C++ PROGRAM THAT SOLVES SIMPLEX ALGORITHM

CSC 301 (STRUCTURED PROGRAMMING)

BY

GROUP 7

U22 CS 1062	HASSAN MARYAM ABDULAZIZ
U22 CS 1064	IBRAHIM AHMAD GARBA
U22 CS 1065	BUHARI IBRAHIM USMAN
U22 CS 1068	JEREMIAH OSARENOKAE IKIEBE
U22CS1069	ISREAL O. DAVID
U22CS1070	ISSA UMAR GBOLAHAN

COMPUTER SCIENCE DEPARTMENT

FACULTY OF COMPUTING
Air Force Institute of Technology, Kaduna.

FEBUARY 2025

TABLE OF CONTENTS

INTRODUCTION

One of the most important things a business can get right is **Decision Making.** Although several strategies exist that allow a company or an organization to get a good solution and one of these is **Linear Programming.**

Linear programming is a method of solving a complex problem, one so complex that is outside the ability of normal human-based decision-making.

Linear programming involves **constraint optimization** (i.e. a set of rules that describe a problem and following these rules will allow us to uncover a solution.

Linear programming has three (3) parts;

1. **The objective function:**
   This is the target we are trying to solve for, either to maximize the objective function or to minimize the objective function.
   Choosing maximization or minimization is always a context-specific decision, but Generally speaking, we choose to maximize good things and minimize bad things, such as;
   - Profit innovation
   - Revenue
   - Return on investment(ROT), etc.

   Example;

   - Maximum revenue
   - Minimum cost
   - Maximum Profit

2. **The Decision variables**
   These are items we are to find, they make up a part of the objective function and the objective function describes them. The descision variables the things such as;

- How many number of items to produce or how many units of products shipped from location A to B, or how many people working, or how many amount of money to be invested, etc.

3. **The Constraints**

   These are the rules that we must follow, and for any type of problem we have a set of rules that define the resources that are available to make production, *“because nobody has unlimited resources”.*

   The constraints have two (2) parts;

   - **The Left-hand side constraints:** These usually describe the current amount of scarce resources that you are consuming.
   - **The Right-hand side constraints:** This tells you the limit of the resources you have at hand.

OVERVIEW OF THE SIMPLEX TABLEAU (THE ALGORITHM STRUCTURE)

The Simplex tableau is a tabular representation of the linear programming model, which includes:

Objective Function

- **Definition**: A mathematical expression that defines the goal of the optimization problem. It represents the quantity to be maximized (e.g., profit) or minimized (e.g., cost).
- **Example**:

  For a business maximizing revenue, the objective function could be:

  $Z = 3x_1 + 5x_2$

  Here:

  - Z: The total revenue to be maximized.
  - x_1 and x_2: Decision variables representing quantities of products.

Decision Variables

- **Definition**: Variables that represent the choices available in the optimization problem. These are the unknowns Solver seeks to find to optimize the objective function.
- **Example**:
  - x_1: Number of products A to produce.
  - x_2: Number of products B to produce.

Constraints

- **Definition**: Conditions that the solution must satisfy. Constraints typically limit the resources available, such as materials, budget, or time.
- **Form**: Constraints are expressed as inequalities (≤, ≥) or equalities (=).
- **Example**:
  - $x_1 + 2x_2 \leq 6$ (limited raw materials).
  - $3x_1 + 2x_2 \leq 12$ (maximum production capacity).

Slack Variables

- **Definition**: Additional variables introduced to transform inequality constraints (≤) into equalities. These represent unused resources in the system.
- **Example**:
  For $x_1 + 2x_2 \leq 6$, adding a slack variable s_1 converts it to: $x_1 + 2x_2 + s_1 = 6$ Here, s_1 indicates the unused portion of the raw material.

Coefficients

- **Definition**: Numerical values representing the relationship between decision variables and the objective function or constraints.
- **Example**:
  In $3x_1 + 5x_2$, the coefficients are 3 (for x_1) and 5 (for x_2).

Maximization Problem:

Maximize the objective function:

$Z = 3x1 + 5x2$

Subject to the constraints:

$x1 + 2x2 \leq 6$

$3x1 + 2x2 \leq 12$

$x1,x2 \geq 0$

To apply the Simplex method, we need to convert these inequalities into equalities by adding slack variables. The inequalities become:

$x1 + 2x2 + s1 = 6$

$3x1 + 2x2 + s2 = 12$

Where s1 and s2 are slack variables representing unused resources.

Step 1: Setting Up the Simplex Tableau

First, we will set up the initial Simplex tableau, which will include the coefficients of the objective function, decision variables, slack variables, and the right-hand side (RHS) values from the constraints.

1.1 Organize the Tableau

The algorithm should have the following structure:

Basic Variable	x1	x2	s1	s2s	RHS
s1	1	2	1	0	6
s2	3	2	0	1	12
Z	-3	-5	0	0	0

Explanation of the Tableau Columns:

- **Basic Variable**: These are the basic variables (initially the slack variables, s1 and s2, and later, the decision variables will become basic variables).
- **Decision Variables**: These are the variables x1 and x2 whose values we are trying to optimize.
- **Slack Variables**: s1 and s2 are introduced to transform inequalities into equalities.
- **RHS**: The right-hand side values of the constraints, which represent the available resources (6 and 12).

1.2 Formula for the Objective Function

In the objective row (row for Z), the coefficients of x1 and x2 in the objective function are entered as negative values (-3 and -5, respectively), as we are trying to **maximize** the function. Slack variables will have 0 coefficients since they don't appear in the objective function.

C++ PROGRAM OF THE ALGORITHM

Link:

This program implements the **Simplex Algorithm** to solve a linear programming problem. It finds the **maximum value** of an objective function. It lets you input the number of variables and constraints, followed by the coefficients of the constraints (including the right-hand side) and the objective function. It then performs the Simplex Algorithm to find the optimal solution.

The Program Structure

1. **Input Handling**

   Code snippet

```
int numVariables, numConstraints;

cout << "Enter the number of variables: ";

cin >> numVariables;

cout << "Enter the number of constraints: ";

cin >> numConstraints;
```

- The user specifies the number of **decision variables** and **constraints**.
- Example: For x1 and x2, i.e numVariables = 2.

2. **Simplex Table Construction**

   Code snippet

```
vector<vector<double>> table(numConstraints + 1, vector<double>(numVariables + numConstraints + 1, 0));
```

This creates a **2D matrix** (Simplex Table) to store:

- Coefficients of decision variables x1,x2,…
- Slack variables s1,s2,…
- The right-hand side (RHS) of constraints.

3. **Fill the Table**

Code snippet

```
for (int i = 0; i < numConstraints; i++) {
    for (int j = 0; j <= numVariables; j++) {
        cin >> table[i][j];
    }
    table[i][numVariables + i + 1] = 1; // Slack variable
}
```

- Input the **constraints** coefficients and RHS values.
- Adds **slack variables** (1 in their own column, 0 elsewhere) to convert inequalities (≤) into equations.

Code snippet

```
for (int j = 1; j <= numVariables; j++) {

    cin >> table.back()[j];

    table.back()[j] *= -1; // Convert to maximization form

}
```

- Input the **objective function** coefficients and store them in the last row (negated to suit the Simplex method).

4. **Printing the Table**

Code snippet

```
void printTable(const vector<vector<double>>& table) {
    for (const auto& row : table) {
        for (double value : row) {
            cout << setw(10) << value << " ";
        }
        cout << endl;
```

```
    }
}
```

- Prints the simplex table neatly for debugging or understanding each step.

5. **Finding the Pivot Column**

   Code Snippet

```
int findPivotColumn(const vector<vector<double>>& table) {
    int pivotColumn = -1;
    double minValue = 0;
    for (int j = 1; j < table[0].size(); j++) {
        if (table.back()[j] < minValue) {
            minValue = table.back()[j];
            pivotColumn = j;
        }
    }
    return pivotColumn;
}
```

- The **pivot column** is the one with the most negative value in the last row (objective function). This indicates which variable can improve the solution.

6. **Finding the Pivot Row**

   Code snippet

```
int findPivotRow(const vector<vector<double>>& table, int pivotColumn) {
    int pivotRow = -1;
    double minRatio = 1e9; // Large value for comparison
    for (int i = 0; i < table.size() - 1; i++) {
        if (table[i][pivotColumn] > 0) {
            double ratio = table[i][0] / table[i][pivotColumn];
            if (ratio < minRatio) {
                minRatio = ratio;
                pivotRow = i;
            }
        }
    }
    return pivotRow;
}
```

- The **pivot row** is determined by the **minimum ratio test**:
  RHS/Pivot Column Coefficient
- This ensures feasibility and avoids unbounded solutions.

7. **Performing Pivoting**

   Code snippet

```
void performPivoting(vector<vector<double>>& table, int pivotRow, int pivotColumn) {
    double pivotValue = table[pivotRow][pivotColumn];
    for (double& value : table[pivotRow]) {
        value /= pivotValue; // Normalize pivot row
    }

    for (int i = 0; i < table.size(); i++) {
        if (i != pivotRow) {
            double factor = table[i][pivotColumn];
            for (int j = 0; j < table[i].size(); j++) {
                table[i][j] -= factor * table[pivotRow][j]; // Eliminate column values
            }
        }
    }
}
```

- This step **normalizes** the pivot row and adjusts the other rows to make the pivot column a unit column (Gaussian elimination).

8. **Simplex Iterations**

   Code snippet

```
void simplex(vector<vector<double>>& table) {
    while (true) {
        int pivotColumn = findPivotColumn(table);
        if (pivotColumn == -1) {
            break; // Optimal solution found
        }
        int pivotRow = findPivotRow(table, pivotColumn);
        if (pivotRow == -1) {
            cout << "Unbounded solution." << endl;
```

```
            return;
        }
        performPivoting(table, pivotRow, pivotColumn);
        cout << "Table after pivoting (Row: " << pivotRow << ", Column: " << pivotColumn << "):\\n";
        printTable(table);
    }

    // Print results
    cout << "Optimal solution found:\\n";
    for (int i = 1; i < table[0].size(); i++) {
        cout << "x" << i << " = ";
        bool found = false;
        for (int j = 0; j < table.size() - 1; j++) {
            if (table[j][i] == 1) {
                cout << table[j][0] << " ";
                found = true;
                break;
            }
        }
        if (!found) {
            cout << "0 ";
        }
    }
    cout << endl;
    cout << "Maximum value: " << table.back()[0] << endl;
}
```

- The algorithm iteratively:
    1. Finds the pivot column and row.
    2. Performs pivoting.
    3. Stops when there are no negative values in the last row (optimal solution).

9. Main Function

Code snippet

```
int main() {
    // Get problem dimensions and inputs
    // Build simplex table
    // Run the simplex algorithm
}
```

- Brings everything together:
    1. Input the problem's data.
    2. Construct the simplex table.
    3. Call the simplex function to solve.

Here’s a line-by-line explanation of the code that shows it logic in detail:

Header Files

```
#include <iostream>

#include <vector>

#include <iomanip>
```

- #include <iostream>: Provides input/output functions (cin, cout).
- #include <vector>: Allows us to use the **std::vector** container to handle dynamic arrays.
- #include <iomanip>: Used to format the output (e.g., controlling decimal precision).

Utility Functions

1. Print Table

```
void printTable(const vector<vector<double>>& table) {
    cout << fixed << setprecision(2);
    for (const auto& row : table) {
        for (double value : row) {
            cout << setw(10) << value << " ";
        }
        cout << endl;
    }
    cout << endl;
}
```

- **Purpose**: Prints the simplex table to the console in a clean format.
- **Details**:
    1. fixed: Ensures numbers are printed in fixed-point notation.
    2. setprecision(2): Displays numbers with two decimal places.
    3. for (const auto& row : table): Iterates over all rows in the table.
    4. for (double value : row): Iterates over all elements in a row and prints them.
    5. setw(10): Ensures column alignment by reserving 10 characters per value.

2. Find Pivot Column

```
int findPivotColumn(const vector<vector<double>>& table) {
    int pivotColumn = -1;
    double minValue = 0;
    for (int j = 1; j < table[0].size(); j++) {
        if (table.back()[j] < minValue) {
            minValue = table.back()[j];
            pivotColumn = j;
        }
```

```
    }
    return pivotColumn;
}
```

- **Purpose**: Finds the **pivot column** (the column corresponding to the most negative value in the last row of the table).
- **Details**:
  1. table.back(): Refers to the last row of the table (objective function row).
  2. pivotColumn = -1: Initialized to -1 (indicating no valid pivot column yet).
  3. for (int j = 1; j < table[0].size(); j++): Iterates over columns, skipping the RHS column (index 0).
  4. if (table.back()[j] < minValue): Checks for the smallest value in the last row. Smaller values indicate potential improvement in the objective function.
  5. Returns the index of the pivot column or -1 if no negative values are found.

3. Find Pivot Row

```
int findPivotRow(const vector<vector<double>>& table, int pivotColumn) {
    int pivotRow = -1;
    double minRatio = 1e9; // Large value for comparison
    for (int i = 0; i < table.size() - 1; i++) {
        if (table[i][pivotColumn] > 0) {
            double ratio = table[i][0] / table[i][pivotColumn];
            if (ratio < minRatio) {
                minRatio = ratio;
                pivotRow = i;
            }
        }
    }
    return pivotRow;
}
```

- **Purpose**: Finds the **pivot row** based on the **minimum ratio test**.
- **Details**:

I. pivotRow = -1: Initialized to -1 (no valid pivot row yet).
II. minRatio = 1e9: Large value to ensure any valid ratio is smaller.
III. if (table[i][pivotColumn] > 0): Only considers rows where the pivot column's value is positive.
IV. double ratio = table[i][0] / table[i][pivotColumn];: Computes the ratio of the RHS to the pivot column value.
V. if (ratio < minRatio): Finds the smallest ratio (ensuring feasibility).
VI. Returns the index of the pivot row or -1 if no valid row exists.

---

4. Perform Pivoting

```
void performPivoting(vector<vector<double>>& table, int pivotRow, int pivotColumn) {
    double pivotValue = table[pivotRow][pivotColumn];
    for (double& value : table[pivotRow]) {
        value /= pivotValue; // Normalize pivot row
    }

    for (int i = 0; i < table.size(); i++) {
        if (i != pivotRow) {
            double factor = table[i][pivotColumn];
            for (int j = 0; j < table[i].size(); j++) {
                table[i][j] -= factor * table[pivotRow][j];
            }
        }
    }
}
```

- **Purpose**: Performs pivoting to make the pivot column a unit column (1 in the pivot row and 0 elsewhere).
- **Details**:
  I. pivotValue = table[pivotRow][pivotColumn];: Stores the pivot value.
  II. value /= pivotValue;: Divides each element in the pivot row by the pivot value, normalizing it.
  III. for (int i = 0; i < table.size(); i++): Iterates through all rows.
  IV. if (i != pivotRow): Skips the pivot row itself.

V. table[i][j] -= factor * table[pivotRow][j];: Subtracts multiples of the pivot row from other rows to make pivot column values 0.

Simplex Algorithm

```
void simplex(vector<vector<double>>& table) {
    while (true) {
        int pivotColumn = findPivotColumn(table);
        if (pivotColumn == -1) {
            break; // Optimal solution found
        }

        int pivotRow = findPivotRow(table, pivotColumn);
        if (pivotRow == -1) {
            cout << "Unbounded solution." << endl;
            return;
        }

        performPivoting(table, pivotRow, pivotColumn);
        cout << "Table after pivoting (Row: " << pivotRow << ", Column: " << pivotColumn << "):\\n";
        printTable(table);
    }

    // Print results
    cout << "Optimal solution found:\\n";
    for (int i = 1; i < table[0].size(); i++) {
        cout << "x" << i << " = ";
```

```
        bool found = false;
        for (int j = 0; j < table.size() - 1; j++) {
            if (table[j][i] == 1) {
                cout << table[j][0] << " ";
                found = true;
                break;
            }
        }
        if (!found) {
            cout << "0 ";
        }
    }
    cout << endl;
    cout << "Maximum value: " << table.back()[0] << endl;
}
```

- **Purpose**: Implements the Simplex Algorithm iteratively.
- **Steps**:
  I. Calls findPivotColumn() to find the next column to improve.
  II. Calls findPivotRow() to find the pivot row based on the minimum ratio test.
  III. Calls performPivoting() to adjust the table.
  IV. Stops when there are no negative values in the last row, indicating optimality.
- **Outputs**:
  I. Values of x1,x2,…x_1, x_2, \dotsx1,x2,….
  II. Maximum value of the objective function.

Main Function

```
int main() {
    int numVariables, numConstraints;
    cout << "Enter the number of variables: ";
    cin >> numVariables;
```

```
    cout << "Enter the number of constraints: ";
    cin >> numConstraints;

    vector<vector<double>> table(numConstraints + 1, vector<double>(numVariables + numConstraints + 1,
0));

    cout << "Enter the coefficients of the constraints (including RHS):\\n";
    for (int i = 0; i < numConstraints; i++) {
        for (int j = 0; j <= numVariables; j++) {
            cin >> table[i][j];
        }
        table[i][numVariables + i + 1] = 1; // Slack variable
    }

    cout << "Enter the coefficients of the objective function (maximize):\\n";
    for (int j = 1; j <= numVariables; j++) {
        cin >> table.back()[j];
        table.back()[j] *= -1; // Convert to maximization form
    }

    cout << "Initial simplex table:\\n";
    printTable(table);

    simplex(table);

    return 0;
}
```

- Handles input for:
    I. Constraints (coefficients and RHS).

    II. Objective function.
- Builds the initial simplex table.
- Calls simplex() to solve the problem.

Benefits of Implementing the Simplex Algorithm in C++

1. **Performance & Efficiency**: C++ offers high execution speed and efficient memory management, making it ideal for handling large-scale linear programming problems.
2. **Low-Level Memory Control**: With direct memory access and optimization techniques, C++ can handle matrix operations efficiently, reducing computational overhead.
3. **Object-Oriented Design**: C++ allows structuring the simplex algorithm using classes and objects, improving code readability and maintainability.
4. **Standard Library Support**: The Standard Template Library (STL) provides useful data structures (like vectors and matrices) for efficient implementation.
5. **Scalability**: C++ can be optimized for parallel computing using multi-threading or GPU acceleration, enhancing performance for complex problems.
6. **Portability**: C++ code can be compiled and executed across multiple platforms with minimal changes, making it flexible for various applications.
7. **Integration with Other Libraries**: C++ can easily integrate with numerical libraries (e.g., Eigen, Gurobi) to enhance computational capabilities.

SUMMARY

C++ can be used to efficiently implement the **Simplex Algorithm** for solving **linear programming problems (LPPs)**, similar to how Excel Solver handles it. The process starts by formulating the problem in **standard form**, converting inequalities into equalities using **slack or surplus variables**.

A **Simplex tableau** is then constructed as a matrix or 2D array in C++, storing the coefficients of decision variables, slack variables, and the right-hand side (RHS) values. The algorithm iterates through **pivot selection**, identifying the entering and leaving variables, and performing **row operations** to update the tableau. This process continues until an **optimal solution** is reached, maximizing or minimizing the objective function while satisfying constraints.

Using **C++'s efficiency and control over memory**, this implementation allows for fast computations, making it suitable for solving large-scale optimization problems. Libraries like **Eigen** or **Boost** can further optimize matrix operations, improving performance.

CONCLUSION

In conclusion, implementing the **Simplex Algorithm** in **C++** provides a powerful and efficient way to solve **Linear Programming (LP) problems**. By constructing a **Simplex tableau** in C++, developers can systematically perform the necessary calculations, track iterations, and determine the **optimal solution** for maximization or minimization problems. Through a structured approach—converting inequalities to equalities, selecting pivot columns and rows, and updating the tableau—C++ offers flexibility and high-performance computation for solving complex optimization tasks.

While C++ requires manual implementation of the algorithm, it provides greater **control, efficiency, and scalability**, especially for **large-scale problems** where performance is critical. Understanding the mechanics of the Simplex method is essential for interpreting results and optimizing the implementation. Ultimately, using C++ for linear optimization enables precise, customizable, and high-speed computation, making it a valuable tool for solving real-world LP problems.