UNIVERSIDAD AUTÓNOMA GABRIEL RENÉ MORENO

FACULTAD DE INGENIERÍA EN CIENCIAS DE LA COMPUTACIÓN Y

TELECOMUNICACIONES



AUTOMATIZACIÓN DE LOS MÉTODOS DE LA GRAN M Y DOS FASES

**Materia:** Investigación Operativa I

**Sigla:** MAT329-SB

**Docente:** Ing. Luz Diana Torrez

Integrantes:

* Rodriguez Araúz Evert
* Molina Cortez Maikol Anthony

Semestre I/2025

Santa Cruz de la Sierra - Bolivia

CONTENIDO

[1. Análisis del Problema 3](#_Toc201501901)

[1.1 Definición de las Variables de Entrada 4](#_Toc201501902)

[1.2 Definición de las Variables de Salida 5](#_Toc201501903)

[1.3 Restricciones del Programa 5](#_Toc201501904)

[2. Fundamentación Teórica 6](#_Toc201501905)

[3. Metodología y Arquitectura del Software 7](#_Toc201501906)

[3.1 Arquitectura de Software 8](#_Toc201501907)

[3.2 Flujo de Datos del Algoritmo 8](#_Toc201501908)

[4. Aplicación y Resultados 9](#_Toc201501909)

[4.1 Pruebas y Validación 10](#_Toc201501910)

[5. Conclusiones y Recomendaciones 18](#_Toc201501911)

[6. Bibliografía 18](#_Toc201501912)

[ANEXOS 19](#_Toc201501913)

[Anexo A: Código Fuente Completo 19](#_Toc201501914)

[Anexo B: Manual del Usuario 48](#_Toc201501915)

[Anexo C: Presentación y Repositorio 49](#_Toc201501916)

**AUTOMATIZACIÓN DE LOS MÉTODOS DE LA GRAN M Y DOS FASES**

# 1. Análisis del Problema

**Contexto:**

La Programación Lineal (PL) es una disciplina matemática fundamental para la optimización de recursos en campos como la ingeniería, la economía y la logística. El método Simplex es el pilar para resolver problemas de PL. Sin embargo, su forma canónica solo se aplica directamente a problemas con restricciones del tipo "menor o igual que" (≤). Para abordar problemas más complejos con restricciones de igualdad (=) o "mayor o igual que" (≥), se requieren algoritmos avanzados como el Método de la Gran M y el Método de las Dos Fases.

La resolución manual de estos métodos es un proceso iterativo, extenso y altamente propenso a errores de cálculo, especialmente a medida que aumenta el número de variables y restricciones. Esta complejidad representa una barrera significativa para su aplicación práctica y eficiente.

**Objetivos:**

**Objetivo General:**  
Desarrollar una aplicación de escritorio robusta y fácil de usar que automatice por completo la resolución de problemas de programación lineal mediante los métodos de la Gran M y de las Dos Fases.

Objetivos Específicos:

1. Diseñar una Interfaz Gráfica de Usuario (GUI) intuitiva que permita al usuario definir la función objetivo, el número de variables, las restricciones y el método de solución.
2. Implementar el núcleo lógico de los algoritmos Simplex, Gran M y Dos Fases, asegurando la correcta manipulación de variables de holgura, exceso y artificiales.
3. Visualizar el proceso de solución paso a paso, mostrando cada tableau (tabla) de la iteración Simplex, junto con las variables de entrada y salida, para facilitar la comprensión y el aprendizaje.
4. Proporcionar un diagnóstico claro de la solución final, incluyendo el valor óptimo de la función objetivo, los valores de las variables de decisión y la detección de casos especiales como soluciones no acotadas o problemas infactibles.

**Alcance y Delimitaciones**

En Alcance:

1. La aplicación resuelve problemas de programación lineal continua.
2. Soporta problemas de maximización y minimización.
3. Implementa los métodos de la Gran M y de las Dos Fases.
4. Permite un número de variables de decisión entre 2 y 100.
5. Permite un número de restricciones entre 1 y 500.
6. Muestra todas las iteraciones del tableau Simplex.

Fuera de Alcance (Delimitaciones):

1. El software no resuelve problemas de programación entera, binaria o no lineal.
2. No incluye análisis de sensibilidad post-óptimo.
3. No implementa otros algoritmos como el Simplex Revisado o métodos de punto interior.

Requerimientos

Funcionales:

1. El usuario debe poder especificar el número de variables y restricciones.
2. El sistema debe generar dinámicamente campos de entrada para los coeficientes de la función objetivo y las restricciones.
3. El usuario debe poder seleccionar el tipo de optimización (Maximizar/Minimizar).
4. El usuario debe poder seleccionar el método de resolución (Gran M/Dos Fases).
5. El usuario debe poder definir el tipo de desigualdad (≤, ≥, =) para cada restricción.
6. El sistema debe calcular la solución óptima y presentarla.
7. El sistema debe mostrar una ventana con el detalle de cada paso del tableau.

No Funcionales:

1. Usabilidad: La interfaz debe ser clara y autoexplicativa, guiando al usuario en la entrada de datos.
2. Corrección: Los cálculos deben ser precisos, con un redondeo controlado a 6 decimales para evitar errores de punto flotante (definido por la variable epsilon = 1e-10 y decimales = 6 en SimplexBase.java).
3. Robustez: La aplicación debe gestionar entradas inválidas (ej. texto en campos numéricos) mostrando alertas de error (NumberFormatException manejado en HelloController.java).

## Definición de las Variables de Entrada

Las entradas del sistema son los datos que el usuario proporciona a través de la interfaz gráfica para definir el problema de programación lineal.

1. Configuración del Modelo:
   1. Número de variables: (Numérico - Entero) Define la cantidad de variables de decisión (ej. x₁, x₂).
   2. Número de restricciones: (Numérico - Entero) Define la cantidad de ecuaciones de restricción.
2. Función Objetivo:
   1. Coeficientes de la función objetivo: (Matricial - Vector numérico de tipo double) Son los valores que acompañan a cada variable en la función Z.
   2. Tipo de objetivo: (Texto) Cadena de texto que indica si el problema es de "Maximizar" o "Minimizar".
   3. Método de solución: (Texto) Cadena de texto que permite elegir entre "Gran M" o "Dos Fases".
3. Restricciones del Modelo:
   1. Coeficientes de las restricciones: (Matricial - Matriz numérica de tipo double) Corresponden a los coeficientes de las variables en cada una de las restricciones.
   2. Operadores de desigualdad: (Texto - Vector de caracteres) Indican el tipo de restricción para cada fila (≤, ≥, =).
   3. Términos independientes (RHS): (Matricial - Vector numérico de tipo double) Son los valores al lado derecho de cada restricción.

## Definición de las Variables de Salida

Las salidas son los resultados generados por el programa, presentados al usuario en una ventana de solución.

1. Proceso de Solución (Pasos):
   1. Tableau por iteración: (Matricial - Matriz numérica de tipo double) Se muestra una tabla completa (GridPane) para cada paso del algoritmo Simplex, con todos sus valores numéricos redondeados.
   2. Descripción del paso: (Texto) Un texto que explica la acción realizada en cada iteración (ej. "Normalización de la fila pivote", "Variable de Entrada: x1, Variable de Salida: s2").
2. Solución Final:
   1. Valores de las variables: (Numérico y Texto) Se presenta el valor numérico final para cada tipo de variable (Decisión, Holgura, Exceso, Artificial), junto con una etiqueta de texto que identifica a cada una.
   2. Valor de la Función Objetivo (Z): (Numérico) El resultado óptimo final del problema.
   3. Diagnóstico del problema: (Texto) Mensajes de texto que informan si la solución es "Óptima", "Infactible" o "No Acotada".

## Restricciones del Programa

El software opera bajo las siguientes limitaciones técnicas y de alcance:

1. Linealidad: El programa está diseñado exclusivamente para resolver problemas de programación lineal. No puede procesar problemas con funciones objetivo o restricciones no lineales.
2. Tipo de Variables: No admite variables de tipo entero o binario; todas las variables de decisión son tratadas como continuas.
3. Límites de Rendimiento: Para asegurar un rendimiento óptimo y evitar tiempos de cálculo excesivos en la plataforma JavaFX, hemos establecido los siguientes límites:
   1. Máximo 100 variables de decisión.
   2. Máximo 500 restricciones.
4. Dependencias: El programa requiere un entorno de ejecución Java (JRE/JDK) con la biblioteca JavaFX incluida para poder ejecutarse.

# 2. Fundamentación Teórica

Para automatizar la resolución de problemas de PL, el proyecto se fundamenta en los siguientes conceptos clave de la Investigación de Operaciones:

**Método Simplex Primal:** Es un algoritmo iterativo que navega por los vértices de una región factible (el conjunto de todas las soluciones posibles). Comienza en un vértice inicial y en cada paso se mueve a un vértice adyacente que mejora el valor de la función objetivo. El proceso termina cuando no es posible encontrar un vértice adyacente mejor, lo que indica que se ha alcanzado la solución óptima.

Variables de Holgura, Exceso y Artificiales:

1. Variable de Holgura (S): Se introduce en restricciones ≤ para convertir la inecuación en una ecuación. Representa un recurso no utilizado.
2. Variable de Exceso (E): Se introduce en restricciones ≥ para representar el excedente sobre el mínimo requerido. Se resta para convertir la inecuación en una ecuación.
3. Variable Artificial (A): No tiene un significado físico. Se introduce en restricciones ≥ y = para crear una solución básica factible inicial (una matriz identidad en el tableau) y dar un punto de partida al algoritmo Simplex. Estas variables deben ser eliminadas de la solución final.

**Método de la Gran M:** Este método integra la penalización de las variables artificiales directamente en la función objetivo.

1. Se asigna un coeficiente de penalización M extremadamente grande (en el código, M\_VALOR = 1E6) a cada variable artificial.
2. Si el objetivo es maximizar, se resta M \* A de la función objetivo.
3. Si el objetivo es minimizar, se suma M \* A.  
   El gran valor de M asegura que cualquier solución que contenga una variable artificial con valor positivo sea subóptima. Si al final del proceso una variable artificial permanece en la base con un valor positivo, el problema se considera infactible. La clase GranM.java implementa esta lógica, ajustando la fila de la función objetivo en su constructor y el método actualizarZ().

**Método de las Dos Fases:** Este método aborda el problema en dos etapas, evitando el uso del valor M.

1. Fase 1: El objetivo original se ignora temporalmente. Se crea una nueva función objetivo auxiliar que busca minimizar la suma de todas las variables artificiales. Se resuelve este nuevo problema con el método Simplex.
2. Si el valor óptimo de esta fase es mayor que cero, significa que al menos una variable artificial no pudo ser eliminada de la base, por lo que el problema original no tiene solución factible.
3. Si el valor óptimo es cero, se ha encontrado una solución básica factible para el problema original.
4. Fase 2: Se elimina la función objetivo auxiliar y las columnas de las variables artificiales. Se restaura la función objetivo original y se continúa con el algoritmo Simplex a partir de la base factible encontrada en la Fase 1 hasta alcanzar la optimalidad. La clase DosFases.java encapsula esta lógica en sus métodos resolverFaseUno() y resolverFaseDos().

**Referencia Clave:**  
Hillier, F. S., & Lieberman, G. J. (2010). *Introducción a la investigación de operaciones* (9a ed.). McGraw-Hill.

# 3. Metodología y Arquitectura del Software

Para el desarrollo de este proyecto adoptamos una metodología de desarrollo de software incremental, dividida en fases secuenciales que permiten una construcción estructurada y organizada.

Nuestro flujo de trabajo se desglosa de la siguiente manera:

1. **Fase de Análisis y Requerimientos:** Se definieron el alcance del problema, los objetivos, y las variables de entrada y salida. En esta etapa se identificó un requerimiento adicional clave: la necesidad de una visualización paso a paso del proceso Simplex para que la herramienta tuviera un valor educativo, además de funcional.
2. **Fase de Diseño:** Se diseñó la arquitectura del software bajo un patrón Modelo-Vista-Controlador (MVC). Se crearon diagramas de clases para estructurar las relaciones entre la lógica del Simplex, las estructuras de datos y la interfaz gráfica. Se diseñó la interfaz de usuario (GUI) en JavaFX, priorizando la usabilidad.
3. **Fase de Implementación:** Se llevó a cabo la codificación en Java, desarrollando primero las clases del Modelo (SimplexBase, VariableInfo, GranM, DosFases), seguido por el Controlador (HelloController) para manejar la lógica de eventos, y finalmente las Vistas (hello-view.fxml, WindowSolucion).
4. **Fase de Pruebas y Depuración:** Se realizaron pruebas unitarias para validar la correcta implementación de los algoritmos. Se ejecutaron pruebas de sistema para asegurar el flujo correcto de datos desde la interfaz hasta la presentación de resultados y se depuraron errores de lógica, ejecución y de interfaz.

## 3.1 Arquitectura de Software

Desarrollamos este proyecto en Java utilizando la plataforma JavaFX para la interfaz gráfica, siguiendo un patrón arquitectónico similar a Modelo-Vista-Controlador (MVC):

1. **Modelo (Model):** Contiene la lógica de negocio y las estructuras de datos.
   1. SimplexBase.java: Clase abstracta que define la estructura y operaciones comunes del tableau Simplex (pivoteo, normalización, búsqueda de pivotes, almacenamiento de pasos).
   2. GranM.java y DosFases.java: Clases concretas que heredan de SimplexBase e implementan la lógica específica de cada método.
   3. VariableInfo.java: Clase que modela cada variable del problema (original, holgura, exceso, artificial), su tipo, su posición en el tableau y su estado (si está en la base o no).
   4. GuardarPasos.java y EstructuraParaGuardarPasos.java: Clases dedicadas a almacenar el estado del tableau en cada iteración para su posterior visualización.
2. **Vista (View):** Responsable de la presentación.
   1. hello-view.fxml: Archivo FXML que define la estructura declarativa de la interfaz principal.
   2. WindowSolucion.java: Clase JavaFX que genera dinámicamente la ventana de resultados, construyendo las tablas (GridPane) para mostrar los tableaus de cada paso y la solución final.
3. **Controlador (Controller):** Actúa como intermediario entre el Modelo y la Vista.
   1. HelloController.java: Maneja todos los eventos de la interfaz principal (ej. clics en botones, cambios en spinners). Recolecta los datos del usuario, instancia las clases del Modelo (GranM o DosFases), invoca el método resolver(), y lanza la WindowSolucion para mostrar los resultados.
   2. HelloApplication.java: Es la clase de entrada de la aplicación JavaFX que carga la vista inicial.

## 3.2 Flujo de Datos del Algoritmo

El proceso completo, desde la entrada del usuario hasta la visualización de la solución, sigue estos pasos:

1. **Configuración Inicial (Vista y Controlador):** El usuario interactúa con los Spinner y ComboBox en la GUI. Al presionar "Generar Modelo", el método generarModelo() de HelloController crea dinámicamente los TextFields necesarios dentro de GridPanes.
2. **Captura de Datos (Controlador):** Al hacer clic en "Resolver Problema", el método resolverProblema():
   1. Lee los valores de todos los TextFields y ComboBoxes.
   2. Valida que los datos sean numéricos (dentro de un bloque try-catch).
   3. Almacena los coeficientes, términos independientes y tipos de restricción en arreglos (double[], double[][], char[]).
3. **Instanciación del Modelo (Controlador -> Modelo):** Se crea una instancia de GranM o DosFases pasándole los datos capturados.
4. Construcción del Tableau Inicial (Modelo):
   1. Dentro del constructor de GranM o DosFases, se llama a inicializarTableauBase().
   2. Este método, a su vez, utiliza contarVariablesPorTipo() para crear una lista de objetos VariableInfo, determinando cuántas variables de holgura, exceso y artificiales se necesitan.
   3. Se construye la matriz M (double[][]) con las dimensiones correctas y se puebla con los coeficientes del problema.
   4. Se realizan los ajustes iniciales a la función objetivo (sumar penalizaciones M o preparar la Fase 1).
   5. Cada paso clave (ej. "Tableau inicializado") se guarda usando guardarPaso().
5. Proceso Iterativo Simplex (Modelo):
   1. El método resolver() inicia un bucle while que se ejecuta mientras la solución no sea óptima (!esOptimo()).
   2. En cada ciclo, se llama a iteraciónSimplex(), que orquesta:  
      a. obtenerColumnaPivote(): Encuentra la columna con el valor más negativo en la fila Z.  
      b. obtenerFilaPivote(): Calcula el criterio del cociente mínimo para hallar la fila pivote.  
      c. realizarPivoteo(): Normaliza la fila pivote y realiza la eliminación gaussiana para actualizar todo el tableau.  
      d. actualizarBaseDeVariables(): Actualiza el estado de los objetos VariableInfo (cuál entra y cuál sale de la base).
   3. Cada operación de pivoteo se guarda como un nuevo paso.
6. Finalización y Presentación (Modelo -> Controlador -> Vista):
   1. Cuando el bucle termina, se llama a finalizarSolucion() para extraer los valores finales.
   2. El control regresa al HelloController, que crea una instancia de WindowSolucion, pasándole el objeto simplex resuelto.
   3. WindowSolucion utiliza los datos almacenados en historialDePasos para renderizar cada tableau y muestra la solución final formateada.

# 4. Aplicación y Resultados

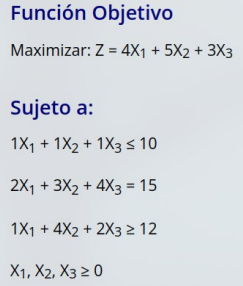
El uso de esta aplicación ofrece ventajas significativas sobre la resolución manual:

1. **Eficiencia:** Reduce drásticamente el tiempo necesario para resolver problemas complejos, pasando de horas o minutos a meros segundos.
2. **Precisión:** Elimina la posibilidad de errores humanos de cálculo, garantizando resultados fiables y correctos.
3. **Valor Educativo:** Al visualizar cada iteración del tableau Simplex, la herramienta facilita la comprensión de los algoritmos y sirve como un potente apoyo para el aprendizaje de la investigación de operaciones.
4. **Análisis Completo:** Detecta automáticamente casos especiales como soluciones infactibles o no acotadas, proporcionando un diagnóstico completo del problema.

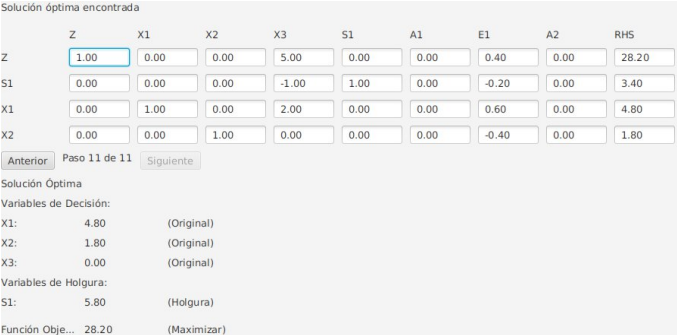
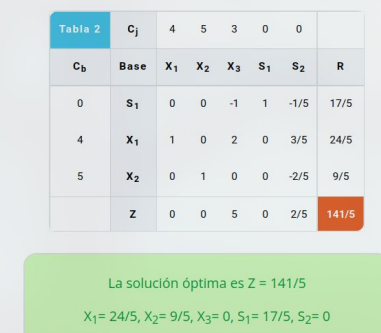
## 4.1 Pruebas y Validación

La aplicación fue sometida a diversas pruebas con problemas de Investigación Operativa que requieren programación lineal.

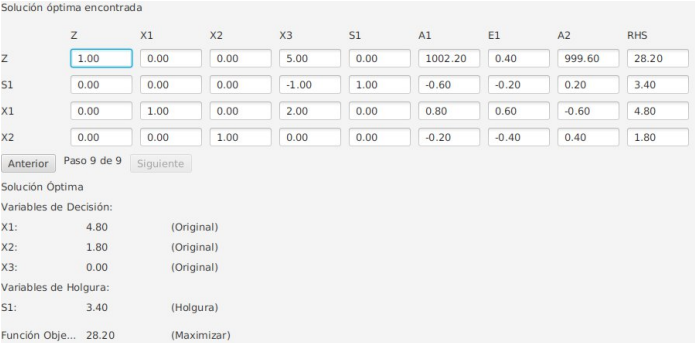
Problema 1:



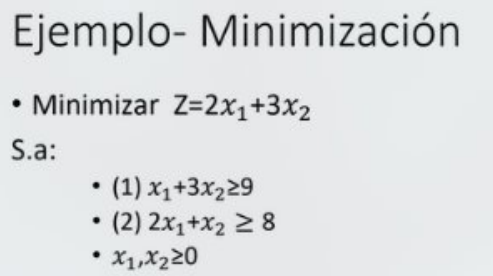
Solución con Dos Fases:



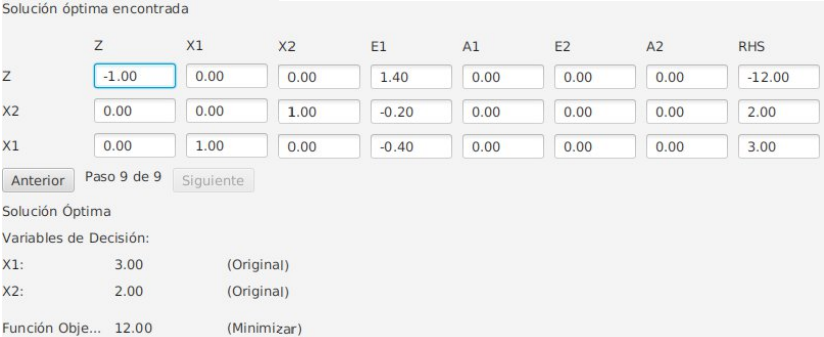
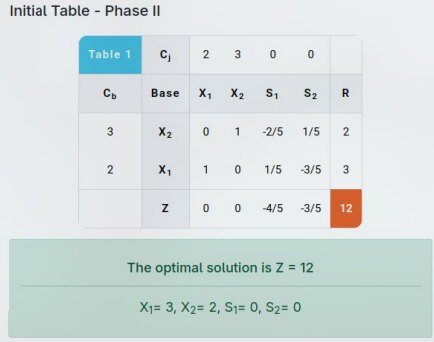
Solución con Gran M:



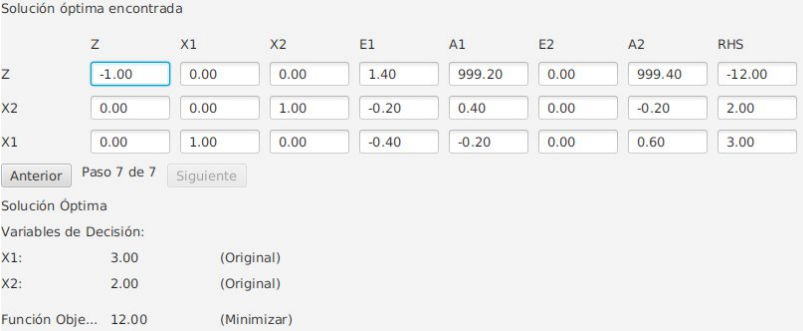
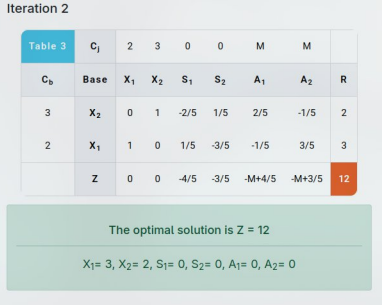
Ejercicio 2:



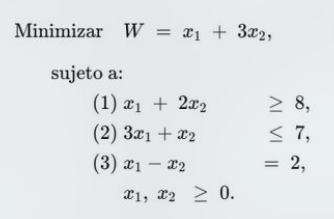
Solución con Dos Fases:



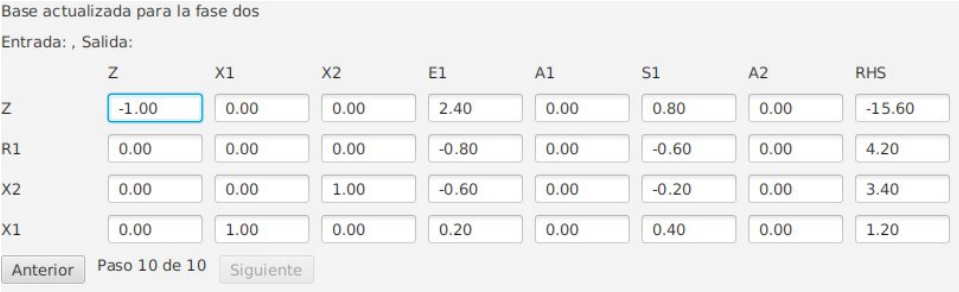
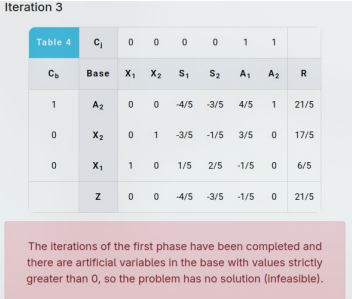
Solución con Gran M:



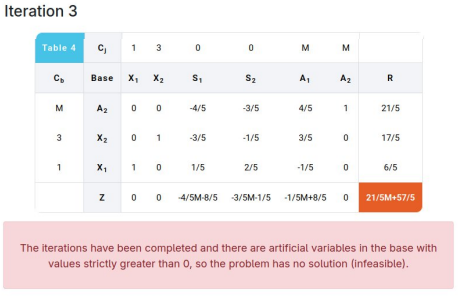
Ejercicio 3:



Solución con Dos Fases:



Solución con Gran M:



Análisis de Resultados

Los resultados demuestran que la aplicación es capaz de:

1. Resolver correctamente problemas estándar de maximización y minimización.
2. Manejar las tres clases de restricciones (≤, ≥, =) de forma precisa.
3. Identificar y notificar al usuario cuando un problema es infactible (si una variable artificial permanece en la base con valor positivo al final).
4. Identificar y notificar cuando un problema tiene una solución no acotada (cuando se identifica una columna pivote pero no se puede determinar una fila pivote).

Depuración y Manejo de Errores

1. **Errores de Compilación:** Resueltos durante el desarrollo, principalmente relacionados con la configuración de JavaFX y las dependencias de Maven/Gradle.
2. **Errores de Ejecución:** El principal riesgo era la NumberFormatException si el usuario ingresaba texto. Esto se mitigó con un bloque try-catch en el método resolverProblema que muestra una Alert al usuario.
3. **Errores Lógicos:** La depuración más compleja se centró en la lógica de pivoteo y la actualización del estado de las variables. El uso de la clase VariableInfo fue crucial para rastrear qué variable correspondía a cada columna/fila y si estaba en la base, simplificando la depuración. El ajuste fino del criterio epsilon fue necesario para manejar comparaciones con cero en punto flotante.

# 5. Conclusiones y Recomendaciones

Conclusiones

1. **Automatización Exitosa:** La aplicación cumple con su objetivo principal de automatizar exitosamente los métodos de la Gran M y Dos Fases, eliminando la necesidad de cálculos manuales y reduciendo drásticamente el tiempo de resolución.
2. **Herramienta Educativa Valiosa:** Al mostrar cada tableau del proceso Simplex, la aplicación sirve como una potente herramienta de aprendizaje. Permite a estudiantes y profesionales visualizar el comportamiento del algoritmo, fortaleciendo su comprensión teórica.
3. **Implementación Robusta:** El software es robusto, manejando correctamente no solo los casos de solución óptima única, sino también los casos especiales de infactibilidad y soluciones no acotadas, que son cruciales para un análisis completo.
4. **Arquitectura Modular:** La arquitectura basada en el patrón MVC y el uso de clases bien definidas (SimplexBase, VariableInfo) resultó en un código mantenible, escalable y más fácil de depurar.

Recomendaciones (Trabajo Futuro)

1. **Funcionalidad de Exportación:** Añadir botones para exportar el historial de pasos y la solución final a formatos como PDF o CSV (Excel), facilitando la creación de informes.
2. **Ampliación de Algoritmos:** Extender la aplicación para incluir otros métodos de optimización, como el Método Simplex Revisado (más eficiente para problemas grandes) o algoritmos para Programación Entera (Branch and Bound).
3. **Análisis de Sensibilidad:** Implementar una sección para el análisis de sensibilidad, que permita al usuario ver cómo cambios en los coeficientes de la función objetivo o los términos independientes afectan la solución óptima.
4. **Mejoras en la Interfaz de Usuario:** Incorporar resaltado visual en la tabla del tableau para indicar claramente la fila, la columna y el elemento pivote en cada paso, mejorando aún más la experiencia de usuario.

# 6. Bibliografía

* Hillier, F. S., & Lieberman, G. J. (2010). *Introducción a la investigación de operaciones* (9a ed.). McGraw-Hill.
* Taha, H. A. (2016). *Investigación de operaciones* (9a ed.). Pearson.

ANEXOS

Anexo A: Código Fuente Completo

A continuación, presentamos el código fuente completo de las clases Java que componen el proyecto.

**Grupo 1: Aplicación Principal y UI**

**HelloApplication:**

**package** org.universidad.automatizacionmetodosgranmydosfases;

**import** **javafx.application.Application**;

**import** **javafx.fxml.FXMLLoader**;

**import** **javafx.scene.Scene**;

**import** **javafx.stage.Stage**;

**import** **java.io.IOException**;

**public** **class** **HelloApplication** **extends** Application {

**@Override**

**public** **void** **start**(Stage stage) **throws** IOException {

FXMLLoader fxmlLoader = **new** FXMLLoader(HelloApplication.class.getResource("hello-view.fxml"));

Scene scene = **new** Scene(fxmlLoader.load(), **800**, **600**); // Ajustado para mejor visualización

stage.setTitle("Solver Programación Lineal: Gran M y Dos Fases");

stage.setScene(scene);

stage.show();

}

**public** **static** **void** **main**(String[] args) {

launch();

}

}

**HelloController:**

**package** org.universidad.automatizacionmetodosgranmydosfases;

**import** **javafx.event.ActionEvent**;

**import** **javafx.fxml.FXML**;

**import** **javafx.scene.control.\***;

**import** **javafx.scene.layout.GridPane**;

**import** **javafx.scene.layout.VBox**;

**import** **org.universidad.automatizacionmetodosgranmydosfases.metodos.DosFases**;

**import** **org.universidad.automatizacionmetodosgranmydosfases.metodos.GranM**;

**import** **org.universidad.automatizacionmetodosgranmydosfases.metodos.SimplexBase**;

**import** **java.util.ArrayList**;

**import** **java.util.List**;

**public** **class** **HelloController** {

SimplexBase simplex;

**private** **double**[] funcionObjetivo;

**private** **double**[][] restricciones;

**private** **double**[] terminosIndependientes;

**private** **char**[] tipoRestricciones;

**private** ComboBox<Character> comboBoxTipoRestricciones;

**private** List<ComboBox<Character>> operadoresRestricciones;

**private** ComboBox<String> comboBoxMetodo;

**private** ComboBox<String> comboBoxTipoObjetivo;

**private** **int** pasoActual;

**@FXML** **private** Spinner<Integer> spinnerVariables;

**@FXML** **private** Spinner<Integer> spinnerRestricciones;

**@FXML** **private** VBox containerFuncionObjetivo;

**@FXML** **private** VBox containerRestricciones;

**@FXML** **private** ScrollPane scrollFuncionObjetivo;

**@FXML** **private** ScrollPane scrollRestricciones;

**@FXML** **private** List<TextField> textFieldsFuncionObjetivo;

**@FXML** **private** List<List<TextField>> textFieldsRestricciones;

**@FXML** **private** List<TextField> textFieldsTerminosIndependientes;

**public** **void** **initialize**() {

SpinnerValueFactory<Integer> spinnerVariablesValueFactory = **new** SpinnerValueFactory.IntegerSpinnerValueFactory(**2**, **100**, **2**);

spinnerVariables.setValueFactory(spinnerVariablesValueFactory);

SpinnerValueFactory<Integer> spinnerRestriccionesValueFactory = **new** SpinnerValueFactory.IntegerSpinnerValueFactory(**1**, **500**, **1**);

spinnerRestricciones.setValueFactory(spinnerRestriccionesValueFactory);

}

**@FXML**

**public** **void** **generarModelo**(ActionEvent actionEvent) {

textFieldsFuncionObjetivo = **new** ArrayList<>();

textFieldsRestricciones = **new** ArrayList<>();

textFieldsTerminosIndependientes = **new** ArrayList<>();

limpiar(actionEvent);

gridPaneFuncionObjetivo();

gridPaneResticciones();

}

**private** **void** **gridPaneFuncionObjetivo**() {

**int** nroVariables = spinnerVariables.getValue();

GridPane gridFuncionObjetivo = crearGridPane();

Label label = **new** Label("Función objetivo (Coeficientes): ");

comboBoxMetodo = **new** ComboBox<>();

comboBoxMetodo.getItems().addAll("Gran M", "Dos Fases");

comboBoxMetodo.setValue("Gran M");

comboBoxTipoObjetivo = **new** ComboBox<>();

comboBoxTipoObjetivo.getItems().addAll("Maximizar", "Minimizar");

comboBoxTipoObjetivo.setValue("Maximizar");

gridFuncionObjetivo.add(label, **0**, **0**, nroVariables, **1**);

gridFuncionObjetivo.add(comboBoxMetodo, **2**, **0**, **1**, **1**);

gridFuncionObjetivo.add(comboBoxTipoObjetivo, **3**, **0**, **1**, **1**);

**for** (**int** i = **0**; i < nroVariables; i++) {

Label labelFuncionObjetivo = **new** Label("x" + (i + **1**));

TextField textFieldCoeficientes = **new** TextField();

textFieldCoeficientes.setPromptText("Coef. X" + (i + **1**));

gridFuncionObjetivo.add(labelFuncionObjetivo, i, **1**);

gridFuncionObjetivo.add(textFieldCoeficientes, i, **2**);

textFieldsFuncionObjetivo.add(textFieldCoeficientes);

}

containerFuncionObjetivo.getChildren().add(gridFuncionObjetivo);

}

**private** **void** **gridPaneResticciones**() {

**int** nroRestricciones = spinnerRestricciones.getValue();

**int** nroVariables = spinnerVariables.getValue();

operadoresRestricciones = **new** ArrayList<>();

**for** (**int** r = **0**; r < nroRestricciones; r++) {

GridPane gridRestriccion = crearGridPane();

Label lblTituloRestriccion = **new** Label("Restricción " + (r + **1**) + ":");

gridRestriccion.add(lblTituloRestriccion, **0**, **0**, nroVariables + **2**, **1**);

List<TextField> camposRestriccion = **new** ArrayList<>();

**for** (**int** v = **0**; v < nroVariables; v++) {

Label lblVariable = **new** Label("X" + (v + **1**) + ":");

TextField txtCoeficiente = **new** TextField();

txtCoeficiente.setPromptText("Coef. X" + (v + **1**));

gridRestriccion.add(lblVariable, v, **1**);

gridRestriccion.add(txtCoeficiente, v, **2**);

camposRestriccion.add(txtCoeficiente);

}

comboBoxTipoRestricciones = **new** ComboBox<>();

comboBoxTipoRestricciones.getItems().addAll('≤', '≥', '=');

comboBoxTipoRestricciones.setValue('≤');

gridRestriccion.add(comboBoxTipoRestricciones, nroVariables, **2**);

operadoresRestricciones.add(comboBoxTipoRestricciones);

// Campo para el término independiente de la restricción

TextField txtTerminoIndependiente = **new** TextField();

txtTerminoIndependiente.setPromptText("Valor");

gridRestriccion.add(**new** Label("Término:"), nroVariables+ **1**, **1**);

gridRestriccion.add(txtTerminoIndependiente, nroVariables + **1**, **2**);

textFieldsTerminosIndependientes.add(txtTerminoIndependiente);

textFieldsRestricciones.add(camposRestriccion);

containerRestricciones.getChildren().add(gridRestriccion);

}

GridPane gridNoNegatividad = crearGridPane();

Label lblTituloNoNeg = **new** Label("Restricciones de no negatividad:");

gridNoNegatividad.add(lblTituloNoNeg, **0**, **0**, nroVariables, **1**);

**for** (**int** i = **0**; i < nroVariables; i++) {

Label lblRestriccion = **new** Label("X" + (i + **1**) + "≥ 0");

gridNoNegatividad.add(lblRestriccion, i, **1**);

}

containerRestricciones.getChildren().add(gridNoNegatividad);

Button btnResolver = **new** Button("Resolver Problema");

btnResolver.setOnAction(**this**::resolverProblema);

btnResolver.getStyleClass().add("button-limpiar");

containerRestricciones.getChildren().add(btnResolver);

}

**@FXML**

**private** **void** **resolverProblema**(ActionEvent actionEvent) {

**try** {

**int** numVariables = spinnerVariables.getValue();

**int** numRestricciones = spinnerRestricciones.getValue();

funcionObjetivo = **new** **double**[numVariables];

**for** (**int** i = **0**; i < numVariables; i++) {

funcionObjetivo[i] = Double.parseDouble(textFieldsFuncionObjetivo.get(i).getText());

}

// Capturar coeficientes de las restricciones y términos independientes

restricciones = **new** **double**[numRestricciones][numVariables];

terminosIndependientes = **new** **double**[numRestricciones];

tipoRestricciones = **new** **char**[numRestricciones];

**boolean** todasRestriccionesSonMenorIgual = **true**;

**for** (**int** r = **0**; r < numRestricciones; r++) {

**for** (**int** v = **0**; v < numVariables; v++) {

restricciones[r][v] = Double.parseDouble(textFieldsRestricciones.get(r).get(v).getText());

}

terminosIndependientes[r] = Double.parseDouble(textFieldsTerminosIndependientes.get(r).getText());

tipoRestricciones[r] = operadoresRestricciones.get(r).getValue();

**if** (tipoRestricciones[r] != '≤')

todasRestriccionesSonMenorIgual = **false**;

}

// Crear instancia del resolvedor y ejecutar el método Simplex

**boolean** maximizar;

**if** (comboBoxTipoObjetivo.getValue().equals("Maximizar"))

maximizar = **true**;

**else**

maximizar = **false**;

String metodo = comboBoxMetodo.getValue();

**if** (metodo.equals("Gran M")) {

simplex = **new** GranM(funcionObjetivo, restricciones, terminosIndependientes, tipoRestricciones, maximizar);

} **else** **if** (metodo.equals("Dos Fases")) {

simplex = **new** DosFases(funcionObjetivo, restricciones, terminosIndependientes, tipoRestricciones, maximizar);

}

simplex.resolver();

WindowSolucion window = **new** WindowSolucion(simplex);

window.mostrar();

} **catch** (NumberFormatException e) {

mostrarAlerta("Error de Entrada", "Por favor ingrese valores numéricos válidos en todos los campos.");

} **catch** (Exception e) {

mostrarAlerta("Error Inesperado", "Ocurrió un error al resolver el problema: " + e.getMessage());

e.printStackTrace();

}

}

**public** **class** **WindowSolucion** {

SimplexBase simplex;

**private** Stage stage;

**private** VBox mainContainer;

**private** GridPane gridPane;

**private** HBox btnContainer;

**private** Label labelPaso;

**private** Button btnAnterior;

**private** Button btnSiguiente;

**private** GridPane gridPaneSolucion;

**private** **int** pasoActual;

**private** **int** fila;

**public** **WindowSolucion**(SimplexBase simplex) {

**this**.simplex = simplex;

stage = **new** Stage();

mainContainer = **new** VBox(**10**);

mainContainer.getStyleClass().add("main-container");

gridPane = crearGridPane();

btnContainer = **new** HBox(**10**);

btnContainer.getStyleClass().add("btn-container");

labelPaso = **new** Label();

labelPaso.getStyleClass().add("label");

btnAnterior = **new** Button("Anterior");

btnSiguiente = **new** Button("Siguiente");

btnAnterior.setOnAction(e -> mostrarPasoAnterior());

btnSiguiente.setOnAction(e -> mostrarPasoSiguiente());

gridPaneSolucion = crearGridPane();

btnContainer.getChildren().addAll(btnAnterior, labelPaso, btnSiguiente);

mainContainer.getChildren().addAll(gridPane, btnContainer, gridPaneSolucion);

ScrollPane scrollPane = **new** ScrollPane();

scrollPane.setContent(mainContainer);

scrollPane.setFitToWidth(**true**);

scrollPane.setFitToHeight(**true**);

scrollPane.getStyleClass().add("scroll-pane");

Scene scene = **new** Scene(scrollPane, **900**, **700**); // Ajustado para mejor visualización

stage.setScene(scene);

stage.setTitle("Proceso del Método Simplex");

}

**public** **void** **mostrar**() {

mostrarPaso(pasoActual);

**if** (simplex.isSolucionEncontrada())

mostrarSolucionFinal();

stage.show();

}

**private** **void** **mostrarPaso**(**int** pasoActual) {

gridPane.getChildren().clear();

gridPane.getColumnConstraints().clear();

gridPane.getRowConstraints().clear();

**double**[][] tableau = simplex.getTablaPaso(pasoActual);

String descripcionPaso = simplex.getDescripcionPaso(pasoActual);

String varEntrada = simplex.getVariableEntradaPaso(pasoActual);

String varSalida = simplex.getVariableSalidaPaso(pasoActual);

dimencinarCeldas(tableau);

// Mostrar descripción del paso

Label labelDescripcionPaso = **new** Label(descripcionPaso);

labelDescripcionPaso.getStyleClass().add("label");

gridPane.add(labelDescripcionPaso, **0**, **0**, tableau[**0**].length + **1**, **1**);

// Mostrar variables de entrada/salida si existen

**if** (!varEntrada.isEmpty() || !varSalida.isEmpty()) {

Label labelVariables = **new** Label("Variable de Entrada: " + varEntrada + ", Variable de Salida: " + varSalida);

labelVariables.getStyleClass().add("label");

gridPane.add(labelVariables, **0**, **1**, tableau[**0**].length + **1**, **1**);

}

// Mostrar encabezados de columnas usando información de variables

mostrarEncabezadosColumnas(tableau[**0**].length);

// Mostrar RHS

Label labelRHS = **new** Label("RHS");

gridPane.add(labelRHS, tableau[**0**].length, **2**);

// Mostrar filas del tableau con etiquetas de variables en base

mostrarFilasTableau(tableau);

labelPaso.setText("Paso " + (pasoActual + **1**) + " de " + simplex.cantPasos());

btnAnterior.setDisable(pasoActual == **0**);

btnSiguiente.setDisable(pasoActual == simplex.cantPasos() - **1**);

}

**private** **void** **mostrarEncabezadosColumnas**(**int** nroColumnas) {

List<VariableInfo> variables = simplex.getVariables();

// Columna 0: Etiqueta para las filas (vacía en el encabezado)

Label labelVacio = **new** Label("Base");

gridPane.add(labelVacio, **0**, **2**);

// Columna 1: Siempre es Z (función objetivo)

Label labelZ = **new** Label("Z");

labelZ.getStyleClass().add("column-header");

gridPane.add(labelZ, **1**, **2**);

// Columnas 2 en adelante: Variables según su orden en el tableau

**for** (**int** j = **1**; j < nroColumnas - **1**; j++) {

String nombreColumna = "";

**for** (VariableInfo var : variables) {

**if** (var.getIndiceColumna() == j) {

nombreColumna = obtenerNombreVisualVariable(var);

**break**;

}

}

Label labelColumna = **new** Label(nombreColumna);

labelColumna.getStyleClass().add("column-header");

// Aplicar estilo especial si es una variable artificial

**boolean** esArtificial = **false**;

**for** (VariableInfo var : variables) {

**if** (var.getIndiceColumna() == j && var.esArtificial()) {

labelColumna.getStyleClass().add("artificial-variable");

esArtificial = **true**;

**break**;

}

}

gridPane.add(labelColumna, j + **1**, **2**);

}

}

**private** **void** **mostrarFilasTableau**(**double**[][] tableau) {

List<VariableInfo> variables = simplex.getVariables();

**for** (**int** i = **0**; i < tableau.length; i++) {

String labelFilaText; // Etiqueta de fila

**if** (i == **0**) {

labelFilaText = "Z";

} **else** {

labelFilaText = "R" + i; // Default

**for** (VariableInfo var : variables) {

**if** (var.isEstaEnBase() && var.getFilaEnBase() == i) {

labelFilaText = obtenerNombreVisualVariable(var);

**break**;

}

}

}

Label labelFila = **new** Label(labelFilaText);

labelFila.getStyleClass().add("row-label");

gridPane.add(labelFila, **0**, i + **3**);

// Valores de la fila

**for** (**int** j = **0**; j < tableau[i].length; j++) {

String valor = String.format("%.2f", tableau[i][j]);

TextField labelValor = **new** TextField(valor.equals("-0.00") ? "0.00" : valor);

labelValor.setEditable(**false**);

**if** (j > **0**) {

**for** (VariableInfo var : variables) {

**if** (var.getIndiceColumna() == j && var.esArtificial()) {

labelValor.getStyleClass().add("artificial-variable");

**break**;

}

}

}

gridPane.add(labelValor, j + **1**, i + **3**);

}

}

}

**private** String **obtenerNombreVisualVariable**(VariableInfo var) {

**switch** (var.getTipo()) {

**case** **ORIGINAL:**

String numero = var.getNombre().substring(**1**);

**return** "X" + numero;

**case** **HOLGURA:**

String numeroH = var.getNombre().substring(**1**);

**return** "S" + numeroH;

**case** **EXCESO:**

String numeroE = var.getNombre().substring(**1**);

**return** "E" + numeroE;

**case** **ARTIFICIAL:**

String numeroA = var.getNombre().substring(**1**);

**return** "A" + numeroA;

**default**:

**return** var.getNombre().toUpperCase();

}

}

**private** **void** **dimencinarCeldas**(**double**[][] tableau) {

**int** totalColumnas = tableau[**0**].length + **1**;

**for** (**int** i = **0**; i < totalColumnas; i++) {

ColumnConstraints columna = **new** ColumnConstraints();

columna.setMinWidth(**80**);

columna.setPrefWidth(**80**);

columna.setMaxWidth(**80**);

gridPane.getColumnConstraints().add(columna);

}

}

**private** **void** **mostrarSolucionFinal**() {

gridPaneSolucion.getChildren().clear();

gridPaneSolucion.getColumnConstraints().clear();

**for** (**int** i = **0**; i < **3**; i++) {

ColumnConstraints columna = **new** ColumnConstraints();

columna.setMinWidth(**120**);

columna.setPrefWidth(**120**);

gridPaneSolucion.getColumnConstraints().add(columna);

}

Label titulo = **new** Label("Solución Óptima");

titulo.getStyleClass().add("title-label");

gridPaneSolucion.add(titulo, **0**, **0**, **3**, **1**);

fila = **1**;

List<VariableInfo> variablesOriginales = simplex.obtenerVariablesPorTipo(VariableInfo.TipoVariable.ORIGINAL);

mostrarResultadoDeLasVariables(variablesOriginales, "Variables de Decisión:", "(Original)");

List<VariableInfo> variablesHolgura = simplex.obtenerVariablesPorTipo(VariableInfo.TipoVariable.HOLGURA);

List<VariableInfo> holguraEnBase = variablesHolgura.stream()

.filter(var -> var.isEstaEnBase() && Math.abs(var.getValor()) > SimplexBase.epsilon)

.toList();

mostrarResultadoDeLasVariables(holguraEnBase, "Variables de Holgura:", "(Holgura)");

List<VariableInfo> variablesExceso = simplex.obtenerVariablesPorTipo(VariableInfo.TipoVariable.EXCESO);

List<VariableInfo> excesoEnBase = variablesExceso.stream()

.filter(var -> var.isEstaEnBase() && Math.abs(var.getValor()) > SimplexBase.epsilon)

.toList();

mostrarResultadoDeLasVariables(excesoEnBase, "Variables de Exceso:", "(Exceso)");

List<VariableInfo> variablesArtificial = simplex.obtenerVariablesPorTipo(VariableInfo.TipoVariable.ARTIFICIAL);

List<VariableInfo> artificialEnBase = variablesArtificial.stream()

.filter(var -> var.isEstaEnBase() && Math.abs(var.getValor()) > SimplexBase.epsilon)

.toList();

mostrarResultadoDeLasVariables(artificialEnBase, "Variables Artificiales:", "(Artificial)");

fila++;

Label labelZ = **new** Label("Función Objetivo (Z):");

labelZ.getStyleClass().add("objective-label");

Label labelValorZ = **new** Label(String.format("%.2f", simplex.solucion.get("z")));

labelValorZ.getStyleClass().add("objective-value");

Label tipoProblema = **new** Label(simplex.isMaximizar() ? "(Maximización)" : "(Minimización)");

tipoProblema.getStyleClass().add("tipo-variable");

gridPaneSolucion.add(labelZ, **0**, fila);

gridPaneSolucion.add(labelValorZ, **1**, fila);

gridPaneSolucion.add(tipoProblema, **2**, fila);

}

**private** **void** **mostrarResultadoDeLasVariables**(List<VariableInfo> variables, String subtitulo, String tipo) {

**if** (!variables.isEmpty()) {

Label subtituloOriginales = **new** Label(subtitulo);

subtituloOriginales.getStyleClass().add("subtitle-label");

gridPaneSolucion.add(subtituloOriginales, **0**, fila++, **3**, **1**);

**for** (VariableInfo var : variables) {

**double** valor = var.getValor();

Label txtVar = **new** Label(obtenerNombreVisualVariable(var) + ":");

Label txtValor = **new** Label(String.format("%.2f", valor));

Label txtTipo = **new** Label(tipo);

txtTipo.getStyleClass().add("tipo-variable");

gridPaneSolucion.add(txtVar, **0**, fila);

gridPaneSolucion.add(txtValor, **1**, fila);

gridPaneSolucion.add(txtTipo, **2**, fila);

fila++;

}

}

}

**private** **void** **mostrarPasoAnterior**() {

**if** (pasoActual > **0**) {

pasoActual--;

mostrarPaso(pasoActual);

}

}

**Grupo 2: Lógica de Algoritmos Core**

**private** **void** **mostrarPasoSiguiente**() {

**if** (pasoActual < simplex.cantPasos() - **1**) {

mostrarPaso(pasoActual); pasoActual++;

}

}

}

**SimplexBase:**

**package** org.universidad.automatizacionmetodosgranmydosfases.metodos;

**import** **org.universidad.automatizacionmetodosgranmydosfases.guardarPasos.GuardarPasos**;

**import** **java.math.BigDecimal**;

**import** **java.math.RoundingMode**;

**import** **java.util.ArrayList**;

**import** **java.util.HashMap**;

**import** **java.util.List**;

**import** **java.util.Map**;

**import** **java.util.stream.Collectors**;

**public** **abstract** **class** **SimplexBase** {

**protected** **double**[][] M;

**protected** **int** nroFilas, nroColumnas, nroVariables, nroRestricciones;

**protected** **boolean** maximizar, soluciónOptima;

**protected** List<VariableInfo> variables;

**protected** Map<String, VariableInfo> variablesPorNombre;

**public** Map<String, Double> solucion;

**protected** GuardarPasos historialDePasos;

**public** **static** **final** **double** epsilon = **1**e-**10**;

**public** **static** **final** **byte** decimales = **6**;

**protected** **SimplexBase**() {

**this**.soluciónOptima = **true**;

**this**.solucion = **new** HashMap<>();

**this**.historialDePasos = **new** GuardarPasos();

**this**.variables = **new** ArrayList<>();

**this**.variablesPorNombre = **new** HashMap<>();

}

**protected** **void** **inicializarTableauBase**(**char**[] tipoRestricciones) {

contarVariablesPorTipo(tipoRestricciones);

**this**.nroFilas = **this**.nroRestricciones + **1**;

**this**.nroColumnas = **1** + getTotalVariables() + **1**; // Z + variables + RHS

**this**.M = **new** **double**[**this**.nroFilas][**this**.nroColumnas];

}

**private** **void** **contarVariablesPorTipo**(**char**[] tipoRestricciones) {

variables.clear(); variablesPorNombre.clear();

**int** indiceColumna = **1**; // Columna 0 es para Z

**for** (**int** i = **0**; i < nroVariables; i++) { // Variables originales

String nombre = "x" + (i + **1**);

VariableInfo var = **new** VariableInfo(indiceColumna++, VariableInfo.TipoVariable.ORIGINAL, nombre);

variables.add(var);

variablesPorNombre.put(nombre, var);

} // Variables de holgura, exceso y artificiales

**int** contadorHolgura = **1**, contadorExceso = **1**, contadorArtificial = **1**;

**int** restriccionIdxExceso = **0**;

**for** (**int** i = **0**; i < tipoRestricciones.length; i++) {

**switch** (tipoRestricciones[i]) {

**case** '≤':

String nombreHolgura = "s" + contadorHolgura++;

VariableInfo varHolgura = **new** VariableInfo(indiceColumna++,

VariableInfo.TipoVariable.HOLGURA, nombreHolgura, i + **1**);

variables.add(varHolgura);

variablesPorNombre.put(nombreHolgura, varHolgura);

**break**;

**case** '≥':

String nombreExceso = "e" + contadorExceso++;

VariableInfo varExceso = **new** VariableInfo(indiceColumna++,

VariableInfo.TipoVariable.EXCESO, nombreExceso);

variables.add(varExceso);

variablesPorNombre.put(nombreExceso, varExceso);

String nombreArtificial1 = "a" + contadorArtificial++;

VariableInfo varArtificial1 = **new** VariableInfo(indiceColumna++,

VariableInfo.TipoVariable.ARTIFICIAL, nombreArtificial1, i + **1**);

variables.add(varArtificial1);

variablesPorNombre.put(nombreArtificial1, varArtificial1);

**break**;

**case** '=':

String nombreArtificial2 = "a" + contadorArtificial++;

VariableInfo varArtificial2 = **new** VariableInfo(indiceColumna++,

VariableInfo.TipoVariable.ARTIFICIAL, nombreArtificial2, i + **1**);

variables.add(varArtificial2);

variablesPorNombre.put(nombreArtificial2, varArtificial2);

**break**;

}

}

}

**protected** **void** **setValor**(**int** fila, **int** columna, **double** valor) {

validarRango(fila, columna);

**this**.M[fila][columna] = valor;

}

**protected** **double** **getValor**(**int** fila, **int** columna) {

validarRango(fila, columna);

**return** **this**.M[fila][columna];

}

**private** **void** **validarRango**(**int** f, **int** c) {

**if** (f < **0** || f >= nroFilas) {

**throw** **new** **IllegalArgumentException**(String.format(

"Índice de fila fuera de rango: %d (válido: 0-%d)", f, nroFilas-**1**));

}

**if** (c < **0** || c >= nroColumnas) {

**throw** **new** **IllegalArgumentException**(String.format(

"Índice de columna fuera de rango: %d (válido: 0-%d)", c, nroColumnas-**1**));

}

}

**protected** **boolean** existenNegativosEnLaFunciónObjetivo() {

**for** (**int** j = **1**; j < nroColumnas - **1**; j++) {

**if** (getValor(**0**, j) < -epsilon) **return** **true**;

}

**return** **false**;

}

**public** **abstract** **void** **resolver**();

**protected** **void** iteraciónSimplex() {

**int** columnaPivote = obtenerColumnaPivote();

**if** (columnaPivote == **0**) {

// No hay más columnas pivote, se alcanzó el óptimo (o una condición de parada)

**return**;

}

**int** filaPivote = obtenerFilaPivote(columnaPivote);

**if** (filaPivote == **0**) {

manejarProblemaNoAcotado("La solución es no acotada.");

**return**;

}

VariableInfo varEntrante = obtenerVariablePorColumna(columnaPivote);

VariableInfo varSaliente = obtenerVariableEnBase(filaPivote);

String nombreVarEntrante = (varEntrante != **null**) ? varEntrante.getNombre() : "N/A";

String nombreVarSaliente = (varSaliente != **null**) ? varSaliente.getNombre() : "N/A";

guardarPaso("Pivoteo. Entra: " + nombreVarEntrante + ", Sale: " + nombreVarSaliente, nombreVarEntrante, nombreVarSaliente);

realizarPivoteo(columnaPivote, filaPivote);

actualizarBaseDeVariables(varEntrante, varSaliente, filaPivote);

}

**protected** **void** **actualizarBaseDeVariables**(VariableInfo varEntrante, VariableInfo varSaliente, **int** filaPivote) {

**if** (varSaliente != **null**) {

varSaliente.sacarDeBase();

}

**if** (varEntrante != **null**) {

**double** valor = getValor(filaPivote, nroColumnas - **1**);

varEntrante.ponerEnBase(filaPivote, valor);

}

}

**protected** **void** **realizarPivoteo**(**int** columnaPivote, **int** filaPivote) {

**double** valorPivote = getValor(filaPivote, columnaPivote);

normalizarFilaPivote(filaPivote, valorPivote);

**for** (**int** i = **0**; i < nroFilas; i++) {

**if** (i == filaPivote) **continue**;

**double** factor = getValor(i, columnaPivote);

**for** (**int** j = **0**; j < nroColumnas; j++) {

setValor(i, j, redondear(getValor(i, j) - factor \* getValor(filaPivote, j)));

}

}

}

**protected** **void** **normalizarFilaPivote**(**int** filaPivote, **double** valorPivote) {

**if** (Math.abs(valorPivote) < epsilon) **return**;

**for** (**int** j = **0**; j < nroColumnas; j++) {

setValor(filaPivote, j, getValor(filaPivote, j) / valorPivote);

}

}

**protected** **abstract** **int** **obtenerColumnaPivote**();

**protected** **int** **obtenerColumnaPivoteMaximizar**() {

**double** maxNegativo = -epsilon;

**int** columnaPivote = **0**;

**for** (**int** j = **1**; j < nroColumnas - **1**; j++) {

**if** (getValor(**0**, j) < maxNegativo) {

maxNegativo = getValor(**0**, j);

columnaPivote = j;

}

}

**return** columnaPivote;

}

**protected** **int** **obtenerFilaPivote**(**int** columnaPivote) {

**double** minimoActual = Double.POSITIVE\_INFINITY;

**int** filaPivote = **0**;

**for** (**int** i = **1**; i < nroFilas; i++) {

**if** (getValor(i, columnaPivote) > epsilon) {

**double** razón = getValor(i, nroColumnas - **1**) / getValor(i, columnaPivote);

**if** (razón < minimoActual) {

minimoActual = razón;

filaPivote = i;

}

}

}

**return** filaPivote;

}

**protected** **abstract** **boolean** **esOptimo**();

**protected** **boolean** **hayVariablesArtificialesEnBase**() {

**return** variables.stream()

.filter(**VariableInfo:**:esArtificial)

.filter(**VariableInfo:**:isEstaEnBase)

.anyMatch(var -> var.getValor() > epsilon);

}

**protected** **void** **manejarProblemaNoAcotado**(String tipo) {

guardarPaso("Problema No Acotado: " + tipo, "", "");

soluciónOptima = **false**;

}

**protected** **void** **finalizarSolucion**() {

guardarPaso("Solución Óptima Encontrada", "", "");

actualizarValoresVariables(); solucion.clear();

variables.stream()

.filter(**VariableInfo:**:esOriginal)

.forEach(var -> solucion.put(var.getNombre(), var.getValor()));

**double** valorZ = getValor(**0**, nroColumnas - **1**);

solucion.put("z", maximizar ? valorZ : -valorZ);

}

**protected** **void** **actualizarValoresVariables**() {

variables.forEach(var -> var.setValor(**0.0**));

variables.stream()

.filter(**VariableInfo:**:isEstaEnBase)

.forEach(var -> {

**double** valor = getValor(var.getFilaEnBase(), nroColumnas - **1**);

var.setValor(redondear(valor));

});

}

**protected** VariableInfo **obtenerVariablePorColumna**(**int** columna) {

**return** variables.stream()

.filter(var -> var.getIndiceColumna() == columna)

.findFirst()

.orElse(**null**);

}

**protected** VariableInfo **obtenerVariableEnBase**(**int** fila) {

**return** variables.stream()

.filter(**VariableInfo:**:isEstaEnBase)

.filter(var -> var.getFilaEnBase() == fila)

.findFirst()

.orElse(**null**);

}

**public** List<VariableInfo> **obtenerVariablesPorTipo**(VariableInfo.TipoVariable tipo) {

**return** variables.stream()

.filter(var -> var.getTipo() == tipo)

.collect(Collectors.toList());

}

**protected** **int** **getTotalVariables**() {

**return** variables.size();

}

**protected** **void** **guardarPaso**(String descripcion, String variableEntrada, String variableSalida) {

historialDePasos.agregarPaso(**this**.M, variableEntrada, variableSalida, descripcion);

}

**public** **static** **double** **redondear**(**double** valor) {

**if** (Math.abs(valor) < epsilon) {

**return** **0.0**;

}

BigDecimal bd = BigDecimal.valueOf(valor);

bd = bd.setScale(decimales, RoundingMode.HALF\_UP);

**return** bd.doubleValue();

}

// Getters

**public** **boolean** **isSolucionEncontrada**() { **return** soluciónOptima; }

**public** **boolean** **isMaximizar**() { **return** maximizar; }

**public** List<VariableInfo> **getVariables**() { **return** **new** ArrayList<>(variables); }

**public** **int** **cantPasos**() { **return** historialDePasos.getListaPasos().size(); }

**public** String **getDescripcionPaso**(**int** i) { **return** historialDePasos.getListaPasos().get(i).getDescripcionPaso(); }

**public** String **getVariableEntradaPaso**(**int** i) { **return** historialDePasos.getListaPasos().get(i).getVariableEntrada(); }

**public** String **getVariableSalidaPaso**(**int** i) { **return** historialDePasos.getListaPasos().get(i).getVariableSalida(); }

**public** **double**[][] **getTablaPaso**(**int** indice) { **return** historialDePasos.getTablaPaso(indice); }

}

**GranM:**

**package** org.universidad.automatizacionmetodosgranmydosfases.metodos;

**public** **class** **GranM** **extends** SimplexBase {

**private** **static** **final** **double** M\_VALOR = **1**E6;

**public** **GranM**(**double**[] funcionObjetivo, **double**[][] restricciones,

**double**[] terminosIndependientes, **char**[] tipoRestricciones, **boolean** maximizar) {

**super**();

**super**.nroVariables = funcionObjetivo.length;

**super**.nroRestricciones = restricciones.length;

**super**.maximizar = maximizar;

**super**.inicializarTableauBase(tipoRestricciones);

construirTableau(funcionObjetivo, restricciones, terminosIndependientes, tipoRestricciones);

**super**.guardarPaso("Tableau Inicial Construido", "", "");

actualizarZParaM();

**super**.guardarPaso("Fila Z ajustada para penalizaciones M", "", "");

}

**private** **void** **construirTableau**(**double**[] funcionObjetivo, **double**[][] restricciones,

**double**[] terminosIndependientes, **char**[] tipoRestricciones) {

setValor(**0**, **0**, **1**);

**for** (VariableInfo var : variables) {

**if** (var.esOriginal()) {

**int** indiceOriginal = Integer.parseInt(var.getNombre().substring(**1**)) - **1**;

**double** coeficiente = maximizar ? -funcionObjetivo[indiceOriginal] : funcionObjetivo[indiceOriginal];

setValor(**0**, var.getIndiceColumna(), coeficiente);

} **else** **if** (var.esArtificial()) {

setValor(**0**, var.getIndiceColumna(), M\_VALOR);

}

}

**for** (**int** i = **0**; i < nroRestricciones; i++) {

**for** (**int** v = **0**; v < nroVariables; v++) {

setValor(i + **1**, v + **1**, restricciones[i][v]);

}

// Configurar variables de holgura, exceso y artificiales

**int** finalI = i;

VariableInfo holgura = variables.stream().filter(v -> v.esHolgura() && v.getFilaEnBase() == finalI + **1**).findFirst().orElse(**null**);

VariableInfo exceso = variables.stream().filter(v -> v.esExceso() && !v.isEstaEnBase()).findFirst();

VariableInfo artificial = variables.stream().filter(v -> v.esArtificial() && v.getFilaEnBase() == finalI + **1**).findFirst().orElse(**null**);

**if**(tipoRestricciones[i] == '≤' && holgura != **null**) {

setValor(i + **1**, holgura.getIndiceColumna(), **1**);

} **else** **if** (tipoRestricciones[i] == '≥') {

// Find corresponding excess and artificial variables for this row

**final** **int** currentArtificialCount = (**int**) variables.stream().filter(v -> v.esArtificial() && v.getFilaEnBase() <= finalI + **1**).count();

VariableInfo artForThisRow = variables.stream().filter(v -> v.esArtificial() && v.getNombre().equals("a" + currentArtificialCount)).findFirst().orElse(**null**);

**final** **int** currentExcesoCount = (**int**) variables.stream().filter(v -> v.esExceso()).count();

VariableInfo excForThisRow = variables.stream().filter(v -> v.esExceso() && v.getNombre().equals("e" + currentArtificialCount)).findFirst().orElse(**null**);

**if** (artForThisRow != **null**) setValor(i + **1**, artForThisRow.getIndiceColumna(), **1**);

**if** (excForThisRow != **null**) setValor(i + **1**, excForThisRow.getIndiceColumna(), -**1**);

} **else** **if** (tipoRestricciones[i] == '=' && artificial != **null**) {

setValor(i + **1**, artificial.getIndiceColumna(), **1**);

}

setValor(i + **1**, nroColumnas - **1**, terminosIndependientes[i]);

}

}

**private** **void** **actualizarZParaM**() {

**for** (VariableInfo var : variables) {

**if** (var.esArtificial() && var.isEstaEnBase()) {

**int** filaVariable = var.getFilaEnBase();

**double** factorM = getValor(**0**, var.getIndiceColumna());

**for** (**int** k = **0**; k < nroColumnas; k++) {

**double** valorActualZ = getValor(**0**, k);

**double** valorFila = getValor(filaVariable, k);

setValor(**0**, k, valorActualZ - factorM \* valorFila);

}

}

}

}

**@Override**

**public** **void** **resolver**() {

**while** (!esOptimo() && soluciónOptima) {

iteraciónSimplex();

}

**if** (soluciónOptima) {

actualizarValoresVariables();

**if** (hayVariablesArtificialesEnBase()) {

**super**.guardarPaso("Solución Infactible: Una o más variables artificiales están en la base con valor positivo.", "", "");

soluciónOptima = **false**;

**return**;

} **else** {

finalizarSolucion();

}

}

}

**@Override**

**protected** **int** **obtenerColumnaPivote**() {

**return** **super**.obtenerColumnaPivoteMaximizar();

}

**@Override**

**protected** **boolean** **esOptimo**() {

**return** !existenNegativosEnLaFunciónObjetivo();

}

}

**DosFases:**

**package** org.universidad.automatizacionmetodosgranmydosfases.metodos;

**import** **java.util.HashMap**;

**import** **java.util.List**;

**import** **java.util.Map**;

**public** **class** **DosFases** **extends** SimplexBase {

**private** **double**[] funcionObjetivoOriginal;

**private** **boolean** maximizarOriginal;

**public** **DosFases**(**double**[] funcionObjetivo, **double**[][] restricciones,

**double**[] terminosIndependientes, **char**[] tipoRestricciones, **boolean** maximizar) {

**super**();

**super**.nroVariables = funcionObjetivo.length;

**super**.nroRestricciones = restricciones.length;

maximizarOriginal = maximizar; // Guardar el objetivo original

**super**.maximizar = **false**; // Fase 1 siempre es minimizar

guardarZOriginal(funcionObjetivo);

**super**.inicializarTableauBase(tipoRestricciones);

construirTableauFase1(restricciones, terminosIndependientes);

**super**.guardarPaso("Tableau inicial para Fase 1", "", "");

}

**private** **void** **guardarZOriginal**(**double**[] funcionObjetivo) {

**this**.funcionObjetivoOriginal = **new** **double**[nroVariables];

System.arraycopy(funcionObjetivo, **0**, **this**.funcionObjetivoOriginal, **0**, nroVariables);

}

**private** **void** **construirTableauFase1**(**double**[][] restricciones, **double**[] terminosIndependientes) {

// Fila Z (Fase 1): Minimizar la suma de artificiales

setValor(**0**, **0**, **1**);

**for** (VariableInfo var : variables) {

**if** (var.esArtificial()) {

setValor(**0**, var.getIndiceColumna(), **1**);

}

}

// Llenar restricciones

**for** (**int** i = **0**; i < nroRestricciones; i++) {

**for** (**int** v = **0**; v < nroVariables; v++) {

setValor(i + **1**, v + **1**, restricciones[i][v]);

}

**int** finalI = i;

VariableInfo holgura = variables.stream().filter(v -> v.esHolgura() && v.getFilaEnBase() == finalI + **1**).findFirst().orElse(**null**);

**if**(holgura != **null**) {

setValor(i + **1**, holgura.getIndiceColumna(), **1**);

}

**final** **int** currentArtificialCount = (**int**) variables.stream().filter(v -> v.esArtificial() && v.getFilaEnBase() <= finalI + **1**).count();

VariableInfo artForThisRow = variables.stream().filter(v -> v.esArtificial() && v.getNombre().equals("a" + currentArtificialCount)).findFirst().orElse(**null**);

**final** **int** currentExcesoCount = (**int**) variables.stream().filter(v -> v.esExceso()).count();

VariableInfo excForThisRow = variables.stream().filter(v -> v.esExceso() && v.getNombre().equals("e" + currentArtificialCount)).findFirst().orElse(**null**);

**if** (artForThisRow != **null**) setValor(i + **1**, artForThisRow.getIndiceColumna(), **1**);

**if** (excForThisRow != **null**) setValor(i + **1**, excForThisRow.getIndiceColumna(), -**1**);

setValor(i + **1**, nroColumnas - **1**, terminosIndependientes[i]);

}

// Ajustar Fila Z para ponerla en forma canónica

**for** (VariableInfo var : variables) {

**if** (var.esArtificial() && var.isEstaEnBase()) {

**int** filaBase = var.getFilaEnBase();

**for** (**int** j = **0**; j < nroColumnas; j++) {

setValor(**0**, j, getValor(**0**, j) - getValor(filaBase, j));

}

}

}

}

**@Override**

**public** **void** **resolver**() {

// --- FASE 1 ---

**super**.maximizar = **true**; // Invertimos porque queremos minimizar la Z auxiliar (que es la suma de artificiales)

**super**.guardarPaso("Inicio Fase 1: Minimizar suma de variables artificiales", "", "");

**while** (!esOptimoFase1() && soluciónOptima) {

iteraciónSimplex();

}

**if** (Math.abs(getValor(**0**, nroColumnas - **1**)) > epsilon) {

soluciónOptima = **false**;

**super**.guardarPaso("Fase 1 terminada. Problema Infactible (Z\_aux > 0)", "", "");

**return**;

}

**super**.guardarPaso("Fase 1 Completada. Base Factible Encontrada.", "", "");

// --- FASE 2 ---

prepararFase2();

**super**.guardarPaso("Inicio Fase 2: Resolver problema original", "", "");

**while** (!esOptimo() && soluciónOptima) {

iteraciónSimplex();

}

**if** (soluciónOptima) {

finalizarSolucion();

}

}

**private** **void** **prepararFase2**() {

**super**.maximizar = maximizarOriginal;

// Restaurar función objetivo original

setValor(**0**, **0**, **1**);

**for**(**int** j = **1**; j < nroColumnas - **1**; j++) setValor(**0**, j, **0**); // Limpiar Z

**for** (**int** i = **0**; i < nroVariables; i++) {

**double** coef = maximizarOriginal ? -funcionObjetivoOriginal[i] : funcionObjetivoOriginal[i];

setValor(**0**, i + **1**, coef);

}

// Ajustar Z para la nueva base

**for** (VariableInfo var : variables) {

**if** (var.isEstaEnBase() && var.esOriginal()) {

**double** coefZ = getValor(**0**, var.getIndiceColumna());

**if** (Math.abs(coefZ) > epsilon) {

**int** filaBase = var.getFilaEnBase();

**for** (**int** j = **0**; j < nroColumnas; j++) {

setValor(**0**, j, getValor(**0**, j) - coefZ \* getValor(filaBase, j));

}

}

}

}

}

**private** **boolean** **esOptimoFase1**() {

// En Fase 1, se minimiza la suma de artificiales (o maximiza -suma)

// El óptimo se alcanza cuando todos los coeficientes de Z son >= 0

**for** (**int** j = **1**; j < nroColumnas - **1**; j++) {

// Ignorar columnas de variables artificiales que ya salieron de la base

VariableInfo var = obtenerVariablePorColumna(j);

**if**(var != **null** && var.esArtificial() && !var.isEstaEnBase()) **continue**;

**if** (getValor(**0**, j) < -epsilon) **return** **false**;

}

**return** **true**;

}

**@Override**

**protected** **int** **obtenerColumnaPivote**() {

**return** **super**.obtenerColumnaPivoteMaximizar();

}

**@Override**

**protected** **boolean** **esOptimo**() {

**return** !existenNegativosEnLaFunciónObjetivo();

}

}

**Grupo 3: Clases de Soporte y Estructuras de Datos**

**VariableInfo:**

**package** org.universidad.automatizacionmetodosgranmydosfases.metodos;

**public** **class** **VariableInfo** {

**public** **enum** TipoVariable {

ORIGINAL,

HOLGURA,

EXCESO,

ARTIFICIAL

}

**private** **final** **int** indiceColumna;

**private** **final** TipoVariable tipo;

**private** **final** String nombre;

**private** **boolean** estaEnBase;

**private** **int** filaEnBase;

**private** **double** valor;

**public** **VariableInfo**(**int** indiceColumna, TipoVariable tipo, String nombre) {

**this**.indiceColumna = indiceColumna;

**this**.tipo = tipo;

**this**.nombre = nombre;

**this**.estaEnBase = **false**;

**this**.filaEnBase = -**1**;

**this**.valor = **0.0**;

}

**public** **VariableInfo**(**int** indiceColumna, TipoVariable tipo, String nombre, **int** filaEnBase) {

**this**(indiceColumna, tipo, nombre);

**this**.estaEnBase = **true**;

**this**.filaEnBase = filaEnBase;

}

**public** **int** **getIndiceColumna**() { **return** indiceColumna; }

**public** TipoVariable **getTipo**() { **return** tipo; }

**public** String **getNombre**() { **return** nombre; }

**public** **boolean** **isEstaEnBase**() { **return** estaEnBase; }

**public** **int** **getFilaEnBase**() { **return** filaEnBase; }

**public** **double** **getValor**() { **return** valor; }

**public** **void** **setValor**(**double** valor) { **this**.valor = valor; }

**public** **boolean** **esArtificial**() { **return** tipo == TipoVariable.ARTIFICIAL; }

**public** **boolean** **esOriginal**() { **return** tipo == TipoVariable.ORIGINAL; }

**public** **boolean** **esHolgura**() { **return** tipo == TipoVariable.HOLGURA; }

**public** **boolean** **esExceso**() { **return** tipo == TipoVariable.EXCESO; }

**public** **void** **sacarDeBase**() {

**this**.estaEnBase = **false**;

**this**.filaEnBase = -**1**;

}

**public** **void** **ponerEnBase**(**int** fila, **double** valor) {

**this**.estaEnBase = **true**;

**this**.filaEnBase = fila;

**this**.valor = valor;

}

**@Override**

**public** String **toString**() {

**return** String.format("%s (col:%d, tipo:%s, base:%s, fila:%d, valor:%.2f)",

nombre, indiceColumna, tipo, estaEnBase ? "Sí" : "No", filaEnBase, valor);

}

}

**GuardarPasos:**

**package** org.universidad.automatizacionmetodosgranmydosfases.guardarPasos;

**import** **java.util.ArrayList**;

**import** **java.util.List**;

**public** **class** **GuardarPasos** {

**private** List<EstructuraParaGuardarPasos> listaPasos;

**public** **GuardarPasos**() {

**this**.listaPasos = **new** ArrayList<>();

}

**public** **void** **agregarPaso**(**double**[][] tablaPasos, String variableEntrada, String variableSalida, String descripcionPaso) {

listaPasos.add(**new** EstructuraParaGuardarPasos(tablaPasos, variableEntrada, variableSalida, descripcionPaso));

}

**public** List<EstructuraParaGuardarPasos> **getListaPasos**() {

**return** listaPasos;

}

**public** **void** **limpiarPasos**(){

listaPasos.clear();

}

**public** **double**[][] **getTablaPaso**(**int** indice){

**return** listaPasos.get(indice).getTablaPasos();

}

}

**EstructuraParaGuardarPasos:**

**package** org.universidad.automatizacionmetodosgranmydosfases.guardarPasos;

**public** **class** **EstructuraParaGuardarPasos** {

**private** **double**[][] tablaPasos;

**private** String variableEntrada;

**private** String variableSalida;

**private** String descripcionPaso;

**public** **EstructuraParaGuardarPasos**(**double**[][] tablaPasos, String variableEntrada, String variableSalida, String descripcionPaso) {

**this**.tablaPasos = copiarTabla(tablaPasos);

**this**.variableEntrada = variableEntrada;

**this**.variableSalida = variableSalida;

**this**.descripcionPaso = descripcionPaso;

}

**private** **double**[][] **copiarTabla**(**double**[][] tablaOriginal) {

**if** (tablaOriginal == **null**) {

**return** **null**;

}

**double**[][] copia = **new** **double**[tablaOriginal.length][];

**for** (**int** i = **0**; i < tablaOriginal.length; i++) {

**if** (tablaOriginal[i] != **null**) {

copia[i] = tablaOriginal[i].clone();

}

}

**return** copia;

}

**public** **double**[][] **getTablaPasos**() {

**return** tablaPasos;

}

**public** String **getVariableEntrada**() {

**return** variableEntrada;

}

**public** String **getVariableSalida**() {

**return** variableSalida;

}

**public** String **getDescripcionPaso**() {

**return** descripcionPaso;

}

}

## Anexo B: Manual del Usuario

1. **Inicio de la Aplicación:**

* Ejecute el programa. Se abrirá la ventana principal.

1. **Configuración del Problema:**

* **Paso 1:** Use los selectores (spinners) en la parte superior para definir el "Número de Variables" y el "Número de Restricciones" de su problema.
* **Paso 2:** Haga clic en el botón **"Generar Modelo"**. La interfaz se actualizará para mostrar los campos necesarios.

1. **Ingreso de Datos:**
2. **Función Objetivo:**
   1. Seleccione "Maximizar" o "Minimizar" en el menú desplegable.
   2. Seleccione el método a utilizar: "Gran M" o "Dos Fases".
   3. Ingrese los coeficientes de cada variable x en los campos de texto correspondientes.
3. **Restricciones:**
   1. Para cada restricción, ingrese los coeficientes de las variables.
   2. Seleccione el tipo de desigualdad (≤, ≥, =) en el menú desplegable.
   3. Ingrese el valor del término independiente (lado derecho de la ecuación) en el campo "Valor".
4. **Ejecución y Resultados:**

* **Paso 1:** Una vez que todos los datos estén ingresados, haga clic en el botón **"Resolver Problema"**.
* **Paso 2:** Si los datos son válidos, se abrirá una nueva ventana titulada "Proceso del Método Simplex".
* **Paso 3:** En esta ventana, podrá ver el tableau inicial. Use los botones **"Siguiente"** y **"Anterior"** para navegar a través de cada iteración del algoritmo.
* **Paso 4:** Al final del proceso, en la parte inferior de la ventana, se mostrará un resumen con la **"Solución Óptima"**, detallando los valores finales de las variables de decisión y el valor óptimo de la función objetivo Z.
* En caso de un error (ej. problema no acotado o infactible), se mostrará un mensaje descriptivo.

1. **Limpiar y Salir:**

* Use el botón **"Limpiar"** en la ventana principal para borrar todos los campos y comenzar un nuevo problema.
* Use el botón **"Salir"** para cerrar la aplicación.

## Anexo C: Presentación y Repositorio

Diapositivas: Realizamos una presentación dentro de una página con html, css y javascript (también se puede obtener en el enlace del repositorio).

Repositorio de Código: El código fuente está disponible en:<https://github.com/Nakroth-5/Automatizacion-Metodos-GranM-Y-DosFases>