Wong Zhing Jieh Jack 201910656

Sub-project 1 (one-phase simplex method)

In the implementation of bland(D,eps), the choice of entering variable is made based on the index – the first encountered entering variable. It is decided at this point that the LP is optimal (a termination criterion) if none of the encountered variable is positive.

The choice of leaving variable is made based on the variable with least negative corresponding ratio. The tie between multiple such variables is broken by choosing the first encountered one. At this point, if all values in the column are non-negative, the LP is set to be unbounded (another termination criterion).

Sub-project 2 (experimentation)

The experimentation to evaluate the performance between data types Fraction and np.float64 for bland's rule is set up as such:

number of decision variables, n: 200
number of constraints, m: 200
coefficients in objective function: a list of numbers randomly generated ranging [-100,100)
coefficients in the m by n matrix: a matrix of numbers randomly generated ranging [-20,100)
right-hand side of constraints, b: a list of numbers randomly generated ranging [300,1000)

The time it takes to run the algorithm to completion is chosen as the performance metric. To be more precise, the timing is calculated as the average of three iterations of execution of the algorithm (There is no particular reason for the magic number three, it may just be a lucky number).

I also ran scipy.optimize.linprog alongside for the same setup.

The experiment is then repeated three times (again, might just be a lucky number).

Results:

Data type	Fraction	np.float64	scipy.optimize.liprog
Average time experiment 1 (ms)	76387.3	424.34	30.96
Average time experiment 2 (ms)	76097.95	426.16	31.05
Average time experiment 3 (ms)	77471.22	406.26	30.84
Average time of all three experiments (ms)	76652.16	418.92	30.95

Observation and discussion:
Based on the above experimentations, clearly the “bland’s” algorithm takes the longest time to run for datatype Fraction, more than 180 times slower than datatype np.float64. This could be due to the Fraction has two parts to process - numerator and denominator whereas np.float64 does not keep the integer parts but only the approximate value for the number, while losing some precision. Meanwhile, scipy.optimize.linprog takes the shortest amount of time to run for the same input, at 13.5 times faster than my “bland’s” algorithm on datatype np.float64.

Side note: The correctness of the implementation is tested based on all examples given, including those requiring two-phase simplex method.

Sub-project 3 (two-phase simplex method)

The two-phase simplex method is implemented using the auxiliary approach. The first step to this feature is having a utility function that checks if auxiliary problem is required. In my code, such function is creatively called isAuxiliaryRequired() which simply checks if any of the RHS values of the constraints is negative.

If auxiliary problem is to be introduced, this means the current dictionary is infeasible. Hence another utility function, makeFeasible() is introduced to convert the current dictionary into a feasible one. This is done by pivoting x_0 (auxiliary variable) with the most infeasible variable as the leaving variable.

Now that the dictionary is feasible, the simplex method is applied on the dictionary in a hope to obtain a feasible solution with $x_0 = 0$, based on the fact that the original problem has a feasible solution if and only if the optimal solution to the auxiliary problem has objective value zero. It is important to note that in my implementation, at each pivoting process, the objective function of the original problem is also being processed.

Sub-project 4 (pivoting rules and further experimentation)

The largest-coefficient pivoting rule is implemented by selecting entering variable with the largest positive coefficient. The choice of leaving variable is made based on the variable with least negative corresponding ratio.

The largest-increase pivoting rule is implemented by selecting the pair of entering and leaving variables that result in the largest increase in current objective function value.

The experimentation to evaluate the performance between data types Fraction and np.float64 for the two pivoting rules is set up as such (similar to the previous experiment):

number of decision variables, n: 200
number of constraints, m: 200
coefficients in objective function: a list of numbers randomly generated ranging [-100,100)
coefficients in the m by n matrix: a matrix of numbers randomly generated ranging [-20,100)
right-hand side of constraints, b: a list of numbers randomly generated ranging [300,1000)

Wong Zhing Jieh Jack 201910656

The time it takes to run the algorithm to completion is chosen as the performance metric. To be more precise, the timing is calculated as the average of three iterations of execution of the algorithm.

The experiment is then repeated three times.

Results:

Data type	Fraction		np.float64		
Pivoting rule	Largest coefficient	Largest increase	Largest coefficient	Largest increase	scipy.optimize.liprog
Average time experiment 1 (ms)	17354.72	16696.84	121.89	686.5	30.96
Average time experiment 2 (ms)	17297.89	16442.79	121.59	715.13	31.05
Average time experiment 3 (ms)	17393.69	16458.32	118.74	715.79	30.84
Average time of all three experiments (ms)	17348.77	16532.65	120.74	705.8	30.95

Observation and discussion:
Based on the experiments, the pivoting rule largest-coefficient takes longer to run than largest-increase pivoting rule for datatype Fraction, but the opposite result is observed for datatype np.float64. Looking at the previous experiment in subproject 2, Bland's rule takes the most amount of time. This could be due to Bland's rule incurring more pivot iterations.

Comparing between the speed for the different datatypes, performing the simplex algorithm on Fraction is generally slower than on np.float64. This could be due to the GCD computation done on fractions during each pivot operation.

Even though largest-coefficient appears slower for datatype Fraction, it cannot be concluded that it is the case in general. In fact, when run on another set of data, largest-coefficient is observed to be faster. This result can be observed in the next sub-project.

Wong Zhing Jieh Jack 201910656

Sub-project 5 (integer pivoting)

Integer pivoting is implemented as described in Lecture 5. The experiment faces a unique restriction due to the increasing size of coefficients in the matrix resulted from integer pivoting. As a result the experiment is constructed on a smaller LP model:

number of decision variables, n: 20
number of constraints, m: 20
coefficients in objective function: a list of numbers randomly generated ranging [-100,100)
coefficients in the m by n matrix: a matrix of numbers randomly generated ranging [-20,100)
right-hand side of constraints, b: a list of numbers randomly generated ranging [300,1000)

The time it takes to run the algorithm to completion is chosen as the performance metric. To be more precise, the timing is calculated as the average of three iterations of execution of the algorithm.

The experiment is then repeated three times.

Results:

Data type	Integer			Fraction		
Pivoting Rule	Bland's	Largest coefficient	Largest increase	Bland's	Largest coefficient	Largest increase
Average time exp1 (ms)	4.14	1.64	2.59	25.47	6.13	11.01
Average time exp2 (ms)	3.99	1.73	2.66	25.3	6.06	11.01
Average time exp3 (ms)	3.96	1.57	2.46	25.94	6.08	11.65
Average of all three (ms)	4.03	1.65	2.57	25.57	14.22	11.22

Observation and discussion:
Within each of the datatypes, Bland's rule turns out to be the slowest among the rest. This could be due to more pivoting iterations are needed before the algorithm terminates.
For both data types, largest-increase pivoting rule takes more than 1.5 times longer for the algorithm to terminate than largest-coefficient rule. This is, by theoretical reasoning, due to the time-consuming process to pick the entering-leaving variable pair for largest-coefficient.

Another observation is be obtained from the experimentation, that is – running the simplex algorithm on integer takes less time than running on Fraction, for every pivoting rule. This

confirms the theorical fact that for fraction, a large part of the computation time is spent on computing GCD (to simplify the fraction) for each pivoting operation.

Side note: The correctness of the implementation is tested based on all examples given, including those requiring two-phase simplex method.