11 The Simplex Method

Lab Objective: *The* Simplex Method *is a straightforward algorithm for finding optimal solutions to optimization problems with linear constraints and cost functions. Because of its simplicity and applicability, this algorithm has been named one of the most important algorithms invented within the last 100 years. In this lab, we implement a standard Simplex solver for the primal problem.*

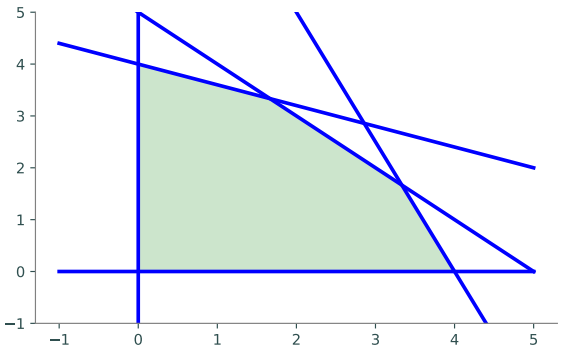
Standard Form

The Simplex Algorithm accepts a linear constrained optimization problem, also called a *linear program*, in the form given below:

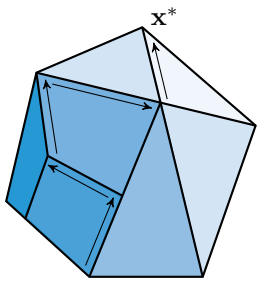
$$\begin{aligned} \text{maximize} \quad & \mathbf{c}^T\mathbf{x} \\ \text{subject to} \quad & A\mathbf{x} \preceq \mathbf{b} \\ & \mathbf{x} \succeq \mathbf{0} \end{aligned}$$

Note that any linear program can be converted to standard form, so there is no loss of generality in restricting our attention to this particular formulation.

Such an optimization problem defines a region in space called the *feasible region*, the set of points satisfying the constraints. Because the constraints are all linear, the feasible region forms a geometric object called a *polytope*, having flat faces and edges (see Figure 11.1). The Simplex Algorithm jumps among the vertices of the feasible region searching for an optimal point. It does this by moving along the edges of the feasible region in such a way that the objective function is always increased after each move.



(a) The feasible region for a linear program with 2-dimensional constraints.



(b) The feasible region for a linear program with 3-dimensional constraints.

Figure 11.1: If an optimal point exists, it is one of the vertices of the polyhedron. The simplex algorithm searches for optimal points by moving between adjacent vertices in a direction that increases the value of the objective function until it finds an optimal vertex.

Implementing the Simplex Algorithm is straightforward, provided one carefully follows the procedure. We will break the algorithm into several small steps, and write a function to perform each one. To become familiar with the execution of the Simplex algorithm, it is helpful to work several examples by hand.

The Simplex Solver

Our program will be more lengthy than many other lab exercises and will consist of a collection of functions working together to produce a final result. It is important to clearly define the task of each function and how all the functions will work together. If this program is written haphazardly, it will be much longer and more difficult to read than it needs to be. We will walk you through the steps of implementing the Simplex Algorithm as a Python class.

For demonstration purposes, we will use the following linear program.

$$\begin{aligned} \text{maximize} \quad & 3x_0 + 2x_1 \\ \text{subject to} \quad & x_0 - x_1 \leq 2 \\ & 3x_0 + x_1 \leq 5 \\ & 4x_0 + 3x_1 \leq 7 \\ & x_0, x_1 \geq 0. \end{aligned}$$

Accepting a Linear Program

Our first task is to determine if we can even use the Simplex algorithm. Assuming that the problem is presented to us in standard form, we need to check that the feasible region includes the origin.

Problem 1. Write a class that accepts the arrays $\mathbf{c}$, A, and $\mathbf{b}$ of a linear optimization problem in standard form. In the constructor, check that the system is feasible at the origin[a]. That is, check that $A\mathbf{x} \preceq \mathbf{b}$ when $\mathbf{x} = \mathbf{0}$. Raise a `ValueError` if the problem is not feasible at the origin.

[a]For now, we only check for feasibility at the origin. A more robust solver sets up the auxiliary problem and solves it to find a starting point if the origin is infeasible.

Adding Slack Variables

The next step is to convert the inequality constraints $A\mathbf{x} \preceq \mathbf{b}$ into equality constraints by introducing a slack variable for each constraint equation. If the constraint matrix A is an $m \times n$ matrix, then there are m slack variables, one for each row of A. Grouping all of the slack variables into a vector $\mathbf{w}$ of length m, the constraints now take the form $A\mathbf{x} + \mathbf{w} = \mathbf{b}$. In our example, we have

$$\mathbf{w} = \begin{bmatrix} x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

When adding slack variables, it is useful to represent all of your variables, both the original primal variables and the additional slack variables, in a convenient manner. One effective way is to refer to a variable by its subscript. For example, we can use the integers 0 through $n-1$ to refer to the original (non-slack) variables x_0 through x_{n-1}, and we can use the integers n through $n+m-1$ to track the slack variables (where the slack variable corresponding to the ith row of the constraint matrix is represented by the index $n+i-1$).

We also need some way to track which variables are *basic* (non-zero) and which variables are *nonbasic* (those that have value 0). A useful representation for the variables is a Python list (or NumPy array), where the elements of the list are integers. Since we know how many basic variables we have (m), we can partition the list so that all the basic variables are kept in the first m locations, and all the non-basic variables are stored at the end of the list. The ordering of this list is important. In particular, if $i \leq m$, the ith element of the list represents the basic variable corresponding to the ith row of A. Henceforth we will refer to this list as the *index list*.

Initially, the basic variables are simply the slack variables, and their values correspond to the values of the vector $\mathbf{b}$. In our example, we have 2 primal variables x_0 and x_1, and we must add 3 slack variables. Thus, we instantiate the following index list:

```
>>> L = [2, 3, 4, 0, 1]
```

Notice how the first 3 entries of the index list are $2, 3, 4$, the indices representing the slack variables. This reflects the fact that the basic variables at this point are exactly the slack variables.

As the Simplex Algorithm progresses, however, the basic variables change, and it will be necessary to swap elements in our index list. For example, suppose the variable represented by the index 4 becomes nonbasic, while the variable represented by index 0 becomes basic. In this case we swap these two entries in the index list.

```
>>> L[2], L[3] = L[3], L[2]
>>> L
[2, 3, 0, 4, 1]
```

Now our index list tells us that the current basic variables have indices $2, 3, 0$.

Problem 2. Design and implement a way to store and track all of the basic and non-basic variables.

Hint: Using integers that represent the index of each variable is useful for Problem 4.

Creating a Tableau

After we have determined that our program is feasible, we need to create the *tableau* (sometimes called the *dictionary*), a data structure to track the state of the algorithm. You may structure the tableau to suit your specific implementation. Remember that your tableau will need to include in some way the slack variables that you created in Problem 2.

There are many different ways to build your tableau. One way is to mimic the tableau that is often used when performing the Simplex Algorithm by hand. Define

$$\bar{A} = \begin{bmatrix} A & I_m \end{bmatrix},$$

where I_m is the $m \times m$ identity matrix, and define

$$\bar{\mathbf{c}} = \begin{bmatrix} \mathbf{c} \\ \mathbf{0} \end{bmatrix}.$$

That is, $\bar{\mathbf{c}} \in \mathbb{R}^{n+m}$ such that the first n entries are $\mathbf{c}$ and the final m entries are zeros. Then the initial tableau has the form

$$T = \begin{bmatrix} 0 & -\bar{\mathbf{c}}^\mathsf{T} & 1 \\ \mathbf{b} & \bar{A} & \mathbf{0} \end{bmatrix} \tag{11.1}$$

The columns of the tableau correspond to each of the variables (both primal and slack), and the rows of the tableau correspond to the basic variables. Using the convention introduced above of representing the variables by indices in the index list, we have the following correspondence:

$$\text{column } i \Leftrightarrow \text{index } i-2, \qquad i = 2, 3, \ldots, n+m+1,$$

and

$$\text{row } j \Leftrightarrow L_{j-1}, \qquad j = 2, 3, \ldots, m+1,$$

where L_{j-1} refers to the $(j-1)$th entry of the index list.

For our example problem, the initial index list is

$$L = (2, 3, 4, 0, 1),$$

and the initial tableau is

$$T = \begin{bmatrix} 0 & -3 & -2 & 0 & 0 & 0 & 1 \\ 2 & 1 & -1 & 1 & 0 & 0 & 0 \\ 5 & 3 & 1 & 0 & 1 & 0 & 0 \\ 7 & 4 & 3 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

The third column corresponds to index 1, and the fourth row corresponds to index 4, since this is the third entry of the index list.

The advantage of using this kind of tableau is that it is easy to check the progress of your algorithm by hand. The disadvantage is that pivot operations require careful bookkeeping to track the variables and constraints.

Problem 3. Add a method to your Simplex solver that will create the initial tableau as a NumPy array.

Pivoting

Pivoting is the mechanism that really makes Simplex useful. Pivoting refers to the act of swapping basic and nonbasic variables, and transforming the tableau appropriately. This has the effect of moving from one vertex of the feasible polytope to another vertex in a way that increases the value of the objective function. Depending on how you store your variables, you may need to modify a few different parts of your solver to reflect this swapping.

When initiating a pivot, you need to determine which variables will be swapped. In the tableau representation, you first find a specific element on which to pivot, and the row and column that contain the pivot element correspond to the variables that need to be swapped. Row operations are then performed on the tableau so that the pivot column becomes an elementary vector.

Let's break it down, starting with the pivot selection. We need to use some care when choosing the pivot element. To find the pivot column, search from left to right along the top row of the tableau (ignoring the first column), and stop once you encounter the first negative value. The index corresponding to this column will be designated the *entering index*, since after the full pivot operation, it will enter the basis and become a basic variable.

Using our initial tableau T in the example, we stop at the second column:

$$T = \left[\begin{array}{c|c|cccc} 0 & -3 & -2 & 0 & 0 & 0 & 1 \\ 2 & 1 & -1 & 1 & 0 & 0 & 0 \\ 5 & 3 & 1 & 0 & 1 & 0 & 0 \\ 7 & 4 & 3 & 0 & 0 & 1 & 0 \end{array}\right]$$

We now know that our pivot element will be found in the second column. The entering index is thus 0.

Next, we select the pivot element from among the positive entries in the pivot column (ignoring the entry in the first row). *If all entries in the pivot column are non-positive, the problem is unbounded and has no solution.* In this case, the algorithm should terminate. Otherwise, assuming our pivot column is the jth column of the tableau and that the positive entries of this column are $T_{i_1,j}, T_{i_2,j}, \ldots, T_{i_k,j}$, we calculate the ratios

$$\frac{T_{i_1,1}}{T_{i_1,j}}, \frac{T_{i_2,1}}{T_{i_2,j}}, \ldots, \frac{T_{i_k,1}}{T_{i_k,j}},$$

and we choose our pivot element to be one that minimizes this ratio. If multiple entries minimize the ratio, then we utilize *Bland's Rule*, which instructs us to choose the entry in the row corresponding to the smallest index (obeying this rule is important, as it prevents the possibility of the algorithm cycling back on itself infinitely). The index corresponding to the pivot row is designated as the *leaving index*, since after the full pivot operation, it will leave the basis and become a nonbasic variable.

In our example, we see that all entries in the pivot column (ignoring the entry in the first row, of course) are positive, and hence they are all potential choices for the pivot element. We then calculate the ratios, and obtain

$$\frac{2}{1} = 2, \quad \frac{5}{3} = 1.66..., \quad \frac{7}{4} = 1.75.$$

We see that the entry in the third row minimizes these ratios. Hence, the element in the second column, third row is our designated pivot element, and our leaving index is $L_2 = 3$:

$$T = \begin{bmatrix} 0 & -3 & -2 & 0 & 0 & 0 & 1 \\ 2 & 1 & -1 & 1 & 0 & 0 & 0 \\ 5 & \boxed{3} & 1 & 0 & 1 & 0 & 0 \\ 7 & 4 & 3 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Problem 4. Write a method that will determine the pivot row and pivot column according to Bland's Rule.

Definition 11.1 (Bland's Rule). *Choose the nonbasic variable with the smallest index that has a positive coefficient in the objective function as the leaving variable. Choose the basic variable with the smallest index among all the binding basic variables.*

Bland's Rule is important in avoiding cycles when performing pivots. This rule guarantees that a feasible Simplex problem will terminate in a finite number of pivots.

The next step is to swap the entering and leaving indices in our index list. In the example, we determined above that these indices are 0 and 3. We swap these two elements in our index list, and the updated index list is now

$$L = (2, 0, 4, 3, 1),$$

so the basic variables are now given by the indices $2, 0, 4$.

Finally, we perform row operations on our tableau in the following way: divide the pivot row by the value of the pivot entry. Then use the pivot row to zero out all entries in the pivot column above and below the pivot entry. In our example, we first divide the pivot row by 3, and then zero out the two entries above the pivot element and the single entry below it:

$$\begin{bmatrix} 0 & -3 & -2 & 0 & 0 & 0 & 1 \\ 2 & 1 & -1 & 1 & 0 & 0 & 0 \\ 5 & 3 & 1 & 0 & 1 & 0 & 0 \\ 7 & 4 & 3 & 0 & 0 & 1 & 0 \end{bmatrix} \to \begin{bmatrix} 0 & -3 & -2 & 0 & 0 & 0 & 1 \\ 2 & 1 & -1 & 1 & 0 & 0 & 0 \\ 5/3 & 1 & 1/3 & 0 & 1/3 & 0 & 0 \\ 7 & 4 & 3 & 0 & 0 & 1 & 0 \end{bmatrix} \to$$

$$\begin{bmatrix} 5 & 0 & -1 & 0 & 1 & 0 & 1 \\ 2 & 1 & -1 & 1 & 0 & 0 & 0 \\ 5/3 & 1 & 1/3 & 0 & 1/3 & 0 & 0 \\ 7 & 4 & 3 & 0 & 0 & 1 & 0 \end{bmatrix} \to \begin{bmatrix} 5 & 0 & -1 & 0 & 1 & 0 & 1 \\ 1/3 & 0 & -4/3 & 1 & -1/3 & 0 & 0 \\ 5/3 & 1 & 1/3 & 0 & 1/3 & 0 & 0 \\ 7 & 4 & 3 & 0 & 0 & 1 & 0 \end{bmatrix} \to$$

$$\begin{bmatrix} 5 & 0 & -1 & 0 & 1 & 0 & 1 \\ 1/3 & 0 & -4/3 & 1 & -1/3 & 0 & 0 \\ 5/3 & 1 & 1/3 & 0 & 1/3 & 0 & 0 \\ 1/3 & 0 & 5/3 & 0 & -4/3 & 1 & 0 \end{bmatrix}.$$

The result of these row operations is our updated Tableau, and the pivot operation is complete.

Problem 5. Add a method to your solver that checks for unboundedness and performs a single pivot operation from start to completion. If the problem is unbounded, raise a `ValueError`.

Termination and Reading the Tableau

Up to this point, our algorithm accepts a linear program, adds slack variables, and creates the initial tableau. After carrying out these initial steps, it then performs the pivoting operation iteratively until the optimal point is found. But how do we determine when the optimal point is found? The answer is to look at the top row of the tableau. More specifically, before each pivoting operation, check whether all of the entries in the top row of the tableau (ignoring the entry in the first column) are nonnegative. If this is the case, then we have found an optimal solution, and so we terminate the algorithm.

The final step is to report the solution. The ending state of the tableau and index list tell us everything we need to know. The maximum value attained by the objective function is found in the upper leftmost entry of the tableau. The nonbasic variables, whose indices are located in the last n entries of the index list, all have the value 0. The basic variables, whose indices are located in the first m entries of the index list, have values given by the first column of the tableau. Specifically, the basic variable whose index is located at the ith entry of the index list has the value $T_{i+1,1}$.

In our example, suppose that our algorithm terminates with the tableau and index list in the following state:

$$T = \begin{bmatrix} 5.2 & 0 & 0 & 0 & .2 & .6 & 1 \\ .6 & 0 & 0 & 1 & -1.4 & .8 & 0 \\ 1.6 & 1 & 0 & 0 & .6 & -.2 & 0 \\ .2 & 0 & 1 & 0 & -.8 & .6 & 0 \end{bmatrix}$$

$$L = (2, 0, 1, 3, 4).$$

Then the maximum value of the objective function is 5.2. The nonbasic variables have indices $3, 4$ and have the value 0. The basic variables have indices $2, 0$, and 1, and have values $.6, 1.6$, and $.2$, respectively. In the notation of the original problem statement, the solution is given by

$$x_0 = 1.6$$
$$x_1 = .2.$$

Problem 6. Write an additional method in your solver called `solve()` that obtains the optimal solution, then returns the maximum value, the basic variables, and the nonbasic variables. The basic and nonbasic variables should be represented as two dictionaries that map the index of the variable to its corresponding value.

For our example, we would return the tuple `(5.2, {0: 1.6, 1: .2, 2: .6}, {3: 0, 4: 0})`.

At this point, you should have a Simplex solver that is simple to use. The following code demonstrates how your solver is expected to behave:

```
>>> import SimplexSolver

# Initialize objective function and constraints.
>>> c = np.array([3., 2])
>>> b = np.array([2., 5, 7])
>>> A = np.array([[1., -1], [3, 1], [4, 3]])
```

```
# Instantiate the simplex solver, then solve the problem.
>>> solver = SimplexSolver(c, A, b)
>>> sol = solver.solve()
>>> print(sol)
(5.200,
 {0: 1.600, 1: 0.200, 2: 0.600},
 {3: 0, 4: 0})
```

If the linear program were infeasible at the origin or unbounded, we would expect the solver to alert the user by raising an error.

Note that this simplex solver is *not* fully operational. It can't handle the case of infeasibility at the origin. This can be fixed by adding methods to your class that solve the *auxiliary problem*, that of finding an initial feasible tableau when the problem is not feasible at the origin. Solving the auxiliary problem involves pivoting operations identical to those you have already implemented, so adding this functionality is not overly difficult.

The Product Mix Problem

We now use our Simplex implementation to solve the *product mix problem*, which in its basic form can be expressed as a simple linear program. Suppose that a manufacturer makes n products using m different resources (labor, raw materials, machine time available, etc). The ith product is sold at a unit price p_i, and there are at most m_j units of the jth resource available. Additionally, each unit of the ith product requires $a_{j,i}$ units of resource j. Given that the demand for product i is d_i units per a certain time period, how do we choose the optimal amount of each product to manufacture in that time period so as to maximize revenue, while not exceeding the available resources?

Let $x_1, x_2, \ldots, x_n$ denote the amount of each product to be manufactured. The sale of product i brings revenue in the amount of $p_i x_i$. Therefore our objective function, the profit, is given by

$$\sum_{i=1}^{n} p_i x_i.$$

Additionally, the manufacture of product i requires $a_{j,i}x_i$ units of resource j. Thus we have the resource constraints

$$\sum_{i=1}^{n} a_{j,i} x_i \leq m_j \text{ for } j = 1, 2, \ldots, m.$$

Finally, we have the demand constraints which tell us not to exceed the demand for the products:

$$x_i \leq d_i \text{ for } i = 1, 2, \ldots, n$$

The variables x_i are constrained to be nonnegative, of course. We therefore have a linear program in the appropriate form that is feasible at the origin. It is a simple task to solve the problem using our Simplex solver.

Problem 7. Solve the product mix problem for the data contained in the file `productMix.npz`. In this problem, there are 4 products and 3 resources. The archive file, which you can load using the function `np.load`, contains a dictionary of arrays. The array with key `'A'` gives the resource coefficients $a_{i,j}$ (i.e. the (i,j)-th entry of the array give $a_{i,j}$). The array with key `'p'` gives the unit prices p_i. The array with key `'m'` gives the available resource units m_j. The array with key `'d'` gives the demand constraints d_i.

Report the number of units that should be produced for each product.

Beyond Simplex

The *Computing in Science and Engineering* journal listed Simplex as one of the top ten algorithms of the twentieth century [**Nash2000**]. However, like any other algorithm, Simplex has its drawbacks.

In 1972, Victor Klee and George Minty Cube published a paper with several examples of worst-case polytopes for the Simplex algorithm [**Klee1972**]. In their paper, they give several examples of polytopes that the Simplex algorithm struggles to solve.

Consider the following linear program from Klee and Minty.

$$\begin{array}{rlllll}
\max & 2^{n-1}x_1 & +2^{n-2}x_2 & +\cdots & +2x_{n-1} & +x_n \\
\text{subject to} & x_1 & & & & \leq 5 \\
& 4x_1 & +x_2 & & & \leq 25 \\
& 8x_1 & +4x_2 & +x_3 & & \leq 125 \\
& \vdots & & & & \vdots \\
& 2^n x_1 & +2^{n-1}x_2 & +\cdots & +4x_{n-1} & +x_n \leq 5
\end{array}$$

Klee and Minty show that for this example, the worst case scenario has exponential time complexity. With only n constraints and n variables, the simplex algorithm goes through 2^n iterations. This is because there are 2^n extreme points, and when starting at the point $x = 0$, the simplex algorithm goes through all of the extreme points before reaching the optimal point $(0, 0, \ldots, 0, 5^n)$. Other algorithms, such as interior point methods, solve this problem much faster because they are not constrained to follow the edges.