Non Linear Optimization Project

Name: Mona Adel Atteya Ahmed Awad

ID: 22010271

Table of Contents

1 Introduction

Implementation of Linear Programming Techniques for Solving Constrained Optimization Problems with a Simple User Interface.

The following report discuss the development of a Python program that can solve constrained optimization problems using Linear Programming (LP) techniques.

The program will allow the user to choose between:

- Graphical method
- Simplex method

The program has multiple functionalities

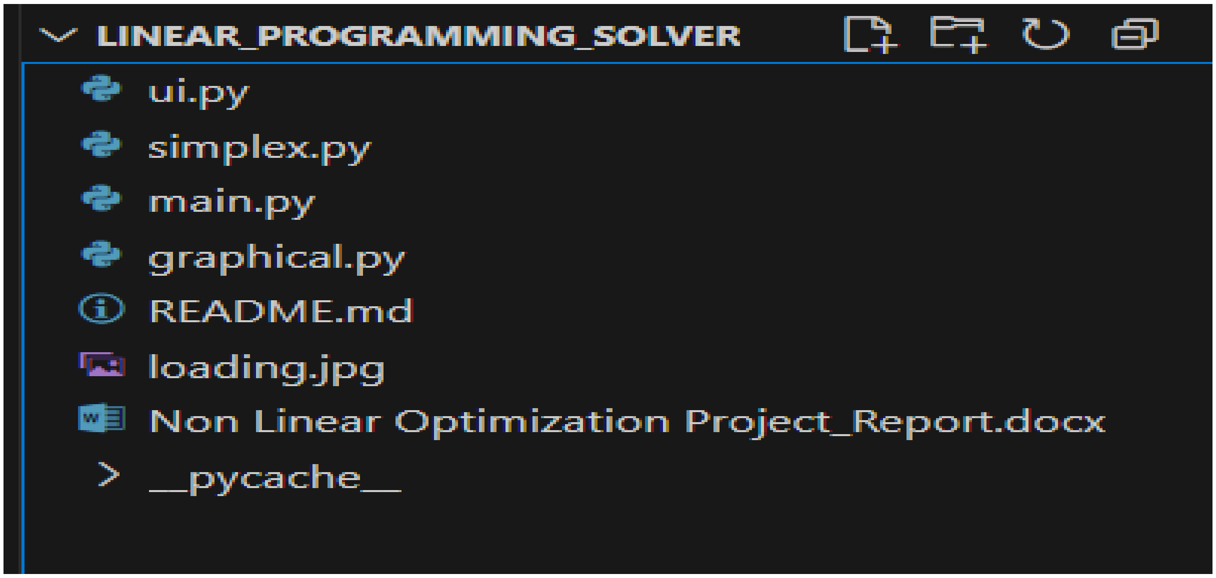
- Entering the problem data
- Determine the optimal solution
- Display results either visually or textually

Libraries used throughout the project:

- Numpy
- Matplotlib.pyplot
- Tkinter for User Interface

We will go through the implementation of the LP methods , User interface and finally a guide for how to run the code as well as using the program followed along with screenshots.

2 Project Structure



Ui.py: A simple user interface

Simplex.py: the simplex method algorithm

Graphical.py: the graphical method algorithm

Main.py: the **entry point** that launches the Tkinter GUI app.

README.md: provides a guide, explaining purpose, required libraries, project structure, how it works and features

3 Graphical Method

3.1 Logic

The graphical method for solving linear programming problems is a powerful visualization tool for problems with two variables. By plotting constraints and identifying the feasible region, one can find the optimal solution by evaluating the objective function at the corner points. This method not only provides insights into the problem but also helps in understanding the impact of each constraint on the solution. However, for problems with more than two variables, other techniques such as the Simplex method are required.

Needed inputs:

- Type of optimization (Max/Min)
- Objective functions and constraints

This program assumes that the non negativity constraint is always true.

3.2 Program Flow Analysis

```python
import numpy as np
import matplotlib.pyplot as plt

def format_constraint(a, b, c, sense):
    """Formats constraint equation as a string"""
    parts = []
    if a != 0:
        parts.append(f"{a:.2f}x" if abs(a) != 1 else ("x" if a > 0 else "-x"))
    if b != 0:
        sign = "+" if (b > 0 and parts) else ("-" if b < 0 else "")
        coeff = abs(b) if abs(b) != 1 else ""
        parts.append(f" {sign} {coeff}y" if coeff else f" {sign} y")
    return f"{''.join(parts)} {sense} {c:.2f}"
```

A helper function used lately in code to plot a legend with the objective function and constraints.

```python
def graphical_solver(opt_type, obj_coeffs, constraints):
```

The main function

```python
    def satisfies(x, y, a, b, c, sense):
        if sense == "<=":
            return a * x + b * y <= c + 1e-9
        elif sense == ">=":
            return a * x + b * y >= c - 1e-9
        elif sense == "=":
            return np.isclose(a * x + b * y, c, atol=1e-9)
        return False

    cons_funcs = [lambda X, Y, a=a, b=b, c=c, s=sense: satisfies(X, Y, a, b, c, s) for a, b, c, sense in constraints]
```

Constraint checker function. (Note: tolerance of 1e-9 to handle floating point rounding errors)

```python
# Find intersection points
points = []
for i in range(len(constraints)):
    for j in range(i+1, len(constraints)):
        a1, b1, c1, _ = constraints[i]
        a2, b2, c2, _ = constraints[j]
        A = np.array([[a1, b1], [a2, b2]])
        B = np.array([c1, c2])
        if np.linalg.det(A) != 0:
            x, y = np.linalg.solve(A, B)
            if x >= -1e-9 and y >= -1e-9:
                if all(cf(x, y) for cf in cons_funcs):
                    points.append((x, y))

# Axis intercepts
for a, b, c, sense in constraints:
    if a != 0:
        x = c / a
        if x >= -1e-9 and all(cf(x, 0) for cf in cons_funcs):
            points.append((x, 0))
    if b != 0:
        y = c / b
        if y >= -1e-9 and all(cf(0, y) for cf in cons_funcs):
            points.append((0, y))

# Unique points
unique_points = []
for p in points:
    if not any(np.isclose(p[0], q[0]) and np.isclose(p[1], q[1]) for q in unique_points):
        unique_points.append(p)
```

1. Find Intersection Points

Solve the system of equations for each pair to find intersection points.

We have a loop to access every unique point stored as tuples

Note: a1, b1, c1, _ = constraints[i] → we ignored the sense variable as it is not needed here.

We check if lines aren't parallel then we solve for (x,y)

Before appending the intersection point we should ensure the non negativity constraint and that the point satisfies every single constraint.

2. Find Axis Intercepts

Assume the coefficient of y=0 and find x then the coefficient of x=0 and find y finally append each point to points list after making sure it satisfies every constraint.

3. Remove Duplicates

Start with empty list unique_points and append points to it if no match found

```python
# Status defaults
status = "optimal"
opt_val = None
opt_pts = []

# Check Infeasiblity
if not unique_points:
    print ("No feasible area solution")
    status = "infeasible"

# Check unbounded solution
elif status != "infeasible":
    obj_dir = np.array(obj_coeffs, dtype=float)
    if opt_type == 'min':
        obj_dir = -obj_dir
    limiting_constraints = 0
    for (a, b, c, sense) in constraints:
        n = np.array([a, b], dtype=float)
        proj = np.dot(n, obj_dir)
        if sense == "<=" and proj > 1e-9:
            limiting_constraints += 1
        elif sense == ">=" and proj < -1e-9:
            limiting_constraints += 1
        elif sense == "=":
            limiting_constraints += 1
    if limiting_constraints == 0:
        status = "unbounded"
```

1. Check infeasibility , if no unique_points then there is no points that satisfies all the constraints and no solution is found

Check unbounded solution by examining constraint directions relative to objective

How this was done?

- A variable **limiting_constraints** is set to zero. This counter will track the number of constraints that restrict movement in the objective function's improvement direction.
- Each constraint is represented as (a, b, c, sense), corresponding to: **ax+by (sense) c**

For each constraint:

1. **Convert to Normal Vector:**
   The coefficients (a, b) are stored in vector form (n).

2. **Project the Objective Direction:**
   The projection of the constraint's normal vector onto the objective direction vector is computed: ***proj= n . obj_dir***

This determines whether movement in the improvement direction will increase or decrease the left-hand side (LHS) of the constraint.

- Determine limiting behavior:
  - For a <= constraint: If `proj > 1e-9`, movement in the objective direction increases the LHS, so this constraint **limits** progress.
  - For a >= constraint: If `proj < -1e-9`, movement in the objective direction decreases the LHS, so this constraint **limits** progress.

    - For an = constraint: This constraint always limits movement because the solution must remain exactly on the defined line.

- **Update Counter ,**if the constraint limits movement, increment limiting_constraints by 1.
- If, after checking all constraints, limiting_constraints remains **zero**, it means there are no constraints restricting movement in the improvement direction. In this case, the solution is classified as **unbounded**.

```python
# If optimal, compute optimum
if status == "optimal":
    values = [(p, obj_coeffs[0] * p[0] + obj_coeffs[1] * p[1]) for p in unique_points]
    if opt_type == 'max':
        opt_val = max(v for _, v in values)
    else:
        opt_val = min(v for _, v in values)
    opt_pts = [p for p, v in values if np.isclose(v, opt_val)]
            # Multiple optimal solutions detection
    if len(opt_pts) > 1:
        p1, p2 = opt_pts[0], opt_pts[-1]
        print(f"Multiple optimal solutions between {p1} and {p2}")
        print(f"All optimal points: P(λ) = λ*{p1} + (1 - λ)*{p2}, 0 ≤ λ ≤ 1")
```

if optimal then compute optimum based on the optimization type (max/min) by calculating z among all feasible points and store the value in opt_val

The list opt_pts is created by selecting all feasible points whose objective value is numerically equal (within floating-point tolerance) to opt_val.

If the list opt_pts contains **more than one point**, it means multiple optimal solutions exist.

In such cases:

- The algorithm selects the first (p1) and last (p2) optimal points from opt_pts.
- It reports that multiple optima exist between these two points.
- It also provides the **parametric form** of all optimal points:

$$\boldsymbol{Z = \lambda\ p1 + (1- \lambda)\ p2\ ,\ where\ 0 \leq \lambda \leq 1}$$

```python
# ==== Plotting ====
if unique_points:
    max_x = max(p[0] for p in unique_points) + 2
    max_y = max(p[1] for p in unique_points) + 2
else:
    max_x, max_y = 10, 10  # default range when no feasible points

x_vals = np.linspace(0, max_x, 400)
y_vals = np.linspace(0, max_y, 400)
X, Y = np.meshgrid(x_vals, y_vals)

feasible_region = np.ones_like(X, dtype=bool)
for cf in cons_funcs:
    feasible_region &= cf(X, Y)

plt.figure(figsize=(8, 8))
plt.contourf(X, Y, feasible_region, levels=[0.5,1], colors=["#87CEFA"], alpha=0.3)
```

1. Sets plot boundaries based on solution points or defaults
2. Create grid for plotting

np.linspace(start,stop,num)→ Generates evenly spaced numbers over a specified interval.

np.meshgrid(x_vals,y_vals)→ Converts one-dimensional coordinate arrays into a full two-dimensional grid of points. Concept: The result is a grid of coordinates (x , y), where every x from x_vals is paired with every y from y_vals.

3. Compute feasible region by applying all constraints to the grid then plot it using plt.contourf (filled contour plots)

```python
# Constraint lines
constraint_colors = plt.cm.tab10(np.linspace(0, 1, len(constraints)))  # Creates a color spectrum

legend_handles = []

# Plot each constraint with unique color and label
for i, (a, b, c, sense) in enumerate(constraints):
    color = constraint_colors[i]  # Assign unique color to each constraint
    label = f"C{i+1}: {format_constraint(a, b, c, sense)}"

    if b != 0:  # Non-vertical line
        plt.plot(x_vals, (c - a * x_vals) / b,
                 linestyle='--',
                 color=color,
                 label=f"Constraint {i+1}: {a:.1f}x + {b:.1f}y {sense} {c:.1f}")
    else:  # Vertical line
        plt.axvline(c / a,
                 linestyle='--',
                 color=color,
                 label=f"Constraint {i+1}: {a:.1f}x {sense} {c:.1f}")

    # Create a custom legend entry
    legend_handles.append(plt.Line2D([], [], color=color, linestyle='--', linewidth=2,label=label))

# Add objective function to legend
obj_label = f"Obj: z = {obj_coeffs[0]:.2f}x + {obj_coeffs[1]:.2f}y ({opt_type})"
legend_handles.append(
    plt.Line2D([], [], color='black', linestyle='-', linewidth=2,label=obj_label))

# Add optimal point to legend
opt_color = 'green' if opt_type == 'max' else 'gold'
opt_label = f"Optimal ({opt_type})"
legend_handles.append(plt.Line2D([], [], color=opt_color, marker='o', linestyle='None',markersize=10, label=opt_label))
```

This code block is responsible for plotting the constraint lines, the objective function, and the optimal point.

1. Each constraint is drawn as a dashed line with a unique color for clarity.
2. The objective function is shown as a solid black line.
3. The optimal solution is highlighted with a colored point (green for maximization, yellow for minimization).
4. A legend is created to explain each line and the optimal point, making the plot easy to read and interpret. **(legend_handles)**

```
# All intersection points
for p in unique_points:
    plt.plot(p[0], p[1], 'ro')
    plt.text(p[0] + 0.1, p[1] + 0.1, f"({p[0]:.2f},{p[1]:.2f})", fontsize=8)

# Optimal points
for p in opt_pts:
    plt.plot(p[0], p[1], 'go' if opt_type == 'max' else 'yo', markersize=10)

plt.xlabel('x')
plt.ylabel('y')
plt.xlim(left=0)  # Start x-axis at 0
plt.ylim(bottom=0)  # Start y-axis at 0
plt.title(f"LP Graphical Solution - {status.capitalize()}")
plt.legend(handles=legend_handles, loc='upper right', bbox_to_anchor=(1.3, 1))
plt.tight_layout()
plt.grid(True)
plt.show()

return {'status': status, 'opt_value': opt_val, 'opt_points': opt_pts}
```

1. Marks all feasible corner points with red dots and coordinates and adds a text label with the point coordinates near the point
2. Highlights optimal points with green (max) or yellow (min) markers
3. Adds labels, title, grid, legend and displays the plot
4. Finally returns solution information as a dictionary

3.3 Conclusion

The code successfully implements the graphical method for linear programming by calculating intersection points, checking feasibility, and determining the optimal solution. It also distinguishes between maximization and minimization problems, highlights optimal points on the graph, and even detects multiple optimal solutions when they occur. Overall, the code is efficient, clear, and provides both numerical and visual outputs that make the results easy to understand.

4 Simplex Method

4.1 Logic

The first step involved in the simplex method is to construct an auxiliary prob lem by introducing certain variables known as artificial variables into the standard form of the linear programming problem. The primary aim of adding the artificial variables is to bring the resulting auxiliary problem into a canonical form from which its basic feasible solution can be obtained immediately. Starting from this canonical form, the optimal solution of the original linear programming problem is sought in two phases. The first phase is intended to find a basic feasible solution to the orig inal linear programming problem. It consists of a sequence of pivot operations that produces a succession of different canonical forms from which the optimal solution of the auxiliary problem can be found. This also enables us to find a basic feasible solution, if one exists, to the original linear programming problem. The second phase is intended to find the optimal solution to the original linear programming problem. It consists of a second sequence of pivot operations that enables us to move from one basic feasible solution to the next of the original linear programming problem. In this process, the optimal solution to the problem, if one exists, will be identified. ***~reference: engineering-optimization-theory-and-practice***

Needed inputs:

- Type of optimization (Max/Min)
- Objective functions and constraints

Note: the following program handles only the ≤ constraints using the **slack variables**.

The **surplus and artificial variables** for **[≥, =]** constraints aren't covere

4.2 Program Flow Analysis

```python
class SimplexResult:
    def __init__(self, status, x=None, z=None, message=""):
        self.status = status          # "optimal", "unbounded", "infeasible", "error"
        self.x = x                    # primal solution (original variable space)
        self.z = z                    # objective value
        self.message = message
```

A container to hold the result of the simplex algorithm.

Attributes:

- **status:** String like "optimal", "unbounded", "infeasible", or "error".
- **x:** decision variables
- **z:** The optimal objective value.
- **message:** Extra details or error messages.

```python
    def __repr__(self):
        return f"SimplexResult(status={self.status!r}, x={self.x}, z={self.z}, message={self.message!r})"
```

string representation of the result when printed.

```python
class SimplexSolver:

    def __init__(self, tol=1e-9, max_iter=10_000):
        self.tol = tol
        self.max_iter = max_iter

    def _pivot(self, T, row, col):
```

Initializes the solver

Utilities

```
def _pivot(self, T, row, col):
    pivot = T[row, col] #pivot element
    T[row, :] /= pivot  #Normalize the pivot row
    m, n = T.shape
    for r in range(m): #Gaussian elemination to update the table
        if r != row:
            T[r, :] -= T[r, col] * T[row, :]
```

1.Pivot Operation:

- Normalize the pivot row
- Eliminate the pivot column (Set all other entries to zero)
- Update the table

```
def _choose_entering(self, row):

    candidates = np.where(row < -self.tol)[0]
    if candidates.size == 0:
        return None
    return candidates[np.argmin(row[candidates])]
```

2.Choose the pivot column:

- Pick the smallest negative coefficient at the objective row.
- If ne negative coefficients then status is already optimal

```
def _choose_leaving(self, T, col):
    rhs = T[:-1, -1]
    col_vals = T[:-1, col]
    mask = col_vals > self.tol

    if not np.any(mask):
        return None  # Unbounded
    ratios = rhs[mask] / col_vals[mask]
    idx = np.argmin(ratios)

    leaving_rows = np.where(mask)[0]
    return leaving_rows[idx]
```

3.Choose the pivot row:

- Minimum ratio test (pick the smallest positive ratio) to determine which variable will leave the basis
- If no positive entries then problem is unbounded

```python
def _build_tableau(self, A, b, c):

    m, n = A.shape
    # Add slack variables
    slack_matrix = np.eye(m)
    full_A = np.hstack([A, slack_matrix])
    # Initial basis consists of slack variables
    basis = list(range(n, n + m))
    # Objective row (maximization)
    c_full = np.zeros(n + m)
    c_full[:n] = -c
    c_full[n:] = 0
    # Build tableau
    T = np.zeros((m + 1, n + m + 1))
    T[:-1, :n + m] = full_A
    T[:-1, -1] = b
    T[-1, :n + m] = c_full

    return T, basis
```

Constructs the initial simplex tableau:

- A: constraint matrix.
  - n: number of constraints
  - m: number of variables
- Add slack variables
  - **np.eye(m)** creates an m×m identity matrix (slack variables).
  - **np.hstack([...])** stacks A and the identity side by side.

- Objective row
  - Since we're maximizing, we put `-c` (because simplex looks for negative coefficients to improve the solution).
  - Slack variables don't appear in the objective, so their coefficients = 0.

- Tableau layout:
  - Rows = constraints + 1 objective row → m+1.
  - Columns = variables (n+m) + RHS → n+m+1.

Filling the tableau:

1. **Constraint rows** (T[:-1, :n+m] = full_A)
   - Copies the constraint matrix with slacks.
2. **Right-hand side** (T[:-1, -1] = b)
   - Stores constraint bounds.
3. **Objective row** (T[-1, :n+m] = c_full)
   - Stores the negated objective coefficients.
4. The last cell T[-1, -1] (bottom-right) is 0 at the start.

Conclusion; _build_tableau converts a human-readable LP into a matrix form where simplex operations (pivoting, entering/leaving, ratio test) can be applied.

```python
def _optimize_tableau(self, T, basis):

    iters = 0
    while iters < self.max_iter:
        iters += 1
        obj_row = T[-1, :-1]
        col = self._choose_entering(obj_row)
        if col is None:
            return "optimal", basis

        row = self._choose_leaving(T, col)
        if row is None:
            return "unbounded", basis

        self._pivot(T, row, col)
        basis[row] = col

    return "iteration_limit", basis
```

Runs simplex iterations:

- Choose entering
- Choose leaving
- Pivot operation
- Return optimal, unbounded or iteration limit

```python
def solve(self, c, A, b, maximize=True):

    c = np.array(c, dtype=float).flatten()
    A = np.array(A, dtype=float)
    b = np.array(b, dtype=float).flatten()

    if A.shape[0] != b.shape[0]:
        return SimplexResult("error", message="Number of constraints in A and b don't match.")
    if A.shape[1] != c.shape[0]:
        return SimplexResult("error", message="Objective length must equal number of variables.")
```

Convert inputs to Numpy arrays and validates input dimensions

```python
    # Convert minimization to maximization
    c_eff = c.copy()
    if not maximize:
        c_eff = -c_eff
```

In case of minimization problem we negate the coefficient of the objective function and treat it like a maximization problem.

```python
    # Ensure b >= 0
    for i in range(len(b)):
        if b[i] < 0:
            return SimplexResult("error", message="All b values must be non-negative for <= constraints.")

    # Build and solve tableau
    T, basis = self._build_tableau(A, b, c_eff)
    status, basis = self._optimize_tableau(T, basis)

    if status == "unbounded":
        return SimplexResult("unbounded", message="Objective is unbounded.")
    if status != "optimal":
        return SimplexResult("error", message=f"Simplex did not converge: {status}")
```

```python
    # Extract solution
    m, n = A.shape[0], A.shape[1]
    x = np.zeros(n)

    # Find values of basic variables
    for i in range(m):
        bj = basis[i]
        if bj < n:
            x[bj] = T[i, -1]

    z = T[-1, -1]
    if not maximize:
        z = -z

    return SimplexResult("optimal", x=x, z=z)
```

Extracts the primal solution x from the tableau.
Note: Slack variables are ignored we only need the decision variables

Start with zeros for all decision variables.
For each row in the tableau:
If the basis variable is an original variable, copy its value from RHS into x.
If it's a slack variable, ignore it.
The result is the optimal solution vector x

Finally adjust the objective value for minimization and return the result.

4.3 Conclusion

The code implements a **basic simplex algorithm solver** in a structured and modular way. The process begins with the ***SimplexResult class***, which encapsulates the solver's outputs, ensuring that solutions, objective values, and error states are clearly reported. The ***SimplexSolver*** class then drives the solution process by:

1. **Building the initial tableau** (_build_tableau) with slack variables, ensuring constraints are properly represented in standard form.
2. **Iteratively optimizing** the tableau (_optimize_tableau) through pivot operations, where entering and leaving variables are systematically chosen to improve the objective.
3. **Extracting the final solution** from the tableau once optimality, unboundedness, or iteration limits are reached.

Each helper method—such as ***_pivot***, ***_choose_entering***, and ***_choose_leaving*** performs a focused task, reflecting the mathematical operations of the simplex algorithm. The solve method integrates these components, handling input validation, converting minimization to maximization, and ensuring feasibility of constraints before optimization.

5 User Interface

5.1 Code Flow Analysis

```
import tkinter as tk
from tkinter import ttk, messagebox
from tkinter import *
import numpy as np
from PIL import Image, ImageTk
from graphical import graphical_solver
from simplex import SimplexSolver
```

Importing Required Libraries

- tkinter : Python's built-in GUI library.
- ttk : Themed Tkinter widgets (for a modern look).
- PIL (Pillow) : For loading and displaying images.
- graphical_solver : Custom module for solving LP problems graphically.
- SimplexSolver : Custom class for solving LP problems using the Simplex method.

```
class LPSolverApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Linear Programming Solver")
        self.root.geometry("900x900")
        image=Image.open('loading.jpg')
        self.icon=ImageTk.PhotoImage(image)
        self.root.iconphoto(True,self.icon)

        # Create notebook for different solvers
        self.notebook = ttk.Notebook(root)
        self.notebook.pack(fill='both' ,expand=True)

        # Graphical Solver Tab
        self.create_graphical_tab()

        # Simplex Solver Tab
        self.create_simplex_tab()
```

-root is the main application window.
-Sets the title and size of the window.
-Loads an image (loading.jpg) to use as the app icon.
-A tabbed interface with two tabs :

- Graphical
- Simplex

Graphical Solver Tab Interface

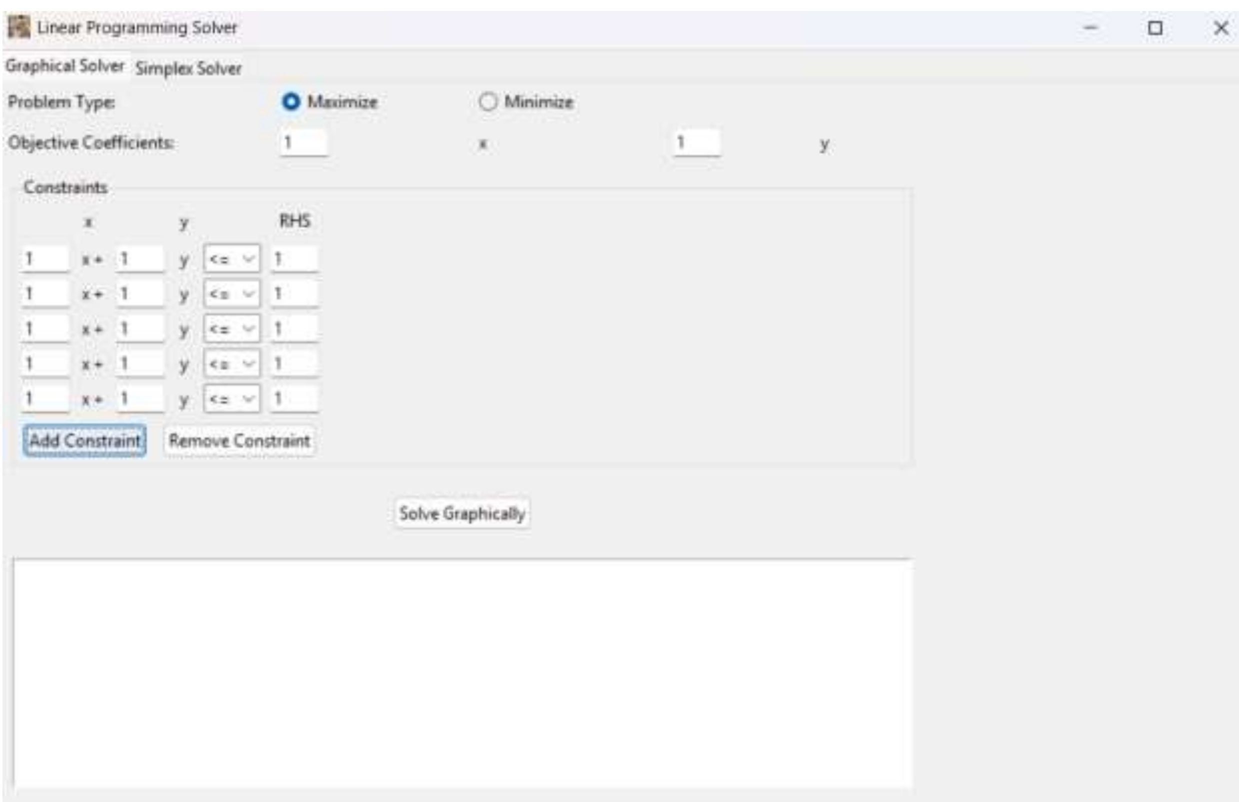
```python
    def create_graphical_tab(self):

        tab = ttk.Frame(self.notebook)
        self.notebook.add(tab, text="Graphical Solver")

        # Problem Type
        ttk.Label(tab, text="Problem Type:").grid(row=0, column=0, padx=5, pady=5, sticky=tk.W)
        self.graphical_opt_type = tk.StringVar(value="max")
        ttk.Radiobutton(tab, text="Maximize", variable=self.graphical_opt_type, value="max").grid(row=0, column=1,sticky=tk.W)
        ttk.Radiobutton(tab, text="Minimize", variable=self.graphical_opt_type, value="min").grid(row=0, column=2,sticky=tk.W)

        # Objective Function
        ttk.Label(tab, text="Objective Coefficients:").grid(row=1, column=0,padx=5, pady=5,sticky=tk.W)
        ttk.Label(tab, text="x").grid(row=1, column=2 ,sticky=tk.W)
        ttk.Label(tab, text="y").grid(row=1, column=4,sticky=tk.W)

        self.obj_x = tk.DoubleVar(value=1)
        self.obj_y = tk.DoubleVar(value=1)
        ttk.Entry(tab, textvariable=self.obj_x, width=5).grid(row=1, column=1,sticky=tk.W)
        ttk.Entry(tab, textvariable=self.obj_y, width=5).grid(row=1, column=3,sticky=tk.W)

        # Constraints Frame
        constraints_frame = ttk.LabelFrame(tab, text="Constraints")
        constraints_frame.grid(row=2, column=0, columnspan=5, padx=10, pady=10, sticky=tk.W+tk.E)

        # Constraints header
        ttk.Label(constraints_frame, text="x  ").grid(row=0, column=2, padx=5, pady=5)
        ttk.Label(constraints_frame, text="y").grid(row=0, column=4, padx=5, pady=5)
        ttk.Label(constraints_frame, text="RHS").grid(row=0, column=6, padx=5, pady=5)
```

```python
        self.constraint_entries = []
        for i in range(3):
            self.add_constraint_row(constraints_frame, i+1)

        # Add/Remove constraint buttons
        button_frame = ttk.Frame(constraints_frame)
        button_frame.grid(row=10, column=0, columnspan=7, pady=5)
        ttk.Button(button_frame, text="Add Constraint", command=lambda: self.add_constraint_row(constraints_frame, len(self.constraint_entries)+1
        ttk.Button(button_frame, text="Remove Constraint", command=self.remove_constraint_row).pack(side=tk.LEFT, padx=5)

        # Solve Button
        ttk.Button(tab, text="Solve Graphically", command=self.solve_graphical).grid(row=3, column=0, columnspan=5, pady=10)

        # Results
        self.graphical_results = Text(tab, height=10, width=80)
        self.graphical_results.grid(row=4, column=0, columnspan=5, padx=10, pady=10)
```

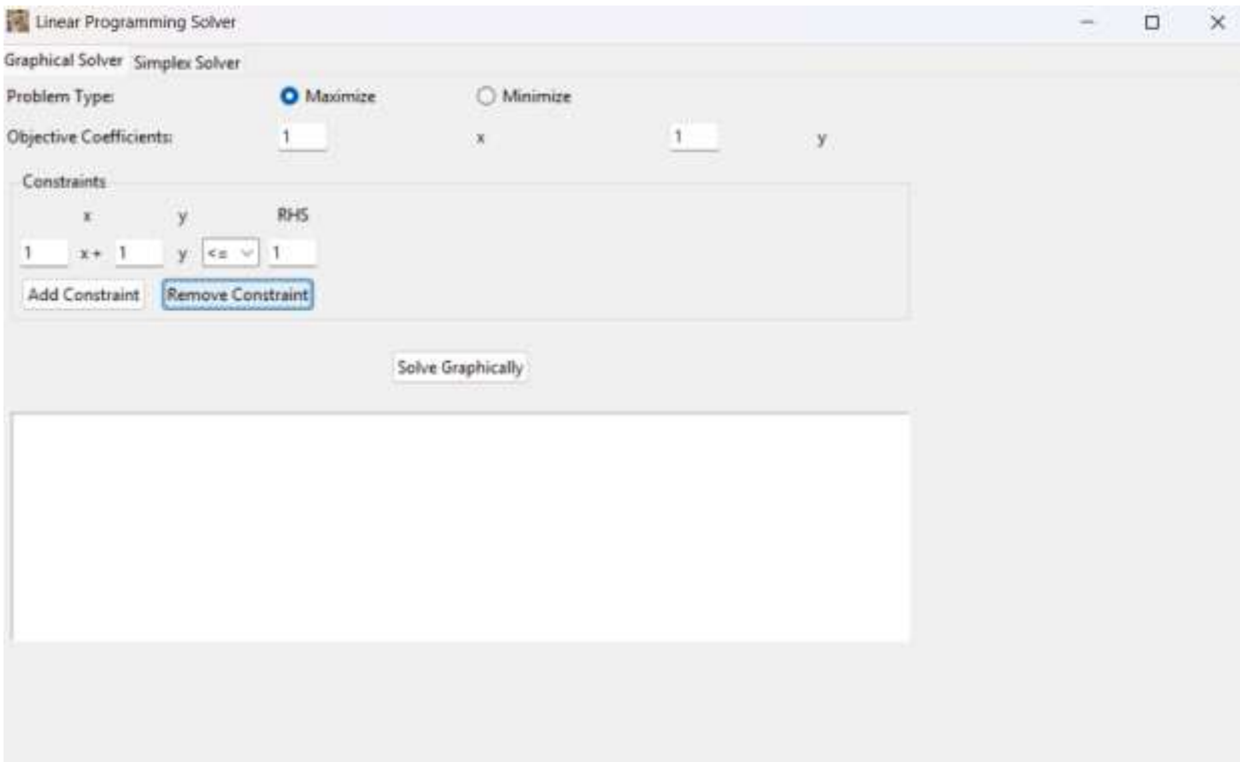
Backend Functions

```python
def add_constraint_row(self, frame, row):

    coeff_x = tk.DoubleVar(value=1)
    coeff_y = tk.DoubleVar(value=1)
    rhs = tk.DoubleVar(value=1)
    sense = tk.StringVar(value="<=")

    # Create all widgets
    x_entry = ttk.Entry(frame, textvariable=coeff_x, width=5)
    plus_label = ttk.Label(frame, text="x +")
    y_entry = ttk.Entry(frame, textvariable=coeff_y, width=5)
    y_label = ttk.Label(frame, text="y")
    sense_combo = ttk.Combobox(frame, textvariable=sense,
                               values=["<=", ">=", "="],
                               width=3, state="readonly")
    rhs_entry = ttk.Entry(frame, textvariable=rhs, width=5)

    # Grid all widgets
    x_entry.grid(row=row, column=1, padx=2, pady=2)
    plus_label.grid(row=row, column=2, padx=0, pady=2)
    y_entry.grid(row=row, column=3, padx=2, pady=2)
    y_label.grid(row=row, column=4, padx=0, pady=2)
    sense_combo.grid(row=row, column=5, padx=2, pady=2)
    rhs_entry.grid(row=row, column=6, padx=2, pady=2)
```

```python
    # Store all widgets and variables
    self.constraint_entries.append({
        'coeff_x': coeff_x,
        'coeff_y': coeff_y,
        'sense': sense,
        'rhs': rhs,
        'widgets': [x_entry, plus_label, y_entry, y_label, sense_combo, rhs_entry]
    })
```



```python
def remove_constraint_row(self):
    if len(self.constraint_entries) > 1:
        # Get the last constraint entry
        last_entry = self.constraint_entries[-1]

        # Destroy all widgets in the row
        for widget in last_entry['widgets']:
            widget.destroy()

        # Remove from our tracking list
        self.constraint_entries.pop()
```



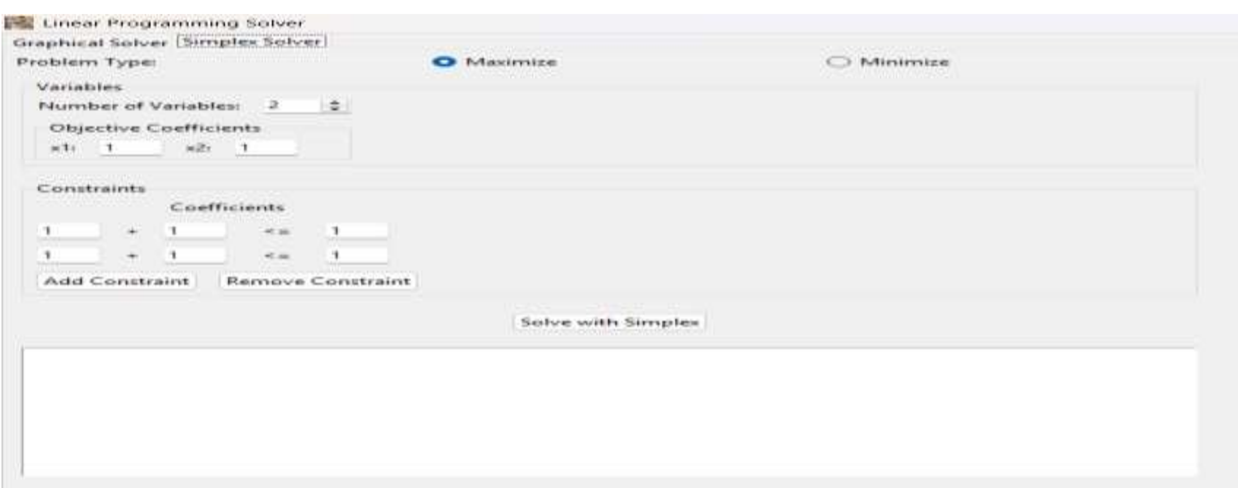
```python
def solve_graphical(self):
    try:
        # Get objective coefficients
        obj_coeffs = (self.obj_x.get(), self.obj_y.get())

        # Get constraints
        constraints = []
        for entry in self.constraint_entries:
            a = entry['coeff_x'].get()
            b = entry['coeff_y'].get()
            c = entry['rhs'].get()
            sense = entry['sense'].get()
            constraints.append((a, b, c, sense))

        # Solve
        result = graphical_solver(
            opt_type=self.graphical_opt_type.get(),
            obj_coeffs=obj_coeffs,
            constraints=constraints
        )

        # Display results
        self.graphical_results.delete(1.0, tk.END)
        self.graphical_results.insert(tk.END, "=== LP Solution ===\n")
        self.graphical_results.insert(tk.END, f"Status: {result['status']}\n")

        if result['status'] == 'optimal':
            self.graphical_results.insert(tk.END, f"Optimal Value: {result['opt_value']:.4f}\n")
            self.graphical_results.insert(tk.END, "Optimal Points:\n")
            for point in result['opt_points']:
                self.graphical_results.insert(tk.END, f"  ({point[0]:.4f}, {point[1]:.4f})\n")

    except Exception as e:
        messagebox.showerror("Error", f"An error occurred:\n{str(e)}")
```

Calls ***graphical_solver()*** and prints results.

Simplex Solver Tab Interface

```python
def create_simplex_tab(self):

    tab = ttk.Frame(self.notebook)
    self.notebook.add(tab, text="Simplex Solver")

    # Problem Type
    ttk.Label(tab, text="Problem Type:").grid(row=0, column=0, padx=5, pady=5, sticky=tk.W)
    self.simplex_opt_type = tk.StringVar(value="max")
    ttk.Radiobutton(tab, text="Maximize", variable=self.simplex_opt_type, value="max").grid(row=0, column=1,  sticky=tk.W)
    ttk.Radiobutton(tab, text="Minimize", variable=self.simplex_opt_type, value="min").grid(row=0, column=2, sticky=tk.W)

    # Variables Frame
    vars_frame = ttk.LabelFrame(tab, text="Variables")
    vars_frame.grid(row=1, column=0, columnspan=3, padx=10, pady=10, sticky=tk.W+tk.E)

    ttk.Label(vars_frame, text="Number of Variables:").grid(row=0, column=0, padx=5, pady=5)
    self.num_vars = tk.IntVar(value=2)
    ttk.Spinbox(vars_frame, from_=1, to=10, textvariable=self.num_vars, width=5, command=self.update_variable_count).grid(row=0, column=1, pac

    # Objective Function
    self.objective_entries = []
    self.objective_frame = ttk.LabelFrame(vars_frame, text="Objective Coefficients")
    self.objective_frame.grid(row=1, column=0, columnspan=2, padx=5, pady=5, sticky=tk.W+tk.E)
    self.update_objective_entries()
```

```python
    # Constraints
    constraints_frame = ttk.LabelFrame(tab, text="Constraints")
    constraints_frame.grid(row=2, column=0, columnspan=3, padx=10, pady=10, sticky=tk.W+tk.E)

    # Constraints header
    ttk.Label(constraints_frame, text="Coefficients").grid(row=0, column=0, columnspan=10, padx=5, pady=5)

    # Constraint entries (start with 2 empty constraints)
    self.simplex_constraint_entries = []
    for i in range(2):
        self.add_simplex_constraint_row(constraints_frame, i+1)

    # Add/Remove constraint buttons
    button_frame = ttk.Frame(constraints_frame)
    button_frame.grid(row=10, column=0, columnspan=10, pady=5)
    ttk.Button(button_frame, text="Add Constraint", command=lambda: self.add_simplex_constraint_row(constraints_frame, len(self.simplex_constr
    ttk.Button(button_frame, text="Remove Constraint", command=self.remove_simplex_constraint_row).pack(side=tk.LEFT, padx=5)

    # Solve Button
    ttk.Button(tab, text="Solve with Simplex", command=self.solve_simplex).grid(row=3, column=0, columnspan=3, pady=10)

    # Results
    self.simplex_results =Text(tab, height=10, width=80)
    self.simplex_results.grid(row=4, column=0, columnspan=3, padx=10, pady=10)
```



Backend Functions

```
def update_variable_count(self):

    self.update_objective_entries()
    self.update_constraints()
```

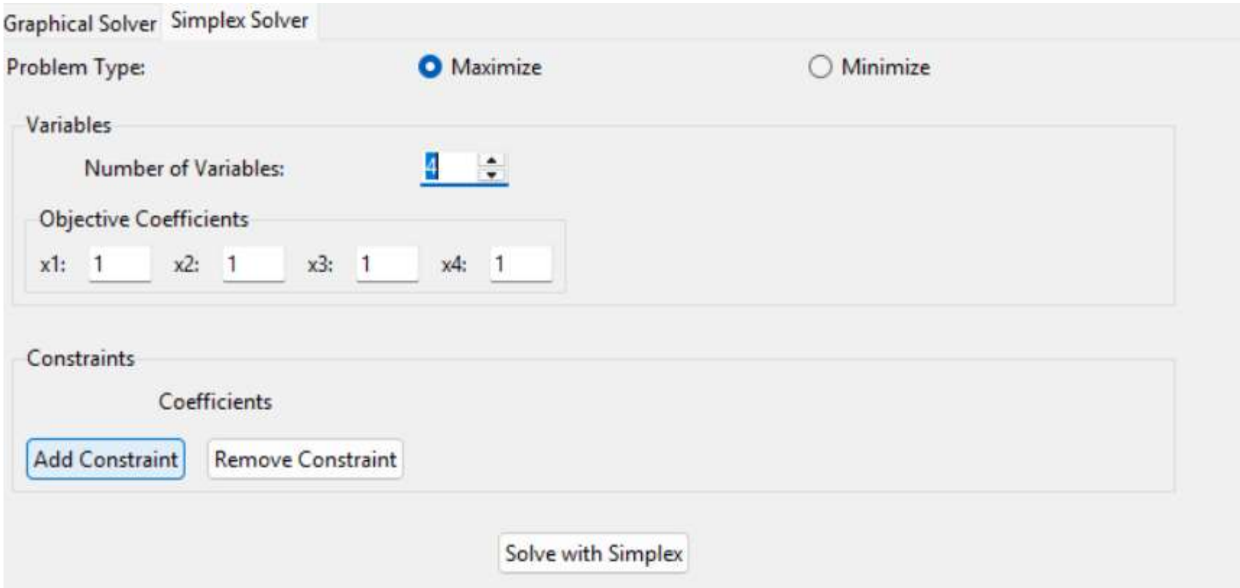
when the user changes the number of variables the entries for objective and constraints functions will change too.

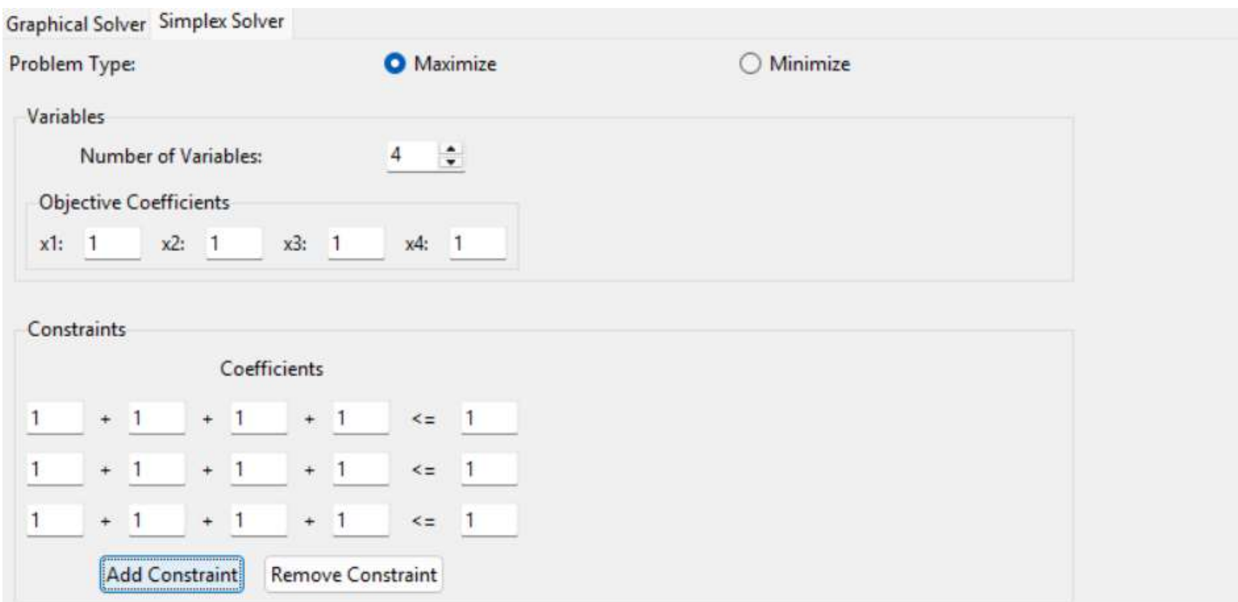
The function is called at *line 174:*

*ttk.Spinbox(vars_frame, from_=1, to=10, textvariable=self.num_vars, width=5, command=**self.update_variable_count**).grid(row=0, column=1, padx=5, pady=5)*

```
def update_objective_entries(self):

    # Clear existing entries
    for widget in self.objective_frame.winfo_children():
        widget.destroy()

    self.objective_entries = []
    num_vars = self.num_vars.get()

    for i in range(num_vars):
        ttk.Label(self.objective_frame, text=f"x{i+1}:").grid(row=0, column=i*2, padx=5, pady=5)
        var = tk.DoubleVar(value=1)
        ttk.Entry(self.objective_frame, textvariable=var, width=5).grid(row=0, column=i*2+1, padx=5, pady=5)
        self.objective_entries.append(var)
```

```
def update_constraints(self):

    # Store current constraints data
    current_constraints = []
    for entry in self.simplex_constraint_entries:
        current_constraints.append({
            'coeffs': [var.get() for var in entry['coeffs']],
            'sense': entry['sense'].get(),
            'rhs': entry['rhs'].get()
        })

    # Clear all existing constraint widgets
    for entry in self.simplex_constraint_entries:
        for widget in entry['widgets']:
            widget.destroy()
    self.simplex_constraint_entries = []

    # Recreate constraints with new variable count
    for i, constraint in enumerate(current_constraints):
        row = i + 1
        self.add_simplex_constraint_row(self.simplex_constraint_entries[0]['widgets'][0].master, row)

        # Set values from stored data (truncate or pad as needed)
        num_vars = self.num_vars.get()
        for j in range(min(num_vars, len(constraint['coeffs']))):
            self.simplex_constraint_entries[-1]['coeffs'][j].set(constraint['coeffs'][j])

        # Set sense and rhs
        self.simplex_constraint_entries[-1]['sense'].set(constraint['sense'])
        self.simplex_constraint_entries[-1]['rhs'].set(constraint['rhs'])
```

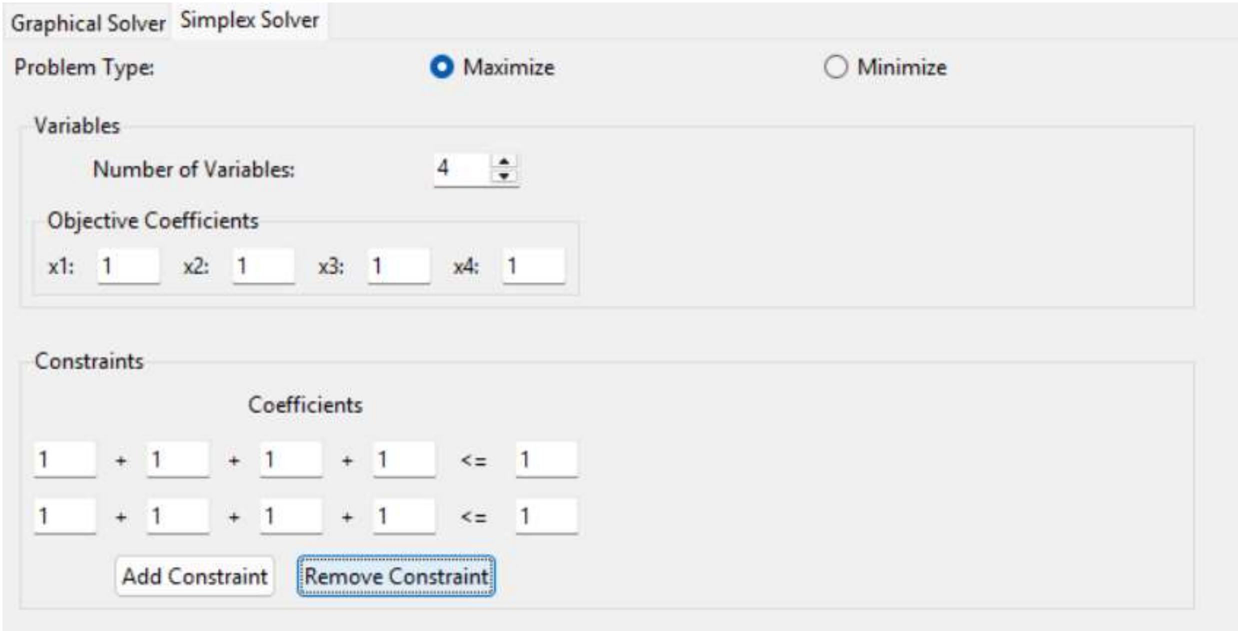

Graphical Solver
Simplex Solver
Problem Type:
Maximize
Minimize
Variables
Number of Variables: 4
Objective Coefficients
x1: 1 x2: 1 x3: 1 x4: 1
Constraints
Coefficients
Add Constraint
Remove Constraint
Solve with Simplex


Graphical Solver
Simplex Solver
Problem Type:
Maximize
Minimize
Variables
Number of Variables: 4
Objective Coefficients
x1: 1 x2: 1 x3: 1 x4: 1
Constraints
Coefficients
1 + 1 + 1 + 1 <= 1
1 + 1 + 1 + 1 <= 1
1 + 1 + 1 + 1 <= 1
Add Constraint
Remove Constraint

```python
def add_simplex_constraint_row(self, frame, row):

    num_vars = self.num_vars.get()
    coeff_vars = []
    widgets = []
    sense = tk.StringVar(value="<=")
    rhs = tk.DoubleVar(value=1)


    for i in range(num_vars):
        var = tk.DoubleVar(value=1)
        entry = ttk.Entry(frame, textvariable=var, width=5)
        entry.grid(row=row, column=i*2, padx=5, pady=5)
        coeff_vars.append(var)
        widgets.append(entry)

        if i < num_vars - 1:
            lbl = ttk.Label(frame, text="+")
            lbl.grid(row=row, column=i*2+1, padx=2)
            widgets.append(lbl)

    col = num_vars * 2
    sense_label = ttk.Label(frame, text="<=", width=3)
    sense_label.grid(row=row, column=col, padx=5, pady=5)
    rhs_entry = ttk.Entry(frame, textvariable=rhs, width=5)
    rhs_entry.grid(row=row, column=col+1, padx=5, pady=5)

    widgets.extend([sense_label, rhs_entry])

    self.simplex_constraint_entries.append({
        'coeffs': coeff_vars,
        'sense': sense,
        'rhs': rhs,
        'widgets': widgets   # save widgets for removal
    })
```

```python
def remove_simplex_constraint_row(self):

    if len(self.simplex_constraint_entries) > 1:
        # Get the last constraint entry
        last_entry = self.simplex_constraint_entries[-1]

        # Destroy all widgets in the row
        for widget in last_entry['widgets']:
            widget.destroy()

        # Remove from our tracking list
        self.simplex_constraint_entries.pop()
```

Removing a constraint also removes all labels not only the entries (widget.destroy)




```python
    def solve_simplex(self):
        try:
            # Get objective coefficients
            c = [var.get() for var in self.objective_entries]

            # Get constraints
            A = []
            b = []
            for entry in self.simplex_constraint_entries:
                A.append([var.get() for var in entry['coeffs']])
                b.append(entry['rhs'].get())

            # Solve
            solver = SimplexSolver()
            result = solver.solve(
                c=c,
                A=A,
                b=b,
                maximize=(self.simplex_opt_type.get() == "max")
            )

            # Display results
            self.simplex_results.delete(1.0, tk.END)
            self.simplex_results.insert(tk.END, "=== Simplex Solution ===\n")
            self.simplex_results.insert(tk.END, f"Status: {result.status}\n")

            if result.status == 'optimal':
                self.simplex_results.insert(tk.END, f"Optimal Value: {result.z:.4f}\n")
                self.simplex_results.insert(tk.END, "Solution:\n")
                for i, val in enumerate(result.x):
                    self.simplex_results.insert(tk.END, f"  x{i+1} = {val:.4f}\n")

            if result.message:
                self.simplex_results.insert(tk.END, f"\nMessage: {result.message}\n")

        except Exception as e:
```

Calls ***SimplexSolver().solve()*** and prints results.

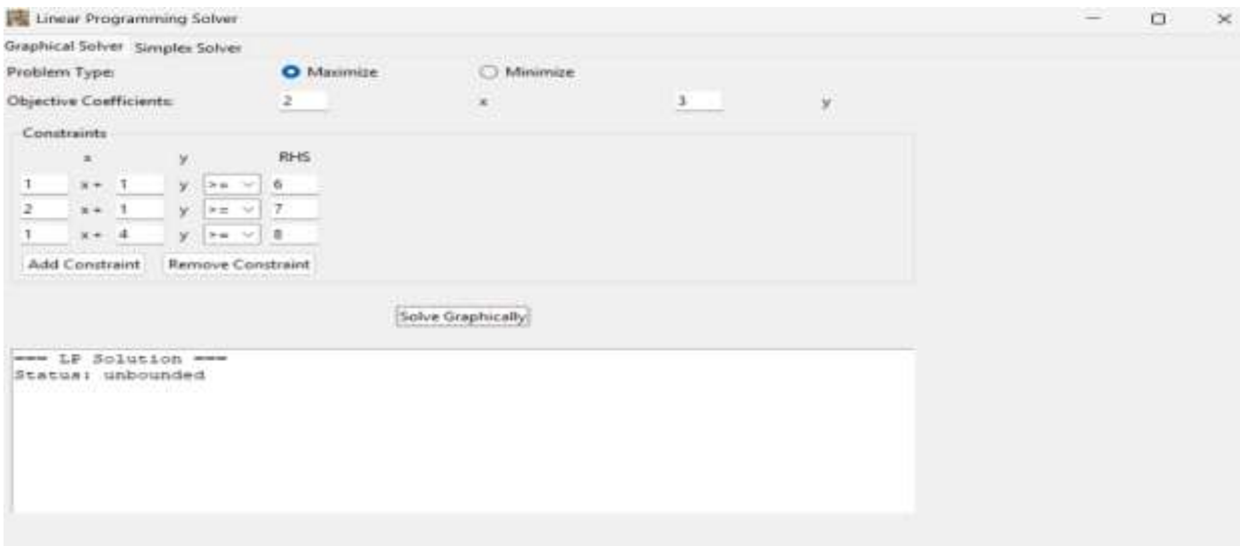
```python
if __name__ == "__main__":
    root = tk.Tk()
    app = LPSolverApp(root)
    root.mainloop()
```

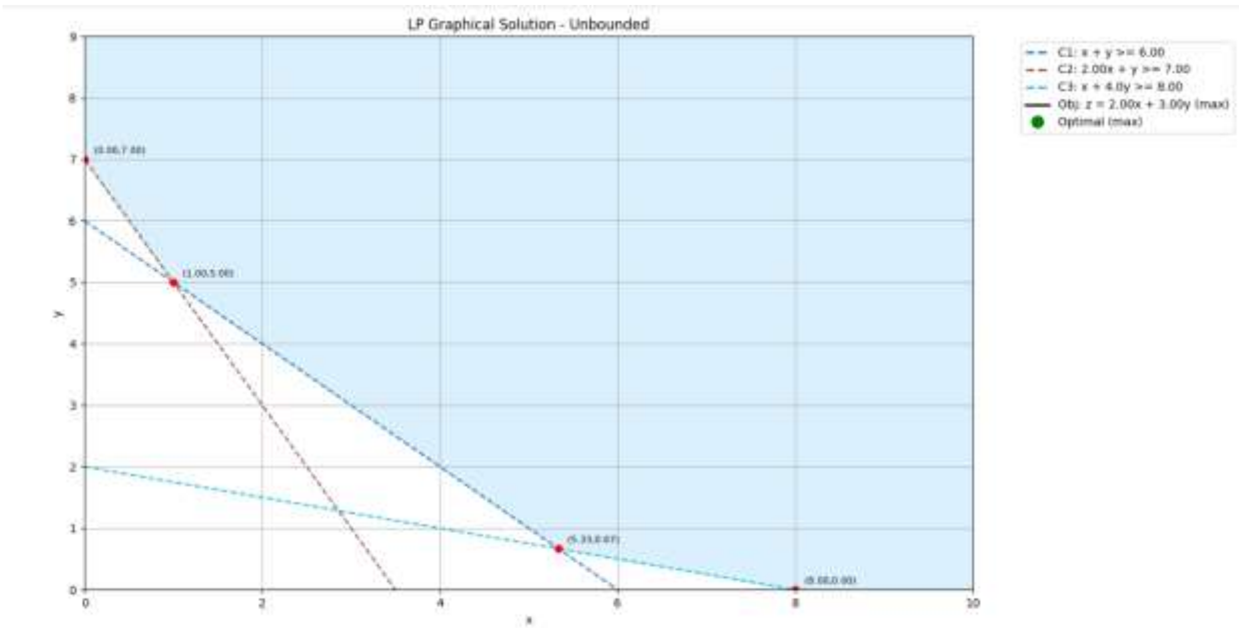
Standard Tkinter main loop.
Runs the application until user closes it.

5.2 Example Usage

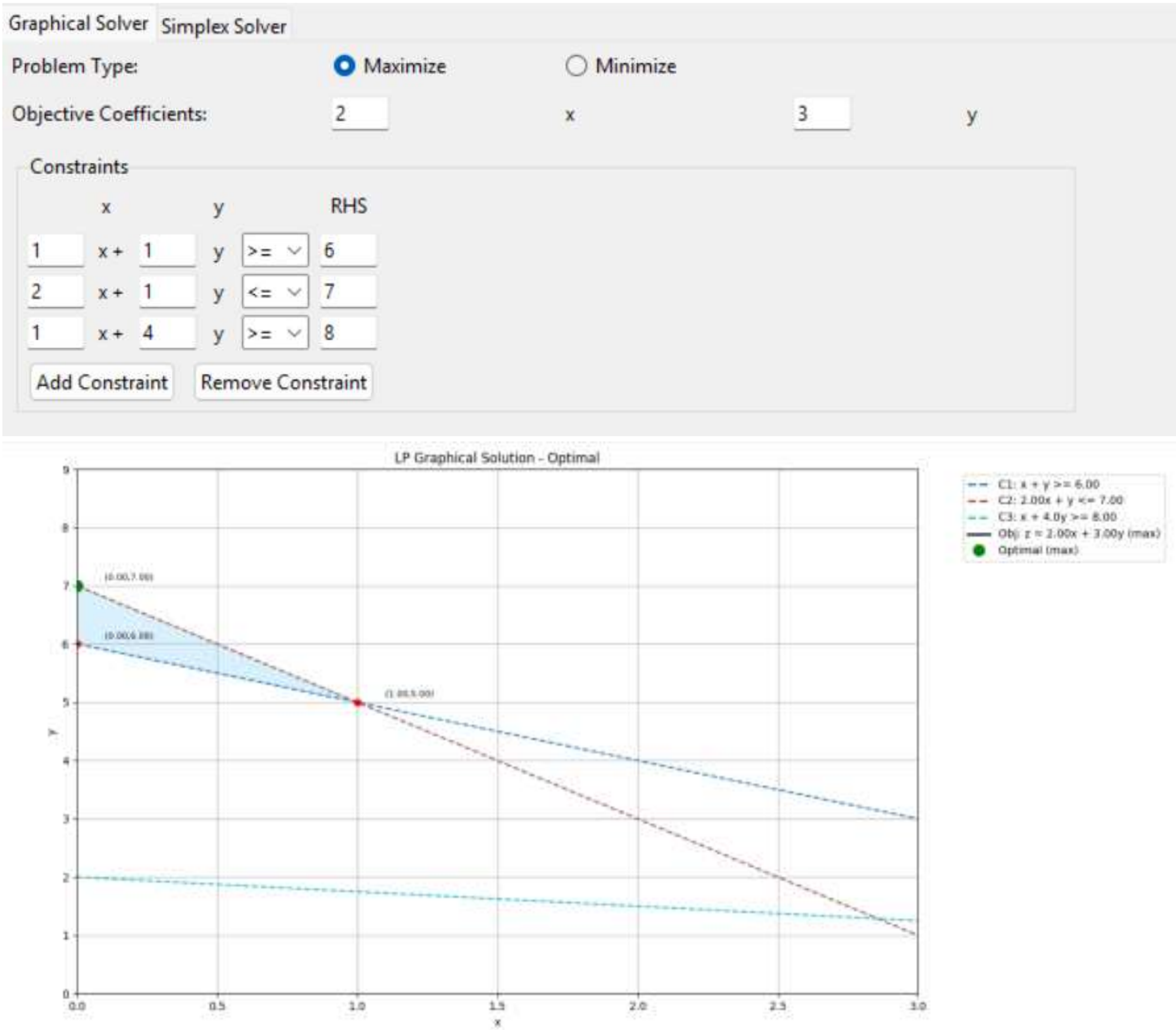
Graphical problem

$$\text{Min } Z = 2x + 3y$$
$$\textit{Subject to}:$$
$$x + y \geq 6,$$
$$2x + y \geq 7,$$
$$x + 4y \geq 8,$$
$$x \geq 0, y \geq 0$$





Ex 2:


Graphical Solver
Simplex Solver
Problem Type:
Maximize
Minimize
Objective Coefficients:
2
x
3
y
Constraints
x
y
RHS
1 x + 1 y >= 6
2 x + 1 y <= 7
1 x + 4 y >= 8
Add Constraint
Remove Constraint
LP Graphical Solution - Optimal
C1: x + y >= 6.00
C2: 2.00x + y <= 7.00
C3: x + 4.0y >= 8.00
Obj: z = 2.00x + 3.00y (max)
Optimal (max)
(0.00,7.00)
(0.00,6.00)
(1.00,5.00)
x
y

Simplex Problem

$$Max\ z = 2x1 + 4x2 + x3 + x4$$

$$Subject\ to:$$

$$x1 + 3x2 + x4 \leq 4,$$

$$2x1 + x2 \leq 3,$$

$$x2 + 4x3 + x4 \leq 3,$$

$$x1, x2, x3, x4 \geq 0$$