

# 1

## 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.*

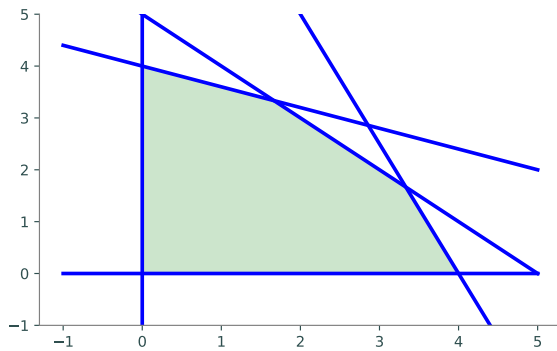
### Standard Form

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

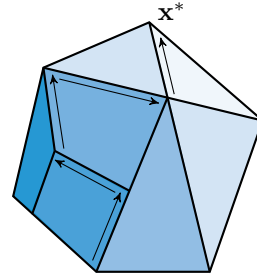
$$\begin{array}{ll} \text{minimize} & \mathbf{c}^\top \mathbf{x} \\ \text{subject to} & A\mathbf{x} \preceq \mathbf{b} \\ & \mathbf{x} \succeq \mathbf{0} \end{array}$$

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 1.1). 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 1.1: 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.

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

## 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. 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.

**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. 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.

## 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  $i$ th row of the constraint matrix is represented by the index  $n + i - 1$ ).

We also need some way to track which variables are *independent* (non-zero) and which variables are *dependent* (those that have value 0). This can be done using the objective function. At anytime during the optimization process, the non-zero variables in the objective function are *independent* and all other variables are *dependent*.

## Creating a Dictionary

After we have determined that our program is feasible, we need to create the *dictionary* (sometimes called the *tableau*), a matrix to track the state of the algorithm.

There are many different ways to build your dictionary. We will do this by setting the corresponding dependent variable equations to 0. For example, if  $x_5$  were a dependent variable we would expect to see a -1 in the column that represents  $x_5$ . Define

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

where  $I_m$  is the  $m \times m$  identity matrix we will use to represent our slack variables, 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 dictionary has the form

$$D = \begin{bmatrix} 0 & \bar{\mathbf{c}}^T \\ \mathbf{b} & -\bar{A} \end{bmatrix} \tag{1.1}$$

The columns of the dictionary correspond to each of the variables (both primal and slack), and the rows of the dictionary correspond to the dependent variables.

For our example the initial dictionary is

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

The advantage of using this kind of dictionary is that it is easy to check the progress of your algorithm by hand.

**Problem 2.** Add a method to your Simplex solver that takes in arrays  $c$ ,  $A$ , and  $b$  to create the initial dictionary ( $D$ ) as a NumPy array.

## Pivoting

Pivoting is the mechanism that really makes Simplex useful. Pivoting refers to the act of swapping dependent and independent variables, and transforming the dictionary 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 dictionary 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 dictionary so that the pivot column becomes a negative 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 dictionary (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 dependent variable.

Using our initial dictionary  $D$  in the example, we stop at the second column:

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

We now know that our pivot element will be found in the second column. The entering index is thus 1.

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

$$\frac{-D_{i_1,0}}{D_{i_1,j}}, \frac{-D_{i_2,0}}{D_{i_2,j}}, \dots, \frac{-D_{i_k,0}}{D_{i_k,j}},$$

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 an independent variable.

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

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

We see that the entry in the third row minimizes these ratios. Hence, the element in the second column (index 1), third row (index 2) is our designated pivot element.

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

**Problem 3.** Write a method that will determine the pivot row and pivot column according to Bland's Rule.

**Definition 1.1 (Bland's Rule).** Choose the independent variable with the smallest index that has a negative coefficient in the objective function as the leaving variable. Choose the dependent variable with the smallest index among all the binding dependent 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. *Hint:* Avoid dividing by zero.

Finally, we perform row operations on our dictionary in the following way: divide the pivot row by the negative 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:

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

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

**Problem 4.** 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 Dictionary

Up to this point, our algorithm accepts a linear program, adds slack variables, and creates the initial dictionary. 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 dictionary, which represents the objective function. More specifically, before each pivoting operation, check whether all of the entries in the top row of the dictionary (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 dictionary and index list tells us everything we need to know. The minimal value attained by the objective function is found in the upper leftmost entry of the dictionary. The dependent variables all have the value 0 in the objective function or first row of our dictionary array. The independent variables have values given by the first column of the dictionary. Specifically, the independent variable whose index is located at the  $i$ th entry of the index list has the value  $T_{i+1,0}$ .

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

$$D = \begin{bmatrix} -5.2 & 0 & 0 & 0 & 0.2 & 0.6 \\ 0.6 & 0 & 0 & -1 & 1.4 & -0.8 \\ 1.6 & -1 & 0 & 0 & -0.6 & 0.2 \\ 0.2 & 0 & -1 & 0 & 0.8 & -0.6 \end{bmatrix}$$

Then the minimal value of the objective function is  $-5.2$ . The independent variables have indices 4, 5 and have the value 0. The dependent variables have indices 3, 1, and 2, 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 5.** Write an additional method in your solver called `solve()` that obtains the optimal solution, then returns the minimal value, the dependent variables, and the independent variables. The dependent and independent 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 ready 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)
(-5.2,
 {0: 1.6, 1: 0.2, 2: 0.6},
 {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 dictionary 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 dependent 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  $i$ th product is sold at a unit price  $p_i$ , and there are at most  $m_j$  units of the  $j$ th resource available. Additionally, each unit of the  $i$ th 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  $x_1, x_2, \dots, x_n$  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

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

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

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

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 6.** 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  $d_i$ .

Return a 1-d numpy array of the number of units that should be produced for each product. (For `productMix.npz`, the function should return an array of length four). *Hint:* Because this is a maximization problem and your solver works with minimizations, you will need to change the sign of the array  $c$ .

## Beyond Simplex

The *Computing in Science and Engineering* journal listed Simplex as one of the top ten algorithms of the twentieth century [?]. 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 [?]. 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.

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

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  $(0, 0, \dots, 0, 5^n)$ . Other algorithms, such as interior point methods, solve this problem much faster because they are not constrained to follow the edges.