# GatorPy: A Custom Implemented Linear Programming Solver

Andres Espinosa
Industrial and Systems Engineering
University of Florida
andresespinosa@ufl.edu

***Abstract*—**

## I. Introduction

GatorPy is a custom Linear Programming (LP) solver implemented entirely in Python. The goal of this project is twofold: First, to serve as an educational tool by providing a clear and simple implementation of an LP solver, exclusively built using Python and NumPy objects. Second, to lay the foundation for an open-source solver that can be enhanced and expanded upon by other UF students, serving as a way for students to dive headfirst into the concepts.

This project report details the implementation of the Simplex algorithm and its modular design, which can easily be built on for future expansions of the solver. The rest of the introduction, section I, serves as a sufficient overview of linear programming, the simplex algorithm, and existing OR solvers. The implementation is detailed in section II, which showcases a few design choices that were made throughout the creation of GatorPy. First, the simplistic modeling syntax is shown, then the specific simplex pseudo-code is explained, then, the Python objects are established that run as the backbone to the project, finally, behind-the-scenes reductions that turns a GatorPy problem into a solvable slack form LP is described.

### A. Linear Programming

Linear Programming (LP) is a mathematical optimization technique for achieving the best outcome in a mathematical model whose requirements are represented by linear relationships. LP has vast applications across various fields, including economics, supply chain, chemical engineering, and industrial engineering. An LP problem typically involves an objective function, which is to be maximized or minimized, subject to a set of linear constraints.

Formally, an LP problem can be expressed as:

$$\text{minimize} \quad \mathbf{c}^\top \mathbf{x}$$

subject to:

$$\mathbf{A}\mathbf{x} \leq \mathbf{b}$$

where:

- $\mathbf{x} \in \mathbb{R}^n$ is the vector of decision variables,
- $\mathbf{c} \in \mathbb{R}^n$ is the vector of coefficients of the objective function,
- $\mathbf{A} \in \mathbb{R}^{m \times n}$ is the matrix of coefficients of the constraints,
- $\mathbf{b} \in \mathbb{R}^m$ is the right-hand side vector of the constraints.

LP problems are highly versatile and many problems that are naturally found across domains can be identically turned into an LP or approximated as one. The goal of an LP solver is to find the values of $\mathbf{x}$ that maximize or minimize the objective function while satisfying all the constraints.

### B. Simplex Algorithm

The Simplex algorithm is one of the most widely used methods for solving linear programming problems. It is an iterative method that navigates along the edges of the feasible region (defined by the constraints) to reach the optimal solution. The algorithm was developed by George Dantzig in 1947 and has since become a cornerstone of operational research and optimization.

The Simplex algorithm operates in two primary phases:

- **Phase 1**: The algorithm begins by finding a basic feasible solution (BFS), if one exists. This is achieved by adding artificial variables to the problem, converting it into a form that is guaranteed to have a feasible starting point.
- **Phase 2**: Once a feasible solution is found, Phase 2 proceeds to optimize the objective function. The algorithm iterates by selecting entering and leaving variables and updating the solution until no further improvements can be made.

The key advantages of the Simplex algorithm include its ability to efficiently handle large problems and its ability to identify both optimal solutions and infeasibility/unboundedness in the problem. However, the algorithm's worst-case performance can be exponential in the number of variables, although in practice, it often performs very well.

In GatorPy, the Simplex algorithm is implemented using the two-phase approach, ensuring that the solver can handle both feasible and infeasible LP problems.

### C. Available Solvers

There are numerous optimization solvers available, both commercial and open-source, that implement LP solvers and other optimization methods. Some of the most popular commercial solvers include:

- **CPLEX**: A high-performance commercial optimization solver by IBM that provides a robust set of optimization algorithms, including the Simplex method and interior-point methods.

- **Gurobi**: Another leading commercial optimization solver, known for its speed and reliability, and used in a wide range of industries.

In the open-source domain, several solvers offer accessible alternatives to commercial solutions:

- **GLPK (GNU Linear Programming Kit)**: A popular open-source LP solver that implements both the Simplex method and interior-point methods.
- **COIN-OR**: A collection of open-source optimization solvers, which includes Clp (the COIN-OR LP solver).
- **CVXPY**: A Python-based optimization modeling language that provides a high-level interface for solving LP and convex optimization problems. CVXPY allows for easy integration of different solvers and has become a widely adopted tool in academic and industrial settings [1].

GatorPy is inspired by these open-source solvers, with a particular focus on educational value and extensibility. By building a custom solver in Python, GatorPy allows for greater transparency and customization, making it an excellent tool for learning about LP and optimization algorithms.

## II. Implementation

The implementation of GatorPy can be sectioned into four parts: First, the algebraic modeling language syntax that was created by GatorPy, this implementation is available in section II-A. Second, the implementation of the Simplex algorithm that receives in a slack form LP and outputs the optimal solution (or an infeasibility/unbounded certificate if applicable). This simplex implementation is available in section II-B. Third, the Python objects that are created to facilitate the GatorPy modeling language. These objects and their overall purpose to the modeling language are available for viewing in section II-C. Finally, a large portion of the work in this project involved a series of reductions that turn any LP problem into an equivalent form that can be processed by the simplex algorithm.

### A. GatorPy Syntax

The overarching goal with the optimization modeling syntax is to maintain a healthy balance between a pythonic syntax and standard optimization linear algebra notation. GatorPy relies heavily on the NumPy numerical processing package in Python. The general structure of a GatorPy problem involves the following steps:

1) Create `Parameter` objects for each parameter in the problem. Each `Parameter` object takes in a `np.array` as the value.
2) Create `Variable` objects for each variable in the problem. Each `Variable` takes in an integer as the shape of the vector. *Note: Each variable must be a vector, this is left as a potential next step in section IV-A.*
3) Create a `Problem` object representing the overall problem. The `Problem` object expects a Python `dict` object with the following key-value pairs:
   - Either `"minimize"` or `"maximize"` as a key with a GatorPy `Expression` as the value.
   - Either `"subject to"` or `"constraints"` as a key with a list of GatorPy `Constraint` objects as the value.

The simple syntax of GatorPy can be best communicated with an example. Consider the following optimization problem with two variables and three constraints:

$$\begin{aligned} \text{maximize} \quad & \mathbf{c}^\top \mathbf{y} \\ \text{subject to} \quad & \mathbf{A}\mathbf{y} \preceq \mathbf{b} \\ & \mathbf{y} \preceq \mathbf{1} \\ & \mathbf{y} \succeq \mathbf{0} \end{aligned}$$

where

$$\mathbf{c} = \begin{bmatrix} 1.2 \\ 0.5 \end{bmatrix}, \quad \mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1.2 & 0.5 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

This above optimization problem can be expressed in GatorPy as:

```
# Parameters
A = Parameter(np.array([[1,1],[1.2,0.5]]))
b = Parameter(np.array([1,1]))
c = Parameter(np.array([1.2,1]))

# Variables
y = Variable(2)

# Problem
problem = Problem({
    'maximize': c.T @ y,
    'subject to': [
        A @ y <= b,
        y <= 1,
        y >= 0
    ]
})

solution = problem.solve()
print(solution)
>>> (1.14, [0.71, 0.29], True)
```

Listing 1. Solving a Linear Program Symbolically

### B. Simplex Implementation

The implementation of the Simplex algorithm in GatorPy uses the two-phase simplex method, which will solve for optimality without needing to explicitly enter a feasible starting point. This version of the simplex algorithm is divided into two main phases: an initial feasibility search phase (Phase 1), and an optimality algorithm (Phase 2). The following defines the overarching purpose and goal of each section in the implementation. The overall pseudo-code algorithm is available in Algorithm 1, while the individual algorithm pseudocodes are located in the Appendix in section V. Here is a high-level overview of how the two phase simplex method is applied in GatorPy.

*1) Phase 1: Finding a Feasible Solution:* The goal of Phase 1 is to find a basic feasible solution (BFS) for the given linear programming problem. If the problem is infeasible, Phase 1 will detect this and terminate. The process involves:

1) Augmenting the constraint matrix $A$ with artificial variables to form an auxiliary problem.
2) Minimizing the sum of the artificial variables to determine feasibility.
3) Using Bland's rule to prevent cycling during pivot selection.
4) Removing artificial variables from the basis if a feasible solution is found.

*2) Phase 2: Optimizing the Objective Function:* Once a feasible solution is obtained, Phase 2 optimizes the objective function. The process involves:

1) Constructing the simplex tableau using the feasible basis obtained from Phase 1.
2) Iteratively selecting entering and leaving variables based on Bland's rule and the minimum ratio test.
3) Performing pivot operations to update the tableau and improve the objective value.
4) Terminating when no entering variable exists, indicating optimality.

*3) Key Functions in the Implementation:* The implementation relies on six key pieces of the full algorithm.

- `pivot`: Performs the pivot operation to update the simplex tableau. This algorithm is summarized in Algorithm 2
- `find_entering_variable`: Determines the entering variable based on the specified rule (e.g., Bland's rule). This algorithm is summarized in Algorithm 3
- `find_leaving_variable`: Identifies the leaving variable using the minimum ratio test. This algorithm is summarized in Algorithm 4
- `simplex_phase_1`: Implements Phase 1 of the simplex algorithm. This algorithm is summarized in Algorithm 5
- `simplex_phase_2`: Implements Phase 2 of the simplex algorithm. This algorithm is summarized in Algorithm 6
- `two_phase_simplex`: Combines Phase 1 and Phase 2 to solve the linear programming problem. This algorithm is summarized in Algorithm 1

**Algorithm 1** Two-Phase Simplex Method

**Input:** matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, vector $\mathbf{b} \in \mathbb{R}^m$, cost vector $\mathbf{c} \in \mathbb{R}^n$
**Output:** optimal value $f^*$, solution $\mathbf{x}^* \in \mathbb{R}^n$, Feasibility

$(\mathbf{A}', \mathbf{b}', B) \leftarrow$ `simplex_phase_1`$(\mathbf{A}, \mathbf{b})$
**if** result is None **then**
    **return** None, None, Infeasible
**end if**
$(f^*, \mathbf{x}^*) \leftarrow$ `simplex_phase_2`$(\mathbf{A}', \mathbf{b}', \mathbf{c}, B)$
**return** $f^*$, $\mathbf{x}^*$, Feasible

## V. Appendix

**Algorithm 2** Pivot Operation

**Input:** tableau $\mathbf{T} \in \mathbb{R}^{m \times n}$, indices $r$ and $c$
**Output:** tableau $\mathbf{T}$

$\mathbf{T}_{r,:} \leftarrow \mathbf{T}_{r,:} / \mathbf{T}r, c$ ▷ normalize pivot row
**for** each row $i \neq r$ **do**
  $\mathbf{T}i, : \leftarrow \mathbf{T}i, : - \mathbf{T}i, c \cdot \mathbf{T}r, :$ ▷ update rows
**end for**
**return T** ▷ return updated tableau

**Algorithm 3** Find Entering Variable (Bland's Rule)

**Input:** tableau $\mathbf{T} \in \mathbb{R}^{m \times n}$
**Output:** index $j$, or None

**for** $j = 1$ to $n - 1$ **do**
  **if** $\mathbf{T}_{m,j} < -\varepsilon$ **then**
    **return** $j$ ▷ return entering index
  **end if**
**end for**
**return** none ▷ no variables found

**Algorithm 4** Find Leaving Variable

**Input:** tableau $\mathbf{T} \in \mathbb{R}^{m \times n}$, index $c$
**Output:** index $r$, or None

Initialize empty list $R$
**for** $i = 1$ to $m - 1$ **do**
  **if** $\mathbf{T}_{i,c} > \varepsilon$ **then**
    Append $(\mathbf{T}_{i,n} / \mathbf{T}_{i,c}, i)$ to $R$
  **end if**
**end for**
**if** $R$ is empty **then**
  **return** None ▷ no variables found
**end if**
$r \leftarrow \arg\min_R q$ ▷ update row index
**return** $r$ ▷ return leaving index

**Algorithm 5** Simplex Phase 1

**Input:** Matrix $\mathbf{A}$, vector $\mathbf{b}$
**Output:** Matrix $\mathbf{A}$, vector $\mathbf{b}$, list $B$, or None

$\mathbf{A}_{aux} := [\mathbf{A}\ \mathbf{I}]$
$\mathbf{T} := \begin{bmatrix} \mathbf{A} & \mathbf{I} & \mathbf{b} \\ -\mathbf{1}^\top \mathbf{A} & -\mathbf{1}^\top & -\mathbf{1}^\top \mathbf{b} \end{bmatrix}$
$B := [\dim(\mathbf{A})_2, \ldots, \dim(\mathbf{A}_{aux})_2]$
**while** true **do**
  $c \leftarrow$ `find_entering_variable`$(\mathbf{T})$
  **if** $c$ is None **then**
    **break**
  **end if**
  $r \leftarrow$ `find_leaving_variable`$(\mathbf{T}, c)$
  **if** none found **then**
    **return** None ▷ return infeasible
  **end if**
  $\mathbf{T} \leftarrow$ `pivot`$(\mathbf{T}, c, r)$
**end while**
**if** $T_{-1,-1} > 0$ **then** ▷ positive objective value
  **return** None ▷ return infeasible
**end if**
$\mathbf{T} \leftarrow$ `pivot`$(\mathbf{T}, c, r)$ ▷ remove aux vars
$\mathbf{A}, \mathbf{b} \leftarrow \mathbf{T}$
**return** $A$, $b$, $B$

**Algorithm 6** Simplex Phase 2

**Input:** Matrix $\mathbf{A}$, vector $\mathbf{b}$, cost vector $\mathbf{c}$, basis $B$
**Output:** Optimal value $f^*$, solution $\mathbf{x}^*$, or Unbounded

$\mathbf{T} := \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ -\mathbf{c}^\top & 0 \end{bmatrix}$
Adjust cost row from $B$
**while** true **do**
  $c \leftarrow$ `find_entering_variable`$(\mathbf{T})$
  **if** $c$ is None **then**
    **break** ▷ optimality reached
  **end if**
  $r \leftarrow$ `find_leaving_variable`$(\mathbf{T}, c)$
  **if** $r$ is None **then**
    **return** `Unbounded`
  **end if**
  $\mathbf{T} \leftarrow$ `pivot`$(\mathbf{T}, r, c)$
  Update basis $B$ with $B[r] \leftarrow c$
**end while**
Extract solution $\mathbf{x}^*$ from $\mathbf{T}$ using basis $B$
$f^* \leftarrow -\mathbf{T}_{-1,-1}$ ▷ optimal value from cost row
**return** $f^*$, $\mathbf{x}^*$, feasible = true

## References


[1] S. Diamond and S. Boyd, "CVXPY: A Python-embedded modeling language for convex optimization," *Journal of Machine Learning Research*, vol. 17, no. 83, pp. 1–5, 2016.