Analytical and Computational Approaches to Linear Programming: From Theory to Python Implementation

Nguyen Nhat Huy
Tran Thi Thuy Van

February 3, 2025

HCM University of Science, VNU-HCM
Faculty of Mathematics and Computer Science

Contents

1 The Standard Form

Linear Programming (LP) is a fundamental technique in mathematical optimization used to determine the best possible outcome—such as maximum profit or minimum cost—in a mathematical model whose requirements are represented by linear relationships. LP is widely applied in fields such as economics, operations research, engineering, and data science.
A **linear programming problem** consists of:

- A linear objective function to be maximized or minimized,
- A set of linear inequality or equality constraints, and
- Non-negativity restrictions on the decision variables.

The **standard form** of a linear programming problem is:

$$\begin{aligned} \text{Maximize} \quad & c_1x_1 + c_2x_2 + \cdots + c_nx_n \\ \text{subject to} \quad & a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1 \\ & a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2 \\ & \vdots \\ & a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m \\ & x_1, x_2, \ldots, x_n \geq 0 \end{aligned}$$

Here:

- $x_1, x_2, \ldots, x_n$ are the *decision variables*,
- $c_1, c_2, \ldots, c_n$ are the *coefficients of the objective function*,
- a_{ij} are the *coefficients of the constraints*, and
- $b_1, b_2, \ldots, b_m$ are the *right-hand-side constants.*

The standard matrix form of a linear programming (LP) problem is written compactly using vector and matrix notation:

$$\begin{aligned} \text{Maximize} \quad & \mathbf{c}^\top \mathbf{x} \\ \text{subject to} \quad & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

Where:

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

2 How to Solve a Linear Programming Problem

2.1 Common Solution Methods for Linear Programming

Solving a linear programming (LP) problem involves finding the optimal values of decision variables that maximize or minimize a linear objective function, subject to a set of linear constraints. There are several well-established methods to solve LP problems, depending on the number of variables, constraints, and the problem structure:

Graphical Method. Applicable only when there are two decision variables. One plots each inequality constraint as a line in the plane, identifies the feasible region by shading, and then superimposes level curves of the objective function. The optimum occurs at a vertex of the feasible polygon, which can be read directly from the plot.

Simplex Method. An iterative vertex-walking procedure introduced by Dantzig. Starting from a basic feasible solution, it moves along edges of the feasible polytope to adjacent vertices with strictly improving objective value. Each step corresponds to a pivot in the simplex tableau until no further improvement is possible.

Bland's Rule. A pivot-selection strategy for the Simplex method that prevents cycling. When multiple entering (or leaving) variables are candidates, always choose the one with the smallest index. This tie-breaking rule guarantees finite termination even in degenerate cases.

Two-Phase Method. A systematic way to find an initial feasible basis when none is obvious.

1. *Phase I:* Introduce artificial variables and minimize their sum to drive them to zero, thereby finding a feasible solution to the original constraints.
2. *Phase II:* Remove artificial variables and apply the Simplex method to optimize the original objective, starting from the feasible basis found in Phase I.

Dual Simplex Method. Solves the dual of the original (primal) LP by maintaining dual feasibility and iterating to restore primal feasibility. Particularly efficient when the primal has a good initial dual solution or when constraints change incrementally.

2.2 Using Scipy in Python

SciPy provides a built-in solver for linear programming via the scipy.optimize.linprog function, which finds the minimum of a linear objective subject to linear equality and/or inequality constraints. In its standard form, a linear program is written as:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & c^T x \\ \text{subject to} \quad & A_{\text{ub}}\, x \le b_{\text{ub}},\, , \\ & A_{\text{eq}}\, x = b_{\text{eq}},\, , \\ & \ell \le x \le u, \end{aligned}$$

where

- $x \in \mathbb{R}^n$ is the decision vector.
- $c \in \mathbb{R}^n$ are the cost coefficients.
- $A_{\text{ub}} \in \mathbb{R}^{m_{\text{ub}} \times n}$, $b_{\text{ub}} \in \mathbb{R}^{m_{\text{ub}}}$ define the inequalities.
- $A_{\text{eq}} \in \mathbb{R}^{m_{\text{eq}} \times n}$, $b_{\text{eq}} \in \mathbb{R}^{m_{\text{eq}}}$ define the equalities.
- $\ell, u \in (\mathbb{R} \cup \{\pm\infty\})^n$ are the variable bounds.

Function signature and main parameters:

```
res = linprog(
    c,                  % 1-D array of length $n$
    A_ub=A_ub,          % 2-D array for inequality LHS
    b_ub=b_ub,          % 1-D array for inequality RHS
    A_eq=A_eq,          % 2-D array for equality LHS
    b_eq=b_eq,          % 1-D array for equality RHS
    bounds=bounds,      % Sequence of (lower, upper) for each variable
    method='highs',     % Solver choice
    options={\dots}     % Solver-specific options
)
```

Parameters:

- `c` (*1-D array*): Coefficients of the linear objective to be *minimized.*
- `A_ub`, `b_ub` (optional, *2-D*, *1-D*): Specify the inequality constraints

$$A_{\text{ub}}\, x \;\leq\; b_{\text{ub}}.$$

- `A_eq`, `b_eq` (optional, *2-D*, *1-D*): Specify the equality constraints

$$A_{\text{eq}}\, x \;=\; b_{\text{eq}}.$$

- `bounds` (optional, sequence of tuples): Bounds for each variable x_i, given as (ℓ_i, u_i), where $\ell_i \leq x_i \leq u_i$. By default $\ell_i = 0$ and $u_i = +\infty$.
- `method` (string): Solver algorithm. Common choices are
  - `'highs'` (default),
  - `'highs-ds'`,
  - `'highs-ipm'`,
  - `'interior-point'` (legacy).
- `options` (dict): Solver-specific settings such as tolerance, maximum iterations, and verbosity.

We use the following program after extracting information of the problem to solve any linear programming problem!

```python
n_vars = len(c)
bounds = [(0, None)] * n_vars
res = linprog(
    c=c,
    A_ub=A_ub,
    b_ub=b_ub,
    A_eq=A_eq,
    b_eq=b_eq,
    bounds=bounds,
    method='highs',  # Modern and recommended solver
)

# === OUTPUT FULL INFORMATION ===
print("Optimization result from scipy.optimize.linprog:")
print(f"Status: {res.message}")
print(f"Success: {res.success}")
print(f"Status Code: {res.status} (0 means success)")

if res.success:
    if lines[0].lower().startswith('maximize'):
        print(f"Optimal Value of Objective Function: {-res.fun}")
    else:
        print(f"Optimal Value of Objective Function: {res.fun}")
    print(f"Optimal Solution x: {res.x}")
    print(f"Number of Iterations: {res.nit}")
    # Slack and Residuals are usually available even if not optimal, but checking success is safer for duals
    if res.slack is not None:
        print(f"Slack Variables (A_ub @ x - b_ub): {res.slack}")
    if res.eqlin.residual is not None: # Use .residual attribute for eqlin
        print(f"Residuals (A_eq @ x - b_eq): {res.eqlin.residual}")
    # Dual prices are typically only available and meaningful for an optimal solution
    if hasattr(res, 'dual') and res.dual is not None:
        print(f"Marginals (dual prices): {res.dual}")
    else:
        print("Marginals (dual prices) not available for this solution status.")
else:
    print("Optimization did not converge to an optimal solution.")
    # You can add more specific messages based on res.status if needed
    # e.g., if res.status == 2: print("Problem is infeasible.")
    # e.g., if res.status == 3: print("Problem is unbounded.")
```

The call to `linprog` returns an `OptimizeResult` object, commonly bound to `res`. Important attributes include:

res.x (array of floats): The optimal values of the decision variables x.

res.fun (float): The optimal value of the objective function, i.e. $\min c^T x$.

res.slack (array of floats): The slack in each inequality constraint, computed as

$$\texttt{slack}_i = b_{\text{ub},i} - (A_{\text{ub}}\, x)_i.$$

res.con (array of floats): The residuals of the equality constraints, computed as

$$\texttt{con}_j = b_{\text{eq},j} - (A_{\text{eq}}\, x)_j.$$

res.status (integer): Solver exit status code:

1. 0 = optimization succeeded;
2. 1 = iteration or time limit reached;
3. 2 = problem appears infeasible;
4. 3 = problem appears unbounded; etc

res.success (boolean): `True` if `res.status == 0`, otherwise `False`.
res.message (string): Human-readable description of the solver outcome.
res.nit (integer): Number of iterations performed (simplex or IPM, depending on the method).
res.crossover_nit (integer, HiGHS only): Number of crossover iterations (simplex phase following interior-point).

2.3 My Input Functions

Instead of determining coefficients of the linear problem, I suggest some python functions that can help us to enter from keyboard.
There are three functions:

1. get_problem_from_keyboard(): This help us to input the mathematical problems even if some clients cannot extract there problems into matrices.
2. parse_objective(line): This extracts coefficients of the objective function.
3. parse_constraints(lines, n_vars): This extracts coefficients of all the constraints.

3 The Graphical Method

3.1 An Analytical Solution

Consider the following linear program:

$$\begin{aligned} \text{maximize} \quad & x + y \\ \text{subject to} \quad & 2x + 3y \leq 5, \\ & -2x + y \leq 1, \\ & 3x - y \leq 4, \\ & x, y \geq 0 \end{aligned}$$

First, we convert each inequality into an equality to find intersection (corner) points of the constraints.

- **Equation (1) and (2):**

$$\begin{aligned} 2x + 3y = 5 \\ -2x + y = 1 \end{aligned} \Rightarrow \text{Point } A = \left(\frac{1}{2}, \frac{3}{2}\right)$$

- **Equation (1) and (3):**

$$
\begin{aligned}
2x + 3y &= 5 \\
3x - y &= 4 \Rightarrow \text{Point } B = \left(\frac{17}{11}, \frac{7}{11}\right)
\end{aligned}
$$

- **Intersection with axes:**

$$
\begin{aligned}
&\text{From } 2x + 3y = 5: \quad (x = 0 \Rightarrow y = \frac{5}{3}), \quad (y = 0 \Rightarrow x = \frac{5}{2}) \\
&\text{From } -2x + y = 1: \quad (x = 0 \Rightarrow y = 1) \\
&\text{From } 3x - y = 4: \quad (y = 0 \Rightarrow x = \frac{4}{3})
\end{aligned}
$$

Only feasible points in the region $x \geq 0, y \geq 0$ and satisfying all constraints, so we don't check the infeasible points, and skip $(0, 0)$ gives $z = 0$.

Point	Feasible	$z = x + y$
$A = (1/2, 3/2)$	Yes	2
$B = (17/11, 7/11)$	Yes	$24/11 \approx 2.18$
$C = (4/3, 0)$	Yes	$4/3 \approx 1.33$

The optimal solution occurs at:

$$\boxed{x = \frac{17}{11}, \quad y = \frac{7}{11}, \quad z = x + y = \frac{24}{11}}$$

3.2 Using Python

Recall the problem:

$$
\begin{aligned}
\text{maximize} \quad & x_1 + x_2 \\
\text{subject to} \quad & 2x_1 + 3x_2 \leq 5, \\
& -2x_1 + x_2 \leq 1, \\
& 3x_1 - x_2 \leq 4, \\
& x_1, x_2 \geq 0
\end{aligned}
$$

We should input the problem as:

```
Enter objective function (e.g., max z = 3x1 + 2x2):
> maximize z = x1 + x2
Enter constraints (one per line, type 'done' to finish):
> 2x1 + 3x2 <= 5
> -2x1 + x2 <= 1
> 3x1 - x2 <= 4
> done
```

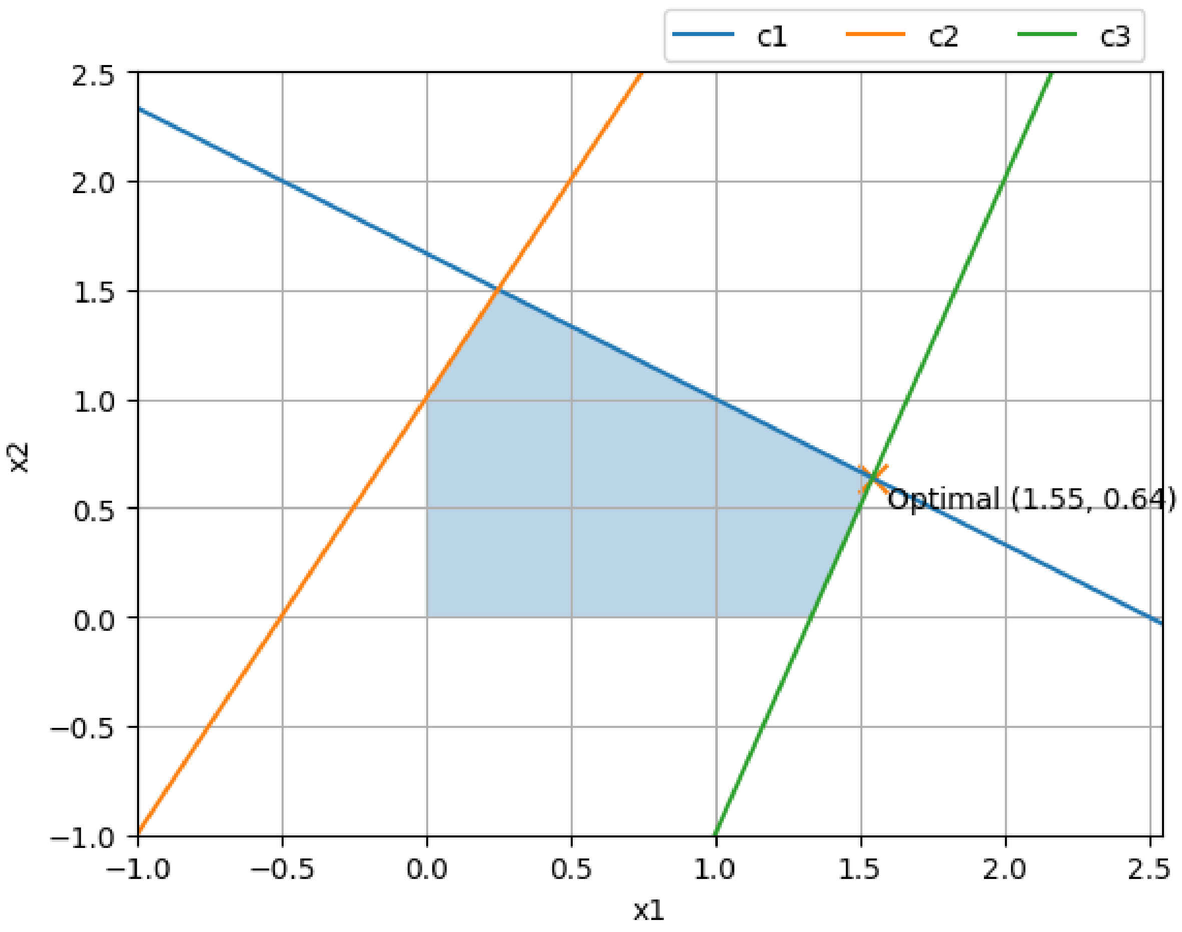
Then, we extract information of the problem:

```
These following parameters will be used for The Standard Form in Scipy
    or any program
If the problem is maximizing, we will convert it to minimizing by
    defining c=-c, then we change the sign of the optimal value
```

```
c =  [-1. -1.]
A_ub =  [[ 2.  3.]
 [-2.  1.]
 [ 3. -1.]]
b_ub =  [5. 1. 4.]
A_eq =  None
b_ed =  None
```

We want to solve through the graphical method, so we call

```
plot_lp_geometric_with_legend(A_ub, b_ub, c, bounds, constraint_names=
    None)
```



4 The Simplex Method (using Bland's rule)

4.1 An Analytical Solution

We wish to solve

$$\begin{aligned} \text{maximize} \quad & x_1 \; + \; 2x_2 \; - \; x_3, \\ \text{subject to} \quad & 2x_1 + x_2 + x_3 \le 14, \\ & 4x_1 + 2x_2 + 3x_3 \le 28, \\ & 2x_1 + 5x_2 + 5x_3 \le 30, \\ & x_1, x_2, x_3 \ge 0. \end{aligned}$$

Introduce slack variables $s_1, s_2, s_3 \ge 0$ to turn inequalities into equations:

$$\begin{cases} 2x_1 + 1x_2 + 1x_3 + s_1 = 14, \\ 4x_1 + 2x_2 + 3x_3 + s_2 = 28, \\ 2x_1 + 5x_2 + 5x_3 + s_3 = 30. \end{cases}$$

Our initial dictionary (s_1, s_2, s_3 basic, x_1, x_2, x_3 nonbasic) is

$$\begin{aligned} s_1 &= 14 \; - \; 2x_1 \; - \; 1x_2 \; - \; 1x_3, \\ s_2 &= 28 \; - \; 4x_1 \; - \; 2x_2 \; - \; 3x_3, \\ s_3 &= 30 \; - \; 2x_1 \; - \; 5x_2 \; - \; 5x_3, \\ z &= 0 \; + \; 1x_1 \; + \; 2x_2 \; - \; 1x_3. \end{aligned}$$

Pivot 1: Enter x_2, Leave s_3

Largest positive coefficient in the z-row is $+2$ at x_2. Ratio test:

$$\frac{s_1}{1} = 14, \quad \frac{s_2}{2} = 14, \quad \frac{s_3}{5} = 6 \quad \Longrightarrow \quad s_3 \text{ leaves.}$$

Solve the third equation for x_2:

$$x_2 = 6 \; - \; 0.4\,x_1 \; - \; 1\,x_3 \; - \; 0.2\,s_3.$$

Substitute into the other rows to obtain the new dictionary:

$$\begin{aligned} s_1 &= 8 \; - \; 1.6x_1 \; + \; 0.2\,s_3, \\ s_2 &= 16 \; - \; 3.2x_1 \; - \; 1x_3 \; + \; 0.4\,s_3, \\ x_2 &= 6 \; - \; 0.4x_1 \; - \; 1\,x_3 \; - \; 0.2\,s_3, \\ z &= 12 \; + \; 0.2x_1 \; - \; 3x_3 \; - \; 0.4\,s_3. \end{aligned}$$

Pivot 2: Enter x_1, Leave s_1

Now the only positive coefficient in the z-row is $+0.2$ at x_1. Ratio test:

$$\frac{s_1}{1.6} = 5, \quad \frac{s_2}{3.2} = 5, \quad \frac{x_2}{0.4} = 15 \quad \Longrightarrow \quad s_1 \text{ leaves.}$$

Solve the first equation for x_1:

$$x_1 = 5 \; - \; 0.625\,s_1 \; + \; 0.125\,s_3.$$

Substitute into the remaining rows:

$$\begin{aligned}
x_1 &= 5 \;-\; 0.625\, s_1 \;+\; 0.125\, s_3,\\
s_2 &= 2\, s_1 \;-\; 1 x_3,\\
x_2 &= 4 \;+\; 0.25\, s_1 \;-\; 1 x_3 \;-\; 0.25\, s_3,\\
z &= 13 \;-\; 0.125\, s_1 \;-\; 3 x_3 \;-\; 0.375\, s_3.
\end{aligned}$$

Optimal Solution

All coefficients in the final z-row are non-positive, so we have reached optimality. Setting the nonbasics $s_1 = s_3 = x_3 = 0$ gives

$$x_1 = 5, \quad x_2 = 4, \quad x_3 = 0, \qquad z_{\max} = 13.$$

4.2 Using Python

We enter the problem

```
Enter objective function:
> maximize z = x1 + 2x2 - x3
Enter constraints (one per line, type 'done' to finish):
> 2x1 + x2 + x3 <= 14
> 4x1 + 2x2 + 3x3 <= 28
> 2x1 + 5x2 + 5x3 <= 30
> done
```

Next, extracting

```
c =  [-1. -2. -1.]
A_ub =  [[2. 1. 1.]
 [4. 2. 3.]
 [2. 5. 5.]]
b_ub =  [14. 28. 30.]
A_eq =  None
b_ed =  None
```

Using simplex method

```
if lines[0].lower().startswith('maximize'):
    simplex(-c, A_ub, b_ub)
else:
    simplex(c,A_ub,b_ub)
```

and obtain

```
Initial Dictionary:
s1 = 14.0-2*x1-1*x2-1*x3
s2 = 28.0-4*x1-2*x2-3*x3
s3 = 30.0-2*x1-5*x2-5*x3
z = 0+1*x1+2*x2+1*x3

Pivot 1: Enter x2, Leave s3
s1 = 8.0-1.6*x1+0*x3
s2 = 16.0-3.2*x1-1*x3
x2 = 6.0-0.4*x1-1*x3
z = 12.0+0.2*x1-1*x3

Pivot 2: Enter x1, Leave s1
```

```
x1 = 5.0+0*x3
s2 = 0.0-1*x3
x2 = 4.0-1*x3
z = 13.0-1*x3

Solution:
s1 = 0
s2 = 0.0
s3 = 0
x1 = 5.0
x2 = 4.0
x3 = 0
z = 13.0
```

5 The Two Phases Simplex Method

5.1 An Analytical Solution

To begin with, we illustrate the following problem:

$$\begin{aligned} \text{maximize} \quad & x_1 - x_2 + x_3 \\ \text{subject to} \quad & 2x_1 - x_2 + 2x_3 \le 4, \\ & 2x_1 - 3x_2 + x_3 \le -5, \\ & -x_1 + x_2 - 2x_3 \le -1, \\ & x_1, x_2, x_3 \ge 0. \end{aligned}$$

In The First Phase, consider the auxiliary problem, which is equivalent to:

$$\begin{aligned} \text{maximize} \quad & -x_0 \\ \text{subject to} \quad & (2x_1 - x_2 + 2x_3) - x_0 \le 4, \\ & (2x_1 - 3x_2 + x_3) - x_0 \le -5, \\ & (-x_1 + x_2 - 2x_3) - x_0 \le -1, \\ & x_{0,1,2,3} \ge 0. \end{aligned}$$

Normally, we solve the auxiliary LP by introducing slack variables to form the initial dictionary, but we still get an infeasible starting dictionary:

$$\begin{aligned} x_4 &= 4 - 2x_1 + x_2 - 2x_3 + x_0 \\ x_5 &= -5 - 2x_1 + 3x_2 - x_3 + x_0 \\ x_6 &= -1 + x_1 - x_2 + 2x_3 + x_0 \\ w &= -x_0 \end{aligned}$$

However, this will becomes feasible after just one pivot. After special pivoting we get:

$$\begin{aligned} x_0 &= 5 + 2x_1 - 3x_2 + x_3 + x_5 \\ x_4 &= 9 + 0x_1 - 2x_2 - x_3 + x_5 \\ x_6 &= 4 + 3x_1 - 4x_2 + 3x_3 + x_5 \\ w &= -5 - 2x_1 + 3x_2 - x_3 - x_5. \end{aligned}$$

This dictionary is feasible!
x_2 enters, x_6 exits:

$$\begin{aligned} x_2 &= 1 + \tfrac{3}{4}x_1 + \tfrac{3}{4}x_3 + \tfrac{1}{4}x_5 - \tfrac{1}{4}x_6 \\ x_0 &= 2 - \tfrac{1}{4}x_1 - \tfrac{5}{4}x_3 + \tfrac{1}{4}x_5 + \tfrac{3}{4}x_6 \\ x_4 &= 7 - \tfrac{3}{2}x_1 - \tfrac{5}{2}x_3 + \tfrac{1}{2}x_5 + \tfrac{1}{2}x_6 \\ w &= -2 + \tfrac{1}{4}x_1 + \tfrac{5}{4}x_3 - \tfrac{1}{4}x_5 - \tfrac{3}{4}x_6 \end{aligned}$$

Now x_3 enters, x_0 exits:

$$\begin{aligned} x_2 &= \tfrac{11}{5} + \tfrac{3}{5}x_1 + \tfrac{2}{5}x_5 + \tfrac{1}{5}x_6 - \tfrac{3}{5}x_0 \\ x_3 &= \tfrac{8}{5} - \tfrac{1}{5}x_1 + \tfrac{1}{5}x_5 + \tfrac{3}{5}x_6 - \tfrac{4}{5}x_0 \\ x_4 &= 3 - x_1 - x_6 + 2x_0 \\ w &= -x_0 \end{aligned}$$

Then, we have the initial feasible dictionary for the original problem as follows:

$$\begin{aligned} x_2 &= \tfrac{11}{5} + \tfrac{3}{5}x_1 + \tfrac{2}{5}x_5 + \tfrac{1}{5}x_6 \\ x_3 &= \tfrac{8}{5} - \tfrac{1}{5}x_1 + \tfrac{1}{5}x_5 + \tfrac{3}{5}x_6 \\ x_4 &= 3 - x_1 - x_6 \\ z &= \tfrac{3}{5} + \tfrac{1}{5}x_1 - \tfrac{1}{5}x_5 + \tfrac{2}{5}x_6. \end{aligned}$$

It is obvious that the solution associated with this dictionary is feasible! Then, we turn into the next phase as solving this problem by The Simplex Method.
For x_6 enters, x_4 exits and we get the final dictionary (which is optimal):

$$\begin{aligned} x_2 &= \tfrac{14}{5} + \tfrac{2}{5}x_1 - \tfrac{1}{5}x_4 + \tfrac{2}{5}x_5 \\ x_3 &= \tfrac{17}{5} - \tfrac{4}{5}x_1 - \tfrac{3}{5}x_4 + \tfrac{1}{5}x_5 \\ x_6 &= 3 - x_1 - x_4 \\ z &= \tfrac{3}{5} - \tfrac{1}{5}x_1 - \tfrac{2}{5}x_4 - \tfrac{1}{5}x_5 \end{aligned}$$

After all, we achieve the maximal $\frac{3}{5}$

5.2 Using Python

First of all, we need to enter the problem that requests the coefficients of b are nonnegative.

```
Enter objective function:
> maximize z=x1 - x2 + x3
Enter constraints (one per line, type 'done' to finish):
> 2x1 - x2 + 2x3 <= 4
> -2x1 + 3x2 - x3 >= 5
> x1 - x2 + 2x3 >= 1
> done
```

Extracting the coefficients

```
c =  [-1.  1. -1.]
A_ub =  [[ 2. -1.  2.]
 [ 2. -3.  1.]
 [-1.  1. -2.]]
b_ub =  [ 4. -5. -1.]
A_eq =  None
b_ed =  None
```

Calling the source:

```python
if lines[0].lower().startswith('maximize'):
    solver = TwoPhaseSimplexSolver(-c, A_ub, b_ub)
    result = solver.solve()
else:
    solver = TwoPhaseSimplexSolver(c, A_ub, b_ub)
    result = solver.solve()

if result:
    if result == "unbounded":
        print("\nThe LP problem is unbounded.")
    else:
        opt_val, sol_vars = result
        print(f"\n--- Summary ---")
        print(f"Maximum value of z: {opt_val} (approx. {float(opt_val)
    :.6f})")
        print("Solution:")
        for var_name, value in sol_vars.items():
            print(f"  {var_name} = {value}")
```

Then, we get:

```
Starting Phase 1...

--- Phase 1 Initial (before special pivot) - Iteration 0 ---
x4 = 4 - 2x1 + x2 - 2x3 + x0
x5 = -5 - 2x1 + 3x2 - x3 + x0
x6 = -1 + x1 - x2 + 2x3 + x0
w = 0 - x0

Performing special pivot for Phase 1: x0 enters, x5 leaves.

--- Phase 1 - Iteration 1 ---
x4 = 9 - 2x2 - x3 + x5
x6 = 4 + 3x1 - 4x2 + 3x3 + x5
x0 = 5 + 2x1 - 3x2 + x3 + x5
w = -5 - 2x1 + 3x2 - x3 - x5

Entering variable: x2 (coeff: 3)
Leaving variable: x6 (ratio: 1)

--- Phase 1 - Iteration 2 ---
x2 = 1 + 3/4x1 + 3/4x3 + 1/4x5 - 1/4x6
x4 = 7 - 3/2x1 - 5/2x3 + 1/2x5 + 1/2x6
x0 = 2 - 1/4x1 - 5/4x3 + 1/4x5 + 3/4x6
w = -2 + 1/4x1 + 5/4x3 - 1/4x5 - 3/4x6

Entering variable: x3 (coeff: 5/4)
Leaving variable: x0 (ratio: 8/5)

--- Phase 1 - Iteration 3 ---
x2 = 11/5 + 3/5x1 + 2/5x5 + 1/5x6 - 3/5x0
x3 = 8/5 - 1/5x1 + 1/5x5 + 3/5x6 - 4/5x0
x4 = 3 - x1 - x6 + 2x0
w = 0 - x0

Optimal solution found in Phase 1.

Phase 1 successful. Optimal w = 0 (x0 = 0).
```

```

Starting Phase 2...

--- Preparing for Phase 2 ---

--- Phase 2 - Iteration 3 ---
x2 = 11/5 + 3/5x1 + 2/5x5 + 1/5x6
x3 = 8/5 - 1/5x1 + 1/5x5 + 3/5x6
x4 = 3 - x1 - x6
z = -3/5 + 1/5x1 - 1/5x5 + 2/5x6

Entering variable: x6 (coeff: 2/5)
Leaving variable: x4 (ratio: 3)

--- Phase 2 - Iteration 4 ---
x2 = 14/5 + 2/5x1 - 1/5x4 + 2/5x5
x3 = 17/5 - 4/5x1 - 3/5x4 + 1/5x5
x6 = 3 - x1 - x4
z = 3/5 - 1/5x1 - 2/5x4 - 1/5x5

Optimal solution found in Phase 2.

Final Optimal Dictionary (Phase 2):

--- Phase 2 Final Optimal - Iteration 4 ---
x2 = 14/5 + 2/5x1 - 1/5x4 + 2/5x5
x3 = 17/5 - 4/5x1 - 3/5x4 + 1/5x5
x6 = 3 - x1 - x4
z = 3/5 - 1/5x1 - 2/5x4 - 1/5x5

Optimal solution found: z = 3/5
Basic variable values:
  x2 = 14/5
  x3 = 17/5
  x6 = 3
Original variable values at optimum:
  x1 = 0
  x2 = 14/5
  x3 = 17/5

--- Summary ---
Maximum value of z: 3/5 (approx. 0.600000)
Solution:
  x1 = 0
  x2 = 14/5
  x3 = 17/5
```

6 The Dual Simplex Method

We will illustrate with the following example:

$$\begin{aligned} \text{maximize} \quad & x_1 - 2x_2 \\ \text{subject to} \quad & x_1 + 3x_2 \le 4, \\ & x_1 - 4x_2 \le 2, \\ & x_1, x_2 \ge 0. \end{aligned} \tag{1}$$

We could convert it into **the dual of the original LP**:

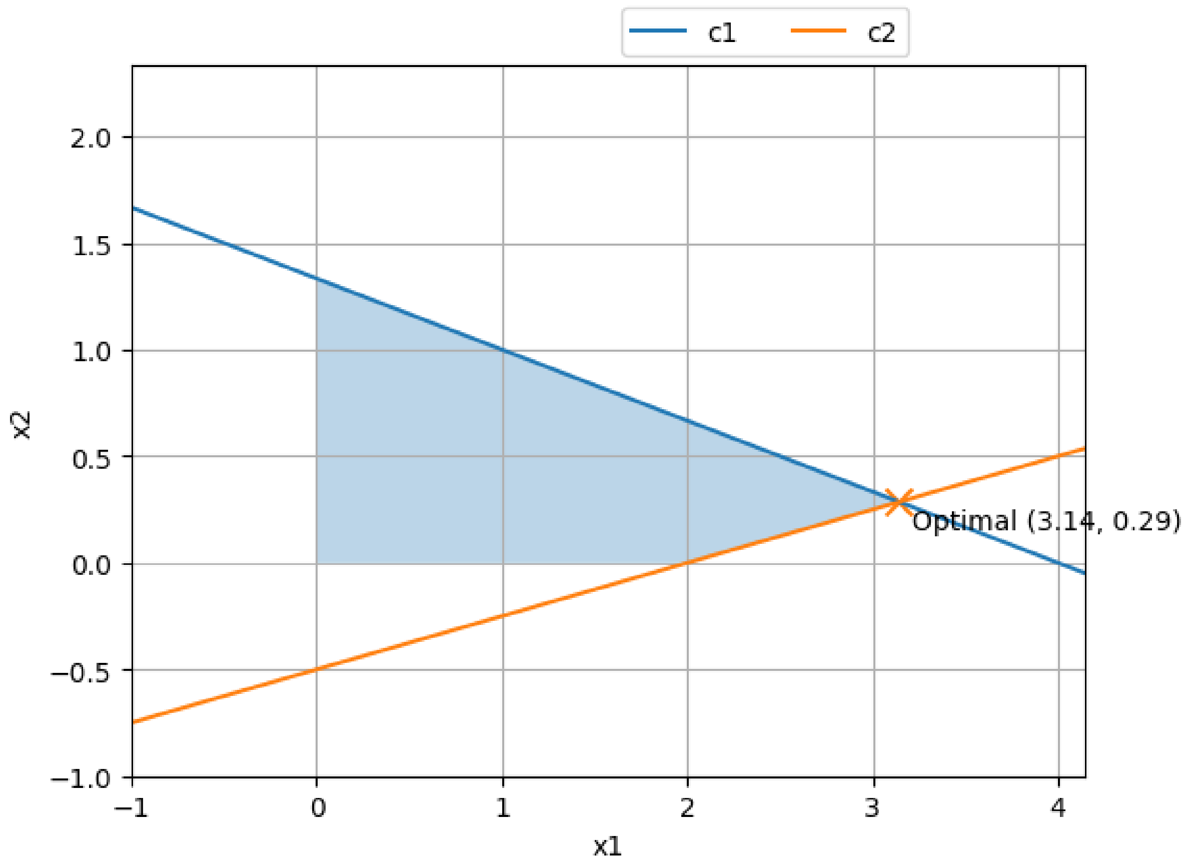
$$\begin{aligned} \text{minimize} \quad & 4y_1 + 2y_2 \\ \text{subject to} \quad & y_1 + y_2 \geq 1, \\ & 3y_1 - 4y_2 \geq -2, \\ & y_1, y_2 \geq 0. \end{aligned} \tag{2}$$

Then, we will use the simplex method or the simplex method with two phases to solve this.
Now, I will use the python program to solve this:

```
--- Primal Problem (Original) ---
Maximize Z = 1.00x1 + -2.00x2
  1.00x1 + 3.00x2 <= 4.00
  1.00x1 + -4.00x2 <= 2.00
  x1 , x2 >= 0

--- Setting up to solve the Dual Problem ---

Dual Task (formulation passed to solver):
Obj: Min W = 4.00y1 + 2.00y2
Constraint 1: -1.00y1 + -1.00y2 <= -1.00
Constraint 2: -3.00y1 + 4.00y2 <= 2.00

--- Solving the Dual Problem ---

Initial Tableau (Iteration 0)
Basis |   y1   |   y2   |   s1   |   s2   |    b    |
-----------------------------------------------------
s1    |  -1.00 |  -1.00 |   1.00 |   0.00 |  -1.00 |
s2    |  -3.00 |   4.00 |   0.00 |   1.00 |   2.00 |
-----------------------------------------------------
  F   |  -4.00 |  -2.00 |   0.00 |   0.00 |   0.00 |

--- Primal Simplex Step (Phase 2) Iteration 1 ---
Basis |   y1   |   y2   |   s1   |   s2   |    b    |
-----------------------------------------------------
y1    |   1.00 |   0.00 |  -0.57 |  -0.14 |   0.29 |
y2    |   0.00 |   1.00 |  -0.43 |   0.14 |   0.71 |
-----------------------------------------------------
  F   |   0.00 |   0.00 |  -3.14 |  -0.29 |   2.57 |

--- Primal Simplex Step (Phase 2) Iteration 2 ---

--- Optimality Condition Met ---

--- Optimal Solution Found ---

Optimal solution for the DUAL problem found.
Solution for dual variables (y):
  y1 = 0.2857
  y2 = 0.7143
Optimal value for DUAL problem (W): 2.5714

By strong duality, optimal value for PRIMAL problem (Z): 2.5714
Solution for PRIMAL variables (x):
  x1 = 3.1429
```

```
x2 = 0.2857
```

Then, I check it again by The Graphical Method, which is completely equivalent to this solution.



7 Advanced Methods in Linear Programming

7.1 Some Advanced Methods

While the Simplex method is a cornerstone of linear programming and highly effective for many practical problems, research has led to several advanced methods. These methods often address the Simplex algorithm's worst-case exponential time complexity or are tailored for very large-scale problems or problems with specific structures.

Interior-Point Methods

Interior-Point Methods (IPMs) gained prominence with Karmarkar's algorithm in 1984 and represent a significant alternative to the Simplex method. Instead of moving along the edges of the feasible polytope, IPMs traverse its interior to reach the optimal solution. Key characteristics include:

- **Polynomial-time complexity:** They were the first class of algorithms for LP with provably polynomial worst-case time complexity.
- **Efficiency for large-scale problems:** IPMs are often more efficient than Simplex methods for very large LPs.

- **Primal-Dual variants:** Most modern IPMs are primal-dual methods, which simultaneously solve the primal and dual LPs.

Ellipsoid Method

Introduced by L. G. Khachiyan in 1979, the Ellipsoid Method was the first polynomial-time algorithm for linear programming. It works by iteratively shrinking an ellipsoid that contains the optimal solution.

- **Theoretical importance:** Its main impact was theoretical, proving that LP problems belong to the class P (solvable in polynomial time).
- **Practical performance:** It is generally not competitive with the Simplex method or IPMs in practice for solving LPs but has applications in other areas of optimization.

Decomposition Techniques

For very large linear programs that exhibit special structures, decomposition techniques can be employed to break the problem into smaller, more manageable subproblems. The solutions to these subproblems are then coordinated to solve the original (master) problem.

- **Dantzig-Wolfe Decomposition:** This technique is suitable for problems with a block-angular structure in their constraints. It reformulates the problem by expressing the solution in terms of extreme points of the subproblem polyhedra. Column generation is often used to solve the master problem.
- **Benders Decomposition:** Useful for problems where fixing a subset of variables makes the remaining subproblem significantly easier (e.g., an LP or a network flow problem). It involves iterating between solving a relaxed master problem and subproblems that generate ”Benders cuts” to refine the master problem.

Cutting Plane Methods

Primarily developed for integer linear programming (ILP) and mixed-integer linear programming (MILP), cutting plane methods iteratively refine the feasible region of the LP relaxation of an integer program.

- **Mechanism:** They add linear inequalities (cuts) that separate the current fractional LP solution from the convex hull of feasible integer solutions, without removing any integer solutions.
- **Goal:** To tighten the LP relaxation, providing better bounds and facilitating the search for an integer optimal solution, often within a branch-and-bound framework (then called branch-and-cut).

7.2 Programming Languages

Various open-source projects provide implementations for solving linear programming problems. These can be found on platforms like GitHub. For example, Google's OR-Tools [5] offers a comprehensive suite for optimization. SciPy includes linear programming solvers [6] as part of its optimization module. PuLP [7] is a popular LP modeler in Python. Here are some notable GitHub repositories related to linear programming:

- **Google OR-Tools:** A suite for combinatorial optimization, including LP. Available at https://github.com/google/or-tools.
- **SciPy (optimize module):** Contains LP solvers like Simplex and Interior-Point methods. The relevant code can be explored starting from https://github.com/scipy/scipy/tree/main/scipy/optimize.
- **PuLP:** A Python library for modeling LP problems. Hosted at https://github.com/coin-or/pulp.
- **HiGHS Solver:** A high-performance open-source LP solver. Found at https://github.com/ERGO-Code/HiGHS.
- **GLPK (Unofficial Mirror):** GNU Linear Programming Kit. An unofficial mirror is at https://github.com/glpk/glpk (official source is GNU Savannah).

References

[1] Bertsimas, D., & Tsitsiklis, J. N. (1997). *Introduction to Linear Optimization.* Athena Scientific. ISBN: 978-1886529009.

[2] Bazaraa, M. S., Jarvis, J. J., & Sherali, H. D. (2009). *Linear Programming and Network Flows* (4th ed.). John Wiley & Sons. ISBN: 978-0470462720.

[3] Chvátal, V. (1983). *Linear Programming.* W. H. Freeman and Company. ISBN: 978-0716715870.

[4] Wright, S. J. (1997). *Primal-Dual Interior-Point Methods.* Society for Industrial and Applied Mathematics (SIAM). ISBN: 978-0898713821.

[5] Google OR-Tools Team. (2024). *Google Optimization Tools (OR-Tools).* GitHub Repository. Retrieved from https://github.com/google/or-tools

[6] SciPy committers. (2024). *SciPy - Scientific Library for Python (scipy.optimize).* GitHub Repository. Retrieved from https://github.com/scipy/scipy

[7] COIN-OR Foundation & PuLP project contributors. (2024). *PuLP - A Linear Programming Modeler in Python.* GitHub Repository. Retrieved from https://github.com/coin-or/pulp

[8] ERGO-Code & HiGHS project contributors. (2024). *HiGHS - High performance serial and parallel software for linear optimization.* GitHub Repository. Retrieved from https://github.com/ERGO-Code/HiGHS