# 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 solver implemented entirely in Python. The purpose of this project is twofold: First, to serve as an educational tool as an example of a simple and custom implementation of an LP solver. Second, to establish a foundation for other students to build upon and contribute to a collaborative open-source University of Florida custom optimization solver.

### A. Linear Programming

### B. Simplex Algorithm

### C. Available Solvers

There are numerous available optimization solvers, both commercial and open-source. One particular source of inspiration for this project is CVXPY, an open-source convex optimization solver [1].

## 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
    ]
})
```

```
18
19 solution = problem.solve()
20 print(solution)
21 >>> (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 ensures feasibility and optimality of the solution. 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 pseudo-code algorithm is available in Algorithm 6

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

**Algorithm 1** Pivot Operation

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

$\mathbf{T}_{r,:} \leftarrow \mathbf{T}_{r,:}/\mathbf{T}r, c$ ▷ 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 2** Find Entering Variable (Bland's Rule)

**Input:** $\mathbf{T} \in \mathbb{R}^{m \times n}$
**Output:** $j$, 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 3** Find Leaving Variable

**Input:** $\mathbf{T} \in \mathbb{R}^{m \times n}$, $c$
**Output:** $r$, 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 6** Two-Phase Simplex Method

**Input:** Constraint matrix $A$, RHS vector $b$, objective coefficients $c$
**Output:** Optimal value $f^*$, solution $x^*$, feasibility status

**Phase 1:** Call SIMPLEX-PHASE-1 with $A, b$
**if** infeasible **then**
  **return** None, None, False
**end if**
**Phase 2:** Call SIMPLEX-PHASE-2 with reduced $A, b, -c$, and basis
**return** $f^*, x^*$, feasible

*4) Example Usage:* The following Python code demonstrates how to use the two-phase simplex implementation to solve a linear programming problem:

**Algorithm 4** Simplex Phase 1

**Input:** $\mathbf{A}$ $\mathbf{b}$
**Output:** $\mathbf{A}$, $\mathbf{b}$, `basis`, or `None`

$\mathbf{A}_{aux} := [\mathbf{A}\ \mathbf{I}]$
$\mathbf{T} := \begin{bmatrix} \mathbf{A} & \mathbf{I} & \mathbf{b} \\ -\mathbf{1}^\top\mathbf{A} & -\mathbf{1}^\top & -\mathbf{1}^\top\mathbf{b} \end{bmatrix}$
**while** `len(A_aux[0]) - len(A[0])` **do**
    Find entering variable using Bland's rule
    **if** none found **then**
        **break**
    **end if**
    Find leaving variable using ratio test
    **if** none found **then**
        **return** `Infeasible`
    **end if**
    Perform pivot and update basis
**end while**
**if** objective value $> \varepsilon$ **then**
    **return** `Infeasible`
**end if**
Remove artificial variables from basis (pivot them out if needed)
**return** Reduced $A$, $b$, basis

**Algorithm 5** Simplex Phase 2

**Input:** Feasible $A$, $b$, cost vector $c$, basis indices
**Output:** Optimal value, solution, or unbounded

Build tableau with $A$, $b$, and $-c$
Adjust cost row based on basic variables
**while** true **do**
    Find entering variable using Bland's rule
    **if** none found **then**
        **break**
    **end if**
    Find leaving variable using ratio test
    **if** none found **then**
        **return** `Unbounded`
    **end if**
    Perform pivot and update basis
**end while**
Extract solution from tableau using basis
Compute optimal value
**return** Optimal value, solution, feasible = true

```python
import numpy as np

# Define the problem
A = np.array([[1, 1], [1.2, 0.5]])
b = np.array([1, 1])
c = np.array([1.2, 0.5])

# Solve using two-phase simplex
optimal_value, solution, feasible =
    two_phase_simplex(A, b, c)

if feasible:
    print(f"Optimal Value: {optimal_value}")
    print(f"Solution: {solution}")
else:
    print("The problem is infeasible.")
```

Listing 2. Example Usage of Two-Phase Simplex

This implementation provides a robust and modular approach to solving linear programming problems using the simplex method.

### C. Python Objects

### D. LP Reductions

## III. Results

### A. Testing Framework

### B. Testing Results

## IV. Discussion

### A. Future Work

### B. Conclusion

## 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.