

GatorPy: A Custom Implemented Linear Programming Solver

Andrés Espinosa

Industrial and Systems Engineering

University of Florida

andresespinosa@ufl.edu

Abstract—GatorPy is an educationally motivated Linear Programming (LP) solver implemented in pure Python and NumPy. The project aims to demystify the internal workings of LP solvers by providing a modular, object-oriented framework that transforms symbolic problem descriptions into slack form and solves them using the two-phase Simplex method. The solver supports a custom modeling language and is designed to be readable, extensible, and transparent for students and practitioners embarking on similar projects.

GatorPy has been validated against CVXPY on a suite of small LPs, successfully passing 6 out of 7 tests. The remaining failure shows one of the solver’s current limitations, motivating iterative improvements as an open-source UF solver. There are several directions for a collaborative further development of GatorPy, including a more robust test suite, performance-oriented reimplementations in a compiled language, and support for mixed-integer programs, convex programs, alternative solver algorithm. GatorPy offers both a functional LP solver and an exciting opportunity as an open-source educational platform for understanding computational optimization fundamentals.

I. INTRODUCTION

GatorPy is a custom Linear Programming (LP) solver implemented entirely in Python. It is built using native Python constructs and NumPy for numerical operations, and follows an object-oriented programming (OOP) approach to ensure modularity and extensibility. GatorPy transforms user-defined LP problems into a structured internal representation, reduces the problem to an equivalent standard form, and applies established solving algorithms to compute optimal solutions. This project report outlines the implementation of GatorPy and its modular design, encouraging its growth as an open-source solver for UF.

The remainder of the introduction, Section I, provides an overview of linear programming, the simplex algorithm, existing Operations Research (OR) solvers, and the motivation behind developing GatorPy. The implementation is discussed in detail in Section II, highlighting key design choices made throughout the development process. The section begins with an overview of GatorPy’s intuitive modeling syntax, followed by an explanation of the simplex algorithm in pseudocode. Next, the core Python objects that serve as the backbone of the framework are introduced. Finally, the internal transformations that convert a user-defined GatorPy problem into a solvable slack form LP are described. Section III presents the testing suite used to validate the functionality of the project, along with instructions on how to deploy and run GatorPy. The

conclusion, Section IV, outlines potential directions for future work and concludes this report.

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:

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

subject to:

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

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.

LPs are highly versatile, with numerous real-world problems across various domains either naturally formulated as LPs or effectively approximated as such.

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 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 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. The Simplex algorithm is also remarkably intuitive, as it moves from vertex to vertex of a polyhedron to identify the point with the optimal objective value. This concept is shown visually in Figure 1.

Several variants of the Simplex algorithm exist, including primal, dual, and revised forms. In GatorPy, the algorithm

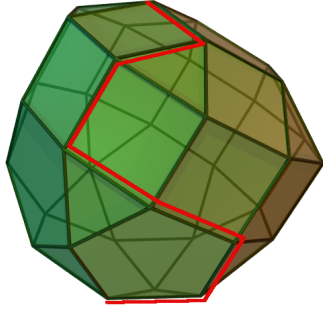


Fig. 1. Visual interpretation of the Simplex traversal [1]

is implemented using the two-phase Simplex method. This approach ensures the user can input any LP in the expected modeling language and solver will handle feasibility, detect unboundedness, and return optimal solutions when they exist.

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.
- **Gurobi:** Another leading commercial optimization solver, known for its speed and reliability, and used in a wide range of industries.
- **Excel Solver:** A built-in optimization tool in Microsoft Excel that allows users to solve LP problems directly within a spreadsheet interface.

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.

GatorPy is primarily inspired by CVXPY, with a strong emphasis on educational value and extensibility. Solvers can often be quite complex due to the advanced techniques they use to optimize computation time. By building a custom solver in Python, GatorPy offers greater transparency and customization, making it an ideal tool for students to better understand the underlying principles.

D. Motivation

The "behind-the-scenes" complexity of solvers often leads students to lose sight of the overall goals and concepts of OR.

Many students need an intuitive understanding of the tools they use in order to truly connect with the material. This project has two main objectives: First, to provide an educational tool by offering a clear and simple implementation of a linear programming (LP) solver, built entirely using Python and NumPy. Second, to lay the groundwork for an open-source solver that can be further developed by future UF students, offering an opportunity for students to engage directly with the core concepts of LP solving.

II. IMPLEMENTATION

The implementation of GatorPy can be divided into four key parts:

- 1) **Algebraic Modeling Language:** The first component of this project focuses on developing a custom algebraic modeling language for GatorPy. This syntax strikes a balance between Pythonic pseudo-code and an algebraic language interface, aiming to provide an intuitive framework for formulating linear programming problems. A more detailed discussion on its design and implementation can be found in Section II-A.
- 2) **Simplex Algorithm:** The second component involves the implementation of the Simplex algorithm, which processes a linear programming problem in slack form to yield the optimal solution. If the problem is infeasible or unbounded, the algorithm provides the appropriate certificates. A thorough explanation of the Simplex algorithm is provided in Section II-B.
- 3) **Python Objects:** The third component is the development of Python objects that support the GatorPy modeling language. These objects are designed to streamline the modeling process, making it easier to define and solve linear programming problems. Their purpose, design, and functionality are discussed in Section II-C.
- 4) **Reductions for Slack Form:** A key focus of this project is the implementation of reductions that convert any given LP problem into a form that can be processed by the Simplex algorithm. This transformation is essential for the solver's operation and is explained in Section II-D.

A. GatorPy Syntax

GatorPy makes extensive use of the NumPy numerical processing package, ensuring high performance while using the famously slow language of Python. To effectively model linear programming problems, GatorPy relies on a set of key user-inputted objects that structure the problem. The structure of a typical 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 primary objective of GatorPy's optimization modeling syntax is to balance a pythonic interface with standard linear algebraic notation. The simple syntax of GatorPy can be best communicated with an example. Consider the following optimization problem with two variables and three constraints:

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

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:

```

1 # Parameters
2 A = Parameter(np.array([[1,1],[1.2,0.5]]))
3 b = Parameter(np.array([1,1]))
4 c = Parameter(np.array([1.2,1]))
5
6 # Variables
7 y = Variable(2)
8
9 # Problem
10 problem = Problem({
11     'maximize': c.T @ y,
12     'subject to': [
13         A @ y <= b,
14         y <= 1,
15         y >= 0
16     ]
17 })
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 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). A high-level pseudo-code of the two-phase simplex algorithm is available in Algorithm 1, while the individual algorithm pseudocodes are located in the Appendix in section V.

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 GatorPy 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`

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

- 1: $(\mathbf{A}', \mathbf{b}', B) \leftarrow \text{simplex_phase_1}(\mathbf{A}, \mathbf{b})$
 - 2: **if** result is None **then**
 - 3: **return** None, None, Infeasible
 - 4: **end if**
 - 5: $(f^*, \mathbf{x}^*) \leftarrow \text{simplex_phase_2}(\mathbf{A}', \mathbf{b}', \mathbf{c}, B)$
 - 6: **return** f^* , \mathbf{x}^* , Feasible
-

C. Python Objects

The GatorPy solver is built on a foundation of modular Python classes that enable symbolic modeling, expression handling, and LP transformation. The object-oriented design ensures that each mathematical concept—such as a variable, parameter, or constraint—is represented by a dedicated object that carries behavior and structure. A fundamental programming concept that is used for these classes is "Operator Overloading". Operator Overloading allows for a program to explicitly override the default behavior of an operation. This is what allows for the sum of two Variable objects to yield a Sum object. This design simplifies user interaction and facilitates internal processing and reduction to slack form. The key classes are outlined below:

1) *Expressions*: The Expression class acts as a wrapper for all objects in GatorPy. The Expression is most importantly defined with a `parents` attribute that allows Expression objects to be linked to each other in a tree.

```
1 expr = A @ x + b
```

2) *Variable*: The Variable class represents decision variables in an LP. Each Variable object is initialized with a shape (i.e., its dimensionality) and internally stores a NumPy vector of symbolic variables.

```
1 # Creating a 2x1 vector variable
2 x = Variable(2)
```

3) *Parameter*: The Parameter class wraps constant problem data such as coefficient vectors or constraint matrices. Each instance stores a NumPy array and supports arithmetic with other Parameters, Variables, and Expressions.

```
1 # Creating a 2x2 matrix A and 2x1 vector b
2 A = Parameter(np.array([[1, 1], [1.2, 0.5]]))
3 b = Parameter(np.array([1, 1]))
```

4) *Sums and LinearOperations*: Two important symbolic objects are Sum and LinearOperation. These classes serve as the intermediate step between Variable objects and Constraints. A LinearOperation is defined as a symbolic representation of a matrix/vector multiplication and consists of a left Parameter object and a right Variable object. A Sum object will collect any Variables, Parameters, or LinearOperations together in a list of terms.

```
1 # Creating two 2x1 variables and one 2x2 parameter
2 x_cold = Variable(2)
3 x_hot = Variable(2)
4 A = Parameter(np.array([[1, 1], [1.2, 0.5]]))
5 # Creating two LinearOperations with "@"
6 lin_op1 = A @ x_hot
7 lin_op2 = - A @ x_cold
8 # Creating a Sum object with "+" or "-"
9 sum_obj = lin_op1 + lin_op2
```

5) *Constraint*: Constraints are created via comparison operations on Expression objects. They encapsulate inequality or equality relationships and are stored in the problem formulation.

```
1 # Creating a 2x2 matrix A and 2x1 vector b
2 A = Parameter(np.array([[1, 1], [1.2, 0.5]]))
3 b = Parameter(np.array([1, 1]))
4 # Creating a 2x1 vector variable
5 x = Variable(2)
6 # Creating two constraints
7 con1 = A @ x <= b
8 con2 = x >= 0
```

6) *Problem*: The Problem class defines the optimization problem. It takes in a dictionary specifying the objective (either "minimize" or "maximize") and a list of constraints. Once instantiated, the problem can be solved using the `solve()` method, which performs symbolic parsing, reduces to slack form, and applies the two-phase simplex method.

```
1 # Creating a 2x2 matrix A and two 2x1 vectors b, c
2 A = Parameter(np.array([[1, 1], [1.2, 0.5]]))
3 b = Parameter(np.array([1, 1]))
4 c = Parameter(np.array([5, 3.5]))
5 # Creating a 2x1 vector variable
6 x = Variable(2)
7
8 prob = Problem({
9     "maximize": c.T @ x,
10    "subject to": [
11        A @ x <= b,
12        x <= 1,
13        x >= 0
14    ]
15 })
16 solution = prob.solve()
```

Figure 2 provides a visual representation of how GatorPy takes in the Parameter and Variable objects and turns them into complete representation of an LP. Each object is connected to the subsequent Expression it contributes to, ultimately forming the final Problem object.

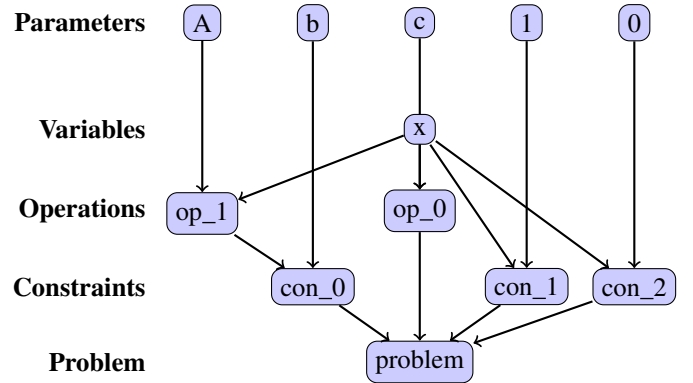


Fig. 2. Directed computation graph demonstrating Expression relationships. The problem data in Code II-C6 is demonstrated as Expressions are linked together to create the Problem.

D. LP Reductions

In order to solve a general linear programming (LP) problem using the simplex algorithm, the problem must first be transformed into *slack form*. The Problem class encapsulates this transformation process by reducing arbitrary LP input definitions into a set of structured matrix equalities and a standardized objective. This section outlines the full reduction pipeline used by the `to_slack_form` method.

An LP problem is provided as a dictionary with an objective function (minimize or maximize) and a list of constraints under subject to or constraints. Constraints may include inequalities, equalities, or non-negativity conditions.

The transformation from a general problem to slack form proceeds through the following sequential reductions:

- 1) **Bounded Variable Collection:** Non-negativity constraints are parsed to identify bounded variables (variables with ≥ 0 restrictions).
- 2) **Non-negativity Stripping:** Non-negativity constraints are removed from the constraint list since they are tracked separately via variable flags.
- 3) **Unbounded Variable Identification:** All variables appearing in the constraint set that were not previously identified as bounded are labeled as unbounded.
- 4) **Equality Standardization:** Inequality constraints are transformed into equalities by introducing slack or surplus variables, which are labeled as bounded.
- 5) **Unbounded Variable Decomposition:** Every unbounded variable is replaced with a pair of bounded variables ($x^+ - x^-$).
- 6) **Constraint Rewriting:** All existing constraints are rewritten in terms of the new bounded variables.
- 7) **Variable Indexing:** The final list of bounded variables is assigned a block index in the variable vector.
- 8) **Matrix Formulation:** Each equality constraint is converted into a matrix row (A_i, b_i) using `linear_equality_to_matrix_equality`.
- 9) **Vector Concatenation:** All bounded variables are concatenated into a single flat vector `x_big`, representing the problem's variable space.
- 10) **Constraint Concatenation:** Each A_i and b_i of each constraint is concatenated into one combined constraint with A_{big}, b_{big} parameters.
- 11) **Objective Concatenation:** The objective function is linearized and mapped to a vector c aligned with the same variable ordering.
- 12) **Minimization Standardization:** If the problem is a maximization, it is converted to a minimization by negating the objective.
- 13) **Solve:** The `two_phase_simplex` function is called and the problem solution is returned.

E. Final Form

After the transformation, the problem is expressed in slack form:

$$\begin{aligned} &\text{minimize} && \mathbf{c}^T \mathbf{x} \\ &\text{subject to} && \mathbf{A}\mathbf{x} = \mathbf{b} \\ &&& \mathbf{x} \geq 0 \end{aligned}$$

Where \mathbf{x} is the flat vector `x_big` composed of all bounded variables, including replacements for unbounded and slack variables. The vectors \mathbf{c} , \mathbf{b} and matrix \mathbf{A} are stored in `c_big`, `b_big`, and `A_big` respectively.

This slack form representation is now compatible with the two-phase simplex method and is fed back in to the algorithm described previously in Algorithm 1

III. RESULTS

A. Testing

The testing suite is available in the `tests.py` file. In order to test the results of GatorPy, **CVXPY** is used as a comparison to ensure that GatorPy returns the correct solution. The testing suite currently involves 7 simple LPs that are ran in both GatorPy and **CVXPY** and compared to each other. GatorPy currently passes 6/7 unit tests, notably failing to drop auxiliary variables from the basis when the auxiliary variables outnumber the original set of variables.

Augmenting the testing suite with a richer set of LPs of varying sizes and problem structure is left as a next step in Section IV-A.

B. GatorPy Usage

To use GatorPy, proceed to the ECH4905 - Andres Espinosa repository GitHub link available in the references at [2]. The files for GatorPy are available at `project/gatorpy`. Before attempting to run your own LP, ensure a Python version with NumPy. This repository was only tested with Python=3.11 and NumPy=1.26.0. The code is likely to work with earlier/later versions but the code has not been verified to work with other versions. To solve your own LP with GatorPy, download this folder, replace the example LP in `solve.py` with your own LP entered in the GatorPy modeling and finally run the `solve.py` file.

Publishing GatorPy as a standalone PyPi package available to install with `pip` is left as a next step in Section IV-A.

IV. DISCUSSION

A. Future Work

Several directions remain to enhance the solver's robustness, generality, and performance. Future efforts will focus on the following areas:

- **Expanded Benchmarking Suite:** The current test suite could be augmented to include problems of varying sizes, particularly those with a large number of variables and constraints. This would allow for a more thorough investigation of solver performance under computationally intensive scenarios.
- **Diverse Problem Structures:** The solver could be evaluated on a broader array of problem structures and formulations. This would help assess the effectiveness and flexibility of the solver's problem reduction mechanisms across different LP representations.
- **Extension to Mixed-Integer Programming (MIP):** Support for mixed-integer programs could easily be introduced by implementing a `BinaryVariable` object. Algorithms such as branch-and-bound and Gomory cut methods could be incorporated to handle integrality constraints while still relying on the already implemented Simplex algorithm.

- **Extension to Convex Optimization:** A difficult, but rewarding extension could be to support general convex programs. This involves introducing a set of `Function` objects representing common convex functions, which can be composed using disciplined convex programming (DCP) rules. These enhancements would evolve the framework into a full-featured convex optimization solver.
- **Multiple Solver Algorithms:** Currently, the Simplex algorithm is the only algorithm available for running LPs through. This could be changed by defining a general `solver` structure that takes in a set of parameters and variables and outputs a solution. This would allow GatorPy to have multiple algorithms implemented such as interior-point methods. This would also be necessary in order to extend the solver to include convex problems, as the Simplex algorithm does not work with convex problems.
- **Efficient Language Switch:** Transitioning the solver's core implementation to a more efficient language such as Rust, Julia, or C++ could significantly improve performance, particularly for large-scale problems. These languages offer better memory management and computational speed, making them ideal for optimization tasks.

B. Conclusion

GatorPy provides a clear, educationally focused implementation of a Linear Programming solver using pure Python and NumPy. By constructing a modular object-oriented framework and employing the two-phase Simplex method, GatorPy demystifies OR solvers and offers students an accessible entry point into optimization.

This project highlights how algorithmic transparency and thoughtful software design can enable a deeper understanding of optimization theory. From the algebraic modeling language to the step-by-step transformations into slack form, every component was engineered to maximize readability and reusability for future students. The solver successfully handles a range of LP problems, validates feasibility, and reliably computes optimal solutions, laying a solid foundation for future development.

However, there are many areas of improvement that this project could benefit from. Extensions to MIPs and Convex Programs is an interesting area of further work that could elevate GatorPy. Rewriting the repository in another language could prove an incredibly rewarding challenge for learning both OR tools and another programming language.

Overall, GatorPy serves as both a functional solver and a pedagogical tool, offering a hands-on open-source platform for students to experiment with and better understand the mechanics of linear programming.

Algorithm 2 pivot**Input:** tableau $\mathbf{T} \in \mathbb{R}^{m \times n}$, indices r and c **Output:** tableau \mathbf{T}

```

1:  $\mathbf{T}_{r,:} \leftarrow \mathbf{T}_{r,:}/\mathbf{T}_{r,c}$   $\triangleright$  normalize pivot row
2: for each row  $i \neq r$  do
3:    $\mathbf{T}_{i,:} \leftarrow \mathbf{T}_{i,:} - \mathbf{T}_{i,c} \cdot \mathbf{T}_{r,:}$   $\triangleright$  update rows
4: end for
5: return  $\mathbf{T}$   $\triangleright$  return updated tableau

```

Algorithm 3 find_entering_variable**Input:** tableau $\mathbf{T} \in \mathbb{R}^{m \times n}$ **Output:** index j , or None

```

1: for  $j = 1$  to  $n - 1$  do
2:   if  $\mathbf{T}_{m,j} < -\varepsilon$  then
3:     return  $j$   $\triangleright$  return entering index
4:   end if
5: end for
6: return none  $\triangleright$  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

```

1: Initialize empty list  $R$ 
2: for  $i = 1$  to  $m - 1$  do
3:   if  $\mathbf{T}_{i,c} > \varepsilon$  then
4:     Append  $(\mathbf{T}_{i,n}/\mathbf{T}_{i,c}, i)$  to  $R$ 
5:   end if
6: end for
7: if  $R$  is empty then
8:   return None  $\triangleright$  no variables found
9: end if
10:  $r \leftarrow \arg \min_R q$   $\triangleright$  update row index
11: return  $r$   $\triangleright$  return leaving index

```

REFERENCES

- [1] User:Sdo, “Elongated pentagonal orthocupolarotunda,” 2007, created using GIMP based on *Image:Elongated_pentagonal_orthocupolarotunda.png*, Licensed under CC BY-SA 3.0. [Online]. Available: <https://commons.wikimedia.org/w/index.php?curid=1295511>
- [2] A. Espinosa, “Ech4905 project repository,” 2025, accessed: 2025-04-17. [Online]. Available: <https://github.com/Hansespinosa2/ech4905-andres-espinosa/tree/main>

Algorithm 5 simplex_phase_1**Input:** Matrix \mathbf{A} , vector \mathbf{b} **Output:** Matrix \mathbf{A} , vector \mathbf{b} , list B , or None

```

1:  $\mathbf{A}_{aux} := [\mathbf{A} \ \mathbf{I}]$ 
2:  $\mathbf{T} := \begin{bmatrix} \mathbf{A} & \mathbf{I} & \mathbf{b} \\ -\mathbf{1}^\top \mathbf{A} & -\mathbf{1}^\top & -\mathbf{1}^\top \mathbf{b} \end{bmatrix}$ 
3:  $B := [\dim(\mathbf{A})_2, \dots, \dim(\mathbf{A}_{aux})_2]$ 
4: while true do
5:    $c \leftarrow \text{find\_entering\_variable}(\mathbf{T})$ 
6:   if  $c$  is None then
7:     break
8:   end if
9:    $r \leftarrow \text{find\_leaving\_variable}(\mathbf{T}, c)$ 
10:  if none found then
11:    return None  $\triangleright$  return infeasible
12:  end if
13:   $\mathbf{T} \leftarrow \text{pivot}(\mathbf{T}, c, r)$ 
14: end while
15: if  $T_{-1,-1} > 0$  then  $\triangleright$  positive objective value
16:   return None  $\triangleright$  return infeasible
17: end if
18:  $\mathbf{T} \leftarrow \text{pivot}(\mathbf{T}, c, r)$   $\triangleright$  remove aux vars
19:  $\mathbf{A}, \mathbf{b} \leftarrow \mathbf{T}$ 
20: return  $\mathbf{A}, \mathbf{b}, B$ 

```

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

```

1:  $\mathbf{T} := \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ -\mathbf{c}^\top & 0 \end{bmatrix}$ 
2: Adjust cost row from  $B$ 
3: while true do
4:    $c \leftarrow \text{find\_entering\_variable}(\mathbf{T})$ 
5:   if  $c$  is None then
6:     break  $\triangleright$  optimality reached
7:   end if
8:    $r \leftarrow \text{find\_leaving\_variable}(\mathbf{T}, c)$ 
9:   if  $r$  is None then
10:    return Unbounded
11:   end if
12:    $\mathbf{T} \leftarrow \text{pivot}(\mathbf{T}, r, c)$ 
13:   Update basis  $B$  with  $B[r] \leftarrow c$ 
14: end while
15: Extract solution  $\mathbf{x}^*$  from  $\mathbf{T}$  using basis  $B$ 
16:  $f^* \leftarrow -\mathbf{T}_{-1,-1}$   $\triangleright$  optimal value from cost row
17: return  $f^*, \mathbf{x}^*$ , feasible = true

```