplex or the interior point method given the same number of scenarios.

4. Conclusions

Based on the results obtained, we can conclude that the primal affine scaling interior point method is an inferior method in comparison to the simplex method due to a lack of accuracy, computational time increases as well as the need for parameter tuning. The only advantage is that coding the interior point method is significantly easier than the simplex method, and this fact could be taken advantage of if the number of scenarios we were solving for were low.

The simplex method excels in comparison to the affine scaling interior point method in terms of its computational speed and accuracy and is also able to output an optimal value for 601 scenarios in approximately 58 minutes while the affine scaling fails at 353 scenarios when running it for 90 minutes.

The linprog function remains the fastest but the optimal solutions generated by the solver after the 89 scenarios raised a pattern which suggested that the objective function value it was outputting was incorrect. This may point to an error in generating the A and c matrix in our scenario generation described in **section 2.1** and **Appendix 6.1.**

5. References and Tools:

1. Introduction to Linear Optimization and Extensions with MATLAB (Dr. Roy Kwon) – For the steps of the implemented simplex method.

2. Lecture Slide Deck for Interior Point method. (North Carolina State University) – For the steps of the implemented primal affine scaling interior point method.

3. https://colab.research.google.com/ - Used as our platform for coding our algorithms as well as running them.

4. NumPy, Pandas and SciPy libraries in Python. – Various coding purposes for our algorithms.

5. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linprog.html - SciPy Linprog function information.

6. Stochastic Farming Problem Slide Deck – For information regarding our initial problem generation.

6. Appendix

6.1 Code for scenario generation

```
for i_s in range(1, 602, 4):

  number_of_scenarios = i_s                                    #deciding the number of variables
  percentage = 0.0036                                          #deciding the percentage increment
  probabilities = 1/number_of_scenarios

  scenarios_list = np.array([])
  for scene in range(number_of_scenarios):
    scenarios_list = np.append(scenarios_list, scene+1)
  percentage_list = ((scenarios_list - np.median(scenarios_list))*percentage)+1   #getting the percentage list based on the number of scenarios

  #it was noticed that the A matrix followed a certain pattern when the scenarios increased. so it's been devided into 4 parts:
  #the initial constraint part (x1+x2+x3+s1 = 500), The coefficient of land acres which will be multiplied with the percentage,
  #other variables coefficients which stays the same and form a super diagonal matrix, the rest of the slacks coefficients which also form a super diagonal matrix

  A_X_normal = np.array([[1, 1, 1, 1]])
  A_X = np.array([[2.5, 0, 0, 0],
                  [0, 3, 0, 0],
                  [0, 0, -20, 0],
                  [0, 0, 0, 0]])
  A_Main_rest = np.array([[1, 0, -1, 0, 0, 0],
                          [0, 1, 0, -1, 0, 0],
                          [0, 0, 0, 0, 1, 1],
                          [0, 0, 0, 0, 1, 0]])
  A_Slacks_rest = np.array([[-1, 0, 0, 0],
                            [0, -1, 0, 0],
                            [0, 0, 1, 0],
                            [0, 0, 0, 1]])

  b_X = np.array([[500]])                                      #the initial constraint b (x1+x2+x3+s1 = 500), is not repeated
  b_Rest = np.array([[200],[240],[0],[6000]])                  #rest of b, repeated as many scenarios
  c_X = np.array([[150],[230],[260], [0]])                     #land acres c, are not repeated
  c_Rest = np.array([[238],[210],[-170],[-150],[-36],[-10]])                    #other variables c, repeated as many scenarios
  c_Slacks_rest = np.array([[0],[0],[0],[0]])                  #slacks c, repeated as many scenarios

  A_Main_rest_new = np.kron(np.eye(number_of_scenarios), A_Main_rest)           #putting this part of the A matrix into the super diagonal matrix form
  A_Main_rest_new[np.isclose(A_Main_rest_new, 0, atol=1e-10)] = 0               #removing any small values generated instead of 0
  A_Slacks_rest_new = np.kron(np.eye(number_of_scenarios), A_Slacks_rest)
  A_Slacks_rest_new[np.isclose(A_Slacks_rest_new, 0, atol=9e-10)] = 0
  A_X_new = np.vstack([A_X * perc for perc in percentage_list])                 #multiplying this part with our percentage set
  A_Rest_new = np.hstack((A_X_new,A_Main_rest_new,A_Slacks_rest_new))           #stacking the three parts into on matrix
  num_columns = A_Rest_new.shape[1]
  zero_rows = np.zeros((1, num_columns - A_X_normal.shape[1]))
  A = np.vstack([np.hstack([A_X_normal, zero_rows]), A_Rest_new])               #adding the initial constraint coefficient and getting our full A matrix

  b_Rest_new = np.vstack([b_Rest for num in range(number_of_scenarios)])   #multiplying the b part with as much constraint as we have then adding the constant part
  b = np.vstack([b_X,b_Rest_new])
                                                              #multiplying the c part with as much constraint as we have then adding the constant part
  c_Rest_new = np.vstack([c_Rest * probabilities for num in range(number_of_scenarios)])
  c_Slacks_rest_new = np.vstack([c_Slacks_rest for num in range(number_of_scenarios)])
  c = np.vstack((c_X,c_Rest_new,c_Slacks_rest_new))

  A = np.round(A, decimals=8)                                   #rounding the values to eliminate any problem with recurring or decimal values
  A[np.isclose(A, 0, atol=1e-8)] = 0
  c = np.round(c, decimals=8)
  c[np.isclose(c, 0, atol=1e-8)] = 0

  start_time = time.time()

  XB, IB, XN, IN = BIG_M(A,b,c)                                 #calling our BIGM function which also solves the LP
  total_index = np.hstack((IB,IN))                              #stacking the indices
  total_X = np.vstack((XB,XN))                                  #stacking the variables
  total_index_flat = total_index.flatten()
  sorting_index = np.argsort(total_index_flat)
  X = total_X[sorting_index]                                    #sorting the ivariables based on their indeces
  Objective_value = c.T@X                                       #getting the objective function value

  end_time = time.time()
  elapsed_time = end_time - start_time
```

6.2 Code for Simplex:

6.2.1 Big M for the Simplex Method

```
def BIG_M(A,b,c):
  #defining the Big M method since our problem needs artificial variables to initial a BFS.
  #it needs Objectve function coefficients, Constraints coefficients, and constraints RHS values.

  m,n = A.shape                                  #storing the shape of A, which indicates the number of constraints and number of variables

  cb_1 = np.ones((m, 1))*1000000                 #creating the objective function coefficient to the new artificial variables (BIG M), as much as we have constaints
  cn_1 = c
  c_1 = np.vstack((cn_1, cb_1))                  #stacking the original c with the BIG M
  XB_1 = b                                       #giving the artificial variables the value of b as an initial basic variables
  XN_1 = np.zeros(((n), 1))                      #giving the rest of the variables a value of 0 as the non-basic variables

  IB_1 = np.zeros((1, m))
  for i in range(m):
    IB_1[:, i] = i+n+1                #storing the basic variables indeces, starting from the number of the last original variable and increasing as much as we have constaints

  IN_1 = np.zeros((1, n))
  for i in range(n):
    IN_1[:, i] = i+1                               #storing the non-basic variables indeces, as much as the original variables.

  identity = np.eye(m)
  A_1 = np.hstack((A, identity))                 #adding an identity matrix with the size of constraints as the A coeffecient of the artificial variables
```

```
XB, IB, XN, IN = Simplex_Method(XB_1,IB_1,XN_1,IN_1,c_1,A_1,b)          #calling the simplex method fucntion to  solve our LP

for IB_index in range(IB.size):
  if int(IB[:,IB_index]) > n:
    if XB[IB_index,0] == 0:
      for IN_index in range(IN.size):
        if int(IN[:,IN_index]) <= n:
          IB[:,IB_index] = int(IN[:,IN_index])    #if we have an artificial variable with a 0 value in our basic variables, it shall be replaced with a non-basic variable
          IN = np.delete(IN, IN_index, axis=1)
          XN = np.delete(XN, IN_index, axis=0)
          break
    else:                                         #if we have an artificial variable with a non 0 value then either infeasible or the BIG M was not big enough
      XB = 'infeasible or need bigger M'
      XN = 'infeasible or need bigger M'

for IN_index in range(IN.size-1, -1, -1):         #removing the artificial variables and there indeces from IN and XN
  if int(IN[:,IN_index]) > n:
    IN = np.delete(IN, IN_index, axis=1)
    XN = np.delete(XN, IN_index, axis=0)

return XB,IB,XN,IN
```

6.2.2 Simplex Method Code

```
import numpy as np
import pandas as pd
import time

#defining the simplex method, It can be used to solve any LP.
#it takes 5 matrices: the basic variables, indeces of the basic matrices, non-basic variables, Indeces of non-basic varibales, Objectve function coefficients, Constraints coefficients
#the constraints RHS values are not needed during the iterations, but were used for debugging.
def Simplex_Method(XB,IB,XN,IN,c,A,b):
  while True:

    m,n = A.shape                        #storing the shape of A, which indicates the number of constraints and number of variables (including slacks and artificial variables)

    B = np.zeros((m, m))                 #storing the current iteration B matrix based on the indeces of the basic variables, stacked column by column from matrix A
    for i in range(m):
      B[:, i] = A[:, int(IB[:,i]-1)]

    cB = np.zeros((m, 1))                #storing the current iteration cB column based on the indeces of the basic variables, stacked row by row from matrix c
    for i in range(m):
      cB[i, :] = c[int(IB[:,i]-1), :]

    pi = np.linalg.solve(B.T, cB)        #computing pi values using linalg function which solves a set of linear equations instead of getting the inverse of the a matrix

    N_loc = 0                                 #initializing the location of each non-basic variable we got through
    for j in range(n-m):                      #going through the non basic variables indeces
      cN = float(c[int(IN[:,j]-1),:])         #getting the c value for the current non-basic variable
      NN = A[:, int(IN[:,j]-1)].reshape(m,1)  #getting the A matrix column associated with the current non-basic variable
      r = cN - (pi.T)@NN                      #Calculating the reduced cost for the non-basic variable
      if r < 0:                               #checking if r is negative
        Vin_enters = int(IN[:,j])             #storing the index of the current non-basic variable with the negative r as the chosed variable to enter
        break                                 #since the indeces are sorted, no need to check for more negative r values. following bland's rule by taking the variable with the smallest index
      else:N_loc +=1                          #updating the location

    if N_loc == (n-m):                  #if all r were positive (aka we went over all the non basic variables without getting a negative r), we're optimal, stop iterating
      break

    d_1 = np.linalg.solve(B, -NN)               #computing d values using linalg function
    e = np.zeros(((n-m), 1))                    #getting the e vector as all 0, same number as the non-basic variables
    e[N_loc, :] = 1                             #assigning 1 to the location of the chosen non-basic variable to enter the basic set
    d_2 = np.vstack((d_1, e))                   #stacking d and e together (for demonstration and debuging only, was not used in the code)

    #initializing Alpha, instead of using a random value, so we can start comparing the alpha values and get the smallest one
    alpha_index = 0
    for d_trial in d_1:                          #going through the values of d - basic
      d_trial = float(d_trial)
      if d_trial < 0:
        alpha = (-1*(float(XB[alpha_index]))/d_trial)+1     #initializing alpha as the first value that we got + 1
        break
      alpha_index += 1

    #checking boundedness and calculating alpha
    d_loc = 0                                           #initializing the location variable
    positive_d = 0                                      #initialising the number of positive d
    for d in d_1:                                       #going throuhg the d values
      d = float(d)
      if d < 0:                                         #checking if d is negative
        alpha_New = (-1*(float(XB[d_loc])))/d           #computing Alpha for each d value < 0
        if alpha_New < alpha:                           #updating alpha to the new smaller alpha
          alpha = alpha_New
          Vin_exits = int(IB[:, d_loc])                 #storing the indix of the variable with the smallest alpha to be removed from the basic variable set
          V_exits_loc = d_loc                           #storing the location of the leaving variable
        d_loc += 1                                      #updating the location
      else:
        positive_d +=1                                  #updating the number of positive d
        d_loc += 1                                      #updating the location
```

```
  if positive_d == d_loc:                          #checking if all d is positive (unbounded)
    XB = 'LP Unbounded'
    XN = 'LP Unbounded'
    IB = 'LP Unbounded'
    IN = 'LP Unbounded'
    break

  XB_new = XB + (alpha * d_1)                      #calculating the new XB
  X_exits = XB_new[V_exits_loc]                    #storing the value of the variable that left (for demonstration and debuging only, was not used in the code since it's 0)

  XN_new = XN + (alpha * e)                        #calculating the new XN
  X_enters = float(XN_new[N_loc])                  #storing the value of the variable that entered

  XB = XB_new
  XB[V_exits_loc] = X_enters                       #replacing the value of the variable that left with the one that entered
  IB[0,V_exits_loc] = Vin_enters                   #replacing the index of the variable that left with the one that entered
  XB = XB[np.argsort(IB)].reshape(m,1)             #sorting XB to get in order based on the indices in IB
  IB = np.sort(IB)                                 #sorting IB to get in order

  XN = XN_new
  XN[N_loc] = 0                                    #replacing the value of the variable that entered with 0
  IN[0,N_loc] = Vin_exits                          #replacing the index of the variable that entered with the one that left
  IN = np.sort(IN)                                 #sorting IN to get in order

return XB, IB, XN, IN
```

6.2.3 Explanation our Simplex Method Code

Once we implemented our scenario generation code, our first approach to solving our linear program was to use the simplex method. In order to generate an initial basic feasible solution, our final approach was to use the Big M method. This would solve the linear program along with generating our initial feasible solution using artificial variables.

Initializing our big M method:

In order to initialize using the big M method to solve our linear program, we need to update our c column vector as well as our A matrix.

1. We defined a function in python that takes the input of A, b and c columns that we generated from our scenario generation code described in section 4.1 of the report.

2. First we defined our m and n variables using the shape function in python to compute the number of rows and columns respectively of our newly generated A matrix that we formed in section 4.1

3. Our column vector c would change. In order to incorporate this change, we used the NumPy ones function to generate m elements of c, each with a coefficient of 1. We then multiped all the

elements of this array to our decided value of M. The last step was then vertically stacking these with the c that we inputted in the function. This stacked c will be called c_1.

4. We then assigned our inputted b to the variable XB_1 to set the value of artificial variables to the value of b, which will have m components. The remaining n variables (our non-basic variables) were all set to zero using the zeros function in NumPy with n as our parameter and named XN_1.

5. Our A matrix would need to be updated as we now have m (number of rows of our scenario generated linear program) artificial variables. Therefore, we created an identity matrix using the eye function in python with m as our parameter and them using NumPy hstack, stacked them horizontally to get our new A matrix. This will be called A_1.

6. In order to assign the values of the indices of both the non-basic and basic variables we run each of them through a for loop. For the basic variables, the indices will be assigned to IB_1 a number of n+1 for each m (our range) as the artificial variables will be the last m variables which will be basic. For the non-basic variables, we loop through each in range of n and assign them an index of 1 to n and assign them to an index (IN_1)

5. The simplex method that is described in the next section, is called and the variables of b that was inputted in section 4.1, c_1, XB_1, XN_1, A_1, IB_1 and IN_1 found in steps 3,4,4,5,6,6 will be used as inputs for the simplex function.

Describing the logic behind our simplex method code:

The following steps were used in implementing the simplex method in python.

1. We recomputed our value of m and n by re-calculating the shape of the new A matrix obtained by initializing the big M method.

2. We started off by extracting our initial Basis matrix B, by extracting the columns of the inputted A from the matrix associated with the index of the basic variables. The same was done in extracting the row of the column of c associated with the basic variables. They were named B and cB respectively.

3. We computed the π (dual) values using linalg.solve (numpy libraries' system of linear equations solver) with our extracted Basis matrix in step 2 and extracted c column in step 2.

4. A variable called N_loc was initialized. Then a for loop was initiated for the range of all non-basic variables n – m, In the loop, the non-basic coefficients cN were extracted from the c column that was inputted in the function using the index of the non-basic variables inputted in the function. The same is done with the column of each non basis matrix from A using the index that we are iterating over. The reduced cost is then calculated.

5. If, when calculating our reduced costs, we went over all the non-basic variables and did not find any non-basic variable with a negative reduced cost (computed by checking if our variable N_loc has a value of m-n meaning it ran all the way to the end of the non basic variables), we would break the loop and return our optimal values.

6. Also, we assign a value to a variable V_in enters, which will be the integer value of the index that is associated with our entering non basic variable. This will be the index of the first non basic variable that has a negative reduced cost.

6. If a non-basic variable with a negative reduced cost was found in step 4, we would continue our function by calculating the column vector d, using linalg.solve with our Basis Matrix found in step 1 and Non Basic Column extracted in step 4 of the first variable that has a negative reduced cost.

7. Then we made a new column vector of all value of 0 called e, using the NumPy zeroes function with n-m as our parameter. Using the value of N_loc which would be computed in step 4 when we found the first non-basic variable with a negative value, we assign a one to this position in our generated array e Then in order to get our final direction we vertically stacked the direction obtained in step 6 with this new column to get our final direction.

8. We then initialized alpha by calculating the first alpha by using a for loop which goes into the range of the total elements of the direction calculated in step 6 and finds the first negative direction. It then uses that index of the range for XB that was inputted in the function to extract XB. Then alpha is calculated.

9. We initialize two variables d_loc and positive_d. We enter another for loop for the same range as step 8 and then compute alpha using the same method as we did for step 8. However, now alpha computed will be stored under alpha_new. If this value is less than alpha, we update the variable alpha. Whenever alpha new is updated to alpha, we update a variable called Vin_exits

and V_exits_loc which represent the integer value of the index of the basic variable that exits and the location of the basic variable that exits respectively.

9. We initialized two direction indexes, d_positive and d_loc. Subsequently, we looped through the other elements of the direction generated in step 6. If the direction was positive, d_positive index was updated. At the end of each iteration d_loc were updated to move to the new iteration.

10. If the d_positive and d_loc are equal, it meant all directions were positive. The function would return a string that says the function is unbounded and our program ends.

11. If we have a value for alpha in step 9, it means that our problem is bounded. Current XB is updated using d_1 and alpha calculated in steps 6 and 9 respectively. Current XN is updated using e and alpha calculated in step 7 and 9 respectively.

12. The value of XN that is entering is calculated using e and alpha in steps 6 and 9 respectively. The entering variable has location index of N_loc.

13. The location for XB that was exiting was was computed in step 9 and is represented by V_Exits_loc. XN in the location N_loc takes the place of XB in index V_Exits_loc and is assigned an integer index of V_in enters.

14. In the location N_loc, the value of N takes a value of 0 now that the exiting variable XB has entered and the same location N_loc is now assigned a value of the integer index V_exits_loc that was calculated in step 9 for the exiting basic variable.

15. Lastly, XB is reordered in the same order as the indices that have entered the basis matrix. We used argsort to order the basic variables in the same order as the current IB indices and then we sort it using in ascending order using the np.sort function to implement blands rule in the next iteration.

16. The same is done for XN, however, argsort is not required for this step as all the Xns are 0s. Therefore, we only sort the indices of the non basic variables using np.sort.

17. The iterations start over from the 1st step till all the reduced costs are positive and the simplex loops break.

Ending steps after the solution is outputted:

After we break the simplex method loop, we must account for the artificial variables as we solved the Big M method.

1.We enter a for loop, which iterates through the range of IB and checks each value. If the index of any of the IB variables is greater than or equal to n, its index is an artificial variable.

2. If the artificial variable is a 0 we enter another loop which loops over the indices of the range of indices for non-basic variables and if the index of the non-basic variable is less than or equal to n, the non-basic variable corresponds to a variable that is not an artificial variable.

3. This variable enters the basic variable set in the loop, where the index for the basic variable which had the artificial variable is now changed to the index of the non-basic variable that enters. The non basic variable value and the position of this non basic variable is deleted from the variable set.

4. If the artificial variable is positive, then XB and XN will be assigned a string saying that the problem is either infeasible or we need to input a bigger M to make the function work.

5. We then enter our final for loop, where we iterate over the ranges of the non-basic variables. If any of the non-basic variables have an index of artificial variables, that elements gets deleted and the index of that non basic variable also gets deleted.

6. Then the function returns our final solution of XB, IB, XN, IN.

6.3 Code for Linprog implementation

```
start_time = time.time()
result = linprog(c, A_eq=A, b_eq=b, method='simplex') #using linprog to solve our linear program
end_time = time.time()
elapsed_time = end_time - start_time
```

6.4 Code for Interior Point Method (Big M and the interior point algorithm.)

6.4.1 Code for BIG M Method for Interior Point Method

```
import numpy as np
import pandas as pd
import time
def solgenerator(A_o, b, c_o, M):

  m,n = A_o.shape         #find the shape of the A matrix and assign the number of rows and columns to m and n

  x_o = np.ones((n,1))        #assign a value of 1's as this will be our initial solution for all the initial x's

  A_I = np.eye(m)         #create an identity matrix that is m*m. This will be used to make the new A matrix for the artificial variables.

  Coeff = (b - (A_o@x_o))  #As per the big M find the coefficeints of the new A matrix associated with the artifical variables.
  A_art = A_I*Coeff        #Assign the value of the coefficients to the A matrix of the artificial variables by multiplying them with the identity matrix of these new variables.

  A = np.hstack((A_o,A_art)) #Get the new A matrix by horizontally combining the A matrix of the original variables with the A matrix of the artifical variables.
  x_a = np.ones((m,1))          #Set all the artificial variables to have value of 1, as we do for the BIG M method for interior point.
  x = np.vstack((x_o,x_a))   #Get all of our values of x together in one column, including the 1's we assigned to the artificial variables.

  c_art = (np.ones((m, 1))*M) #find the coefficients of the c values in the objective function by multiplying it with an assigned Big M in the function for the

  c = np.vstack((c_o,c_art))  #make a final c row array that has all the values of c associated with the original variables and the artificial variables

  return A, b, c, x
```

6.4.2 Code for Interior Point Method

```
def interiorpoint(A, b, c, X, e):
  while True:
    Xk = np.eye(X.shape[0])*X    #forming a diagonal matrix with the initial solution we got from the solution generator
    y = np.ones((X.shape[0],1))   #assigning the initial value of y that will be all 1s

    w_1 = A @ Xk
    w_2 = w_1 @ Xk
    w_3 = w_2 @ c
    A_T = A.T
    w_4 = w_2 @ A_T

    w = np.linalg.solve(w_4, w_3) #solving for the dual values using linalg solve
    r = c - A_T @ w                    #solving for the reduced costs

    yXr_1 = y.T @ Xk
    yXr = yXr_1 @ r #solving for the dual gap to check optimality conditions

    if abs(yXr) <= e: # Optimality check
      X_optimal = X
      OF = c.T @ X_optimal
      return X_optimal, OF

    d_y = -Xk @ r #if we are not at optimality, we compute the direction for y

    alpha = 1000000 #Initializing the alpha values

    for d in d_y: #unboundedness checking and generation of step length
      if d < 0: #we compute the step length only for the direction items that have a negative component associated with it.
        alpha_new = -0.5/d
        if alpha_new < alpha:
          alpha = alpha_new
    X = X + alpha * (Xk @ d_y) #next x_point computation
```

6.4.3 Explanation of our code for the Interior Point Method

In order to generate an initial feasible solution to be used in the interior point algorithm, we used the BIG M method where all values of x and artificial variables x take on a value of 1.

The following steps highlights how we achieved this in code:

1. We created a function that takes A, b and values generated in section 4.1 and an M value of 1000000. (The reasoning behind this M value as the same as described in step 2 Big M of Simplex in section 4.2.)

1. We assigned the number of rows and columns of our inputted A matrix to m and n variables.

2. Initially the value of all x's were set to 1 for n variables using np.ones.

3. An identity matrix of size m was constructed which will be our initial matrix for the artificial variables as we have would have m constraints, where the ones will take a different value as shown in the next step.

4. We found the coefficients of the artificial variables using numpy matrix multiplication of the inputted A and x that was generated in step 2. We subtract this column vector generated from inputted b to get our coefficient values for our artificial variables.

5. We used regular multiplication of the identity matrix generated in step 3 and the column vector generated in step 4 to get our new A_art matrix for the artificial variables. This was horizontally stacked with the A matrix inputted using hstack.

6. We created an array of m variables using np.ones which would be the initial values of our x variables. This was then vstacked with the array generated in step 2 to get our initial feasible solution for the interior point algorithm.

7. To get our updated c values, we used the same strategy as we did for generating an initial feasible solution for the artificial variables and then multiplied the inputted M to each one of these elements (they have a value of 1) in the generated c array for the artificial variables. Then this vstacked with out inputted c in the function.

8. Finally, the function returns the values A, b, c and x that will be used for our interior point algorithm.

Explaining our code for the interior point method:

Implementing the code for interior point was simpler than the simplex method as we did not need to rearrange our indexes and variables after each iteration. The following steps describe our approach to coding the interior point method:

1. Created a function that takes A, b, c and x and as our inputs which we generate from the previous section and an epsilon which we use as 0.01. This would allow the interior point to converge quicker while not sacrificing accuracy.

2. We got our diagonal matrix Xk by using the numpy eye to create an identity matrix of size X[0] which is the number of rows of our input x obtained in step 6 of the previous section. This will be multiplied with the X values that are inputted or obtained in step " " to get Xk.

3. Subsequently, we set our initial value of the scaled LP which would be a vector of all 1s and name this variable y. This would be the same for each loop and hence we use the the np.ones function to generate an array of all 1^s based on the number of rows of our input X.

3. We transposed the inputted A and stored it as a value as it does not change with each iteration. linalg solve was used to compute dual values by using the inputs c and A in our function and calculating variables w1, w2, w3 and w4.

4. Reduced costs were then calculated using the numpy matrix operations where the transposed A in step 3 was matrix multiplied with the dual values obtained in step 3 and subtracted from the c column that was entered as an input in the function.

5. Optimality was checked by using the y obtained in step 3, by transposing it and matrix multiplying it with the diagonal matrix obtained in step 2. This value was then multiplied with the reduced cost column obtained in step 4.

6. If optimality gap was less than our required epsilon that was inputted in the function, we would return our optimal x that was the solution that was either inputted or x solution that would be generated in step " ".

7. If our dual gap is not less than the required epsilon, we computed the direction by numpy matrix multiplying the – ve of the diagonal matrix obtained in step 2 with the reduced costs obtained in step 4 and name this d_y.

8. We choose our value of alpha to be 0.5 for all iterations to compute the value of alpha (k). We found that using trial and error, if the alpha value was too high, the accuracy of the optimal solution answers was being compromised. With an alpha value of 0.5, the algorithm was a bit slower but it would always lead to the most accurate answers.

9. We then initialized alpha the same way we initialized alpha for the simplex. We calculated the first alpha for the first negative direction by loo ping through each element of the direction vector found in step 7 and storing this value.

10. We then looped through all the elements of the d_y column that we generated, and if a d was negative, we computed a alpha_new. If this new alpha was lower than the alpha value calculated in step 9, the value would be stored in the variable alpha, otherwise we do not change the value of the alpha variable.

11. We then calculate the next x value for our next iteration, which will be computed using values of Xk, d_y and alpha obtained in steps 2, steps 7 and 10 respectively. We then continue the loop from the first iteration till the function returns a value.

6.5 Tables for data generated for each scenario for each algorithm. (all for 0.36% increment)

6.5.1 Tables Simplex Method

Scenarios number	Objective Function Values	Time of Computing
1	-118600	0.003801346
5	-118262.6969	0.022854328
9	-117978.5446	0.054746866
13	-117702.2689	0.1073277
17	-117422.4902	0.232102156
21	-117149.3214	0.302447319
25	-116875.1444	0.436136961
29	-116600.0568	0.590668917
33	-116329.8784	0.747664928
37	-116059.056	1.200236559
41	-115788.159	2.665186167
45	-115519.9655	2.634509087
49	-115252.7143	2.495428085
53	-114985.8928	2.266235352
57	-114719.5315	2.814376354
61	-114455.7874	6.779640675
65	-114192.8488	4.616142273
69	-113930.4362	4.896131992
73	-113669.0181	8.802405119
77	-113408.7316	7.306910276
81	-113150.3528	10.11684036
85	-112892.747	11.58025408
89	-112636.1747	12.65902209
93	-112380.5093	13.92453837
97	-112126.002	14.76356053
101	-111872.4657	16.25335765
105	-111620.5429	18.16031885
109	-111369.7861	26.68892741
113	-111120.1376	21.72926831
117	-110871.5976	26.07747293
121	-110624.1078	27.81564164
125	-110377.8582	29.84041476
129	-110132.7053	32.77152014
133	-109888.667	35.17418861

137	-109645.8788	39.3356266
141	-109404.2234	42.94695115
145	-109163.7036	45.24280334
149	-108924.4064	48.44955993
153	-108686.2871	56.60968089
157	-108449.3149	69.90301394
161	-108213.5019	68.54078889
165	-107978.8868	73.22747207
169	-107745.4535	70.50411248
173	-107513.1805	75.86663079
177	-107282.0498	85.04390311
181	-107052.0765	83.37924623
185	-106823.2423	89.39170671
189	-106595.5667	100.0136504
193	-106369.0585	97.29019785
197	-106143.9508	105.7214539
201	-105919.9816	111.9418867
205	-105697.1464	117.0731018
253	-103112.8449	210.9245594
301	-100692.7284	364.119457
353	593.7706518	-98214.78941
401	889.9442563	-96000.49266
453	1457.932241	-93665.6459
501	1963.798057	-91558.30785
601	3496.286156	-58072.29424

6.5.2 Interior Point Method solution

Scenarios number	Objective Function Value	Time of Computing
1	-118599.9945	0.00696969
5	-118262.6967	0.039410591
9	-117978.5439	0.089310408
13	-117702.268	0.174672842
17	-117422.477	0.266320229
21	-117149.3074	0.423526287
25	-116875.1447	0.601302624
29	-116600.0534	0.897898912
33	-116329.8899	1.252330065
37	-116059.0598	1.682071447
41	-115788.1655	3.706790686
45	-115519.9368	2.975915909
49	-115252.71	3.647416592

53	-114985.8692	6.307530403
57	-114719.5181	7.557775021
61	-114455.8058	8.936638117
65	-114192.7671	9.910642862
69	-113930.4197	12.64263868
73	-113668.9986	15.19316888
77	-113408.4257	18.96725535
81	-113150.3382	21.73653388
85	-112892.6831	26.4628346
89	-112632.9178	29.05319953
93	-112380.5412	34.09995413
97	-112125.9865	40.38836408
101	-111872.5344	46.55273151
105	-111620.4475	54.36412239
109	-111369.0275	59.73673987
113	-111118.95	68.50637197
117	-110871.4773	80.2618103
121	-110622.5879	85.64042902
125	-110375.7499	98.71567369
129	-110130.7705	109.0725374
133	-109886.6073	122.3604228
137	-109643.8663	136.0035284
141	-109396.1934	145.0339146
145	-109146.7454	156.1343832
149	-108922.4284	191.9791274
153	-108684.288	214.7828333
157	-108408.1026	201.1870408
161	-108198.955	240.1271672
165	-107977.1614	286.7534595
169	-107743.9395	310.798085
173	-107512.004	346.7908499
177	-107277.1868	351.5185537
181	-107041.5768	379.1597807
185	-106819.6552	415.9144878
189	-106592.8006	466.74209
193	-106366.3863	497.1700492
197	-106141.6133	541.5873556
201	-105913.9099	573.0194662
205	-105695.5542	620.7653856
253	-103109.6371	1359.118699
301	-100687.8992	2626.46198

6.5.3 Table of generated values for linprog

Scenarios number	Objective Function Value	Time of Computing
1	-118600	0.004739761
5	-118262.6969	0.012312412
9	-117978.5446	0.024893522
13	-117702.2689	0.04105258
17	-117422.4902	0.062646866
21	-117149.3214	0.133086443
25	-116875.1444	0.156049967
29	-116600.0568	0.198174953
33	-116329.8784	0.224866152
37	-116059.056	0.276377916
41	-115788.159	0.298697948
45	-115519.9655	0.348191977
49	-115252.7143	0.409965277
53	-114985.8928	0.524421692
57	-114719.5315	0.616081476
61	-114455.7874	1.375943899
65	-114192.8488	1.541180849
69	-113930.4362	1.638776064
73	-113669.0181	1.174001932
77	-113408.7316	1.318868399
81	-113150.3528	1.661393642
85	-112892.747	1.719643831
89	-112636.1747	1.951741695
93	-95193.84871	1.343654633
97	-112126.002	3.671552658
101	-111872.4657	3.081896067
105	-90558.05805	1.845370293
109	-111369.7861	3.446776152
113	-111120.1376	4.452422857
117	-85643.38893	3.209437132
121	-110624.1078	4.154047728
125	-109804.1651	4.735976458
129	-80432.7792	4.412416697
133	-105595.8008	4.74654603
137	-104031.5955	4.927176714
141	-74905.09584	5.235765934
145	-101290.9137	5.74646616
149	-99180.1557	7.297511101
153	-96862.61508	6.035586357

157	-94501.41235	7.804136038
161	-92094.87247	6.190003395
165	-90031.99844	7.850985527
169	-85751.6373	7.692865849
173	-58982.36055	7.592777252
177	-59656.49914	6.609559059
181	-66851.81286	8.113603592
185	-76413.50354	6.633782625
189	-83224.47499	8.631051064
193	-83450.16515	7.9360919
197	-83933.96312	8.598524809
201	-84424.23878	9.473888159
205	-84927.57963	8.497603416
253	-90235.1258	14.05615139
301	-75982.92522	15.70628452
353	-27959.67025	23.31602502
401	-53864.65647	25.58690715
453	-54476.12687	32.76122808
501	-55887.26977	40.06821489
601	-60889.8	76.83692

Appendix Complete