Instituto Tecnológico Superior del Oriente del Estado de Hidalgo
Ingeniería en Sistemas Computacionales

Métodos Numéricos

Grupo: 4F21 **Semestre**: 4to

Actividad:
Problemario

Integrantes
Diego Alonso Coronel Vargas
Brandon García Ordaz
Oscar Aaron Delgadillo Fernandez

Contenido

Introducción **3**
Ejercicios de Gauss Jordan **4**
Ejercicio 1 9
Ejercicio 2 10
Ejercicio 3 12
Ejercicio 4 14
Ejercicio 5 15
Ejercicio de eliminación gaussiana con pivoteo **17**
Pseudocódigo 20
Ejercicio 1 22
Ejercicio 2 23
Ejercicio 3 25
Ejercicio 4 27
Ejercicio 5 29
Ejercicios de Jacobi **31**
Pseudocódigo 31
Ejercicio 1 37
Ejercicio 2 38
Ejercicio 3 40
Ejercicio 4 43
Ejercicio 5 45
Ejercicios de Gauss-Seidel **51**
Pseudocódigo 51
Ejercicio 1 56
Ejercicio 2 56
Ejercicio 3 60
Ejercicio 4 64
Ejercicio 5 68
Conclusiones **73**
Bibliografía **74**
Extras **75**
Distribución del trabajo 75

Introducción

En el presente trabajo se desarrolla un problemario de métodos numéricos en Java, enfocado en la resolución de sistemas de ecuaciones lineales mediante cuatro técnicas fundamentales: Jacobi, Gauss-Seidel, Eliminación Gaussiana y Gauss-Jordan. Estos métodos, ampliamente utilizados en el análisis numérico, permiten obtener soluciones exactas o aproximadas dependiendo del enfoque aplicado. A lo largo del problemario, se presentan ejercicios implementados en Java que muestran el funcionamiento de cada método. Además, se analiza el comportamiento de cada algoritmo en distintos escenarios, permitiendo una mejor comprensión de su eficiencia.

Ejercicios de Gauss Jordan

Código general:

```
import java.util.Random;
import java.util.Scanner;

/**
 * Aplicación que implementa el algoritmo de eliminación de Gauss-Jordan.
 * La matriz se genera automáticamente con valores aleatorios en función del
tamaño ingresado por el usuario.
 *
 * @author Diego Alonso Coronel Vargas
 * @version 4.0
 * @since 2025-03-13
 */
public class GaussJordanDiego {
    public static void main(String[] args) {
        Scanner read = new Scanner(System.in);

        System.out.print("Número de filas: ");
        int rows = read.nextInt();
        System.out.print("Número de columnas: ");
        int cols = read.nextInt();

        double[][] matrix = new double[rows][cols];

        llenarMatriz(matrix, rows, cols);
        System.out.println("Matriz original:");
        imprimirMatriz(matrix);

        gaussJordan(matrix);

        System.out.println("\nMatriz después de aplicar Gauss-Jordan:");
        imprimirMatriz(matrix);
    }

    /**
     * Llena una matriz con valores aleatorios entre 0 y 20.
     *
     * @param matrix Matriz a llenar.
     * @param rows   Número de filas.
     * @param cols   Número de columnas.
     */
    private static void llenarMatriz(double[][] matrix, int rows, int cols) {
        Random random = new Random();

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                matrix[i][j] = random.nextInt(21); // Valores entre 0 y 20
```

```
            }
        }
    }

    /**
     * Imprime una matriz en formato legible.
     *
     * @param matrix Matriz a imprimir.
     */
    private static void imprimirMatriz(double[][] matrix) {
        int rows = matrix.length;
        int cols = matrix[0].length;

        for (int i = 0; i < rows; i++) {
            System.out.println();
            for (int j = 0; j < cols; j++) {
                System.out.printf("[%6.2f] ", matrix[i][j]);
            }
        }
        System.out.println();
    }

    /**
     * Aplica el método de eliminación de Gauss-Jordan a la matriz.
     *
     * @param matrix Matriz a transformar en su forma escalonada reducida.
     */
    private static void gaussJordan(double[][] matrix) {
        int rows = matrix.length;
        int cols = matrix[0].length;

        for (int pivotRow = 0; pivotRow < rows; pivotRow++) {
            double pivot = matrix[pivotRow][pivotRow];

            // Si el pivote es 0, intercambiar filas
            if (pivot == 0) {
                boolean found = false;
                for (int i = pivotRow + 1; i < rows; i++) {
                    if (matrix[i][pivotRow] != 0) {
                        double[] temp = matrix[pivotRow];
                        matrix[pivotRow] = matrix[i];
                        matrix[i] = temp;
                        pivot = matrix[pivotRow][pivotRow];
                        found = true;
                        break;
                    }
                }
                if (!found) continue; // Si no se encuentra un pivote válido,
saltar
            }
```

```
            // Normalizar la fila del pivote
            for (int j = pivotRow; j < cols; j++) {
                matrix[pivotRow][j] /= pivot;
            }

            // Convertir en ceros las demás filas en la columna del pivote
            for (int i = 0; i < rows; i++) {
                if (i == pivotRow) continue; // Saltar la fila del pivote

                double factor = matrix[i][pivotRow]; // Factor de eliminación
                for (int j = pivotRow; j < cols; j++) {
                    matrix[i][j] -= factor * matrix[pivotRow][j];
                }
            }
        }
    }
}
```

Pseudocódigo:

```
Inicio

    Definir función principal
        Leer número de filas (rows)
        Leer número de columnas (cols)

        Crear matriz (matrix) de tamaño [rows][cols]

        LlenarMatriz(matrix, rows, cols)
        Imprimir "Matriz original:"
        ImprimirMatriz(matrix)

        GaussJordan(matrix)

        Imprimir "Matriz después de aplicar Gauss-Jordan:"
        ImprimirMatriz(matrix)
    Fin función principal

    Función LlenarMatriz(matrix, rows, cols)
        Para i desde 0 hasta rows - 1 hacer
            Para j desde 0 hasta cols - 1 hacer
                matrix[i][j] = Generar número aleatorio entre 0 y 20
            Fin Para
        Fin Para
    Fin Función LlenarMatriz

    Función ImprimirMatriz(matrix)
        Definir rows como longitud de matrix
        Definir cols como longitud de matrix[0]

        Para i desde 0 hasta rows - 1 hacer
            Imprimir nueva línea
            Para j desde 0 hasta cols - 1 hacer
                Imprimir "[" + Formatear(matrix[i][j]) + "] "
            Fin Para
        Fin Para
        Imprimir nueva línea
    Fin Función ImprimirMatriz

    Función GaussJordan(matrix)
        Definir rows como longitud de matrix
        Definir cols como longitud de matrix[0]

        Para pivotRow desde 0 hasta rows - 1 hacer
            Definir pivot como matrix[pivotRow][pivotRow]

            Si pivot es 0 entonces
                Definir found como falso
```

```
                Para i desde pivotRow + 1 hasta rows - 1 hacer
                    Si matrix[i][pivotRow] ≠ 0 entonces
                        Intercambiar matrix[pivotRow] con matrix[i]
                        pivot = matrix[pivotRow][pivotRow]
                        found = verdadero
                        Salir del bucle
                    Fin Si
                Fin Para
                Si !found entonces
                    Continuar con la siguiente iteración
                Fin Si
            Fin Si

            Para j desde pivotRow hasta cols - 1 hacer
                matrix[pivotRow][j] = matrix[pivotRow][j] / pivot
            Fin Para

            Para i desde 0 hasta rows - 1 hacer
                Si i = pivotRow entonces
                    Continuar
                Fin Si

                Definir factor como matrix[i][pivotRow]
                Para j desde pivotRow hasta cols - 1 hacer
                    matrix[i][j] = matrix[i][j] - factor * matrix[pivotRow][j]
                Fin Para
            Fin Para
        Fin Para
    Fin Función GaussJordan

Fin
```

Ejercicio 1

Método de Gauss Jordan empezando por la esquina superior izquierda.

Matriz:

```
[  4.00] [ 12.00] [ 15.00] [ 16.00]
[  4.00] [ 13.00] [  2.00] [  7.00]
[  1.00] [  7.00] [ 17.00] [  5.00]
```

Ejecución:

```
User program running
N?mero de filas:
3

N?mero de columnas:
4

User program finished
Matriz original:

[  4.00] [ 12.00] [ 15.00] [ 16.00]
[  4.00] [ 13.00] [  2.00] [  7.00]
[  1.00] [  7.00] [ 17.00] [  5.00]

Matriz despu?s de aplicar Gauss-Jordan:

[  1.00] [  0.00] [  0.00] [  6.76]
[  0.00] [  1.00] [  0.00] [ -1.63]
[  0.00] [  0.00] [  1.00] [  0.57]
```

Resultado:
x: 6.76, y: -1.63, z: 0.57

Ejercicio 2

Método de Gauss Jordan empezando por la esquina superior izquierda.

Matriz:

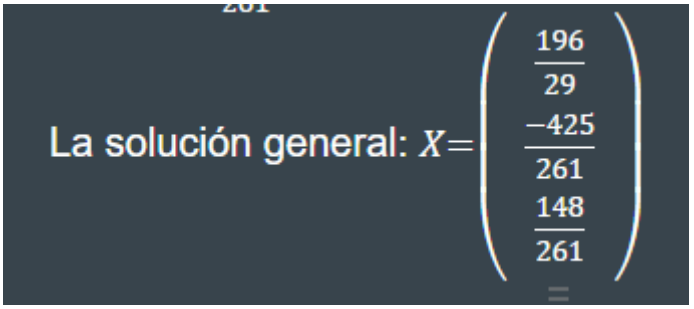
```
[  5.00] [  2.00] [  4.00] [ 16.00]
[  7.00] [ 16.00] [ 17.00] [  5.00]
[ 10.00] [  9.00] [  0.00] [ 16.00]
```

Ejecución:

```
N?mero de filas:
3

N?mero de columnas:
4

User program finished
Matriz original:

[  5.00] [  2.00] [  4.00] [ 16.00]
[  7.00] [ 16.00] [ 17.00] [  5.00]
[ 10.00] [  9.00] [  0.00] [ 16.00]

Matriz despu?s de aplicar Gauss-Jordan:

[  1.00] [  0.00] [  0.00] [  3.38]
[  0.00] [  1.00] [  0.00] [ -1.98]
[  0.00] [  0.00] [  1.00] [  0.76]
```

Resultado:
x: 3.38, y: -1.98, z: 0.76

Solución con Matrixcalc:



Ejercicio 3

Método de Gauss Jordan empezando por la esquina inferior derecha.

Código modificado:

```
private static void gaussJordan(double[][] matrix) {
        int rows = matrix.length;
        int cols = matrix[0].length;

        // Recorremos las filas de abajo hacia arriba para la eliminación
inversa
        for (int pivotRow = rows - 1; pivotRow >= 0; pivotRow--) {
            int pivotCol = pivotRow; // Usamos la diagonal principal
(coeficiente)
            double pivot = matrix[pivotRow][pivotCol];

            // Si el pivote es 0, buscamos intercambiar con alguna fila superior
que tenga un pivote no nulo
            if (pivot == 0) {
                boolean found = false;
                for (int i = pivotRow - 1; i >= 0; i--) {
                    if (matrix[i][pivotCol] != 0) {
                        double[] temp = matrix[pivotRow];
                        matrix[pivotRow] = matrix[i];
                        matrix[i] = temp;
                        pivot = matrix[pivotRow][pivotCol];
                        found = true;
                        break;
                    }
                }
                if (!found) continue; // Si no se encuentra, se salta esta fila
            }

            // Normalizamos la fila del pivote recorriendo de derecha a
izquierda
            for (int j = cols - 1; j >= 0; j--) {
                matrix[pivotRow][j] /= pivot;
            }

            // Eliminamos el elemento en la columna pivote en todas las demás
filas
            for (int i = 0; i < rows; i++) {
                if (i == pivotRow) continue;
                double factor = matrix[i][pivotCol];
                for (int j = cols - 1; j >= 0; j--) {
                    matrix[i][j] -= factor * matrix[pivotRow][j];
                }
```

```
            }
        }
    }
```

Matriz:

```
[ 18.00] [ 13.00] [  4.00] [ 19.00]
[ 17.00] [  8.00] [  1.00] [ 16.00]
[ 15.00] [ 16.00] [  5.00] [  7.00]
```

Ejecución:

```
User program running
N?mero de filas:
3

4

N?mero de columnas: Matriz original:

[ 18.00] [ 13.00] [  4.00] [ 19.00]
[ 17.00] [  8.00] [  1.00] [ 16.00]
[ 15.00] [ 16.00] [  5.00] [  7.00]

Matriz despu?s de aplicar Gauss-Jordan:

[  1.00] [  0.00] [  0.00] [  2.36]
[  0.00] [  1.00] [  0.00] [ -3.85]
[  0.00] [  0.00] [  1.00] [  6.62]
User program finished
```

Resultado:
x: 2.36, y: -3.85, z: 6.62

Ejercicio 4

Método de Gauss Jordan empezando por la esquina inferior derecha.

Matriz:

```
[  2.00] [  4.00] [  8.00] [  4.00]
[ 20.00] [  2.00] [  5.00] [  3.00]
[ 18.00] [ 16.00] [ 12.00] [ 13.00]
```

Ejecución:

```
User program running
N?mero de filas:
3

N?mero de columnas:
4

Matriz original:

[  2.00] [  4.00] [  8.00] [  4.00]
[ 20.00] [  2.00] [  5.00] [  3.00]
[ 18.00] [ 16.00] [ 12.00] [ 13.00]

Matriz despu?s de aplicar Gauss-Jordan:

[  1.00] [ -0.00] [ -0.00] [  0.04]
[  0.00] [  1.00] [ -0.00] [  0.63]
[  0.00] [  0.00] [  1.00] [  0.17]
User program finished
```

Resultado:
x: 0.4, y: 0.63, z: 0.17

Ejercicio 5

Método de Gauss Jordan en un caso en donde falla.

Código modificado:

```
public static void main(String[] args)
    {
        double[][] matrix = {
            {1, 2, 3, 4},
            {5, 6, 7, 8},
            {0, 0, 0, 0} // Esta fila de ceros causará un fallo en el método
        };

        System.out.println("Matriz original:");
        imprimirMatriz(matrix);

        gaussJordan(matrix);

        System.out.println("\nMatriz después de aplicar Gauss Jordan:");
        imprimirMatriz(matrix);
    }
```

Matriz:

```
[  1.00] [  2.00] [  3.00] [  4.00]
[  5.00] [  6.00] [  7.00] [  8.00]
[  0.00] [  0.00] [  0.00] [  0.00]
```

Ejecución:

```
User program running
User program finished
Matriz original:

[  1.00] [  2.00] [  3.00] [  4.00]
[  5.00] [  6.00] [  7.00] [  8.00]
[  0.00] [  0.00] [  0.00] [  0.00]

Matriz despu◆s de aplicar Gauss Jordan:

[  0.00] [  0.00] [  0.00] [  0.00]
[  2.00] [  1.00] [ -0.00] [ -1.00]
[ -1.00] [  0.00] [  1.00] [  2.00]
```

Resultado:
x: indefinido, y: indefinido, z: indefinido

Ejercicio de eliminación gaussiana con pivoteo

Código general:

```java
import java.util.Random;
import java.util.Scanner;

/**
 * Aplicación que implementa el algoritmo de eliminación de Gauss-Jordan.
 * La matriz se genera automáticamente con valores aleatorios en función del
tamaño ingresado por el usuario.
 *
 * @author Diego Alonso Coronel Vargas
 * @version 4.0
 * @since 2025-03-13
 */
public class debug_code {
    public static void main(String[] args) {
        Scanner read = new Scanner(System.in);

        System.out.print("Número de filas: ");
        int rows = read.nextInt();
        System.out.print("Número de columnas: ");
        int cols = read.nextInt();

        double[][] matrix = new double[rows][cols];

        llenarMatriz(matrix, rows, cols);
        System.out.println("Matriz original:");
        imprimirMatriz(matrix);

        gaussPivote(matrix);

        System.out.println("\nMatriz después de aplicar Eliminación Gaussiana
por Pivote:");
        imprimirMatriz(matrix);
    }

    /**
     * Llena una matriz con valores aleatorios entre 0 y 20.
     *
     * @param matrix Matriz a llenar.
     * @param rows   Número de filas.
     * @param cols   Número de columnas.
     */
    private static void llenarMatriz(double[][] matrix, int rows, int cols) {
        Random random = new Random();

        for (int i = 0; i < rows; i++) {
```

```
            for (int j = 0; j < cols; j++) {
                matrix[i][j] = random.nextInt(21); // Valores entre 0 y 20
            }
        }
    }

    /**
     * Imprime una matriz en formato legible.
     *
     * @param matrix Matriz a imprimir.
     */
    private static void imprimirMatriz(double[][] matrix) {
        int rows = matrix.length;
        int cols = matrix[0].length;

        for (int i = 0; i < rows; i++) {
            System.out.println();
            for (int j = 0; j < cols; j++) {
                System.out.printf("[%6.2f] ", matrix[i][j]);
            }
        }
        System.out.println();
    }

    /**
     * Aplica el método de eliminación de Gauss con pivoteo a matrices
rectangulares.
     *
     * @param matrix Matriz a transformar en su forma escalonada.
     */
    public static void gaussPivote(double[][] matrix) {
        int rows = matrix.length;
        int cols = matrix[0].length;
        int min = Math.min(rows, cols); // Número de pivotes a procesar

        for (int pivotRow = 0; pivotRow < min; pivotRow++) {
            double pivot = matrix[pivotRow][pivotRow];

            // Si el pivote es 0, buscamos intercambiar con alguna fila inferior
que tenga un pivote no nulo
            if (pivot == 0) {
                boolean swapped = false;
                for (int i = pivotRow + 1; i < rows; i++) {
                    if (matrix[i][pivotRow] != 0) {
                        double[] temp = matrix[pivotRow];
                        matrix[pivotRow] = matrix[i];
                        matrix[i] = temp;
                        pivot = matrix[pivotRow][pivotRow];
                        swapped = true;
                        break;
                    }
```

```
                }
                if (!swapped) {
                    // Si no se encuentra un pivote no nulo, se continúa con la
siguiente iteración
                    continue;
                }
            }

            // Normalizamos la fila del pivote (se divide cada elemento desde la
columna del pivote hasta el final)
            for (int j = pivotRow; j < cols; j++) {
                matrix[pivotRow][j] /= pivot;
            }

            // Convertimos en cero los elementos de la columna pivote en las
filas inferiores
            for (int i = pivotRow + 1; i < rows; i++) {
                double factor = matrix[i][pivotRow];
                for (int j = pivotRow; j < cols; j++) {
                    matrix[i][j] -= factor * matrix[pivotRow][j];
                }
            }
        }
    }
}
```

Pseudocódigo:

```
Inicio

    Definir función principal
        Leer número de filas (rows)
        Leer número de columnas (cols)

        Crear matriz (matrix) de tamaño [rows][cols]

        LlenarMatriz(matrix, rows, cols)
        Imprimir "Matriz original:"
        ImprimirMatriz(matrix)

        GaussPivote(matrix)

        Imprimir "Matriz después de aplicar Eliminación Gaussiana por Pivote:"
        ImprimirMatriz(matrix)
    Fin función principal

    Función LlenarMatriz(matrix, rows, cols)
        Definir random como nuevo generador de números aleatorios

        Para i desde 0 hasta rows - 1 hacer
            Para j desde 0 hasta cols - 1 hacer
                matrix[i][j] = Generar número aleatorio entre 0 y 20
            Fin Para
        Fin Para
    Fin Función LlenarMatriz

    Función ImprimirMatriz(matrix)
        Definir rows como longitud de matrix
        Definir cols como longitud de matrix[0]

        Para i desde 0 hasta rows - 1 hacer
            Imprimir nueva línea
            Para j desde 0 hasta cols - 1 hacer
                Imprimir "[" + Formatear(matrix[i][j]) + "] "
            Fin Para
        Fin Para
        Imprimir nueva línea
    Fin Función ImprimirMatriz

    Función GaussPivote(matrix)
        Definir rows como longitud de matrix
        Definir cols como longitud de matrix[0]
        Definir min como mínimo entre rows y cols

        Para pivotRow desde 0 hasta min - 1 hacer
            Definir pivot como matrix[pivotRow][pivotRow]
```

```
            Si pivot es 0 entonces
                Definir swapped como falso
                Para i desde pivotRow + 1 hasta rows - 1 hacer
                    Si matrix[i][pivotRow] ≠ 0 entonces
                        Intercambiar matrix[pivotRow] con matrix[i]
                        pivot = matrix[pivotRow][pivotRow]
                        swapped = verdadero
                        Salir del bucle
                    Fin Si
                Fin Para
                Si !swapped entonces
                    Continuar con la siguiente iteración
                Fin Si
            Fin Si

            Para j desde pivotRow hasta cols - 1 hacer
                matrix[pivotRow][j] = matrix[pivotRow][j] / pivot
            Fin Para

            Para i desde pivotRow + 1 hasta rows - 1 hacer
                Definir factor como matrix[i][pivotRow]
                Para j desde pivotRow hasta cols - 1 hacer
                    matrix[i][j] = matrix[i][j] - factor * matrix[pivotRow][j]
                Fin Para
            Fin Para
        Fin Para
    Fin Función GaussPivote

Fin
```

Ejercicio 1

Eliminación gaussiana con pivoteo empezando por la esquina superior izquierda.

Matriz:

```
[ 14.00] [  1.00] [ 18.00] [  8.00]
[ 15.00] [ 17.00] [ 20.00] [  7.00]
[  0.00] [  8.00] [  4.00] [ 16.00]
```

Ejecución:

```
User program running
N?mero de filas:
3

N?mero de columnas:
4

Matriz original:

[ 14.00] [  1.00] [ 18.00] [  8.00]
[ 15.00] [ 17.00] [ 20.00] [  7.00]
[  0.00] [  8.00] [  4.00] [ 16.00]

Matriz despu?s de aplicar Gauss-Jordan:

[  1.00] [  0.07] [  1.29] [  0.57]
[  0.00] [  1.00] [  0.04] [ -0.10]
[  0.00] [  0.00] [  1.00] [  4.61]
User program finished
```

Resultado:
x: por calcular, y: por calcular, z: 4.61

Ejercicio 2

Eliminación gaussiana con pivoteo parcial.

Código modificado:

```
public static void gaussPivoteParcial(double[][] matrix) {
        int rows = matrix.length;
        int cols = matrix[0].length;
        int min = Math.min(rows, cols); // Número de pivotes a procesar

        for (int pivotRow = 0; pivotRow < min; pivotRow++) {
            // Búsqueda del pivote parcial: seleccionar la fila con el mayor
valor absoluto en la columna 'pivotRow'
            int maxRow = pivotRow;
            for (int i = pivotRow + 1; i < rows; i++) {
                if (Math.abs(matrix[i][pivotRow]) >
Math.abs(matrix[maxRow][pivotRow])) {
                    maxRow = i;
                }
            }

            // Intercambiamos la fila actual con la fila que tiene el mayor
pivote
            if (maxRow != pivotRow) {
                double[] temp = matrix[pivotRow];
                matrix[pivotRow] = matrix[maxRow];
                matrix[maxRow] = temp;
            }

            double pivot = matrix[pivotRow][pivotRow];

            // Si el pivote es 0, la columna es nula y se omite este pivote
            if (pivot == 0) {
                continue;
            }

            // Normalizamos la fila del pivote (se divide cada elemento desde la
columna del pivote hasta el final)
            for (int j = pivotRow; j < cols; j++) {
                matrix[pivotRow][j] /= pivot;
            }

            // Eliminamos el valor en la columna del pivote en las filas
inferiores
            for (int i = pivotRow + 1; i < rows; i++) {
                double factor = matrix[i][pivotRow];
                for (int j = pivotRow; j < cols; j++) {
```

```
                    matrix[i][j] -= factor * matrix[pivotRow][j];
                }
            }
        }
    }
```

Matriz:

```
[  7.00] [  7.00] [  5.00] [ 17.00]
[ 14.00] [ 15.00] [  7.00] [  6.00]
[  9.00] [  1.00] [  0.00] [  4.00]
```

Ejecución:

```
User program running
N◆mero de filas:
3

N◆mero de columnas:
4

User program finished
Matriz original:

[  7.00] [  7.00] [  5.00] [ 17.00]
[ 14.00] [ 15.00] [  7.00] [  6.00]
[  9.00] [  1.00] [  0.00] [  4.00]

Matriz despu◆s de aplicar Eliminaci◆n Gaussiana por Pivote:

[  1.00] [  1.07] [  0.50] [  0.43]
[  0.00] [  1.00] [  0.52] [ -0.02]
[  0.00] [  0.00] [  1.00] [  7.95]
```

Resultado:
x: por calcular, y: por calcular, z: 7.95

Ejercicio 3

Eliminación gaussiana con pivoteo total.

Código modificado:

```
public static void gaussPivote(double[][] matrix) {
        int size = matrix.length;

        for (int pivotRow = 0; pivotRow < size; pivotRow++) {
            int maxRow = pivotRow, maxCol = pivotRow;
            double maxValue = Math.abs(matrix[pivotRow][pivotRow]);

            // Buscar el elemento de mayor valor absoluto en la submatriz
restante
            for (int i = pivotRow; i < size; i++) {
                for (int j = pivotRow; j < size; j++) {
                    if (Math.abs(matrix[i][j]) > maxValue) {
                        maxValue = Math.abs(matrix[i][j]);
                        maxRow = i;
                        maxCol = j;
                    }
                }
            }

            // Intercambio de filas si es necesario
            if (maxRow != pivotRow) {
                double[] temp = matrix[pivotRow];
                matrix[pivotRow] = matrix[maxRow];
                matrix[maxRow] = temp;
            }

            // Intercambio de columnas si es necesario
            if (maxCol != pivotRow) {
                for (int i = 0; i < size; i++) {
                    double temp = matrix[i][pivotRow];
                    matrix[i][pivotRow] = matrix[i][maxCol];
                    matrix[i][maxCol] = temp;
                }
            }

            // Normalización de la fila del pivote
            double pivot = matrix[pivotRow][pivotRow];
            for (int j = 0; j < size; j++) {
                matrix[pivotRow][j] /= pivot;
            }

            // Eliminación en las filas inferiores
```

```
            for (int i = pivotRow + 1; i < size; i++) {
                double factor = matrix[i][pivotRow];
                for (int j = 0; j < size; j++) {
                    matrix[i][j] -= factor * matrix[pivotRow][j];
                }
            }
        }
    }
```

Matriz:

```
[  4.00] [  2.00] [ 19.00] [ 15.00]
[ 10.00] [ 14.00] [ 13.00] [ 11.00]
[  3.00] [ 17.00] [ 14.00] [ 19.00]
```

Ejecución:

```
User program running
N◆mero de filas:
3

N◆mero de columnas:
4

Matriz original:

[  4.00] [  2.00] [ 19.00] [ 15.00]
[ 10.00] [ 14.00] [ 13.00] [ 11.00]
[  3.00] [ 17.00] [ 14.00] [ 19.00]

Matriz despu◆s de aplicar Eliminaci◆n Gaussiana por Pivote:

[  1.00] [  0.11] [  0.21] [  0.79]
[  0.00] [  1.00] [  0.00] [  0.51]
[  0.00] [  0.00] [  1.00] [ -0.79]
User program finished
```

Resultado:

x: por calcular, y: por calcular, z: -0.79

Ejercicio 4

Eliminación gaussiana con pivoteo escalado.

Código modificado:

```
public static void gaussPivote(double[][] matrix) {
        int size = matrix.length;
        double[] escala = new double[size];

        // Calcular los factores de escala (máximo valor absoluto en cada fila)
        for (int i = 0; i < size; i++) {
            double maxValor = 0;
            for (int j = 0; j < size; j++) {
                maxValor = Math.max(maxValor, Math.abs(matrix[i][j]));
            }
            escala[i] = maxValor;
        }

        for (int pivotRow = 0; pivotRow < size; pivotRow++) {
            // Encontrar la mejor fila para el pivote considerando la escala
            int maxRow = pivotRow;
            double maxRatio = Math.abs(matrix[pivotRow][pivotRow]) /
escala[pivotRow];

            for (int i = pivotRow + 1; i < size; i++) {
                double ratio = Math.abs(matrix[i][pivotRow]) / escala[i];
                if (ratio > maxRatio) {
                    maxRatio = ratio;
                    maxRow = i;
                }
            }

            // Intercambiar filas si es necesario
            if (maxRow != pivotRow) {
                double[] temp = matrix[pivotRow];
                matrix[pivotRow] = matrix[maxRow];
                matrix[maxRow] = temp;

                double tempEscala = escala[pivotRow];
                escala[pivotRow] = escala[maxRow];
                escala[maxRow] = tempEscala;
            }

            // Normalizar la fila del pivote
            double pivot = matrix[pivotRow][pivotRow];
            for (int j = 0; j < size; j++) {
                matrix[pivotRow][j] /= pivot;
```

```
            }

            // Eliminación en las filas inferiores
            for (int i = pivotRow + 1; i < size; i++) {
                double factor = matrix[i][pivotRow];
                for (int j = 0; j < size; j++) {
                    matrix[i][j] -= factor * matrix[pivotRow][j];
                }
            }
        }
    }
```

Matriz:

```
[ 11.00] [ 12.00] [  4.00] [ 17.00]
[  9.00] [ 19.00] [  1.00] [  9.00]
[ 13.00] [ 15.00] [ 19.00] [  2.00]
```

Ejecución:

```
User program running
N◆mero de filas:
3

N◆mero de columnas:
4

Matriz original:

[ 11.00] [ 12.00] [  4.00] [ 17.00]
[  9.00] [ 19.00] [  1.00] [  9.00]
[ 13.00] [ 15.00] [ 19.00] [  2.00]

Matriz despu◆s de aplicar Eliminaci◆n Gaussiana por Pivote:

[  1.00] [  1.15] [  1.46] [  0.15]
[  0.00] [  1.00] [ -1.41] [  0.88]
[  0.00] [  0.00] [  1.00] [ -1.22]
User program finished
```

Resultado:

x: por calcular, y: por calcular, z: -1.22

Ejercicio 5

Eliminación gaussiana con pivoteo en un caso en donde falla.

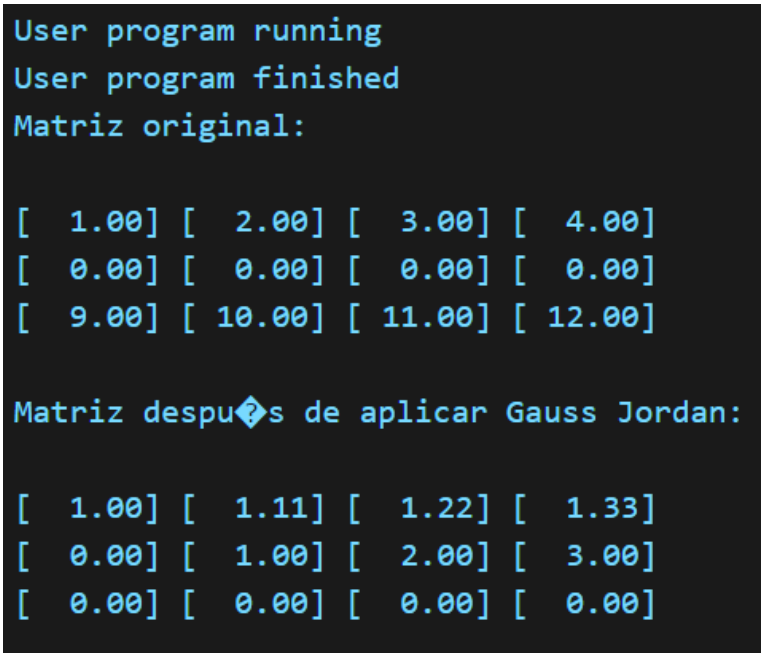
Código modificado:

```
public static void main(String[] args)
    {
        double[][] matrix = {
            {1, 2, 3, 4},
            {0, 0, 0, 0},  // Esta fila de ceros causará un fallo en el método
            {9, 10, 11, 12}
        };

        System.out.println("Matriz original:");
        imprimirMatriz(matrix);

        gaussPivoteEscalado(matrix);

        System.out.println("\nMatriz después de aplicar Gauss Jordan:");
        imprimirMatriz(matrix);
    }
```

Matriz:

```
[  1.00] [  2.00] [  3.00] [  4.00]
[  0.00] [  0.00] [  0.00] [  0.00]
[  9.00] [ 10.00] [ 11.00] [ 12.00]
```

Ejecución:



Resultado:
x: indefinido, y: indefinido, z: indefinido

Ejercicios de Jacobi

Pseudocódigo

```
INICIO

    // Leer el tamaño de la matriz
    ESCRIBIR "Introduce el tamaño de la matriz (N): "
    LEER N

    // Declarar la matriz A y el vector b
    DECLARAR A[N][N]
    DECLARAR b[N]

    // Llenar la matriz A y el vector b con valores aleatorios asegurando
diagonal dominante
    PROCEDIMIENTO llenarMatricesDiagonalDominante(A, b, N)
        PARA i DESDE 0 HASTA N-1 HACER
            b[i] ← número aleatorio entre 0 y 10
            sumaFila ← 0
            PARA j DESDE 0 HASTA N-1 HACER
                A[i][j] ← número aleatorio entre 0 y 10
                sumaFila ← sumaFila + ABS(A[i][j])
            FIN PARA
            A[i][i] ← sumaFila + número aleatorio entre 0 y 10 // Asegurar
diagonal dominante
        FIN PARA
    FIN PROCEDIMIENTO

    // Imprimir la matriz A
    PROCEDIMIENTO imprimirMatriz(A)
        PARA i DESDE 0 HASTA N-1 HACER
            PARA j DESDE 0 HASTA N-1 HACER
                IMPRIMIR A[i][j] CON DOS DECIMALES
            FIN PARA
            IMPRIMIR NUEVA LÍNEA
        FIN PARA
    FIN PROCEDIMIENTO

    // Imprimir el vector b
    PROCEDIMIENTO imprimirVector(b)
        PARA i DESDE 0 HASTA N-1 HACER
            IMPRIMIR b[i] CON DOS DECIMALES
        FIN PARA
        IMPRIMIR NUEVA LÍNEA
    FIN PROCEDIMIENTO

    // Llamar al método de Jacobi
    FUNCION jacobiEstandar(A, b, tol, maxIter) DEVUELVE VECTOR
        DECLARAR x[N] ← 0 // Inicializar vector solución en ceros
```

```
        DECLARAR xNuevo[N]
        iteraciones ← 0

        PARA iter DESDE 0 HASTA maxIter-1 HACER
            PARA i DESDE 0 HASTA N-1 HACER
                suma ← 0
                PARA j DESDE 0 HASTA N-1 HACER
                    SI i ≠ j ENTONCES
                        suma ← suma + (A[i][j] * x[j])
                    FIN SI
                FIN PARA
                xNuevo[i] ← (b[i] - suma) / A[i][i]
            FIN PARA

            // Verificar convergencia
            SI convergio(x, xNuevo, tol) ENTONCES
                iteraciones ← iter + 1
                IMPRIMIR "Convergió en ", iteraciones, " iteraciones."
                RETORNAR xNuevo
            FIN SI

            // Copiar valores de xNuevo a x
            PARA i DESDE 0 HASTA N-1 HACER
                x[i] ← xNuevo[i]
            FIN PARA
        FIN PARA

        // Si no converge
        IMPRIMIR "No convergió en el número máximo de iteraciones."
        RETORNAR x
    FIN FUNCIÓN

    // Función para verificar convergencia
    FUNCION convergio(x, xNuevo, tol) DEVUELVE BOOLEANO
        maxError ← 0
        PARA i DESDE 0 HASTA N-1 HACER
            error ← ABS(xNuevo[i] - x[i])
            SI error > maxError ENTONCES
                maxError ← error
            FIN SI
        FIN PARA
        RETORNAR maxError < tol
    FIN FUNCIÓN

    // Ejecutar el método de Jacobi con tolerancia 1e-6 y 100 iteraciones
máximas
    resultado ← jacobiEstandar(A, b, 1e-6, 100)

    // Imprimir el resultado con redondeo a 4 decimales
    PROCEDIMIENTO imprimirVectorRedondeado(b, decimales)
        PARA i DESDE 0 HASTA N-1 HACER
```

```
            IMPRIMIR b[i] REDONDEADO A "decimales" DECIMALES
        FIN PARA
        IMPRIMIR NUEVA LÍNEA
    FIN PROCEDIMIENTO

    IMPRIMIR "Solución:"
    imprimirVectorRedondeado(resultado, 4)

FIN
```

Código general:

```java
import java.util.Scanner;

/**
 * Aplicación que implementa el método iterativo de Jacobi para resolver
 * sistemas de ecuaciones lineales. La matriz de coeficientes se genera
 * automáticamente con valores aleatorios garantizando que sea diagonalmente
dominante.
 *
 * @author Brandon García Ordaz
 * @version 1.0
 * @since 2025-03-16
 */
public class JacobiBrandon {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Solicitar tamaño N
        System.out.print("Introduce el tamaño de la matriz (N): ");
        int N = scanner.nextInt(); // Obtener el tamaño de la matriz

        // Inicializar la matriz A y el vector b
        double[][] A = new double[N][N];
        double[] b = new double[N];

        // Llenar las matrices A y el vector b con valores aleatorios,
asegurando la diagonal dominante
        llenarMatricesDiagonalDominante(A, b, N);

        // Mostrar la matriz A y el vector b
        System.out.println("Matriz A generada automáticamente:");
        imprimirMatriz(A);
        System.out.println("\nVector b generado automáticamente:");
        imprimirVector(b);
        System.out.println();

        // Resolver el sistema de ecuaciones usando el método de Jacobi
        double[] resultado = jacobiEstandar(A, b, 1e-6, 100);

        // Mostrar el resultado redondeado
        System.out.println("Solución:");
        imprimirVectorRedondeado(resultado, 4); // Redondear los resultados a 4
decimales

        scanner.close();
    }

    /**
     * Llena la matriz A y el vector b con valores aleatorios, asegurando que A
```

```
sea diagonalmente dominante.
     *
     * @param A Matriz de coeficientes.
     * @param b Vector de términos independientes.
     * @param N Tamaño de la matriz (N x N).
     */
    public static void llenarMatricesDiagonalDominante(double[][] A, double[] b,
int N) {
        for (int i = 0; i < N; i++) {
            b[i] = Math.random() * 10; // Llenar b con números aleatorios
            double sumaFila = 0;
            for (int j = 0; j < N; j++) {
                A[i][j] = Math.random() * 10; // Llenar A con números aleatorios
                sumaFila += Math.abs(A[i][j]);
            }
            A[i][i] = sumaFila + Math.random() * 10; // Asegurar que la diagonal
sea dominante
        }
    }

    /**
     * Imprime la matriz A en formato legible.
     *
     * @param A Matriz de coeficientes.
     */
    public static void imprimirMatriz(double[][] A) {
        for (double[] fila : A) {
            for (double valor : fila) {
                System.out.print("[" + String.format("%.2f", valor) + "] ");
            }
            System.out.println();
        }
    }

    /**
     * Imprime un vector en formato legible.
     *
     * @param b Vector a imprimir.
     */
    public static void imprimirVector(double[] b) {
        for (double valor : b) {
            System.out.print("[" + String.format("%.2f", valor) + "] ");
        }
        System.out.println();
    }

    /**
     * Imprime un vector con valores redondeados a una cantidad específica de
decimales.
     *
     * @param b Vector a imprimir.
```

```
     * @param decimales Número de decimales a mostrar.
     */
    public static void imprimirVectorRedondeado(double[] b, int decimales) {
        for (double valor : b) {
            System.out.print("[" + String.format("%." + decimales + "f", valor)
+ "] ");
        }
        System.out.println();
    }

    /**
     * Implementación estándar del método iterativo de Jacobi.
     *
     * @param A Matriz de coeficientes.
     * @param b Vector de términos independientes.
     * @param tol Tolerancia para la convergencia.
     * @param maxIter Número máximo de iteraciones.
     * @return Vector solución del sistema de ecuaciones.
     */
    public static double[] jacobiEstandar(double[][] A, double[] b, double tol,
int maxIter) {
        int n = A.length;
        double[] x = new double[n]; // Vector solución inicializado en 0
        double[] xNuevo = new double[n]; // Vector para la nueva solución
        int iteraciones = 0; // Contador de iteraciones

        // Iterar hasta alcanzar la tolerancia o el número máximo de iteraciones
        for (int iter = 0; iter < maxIter; iter++) {
            for (int i = 0; i < n; i++) {
                double suma = 0;
                for (int j = 0; j < n; j++) {
                    if (i != j) suma += A[i][j] * x[j];
                }
                xNuevo[i] = (b[i] - suma) / A[i][i]; // Calcular el nuevo valor
para x[i]
            }

            // Verificar la convergencia
            if (convergio(x, xNuevo, tol)) {
                iteraciones = iter + 1;
                System.out.println("Convergió en " + iteraciones + "
iteraciones.");
                break;
            }

            // Copiar los valores de xNuevo a x para la siguiente iteración
            System.arraycopy(xNuevo, 0, x, 0, n);
        }

        // Si no ha convergido en el número máximo de iteraciones, mostrar un
mensaje
```

```
        if (iteraciones == 0) {
            System.out.println("No convergió en el número máximo de
iteraciones.");
        }

        return x;
    }

    /**
     * Comprueba si el método de Jacobi ha convergido.
     *
     * @param x Vector de la iteración anterior.
     * @param xNuevo Vector de la iteración actual.
     * @param tol Tolerancia para la convergencia.
     * @return Verdadero si la diferencia es menor que la tolerancia, falso en
caso contrario.
     */
    public static boolean convergio(double[] x, double[] xNuevo, double tol) {
        double maxError = 0;
        for (int i = 0; i < x.length; i++) {
            maxError = Math.max(maxError, Math.abs(xNuevo[i] - x[i]));
        }
        return maxError < tol;
    }
}
```

Ejercicio 1

Método de Jacobi de la forma estándar (izquierda a derecha)

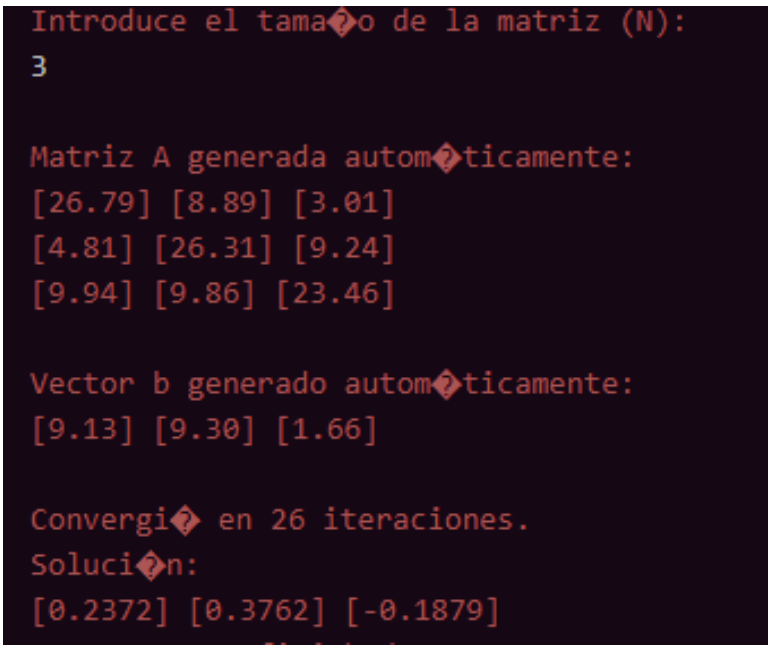
Matriz A:

26.79	8.89	3.01
4.81	26.31	9.24
9.94	9.86	23.46

Vector b:

9.13	9.30	1.66

Ejecución:



Ejercicio 2

Método de Jacobi de abajo hacia arriba

Código modificado:

```
    /**
     * Implementación del método de Jacobi "de abajo hacia arriba".
     *
     * @param A Matriz de coeficientes.
     * @param b Vector de términos independientes.
     * @param tol Tolerancia para la convergencia.
     * @param maxIter Número máximo de iteraciones.
     * @return Vector solución del sistema de ecuaciones.
     */
    public static double[] jacobiAbajoHaciaArriba(double[][] A, double[] b,
double tol, int maxIter) {
        int n = A.length;
```

```
        double[] x = new double[n]; // Vector solución inicializado en 0
        double[] xNuevo = new double[n]; // Vector para la nueva solución
        int iteraciones = 0; // Contador de iteraciones

        // Iterar hasta alcanzar la tolerancia o el número máximo de iteraciones
        for (int iter = 0; iter < maxIter; iter++) {
            for (int i = n - 1; i >= 0; i--) { // Iterar de abajo hacia arriba
                double suma = 0;
                for (int j = 0; j < n; j++) {
                    if (i != j) suma += A[i][j] * xNuevo[j];
                }
                xNuevo[i] = (b[i] - suma) / A[i][i]; // Calcular el nuevo valor
para x[i]
            }

            // Verificar la convergencia
            if (convergio(x, xNuevo, tol)) {
                iteraciones = iter + 1;
                System.out.println("Convergió en " + iteraciones + "
iteraciones.");
                break;
            }

            // Copiar los valores de xNuevo a x para la siguiente iteración
            System.arraycopy(xNuevo, 0, x, 0, n);
        }

        // Si no ha convergido en el número máximo de iteraciones, mostrar un
mensaje
        if (iteraciones == 0) {
            System.out.println("No convergió en el número máximo de
iteraciones.");
        }

        return x;
    }
```

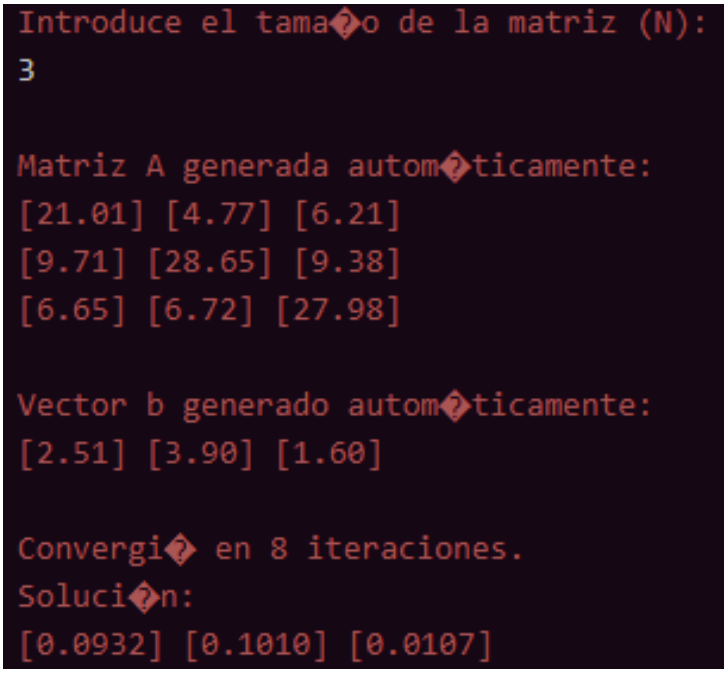
Matriz A:

21.01	4.77	6.21
9.71	28.65	9.38
6.65	6.72	27.98

Vector b:

2.51	3.90	1.60

Ejecución:



Ejercicio 3

Método de Jacobi Intercalado (Pares primero, luego impares)

Código modificado:

```
/**
     * Implementación del método iterativo de Jacobi con el método intercalado
```

```
(pares primero, luego impares).
     *
     * @param A Matriz de coeficientes.
     * @param b Vector de términos independientes.
     * @param tol Tolerancia para la convergencia.
     * @param maxIter Número máximo de iteraciones.
     * @return Vector solución del sistema de ecuaciones.
     */
    public static double[] jacobiIntercalado(double[][] A, double[] b, double
tol, int maxIter) {
        int n = A.length;
        double[] x = new double[n]; // Vector solución inicializado en 0
        double[] xNuevo = new double[n]; // Vector para la nueva solución
        int iteraciones = 0; // Contador de iteraciones

        // Iterar hasta alcanzar la tolerancia o el número máximo de iteraciones
        for (int iter = 0; iter < maxIter; iter++) {
            // Primer paso: Iterar sobre los índices pares
            for (int i = 0; i < n; i += 2) {
                double suma = 0;
                for (int j = 0; j < n; j++) {
                    if (i != j) suma += A[i][j] * x[j];
                }
                xNuevo[i] = (b[i] - suma) / A[i][i]; // Calcular el nuevo valor
para x[i]
            }

            // Segundo paso: Iterar sobre los índices impares
            for (int i = 1; i < n; i += 2) {
                double suma = 0;
                for (int j = 0; j < n; j++) {
                    if (i != j) suma += A[i][j] * x[j];
                }
                xNuevo[i] = (b[i] - suma) / A[i][i]; // Calcular el nuevo valor
para x[i]
            }

            // Verificar la convergencia
            if (convergio(x, xNuevo, tol)) {
                iteraciones = iter + 1;
                System.out.println("Convergió en " + iteraciones + "
iteraciones.");
                break;
            }

            // Copiar los valores de xNuevo a x para la siguiente iteración
            System.arraycopy(xNuevo, 0, x, 0, n);
        }

        // Si no ha convergido en el número máximo de iteraciones, mostrar un
mensaje
```

```
        if (iteraciones == 0) {
            System.out.println("No convergió en el número máximo de
iteraciones.");
        }

        return x;
    }
```

Matriz A:

10.43	1.27	6.46
9.84	19.87	6.70
2.24	7.31	20.69

Vector b:

2.68	7.71	0.35

Ejecución:

```
Introduce el tama?o de la matriz (N):
3

Matriz A generada autom?ticamente:
[10.43] [1.27] [6.46]
[9.84] [19.87] [6.70]
[2.24] [7.31] [20.69]

Vector b generado autom?ticamente:
[2.68] [7.71] [0.35]

Convergi? en 30 iteraciones.
Soluci?n:
[0.2936] [0.2813] [-0.1145]
```

Ejercicio 4

Método de Jacobi por Bloques (dos filas a la vez)

Código modificado:

```
/**
     * Implementación del método iterativo de Jacobi con el método por bloques
(dos filas a la vez).
     *
     * @param A Matriz de coeficientes.
     * @param b Vector de términos independientes.
     * @param tol Tolerancia para la convergencia.
     * @param maxIter Número máximo de iteraciones.
     * @return Vector solución del sistema de ecuaciones.
     */
    public static double[] jacobiPorBloques(double[][] A, double[] b, double
tol, int maxIter) {
        int n = A.length;
        double[] x = new double[n]; // Vector solución inicializado en 0
        double[] xNuevo = new double[n]; // Vector para la nueva solución
        int iteraciones = 0; // Contador de iteraciones

        // Iterar hasta alcanzar la tolerancia o el número máximo de iteraciones
        for (int iter = 0; iter < maxIter; iter++) {
            // Primer paso: Iterar sobre los bloques de dos filas
            for (int i = 0; i < n; i += 2) {
                // Actualizar las dos filas en el bloque
                for (int j = i; j < Math.min(i + 2, n); j++) {
                    double suma = 0;
                    for (int k = 0; k < n; k++) {
                        if (j != k) suma += A[j][k] * x[k];
                    }
                    xNuevo[j] = (b[j] - suma) / A[j][j]; // Calcular el nuevo
valor para x[j]
                }
            }

            // Verificar la convergencia
            if (convergio(x, xNuevo, tol)) {
                iteraciones = iter + 1;
                System.out.println("Convergió en " + iteraciones + "
iteraciones.");
                break;
            }

            // Copiar los valores de xNuevo a x para la siguiente iteración
            System.arraycopy(xNuevo, 0, x, 0, n);
        }
```

```
        // Si no ha convergido en el número máximo de iteraciones, mostrar un
mensaje
        if (iteraciones == 0) {
            System.out.println("No convergió en el número máximo de
iteraciones.");
        }

        return x;
    }
```

Matriz A:

17.04	3.10	9.76
4.52	14.26	3.83
8.48	3.01	28.08

Vector b:

8.52	4.80	4.98

Ejecución:

```
Introduce el tama□o de la matriz (N):
3

User program finished
Matriz A generada autom□ticamente:
[17.04] [3.10] [9.76]
[4.52] [14.26] [3.83]
[8.48] [3.01] [28.08]

Vector b generado autom□ticamente:
[8.52] [4.80] [4.98]

Convergi□ en 24 iteraciones.
Soluci□n:
[0.4545] [0.1873] [0.0202]
```

Ejercicio 5

Método de Jacobi cuando falla

Código modificado:

```java
import java.util.Scanner;

/**
 * Aplicación que implementa el método iterativo de Jacobi para resolver
 * sistemas de ecuaciones lineales. La matriz de coeficientes se genera
 * automáticamente con valores aleatorios garantizando que sea diagonalmente
dominante.
 *
 * @author Brandon García Ordaz
 * @version 1.5
 * @since 2025-03-16
 */
public class JacobiBrandon {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Solicitar tamaño N
        System.out.print("Introduce el tamaño de la matriz (N): ");
        int N = scanner.nextInt(); // Obtener el tamaño de la matriz

        // Inicializar la matriz A y el vector b
        double[][] A = new double[N][N];
        double[] b = new double[N];

        // Llenar las matrices A y el vector b con valores aleatorios
        llenarMatricesDiagonalDominante(A, b, N);
```

```
        // Mostrar la matriz A y el vector b
        System.out.println("Matriz A generada automáticamente:");
        imprimirMatriz(A);
        System.out.println("\nVector b generado automáticamente:");
        imprimirVector(b);
        System.out.println();

        // **Verificar si la matriz es diagonalmente dominante**
        if (!esDiagonalDominante(A)) {
            throw new IllegalArgumentException("ERROR: La matriz no es
diagonalmente dominante. El método de Jacobi podría no converger.");
        }

        // Resolver el sistema de ecuaciones usando el método de Jacobi
        double[] resultado = jacobiEstandar(A, b, 1e-6, 100);

        // Mostrar el resultado redondeado
        System.out.println("Solución:");
        imprimirVectorRedondeado(resultado, 4); // Redondear los resultados a 4
decimales

        scanner.close();
    }

    /**
     * Llena la matriz A y el vector b con valores aleatorios, asegurando que A
sea diagonalmente dominante.
     *
     * @param A Matriz de coeficientes.
     * @param b Vector de términos independientes.
     * @param N Tamaño de la matriz (N x N).
     */
    public static void llenarMatricesDiagonalDominante(double[][] A, double[] b,
int N) {
        for (int i = 0; i < N; i++) {
            b[i] = Math.random() * 10; // Llenar b con números aleatorios
            double sumaFila = 0;
            for (int j = 0; j < N; j++) {
                A[i][j] = Math.random() * 10; // Llenar A con números aleatorios
                if (i != j) {
                    sumaFila += Math.abs(A[i][j]);
                }
            }
            A[i][i] = Math.random() * 5; // **Modificar diagonal para provocar
error a veces**
        }
    }

    /**
     * Verifica si una matriz es diagonalmente dominante.
```

```
     *
     * @param A Matriz a verificar.
     * @return Verdadero si la matriz es diagonalmente dominante, falso en caso
contrario.
     */
    public static boolean esDiagonalDominante(double[][] A) {
        for (int i = 0; i < A.length; i++) {
            double sumaFila = 0;
            for (int j = 0; j < A.length; j++) {
                if (i != j) {
                    sumaFila += Math.abs(A[i][j]);
                }
            }
            if (Math.abs(A[i][i]) < sumaFila) { // La diagonal debe ser mayor o
igual a la suma de los demás elementos de la fila
                return false;
            }
        }
        return true;
    }

    /**
     * Imprime la matriz A en formato legible.
     *
     * @param A Matriz de coeficientes.
     */
    public static void imprimirMatriz(double[][] A) {
        for (double[] fila : A) {
            for (double valor : fila) {
                System.out.print("[" + String.format("%.2f", valor) + "] ");
            }
            System.out.println();
        }
    }

    /**
     * Imprime un vector en formato legible.
     *
     * @param b Vector a imprimir.
     */
    public static void imprimirVector(double[] b) {
        for (double valor : b) {
            System.out.print("[" + String.format("%.2f", valor) + "] ");
        }
        System.out.println();
    }

    /**
     * Imprime un vector con valores redondeados a una cantidad específica de
decimales.
     *
```

```
     * @param b Vector a imprimir.
     * @param decimales Número de decimales a mostrar.
     */
    public static void imprimirVectorRedondeado(double[] b, int decimales) {
        for (double valor : b) {
            System.out.print("[" + String.format("%." + decimales + "f", valor)
+ "] ");
        }
        System.out.println();
    }

    /**
     * Implementación estándar del método iterativo de Jacobi.
     *
     * @param A Matriz de coeficientes.
     * @param b Vector de términos independientes.
     * @param tol Tolerancia para la convergencia.
     * @param maxIter Número máximo de iteraciones.
     * @return Vector solución del sistema de ecuaciones.
     */
    public static double[] jacobiEstandar(double[][] A, double[] b, double tol,
int maxIter) {
        int n = A.length;
        double[] x = new double[n]; // Vector solución inicializado en 0
        double[] xNuevo = new double[n]; // Vector para la nueva solución
        int iteraciones = 0; // Contador de iteraciones

        // Iterar hasta alcanzar la tolerancia o el número máximo de iteraciones
        for (int iter = 0; iter < maxIter; iter++) {
            for (int i = 0; i < n; i++) {
                double suma = 0;
                for (int j = 0; j < n; j++) {
                    if (i != j) suma += A[i][j] * x[j];
                }
                xNuevo[i] = (b[i] - suma) / A[i][i]; // Calcular el nuevo valor
para x[i]
            }

            // Verificar la convergencia
            if (convergio(x, xNuevo, tol)) {
                iteraciones = iter + 1;
                System.out.println("Convergió en " + iteraciones + "
iteraciones.");
                break;
            }

            // Copiar los valores de xNuevo a x para la siguiente iteración
            System.arraycopy(xNuevo, 0, x, 0, n);
        }

        // Si no ha convergido en el número máximo de iteraciones, mostrar un
```

```
mensaje
        if (iteraciones == 0) {
            System.out.println("No convergió en el número máximo de
iteraciones.");
        }

        return x;
    }

    /**
     * Comprueba si el método de Jacobi ha convergido.
     *
     * @param x Vector de la iteración anterior.
     * @param xNuevo Vector de la iteración actual.
     * @param tol Tolerancia para la convergencia.
     * @return Verdadero si la diferencia es menor que la tolerancia, falso en
caso contrario.
     */
    public static boolean convergio(double[] x, double[] xNuevo, double tol) {
        double maxError = 0;
        for (int i = 0; i < x.length; i++) {
            maxError = Math.max(maxError, Math.abs(xNuevo[i] - x[i]));
        }
        return maxError < tol;
    }
}
```

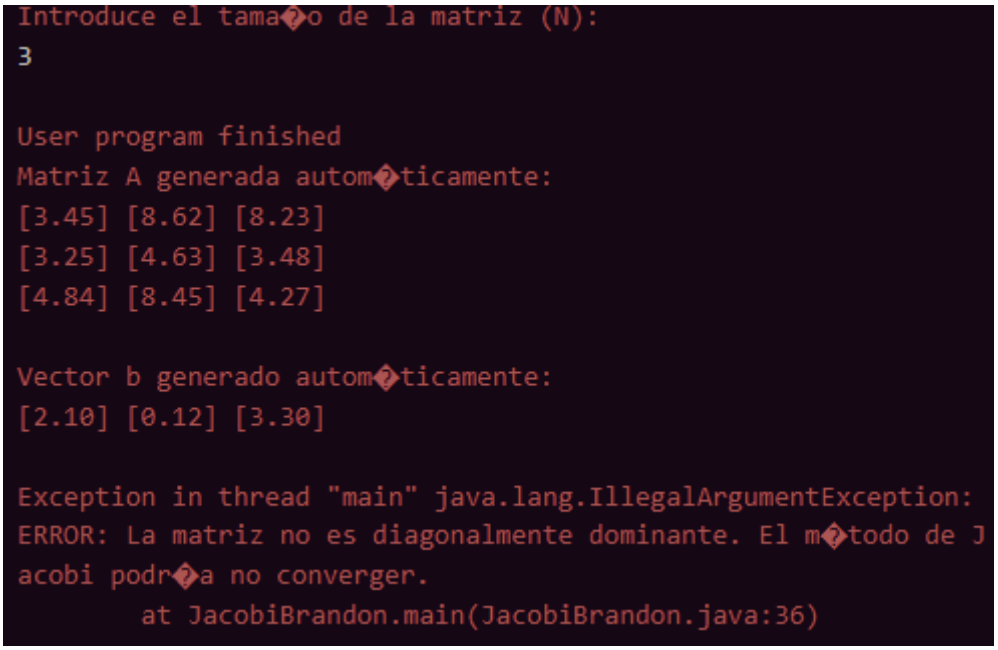
Matriz A:

3.45	8.62	8.23
3.25	4.63	3.48
4.84	8.45	4.27

Vector b:

2.10	0.12	3.30

Ejecución:



Ejercicios de Gauss-Seidel

Pseudocódigo

```
INICIO

CONSTANTE TOLERANCIA = 1e-6

FUNCION PRINCIPAL
    LEER tamaño de la matriz
    INICIALIZAR matriz como arreglo de tamaño x tamaño
    INICIALIZAR vectorB como arreglo de tamaño
    INICIALIZAR x0 como arreglo de tamaño con ceros

    LLENAR MATRIZ(matrix, vectorB, tamaño)
    IMPRIMIR MATRIZ(matrix, vectorB)

    SOLUCIÓN = GAUSS_SEIDEL(matriz, vectorB, x0, 100)

    IMPRIMIR SOLUCIÓN
FIN FUNCION

FUNCION LLENAR MATRIZ(matrix, vectorB, tamaño)
    INICIALIZAR random
    PARA i DESDE 0 HASTA tamaño - 1 HACER
        INICIALIZAR suma = 0
        PARA j DESDE 0 HASTA tamaño - 1 HACER
            SI i ES IGUAL A j ENTONCES
                matrix[i][j] = 0
            SINO
                matrix[i][j] = random.NÚMERO_ENTRE(1, 20)
            FIN SI
            suma = suma + ABS(matrix[i][j])
        FIN PARA
        matrix[i][i] = suma + random.NÚMERO_ENTRE(1, 10)
```

```
        vectorB[i] = random.NÚMERO_ENTRE(1, 50)
    FIN PARA
FIN FUNCION

FUNCION GAUSS_SEIDEL(A, b, x, maxIteraciones)
    n = LONGITUD(A)
    INICIALIZAR xNuevo como arreglo de tamaño n
    COPIAR x A xNuevo

    PARA k DESDE 0 HASTA maxIteraciones - 1 HACER
        xAntiguo = COPIAR xNuevo

        PARA i DESDE 0 HASTA n - 1 HACER
            INICIALIZAR suma = 0
            PARA j DESDE 0 HASTA n - 1 HACER
                SI j NO ES IGUAL A i ENTONCES
                    suma = suma + A[i][j] * xNuevo[j]
                FIN SI
            FIN PARA
            xNuevo[i] = (b[i] - suma) / A[i][i]
        FIN PARA

        SI CONVERGIÓ(xNuevo, xAntiguo) ENTONCES
            IMPRIMIR "Convergió en " + (k + 1) + " iteraciones."
            SALIR
        FIN SI
    FIN PARA
    RETORNAR xNuevo
FIN FUNCION

FUNCION CONVERGIÓ(xNuevo, xAntiguo)
    PARA i DESDE 0 HASTA LONGITUD(xNuevo) - 1 HACER
        SI ABS(xNuevo[i] - xAntiguo[i]) > TOLERANCIA ENTONCES
            RETORNAR FALSO
        FIN SI
    FIN PARA
    RETORNAR VERDADERO
FIN FUNCION

FUNCION IMPRIMIR MATRIZ(matrix, vectorB)
    tamaño = LONGITUD(matrix)
    PARA i DESDE 0 HASTA tamaño - 1 HACER
```

```
        PARA j DESDE 0 HASTA tamaño - 1 HACER
            IMPRIMIR "[" + matrix[i][j] + "]"
        FIN PARA
        IMPRIMIR " | [" + vectorB[i] + "]"
    FIN PARA
FIN FUNCION

FUNCION IMPRIMIR VECTOR(vector)
    PARA cada v EN vector HACER
        IMPRIMIR "[" + v + "]"
    FIN PARA
    IMPRIMIR NUEVA_LINEA
FIN FUNCION

FIN
```

Código general

```java
import java.util.Random;
import java.util.Scanner;

/**
 * Aplicación que implementa el método iterativo de Gauss-Seidel.
  *  La  matriz  y  el  vector  de  términos  independientes  se  generan
automáticamente con valores aleatorios.
 *
 * @author Oscar Aaron Delgadillo Fernandez
 * @version 1.0
 * @since 2025-03-16
 */

public class GaussSeidelOscar {
    static final double TOLERANCE = 1e-6; // Tolerancia para la convergencia

    public static void main(String[] args) {
        Scanner read = new Scanner(System.in);

        System.out.println("Resolución de matrices por Gauss-Seidel\n");
        System.out.print("Tamaño de la matriz: ");
        int size = read.nextInt();

        double[][] matrix = new double[size][size];
        double[] vectorB = new double[size];
        double[] x0 = new double[size]; // Valores iniciales en cero

        llenarMatriz(matrix, vectorB, size);
```

```
            System.out.println("Matriz de coeficientes y vector b generados aleatoriamente:");
        imprimirSistema(matrix, vectorB);

        double[] solution = gaussSeidel(matrix, vectorB, x0, 100);

        System.out.println("\nSolución encontrada:");
        imprimirVector(solution);
    }

    /**
       * Llena la matriz con valores aleatorios entre 1 y 20, asegurando diagonal dominante.
     * Llena el vector de términos independientes con valores entre 1 y 50.
     *
     * @param matrix Matriz de coeficientes.
     * @param vectorB Vector de términos independientes.
     * @param size Tamaño de la matriz.
     */
     private static void llenarMatriz(double[][] matrix, double[] vectorB, int size) {
        Random random = new Random();
        for (int i = 0; i < size; i++) {
            double sum = 0;
            for (int j = 0; j < size; j++) {
                matrix[i][j] = (i == j) ? 0 : random.nextInt(20) + 1;
                sum += Math.abs(matrix[i][j]);
            }
                  matrix[i][i] = sum + random.nextInt(10) + 1; // Asegurar diagonal dominante
            vectorB[i] = random.nextInt(50) + 1;
        }
    }

    /**
     * Aplica el método iterativo de Gauss-Seidel para resolver el sistema.
     *
     * @param A Matriz de coeficientes.
     * @param b Vector de términos independientes.
     * @param x Vector inicial de soluciones.
     * @param maxIterations Número máximo de iteraciones.
     * @return Vector solución.
     */
     private static double[] gaussSeidel(double[][] A, double[] b, double[] x, int maxIterations) {
        int n = A.length;
        double[] xNew = new double[n];
        System.arraycopy(x, 0, xNew, 0, n);

        for (int k = 0; k < maxIterations; k++) {
            double[] xOld = xNew.clone();

            for (int i = 0; i < n; i++) {
```

```
                double sum = 0;
                for (int j = 0; j < n; j++) {
                    if (j != i) {
                        sum += A[i][j] * xNew[j];
                    }
                }
                xNew[i] = (b[i] - sum) / A[i][i];
            }

            if (converged(xNew, xOld)) {
                    System.out.println("\nConvergió en " + (k + 1) + "
iteraciones.");
                break;
            }
        }
        return xNew;
    }

    /**
     * Verifica la convergencia comparando los valores entre iteraciones.
     *
     * @param xNew Vector de soluciones actual.
     * @param xOld Vector de soluciones anterior.
     * @return Verdadero si la diferencia es menor que la tolerancia.
     */
    private static boolean converged(double[] xNew, double[] xOld) {
        for (int i = 0; i < xNew.length; i++) {
            if (Math.abs(xNew[i] - xOld[i]) > TOLERANCE) {
                return false;
            }
        }
        return true;
    }

    /**
       * Imprime la matriz de coeficientes junto con el vector de términos
independientes.
     *
     * @param matrix Matriz de coeficientes.
     * @param vectorB Vector de términos independientes.
     */
    private static void imprimirSistema(double[][] matrix, double[] vectorB)
{
        int size = matrix.length;
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                System.out.printf("[%5.2f] ", matrix[i][j]);
            }
            System.out.printf(" | [%5.2f]\n", vectorB[i]);
        }
    }

    /**
```

```java
     * Imprime un vector en formato legible.
     *
     * @param vector Vector a imprimir.
     */
    private static void imprimirVector(double[] vector) {
        for (double v : vector) {
            System.out.printf("[%5.4f] ", v);
        }
        System.out.println();
    }
}
```

Ejercicio 1:
Ejercicio resuelto usando el orden convencional de arriba hacia abajo.

Matriz:

30	7	13	20
10	25	12	30
19	11	38	6

Ejecución:

```
Resolución de matrices por Gauss-Seidel

Tamaño de la matriz: 3
Matriz de coeficientes y vector b generados aleatoriamente:
[30.00] [ 7.00] [13.00]  | [20.00]
[10.00] [25.00] [12.00]  | [30.00]
[19.00] [11.00] [38.00]  | [ 6.00]

Convergió en 11 iteraciones.

Solución encontrada:
[0.5992] [1.1943] [-0.4874]
```

Ejercicio 2:
Ejercicio resuelto usando el orden de abajo hacia arriba.

Código modificado:

```java
import java.util.Random;
import java.util.Scanner;

/**
 * Aplicación que implementa el método iterativo de Gauss-Seidel en orden de abajo hacia arriba.
  * La matriz y el vector de términos independientes se generan automáticamente con valores aleatorios.
 *
 * @author Oscar Aaron Delgadillo Fernandez
 * @version 1.1
 * @since 2025-03-16
 */

public class GaussSeidelOscar1 {
        static final double TOLERANCE = 1e-6; // Tolerancia para la convergencia

    public static void main(String[] args) {
        Scanner read = new Scanner(System.in);

           System.out.println("Resolución de matrices por Gauss-Seidel de abajo hacia arriba\n");
        System.out.print("Tamaño de la matriz: ");
        int size = read.nextInt();

        double[][] matrix = new double[size][size];
        double[] vectorB = new double[size];
        double[] x0 = new double[size]; // Valores iniciales en cero

        llenarMatriz(matrix, vectorB, size);
          System.out.println("Matriz de coeficientes y vector b generados aleatoriamente:");
        imprimirSistema(matrix, vectorB);

        double[] solution = gaussSeidel(matrix, vectorB, x0, 100);

        System.out.println("\nSolución encontrada:");
        imprimirVector(solution);
    }

    /**
       * Llena la matriz con valores aleatorios entre 1 y 20, asegurando diagonal dominante.
     * Llena el vector de términos independientes con valores entre 1 y 50.
     *
     * @param matrix Matriz de coeficientes.
     * @param vectorB Vector de términos independientes.
     * @param size Tamaño de la matriz.
     */
     private static void llenarMatriz(double[][] matrix, double[] vectorB, int size) {
```

```
        Random random = new Random();
        for (int i = 0; i < size; i++) {
            double sum = 0;
            for (int j = 0; j < size; j++) {
                matrix[i][j] = (i == j) ? 0 : random.nextInt(20) + 1;
                sum += Math.abs(matrix[i][j]);
            }
                matrix[i][i] = sum + random.nextInt(10) + 1; // Asegurar
diagonal dominante
            vectorB[i] = random.nextInt(50) + 1;
        }
    }

    /**
     * Aplica el método iterativo de Gauss-Seidel en orden de abajo hacia
arriba.
     *
     * @param A Matriz de coeficientes.
     * @param b Vector de términos independientes.
     * @param x Vector inicial de soluciones.
     * @param maxIterations Número máximo de iteraciones.
     * @return Vector solución.
     */
    private static double[] gaussSeidel(double[][] A, double[] b, double[]
x, int maxIterations) {
        int n = A.length;
        double[] xNew = new double[n];
        System.arraycopy(x, 0, xNew, 0, n);

        for (int k = 0; k < maxIterations; k++) {
            double[] xOld = xNew.clone();

             for (int i = n - 1; i >= 0; i--) { // Recorrer de abajo hacia
arriba
                double sum = 0;
                for (int j = 0; j < n; j++) {
                    if (j != i) {
                        sum += A[i][j] * xNew[j];
                    }
                }
                xNew[i] = (b[i] - sum) / A[i][i];
            }

            if (converged(xNew, xOld)) {
                    System.out.println("\nConvergió en " + (k + 1) + "
iteraciones.");
                break;
            }
        }
        return xNew;
    }

    /**
```

```
     * Verifica la convergencia comparando los valores entre iteraciones.
     *
     * @param xNew Vector de soluciones actual.
     * @param xOld Vector de soluciones anterior.
     * @return Verdadero si la diferencia es menor que la tolerancia.
     */
    private static boolean converged(double[] xNew, double[] xOld) {
        for (int i = 0; i < xNew.length; i++) {
            if (Math.abs(xNew[i] - xOld[i]) > TOLERANCE) {
                return false;
            }
        }
        return true;
    }

    /**
      * Imprime la matriz de coeficientes junto con el vector de términos
independientes.
     *
     * @param matrix Matriz de coeficientes.
     * @param vectorB Vector de términos independientes.
     */
        private static void imprimirSistema(double[][] matrix, double[]
vectorB) {
        int size = matrix.length;
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                System.out.printf("[%5.2f] ", matrix[i][j]);
            }
            System.out.printf(" | [%5.2f]\n", vectorB[i]);
        }
    }

    /**
     * Imprime un vector en formato legible.
     *
     * @param vector Vector a imprimir.
     */
    private static void imprimirVector(double[] vector) {
        for (double v : vector) {
            System.out.printf("[%5.4f] ", v);
        }
        System.out.println();
    }
}
```

Matriz:

18	6	11	35
5	21	12	8
16	12	36	15

Ejecución:

```
Resolución de matrices por Gauss-Seidel de abajo hacia arriba

Tamaño de la matriz: 3
Matriz de coeficientes y vector b generados aleatoriamente:
[18.00] [ 6.00] [11.00]  | [35.00]
[ 5.00] [21.00] [12.00]  | [ 8.00]
[16.00] [12.00] [36.00]  | [15.00]

Convergió en 13 iteraciones.

Solución encontrada:
[2.2806] [0.2212] [-0.6706]
```

Ejercicio 3:
Ejercicio resuelto usando el orden aleatorio.

Código modificado:

```
import java.util.Scanner;
import java.util.Random;
import java.util.Collections;
import java.util.List;
import java.util.ArrayList;

/**
 * Aplicación que implementa el método iterativo de Gauss-Seidel en orden aleatorio.
```

```
  * La matriz y el vector de términos independientes se generan automáticamente con valores aleatorios.
 *
 * @author Oscar Aaron Delgadillo Fernandez
 * @version 1.2
 * @since 2025-03-16
 */

public class GaussSeidelOscar2 {
        static final double TOLERANCE = 1e-6; // Tolerancia para la convergencia

    public static void main(String[] args) {
        Scanner read = new Scanner(System.in);

             System.out.println("Resolución de matrices por Gauss-Seidel aleatorio\n");
        System.out.print("Tamaño de la matriz: ");
        int size = read.nextInt();

        double[][] matrix = new double[size][size];
        double[] vectorB = new double[size];
        double[] x0 = new double[size]; // Valores iniciales en cero

        llenarMatriz(matrix, vectorB, size);
           System.out.println("Matriz de coeficientes y vector b generados aleatoriamente:");
        imprimirSistema(matrix, vectorB);

        double[] solution = gaussSeidel(matrix, vectorB, x0, 100);

        System.out.println("\nSolución encontrada:");
        imprimirVector(solution);
    }

    /**
       * Llena la matriz con valores aleatorios entre 1 y 20, asegurando diagonal dominante.
     * Llena el vector de términos independientes con valores entre 1 y 50.
     *
     * @param matrix Matriz de coeficientes.
     * @param vectorB Vector de términos independientes.
     * @param size Tamaño de la matriz.
     */
     private static void llenarMatriz(double[][] matrix, double[] vectorB, int size) {
        Random random = new Random();
        for (int i = 0; i < size; i++) {
            double sum = 0;
            for (int j = 0; j < size; j++) {
                matrix[i][j] = (i == j) ? 0 : random.nextInt(20) + 1;
                sum += Math.abs(matrix[i][j]);
            }
```

```
                matrix[i][i] = sum + random.nextInt(10) + 1; // Asegurar
diagonal dominante
            vectorB[i] = random.nextInt(50) + 1;
        }
    }

    /**
     * Aplica el método iterativo de Gauss-Seidel en orden aleatorio.
     *
     * @param A Matriz de coeficientes.
     * @param b Vector de términos independientes.
     * @param x Vector inicial de soluciones.
     * @param maxIterations Número máximo de iteraciones.
     * @return Vector solución.
     */
    private static double[] gaussSeidel(double[][] A, double[] b, double[]
x, int maxIterations) {
        int n = A.length;
        double[] xNew = new double[n];
        System.arraycopy(x, 0, xNew, 0, n);
        Random rand = new Random();
        List<Integer> indices = new ArrayList<>();

        for (int i = 0; i < n; i++) {
            indices.add(i);
        }

        for (int k = 0; k < maxIterations; k++) {
            double[] xOld = xNew.clone();
             Collections.shuffle(indices, rand); // Orden aleatorio en cada
iteración

            for (int i : indices) {
                double sum = 0;
                for (int j = 0; j < n; j++) {
                    if (j != i) {
                        sum += A[i][j] * xNew[j];
                    }
                }
                xNew[i] = (b[i] - sum) / A[i][i];
            }

            if (converged(xNew, xOld)) {
                    System.out.println("\nConvergió en " + (k + 1) + "
iteraciones.");
                break;
            }
        }
        return xNew;
    }

    /**
     * Verifica la convergencia comparando los valores entre iteraciones.
```

```
     *
     * @param xNew Vector de soluciones actual.
     * @param xOld Vector de soluciones anterior.
     * @return Verdadero si la diferencia es menor que la tolerancia.
     */
    private static boolean converged(double[] xNew, double[] xOld) {
        for (int i = 0; i < xNew.length; i++) {
            if (Math.abs(xNew[i] - xOld[i]) > TOLERANCE) {
                return false;
            }
        }
        return true;
    }

    /**
      * Imprime la matriz de coeficientes junto con el vector de términos
independientes.
     *
     * @param matrix Matriz de coeficientes.
     * @param vectorB Vector de términos independientes.
     */
      private static void imprimirSistema(double[][] matrix, double[]
vectorB) {
        int size = matrix.length;
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                System.out.printf("[%5.2f] ", matrix[i][j]);
            }
            System.out.printf(" | [%5.2f]\n", vectorB[i]);
        }
    }

    /**
     * Imprime un vector en formato legible.
     *
     * @param vector Vector a imprimir.
     */
    private static void imprimirVector(double[] vector) {
        for (double v : vector) {
            System.out.printf("[%5.4f] ", v);
        }
        System.out.println();
    }
}
```

Matriz:

32	19	4	4
15	33	14	8
17	17	40	21

Ejecución:

```
Resolución de matrices por Gauss-Seidel aleatorio

Tamaño de la matriz: 3
Matriz de coeficientes y vector b generados aleatoriamente:
[32.00] [19.00] [ 4.00]  | [ 4.00]
[15.00] [33.00] [14.00]  | [ 8.00]
[17.00] [17.00] [40.00]  | [21.00]

Convergió en 14 iteraciones.

Solución encontrada:
[0.0606] [0.0038] [0.4977]
```

Ejercicio 4:

Ejercicio resuelto usando el orden por mayor coeficiente diagonal.

Código modificado:

```
import java.util.Scanner;
import java.util.Random;
import java.util.Arrays;
import java.util.Comparator;

/**
 * Aplicación que implementa el método iterativo de Gauss-Seidel en orden basado en el mayor coeficiente diagonal.
  * La matriz y el vector de términos independientes se generan automáticamente con valores aleatorios.
 *
 * @author Oscar Aaron Delgadillo Fernandez
 * @version 1.3
 * @since 2025-03-16
 */

public class GaussSeidelOscar3 {
        static final double TOLERANCE = 1e-6; // Tolerancia para la convergencia

    public static void main(String[] args) {
        Scanner read = new Scanner(System.in);
```

```
         System.out.println("Resolución de matrices por Gauss-Seidel con el mayor coeficiente diagonal\n");
        System.out.print("Tamaño de la matriz: ");
        int size = read.nextInt();

        double[][] matrix = new double[size][size];
        double[] vectorB = new double[size];
        double[] x0 = new double[size]; // Valores iniciales en cero

        llenarMatriz(matrix, vectorB, size);
           System.out.println("Matriz de coeficientes y vector b generados aleatoriamente:");
        imprimirSistema(matrix, vectorB);

        double[] solution = gaussSeidel(matrix, vectorB, x0, 100);

        System.out.println("\nSolución encontrada:");
        imprimirVector(solution);
    }

    /**
       * Llena la matriz con valores aleatorios entre 1 y 20, asegurando diagonal dominante.
     * Llena el vector de términos independientes con valores entre 1 y 50.
     *
     * @param matrix Matriz de coeficientes.
     * @param vectorB Vector de términos independientes.
     * @param size Tamaño de la matriz.
     */
     private static void llenarMatriz(double[][] matrix, double[] vectorB, int size) {
        Random random = new Random();
        for (int i = 0; i < size; i++) {
            double sum = 0;
            for (int j = 0; j < size; j++) {
                matrix[i][j] = (i == j) ? 0 : random.nextInt(20) + 1;
                sum += Math.abs(matrix[i][j]);
            }
                 matrix[i][i] = sum + random.nextInt(10) + 1; // Asegurar diagonal dominante
            vectorB[i] = random.nextInt(50) + 1;
        }
    }

    /**
      * Aplica el método iterativo de Gauss-Seidel en orden según el mayor coeficiente diagonal.
     *
     * @param A Matriz de coeficientes.
     * @param b Vector de términos independientes.
     * @param x Vector inicial de soluciones.
     * @param maxIterations Número máximo de iteraciones.
```

```
     * @return Vector solución.
     */
    private static double[] gaussSeidel(double[][] A, double[] b, double[]
x, int maxIterations) {
        int n = A.length;
        double[] xNew = new double[n];
        System.arraycopy(x, 0, xNew, 0, n);
        Integer[] indices = new Integer[n];

        for (int i = 0; i < n; i++) {
            indices[i] = i;
        }

        // Ordenar índices según el mayor coeficiente diagonal
                    Arrays.sort(indices,   Comparator.comparingDouble(i   ->
-Math.abs(A[i][i])));

        for (int k = 0; k < maxIterations; k++) {
            double[] xOld = xNew.clone();

            for (int i : indices) {
                double sum = 0;
                for (int j = 0; j < n; j++) {
                    if (j != i) {
                        sum += A[i][j] * xNew[j];
                    }
                }
                xNew[i] = (b[i] - sum) / A[i][i];
            }

            if (converged(xNew, xOld)) {
                      System.out.println("\nConvergió en " + (k + 1) + "
iteraciones.");
                break;
            }
        }
        return xNew;
    }

    /**
     * Verifica la convergencia comparando los valores entre iteraciones.
     *
     * @param xNew Vector de soluciones actual.
     * @param xOld Vector de soluciones anterior.
     * @return Verdadero si la diferencia es menor que la tolerancia.
     */
    private static boolean converged(double[] xNew, double[] xOld) {
        for (int i = 0; i < xNew.length; i++) {
            if (Math.abs(xNew[i] - xOld[i]) > TOLERANCE) {
                return false;
            }
        }
        return true;
```

```
    }

    /**
      * Imprime la matriz de coeficientes junto con el vector de términos
independientes.
     *
     * @param matrix Matriz de coeficientes.
     * @param vectorB Vector de términos independientes.
     */
        private static void imprimirSistema(double[][] matrix, double[]
vectorB) {
        int size = matrix.length;
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                System.out.printf("[%5.2f] ", matrix[i][j]);
            }
            System.out.printf(" | [%5.2f]\n", vectorB[i]);
        }
    }

    /**
     * Imprime un vector en formato legible.
     *
     * @param vector Vector a imprimir.
     */
    private static void imprimirVector(double[] vector) {
        for (double v : vector) {
            System.out.printf("[%5.4f] ", v);
        }
        System.out.println();
    }
}
```

Matriz:

20	2	12	40
10	28	15	9
2	5	11	33

Ejecución:

```
Resolución de matrices por Gauss-Seidel con el mayor coeficiente diagonal

Tamaño de la matriz: 3
Matriz de coeficientes y vector b generados aleatoriamente:
[20.00] [ 2.00] [12.00]  | [40.00]
[10.00] [28.00] [15.00]  | [ 9.00]
[ 2.00] [ 5.00] [11.00]  | [33.00]

Convergió en 15 iteraciones.

Solución encontrada:
[-0.0985] [-1.6658] [3.7751]
```

Ejercicio 5:
Ejercicio resuelto cuando falla.

Código modificado:

```java
import java.util.Random;
import java.util.Scanner;

/**
 * Aplicación que implementa el algoritmo de Gauss-Seidel con inyección de errores.
 * Se incluyen verificaciones para detectar errores como división por cero,
 * falta de convergencia e inestabilidad numérica (NaN).
 *
 * @author Oscar Aaron Delgadillo Fernandez
 * @version 1.4
 * @since 2025-03-16
 */

public class GaussSeidelOscarError {
    static final int MAX_ITER = 1000; // Límite de iteraciones
        static final double TOLERANCE = 1e-6; // Tolerancia para la convergencia

    public static void main(String[] args) {
        Scanner read = new Scanner(System.in);

         System.out.println("Resolución de matrices por Gauss-Seidel cuando falla\n");
        System.out.print("Tamaño de la matriz: ");
        int size = read.nextInt();

        double[][] matrix = new double[size][size + 1]; // Matriz aumentada
            llenarMatriz(matrix, size); // Se inyectarán errores en esta función

        System.out.println("\nMatriz original:");
```

```
        imprimirMatriz(matrix);

        // Verifica si la matriz es diagonalmente dominante
        if (!esDiagonalmenteDominante(matrix, size)) {
                    System.err.println("\nADVERTENCIA: La matriz no es
diagonalmente dominante. Gauss-Seidel podría no converger.");
        }

        double[] resultado = gaussSeidel(matrix, size);

        // Verifica si se encontró una solución válida
        if (resultado != null) {
            System.out.println("\nSolución encontrada:");
            for (int i = 0; i < size; i++) {
                System.out.printf("x[%d] = %.6f\n", i, resultado[i]);
            }
        } else {
               System.err.println("\nERROR: El método Gauss-Seidel no pudo
encontrar una solución.");
        }
    }

    /**
     * Llena una matriz con valores controlados para forzar errores.
     *
     * @param matrix Matriz aumentada a llenar.
     * @param size Tamaño de la matriz.
     */
    private static void llenarMatriz(double[][] matrix, int size) {
        Random random = new Random();

        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                matrix[i][j] = random.nextInt(20); // Valores entre 0 y 20
            }
                    matrix[i][size] = random.nextInt(50); // Términos
independientes
        }

        // FORZAR ERROR: División por cero en la diagonal
        if (size > 1) {
            matrix[0][0] = 0; // Esto provocará una división por cero
        }

        // FORZAR ERROR: Falta de convergencia (no diagonalmente dominante)
        if (size > 2) {
            matrix[1][1] = 1;
            matrix[1][0] = 10;
                matrix[1][2] = 10; // Provocamos que no sea diagonalmente
dominante
        }

        // FORZAR ERROR: NaN (inestabilidad numérica)
```

```
        if (size > 3) {
                matrix[2][1] = Double.NaN; // Se inyecta un valor NaN en la
matriz
        }
    }

    /**
     * Verifica si la matriz es diagonalmente dominante.
     *
     * @param matrix Matriz aumentada a verificar.
     * @param size Tamaño de la matriz.
      * @return true si la matriz es diagonalmente dominante, false en caso
contrario.
     */
    private static boolean esDiagonalmenteDominante(double[][] matrix, int
size) {
        for (int i = 0; i < size; i++) {
            double sumaFila = 0;
            for (int j = 0; j < size; j++) {
                if (i != j) sumaFila += Math.abs(matrix[i][j]);
            }
            if (Math.abs(matrix[i][i]) < sumaFila) {
                return false;
            }
        }
        return true;
    }

    /**
        * Aplica el método de Gauss-Seidel para resolver el sistema de
ecuaciones lineales.
     *
     * @param matrix Matriz aumentada del sistema de ecuaciones.
     * @param size Tamaño de la matriz.
       * @return Un vector con la solución del sistema, o null si no se
encuentra una solución.
     */
    private static double[] gaussSeidel(double[][] matrix, int size) {
        double[] x = new double[size]; // Soluciones inicializadas en 0
        int iteraciones = 0;
        boolean converge;

        do {
            double[] xOld = x.clone();
            converge = true;

            for (int i = 0; i < size; i++) {
                // Verifica división por cero
                if (matrix[i][i] == 0) {
                        System.err.println("ERROR: División por cero en la
diagonal.");
                    return null;
                }
```

```
                double suma = matrix[i][size]; // Término independiente
                for (int j = 0; j < size; j++) {
                    if (i != j) {
                        suma -= matrix[i][j] * x[j];
                    }
                }
                x[i] = suma / matrix[i][i];

                    // Verifica si hay valores NaN (error de inestabilidad
numérica)
                if (Double.isNaN(x[i])) {
                         System.err.println("ERROR: Valores NaN detectados,
posible inestabilidad numérica.");
                    return null;
                }

                // Comprobación de convergencia
                if (Math.abs(x[i] - xOld[i]) > TOLERANCE) {
                    converge = false;
                }
            }
            iteraciones++;

            // Si se exceden las iteraciones máximas, el método no converge
            if (iteraciones > MAX_ITER) {
                System.err.println("ERROR: El método no converge después de
" + MAX_ITER + " iteraciones.");
                return null;
            }

        } while (!converge);

        return x;
    }

    /**
     * Imprime una matriz en formato legible.
     *
     * @param matrix Matriz a imprimir.
     */
    private static void imprimirMatriz(double[][] matrix) {
        int rows = matrix.length;
        int cols = matrix[0].length;

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                System.out.printf("[%6.2f] ", matrix[i][j]);
            }
            System.out.println();
        }
    }
}
```

Matriz:

0	0	13	31
10	1	10	36
3	17	8	9

Ejecución:

```
Resolución de matrices por Gauss-Seidel cuando falla

Tamaño de la matriz: 3

Matriz original:
[  0.00] [  0.00] [ 13.00] [ 31.00]
[ 10.00] [  1.00] [ 10.00] [ 36.00]
[  3.00] [ 17.00] [  8.00] [  9.00]

ADVERTENCIA: La matriz no es diagonalmente dominante. Gauss-Seidel podría no converger.
ERROR: División por cero en la diagonal.

ERROR: El método Gauss-Seidel no pudo encontrar una solución.
```

Conclusiones

Brandon García Ordaz

A través de este problemario, pude comprender mejor el funcionamiento de los métodos numéricos en la resolución de sistemas de ecuaciones lineales. La implementación en Java me permitió analizar el comportamiento de cada técnica en distintos escenarios y reforzar mi conocimiento sobre su eficiencia. Este trabajo me ayudó a ver cómo la programación puede ser una herramienta clave para automatizar y visualizar soluciones matemáticas de manera efectiva.

Diego Alonso Coronel Vargas

Mediante este problemario, he adquirido una comprensión más profunda sobre cómo funcionan los métodos numéricos para la resolución de sistemas de ecuaciones lineales. La implementación en Java me ofreció la oportunidad de observar el comportamiento de cada técnica en diferentes contextos, lo que ha fortalecido mi conocimiento sobre su efectividad. Este trabajo me ha permitido apreciar cómo la programación puede ser una herramienta crucial para automatizar y visualizar soluciones matemáticas de manera eficiente.

Oscar Aaron Delgadillo Fernandez

La realización de este problemario me permitió mejorar mi comprensión sobre los métodos numéricos aplicados a la resolución de sistemas de ecuaciones lineales. A través de su implementación en Java, pude analizar el desempeño de cada técnica en distintos casos, evaluando su eficiencia y precisión. Además, este trabajo me permitió apreciar cómo la programación facilita la automatización y visualización de soluciones matemáticas, convirtiéndose en una herramienta fundamental para el análisis y resolución de problemas complejos.

Bibliografía

The Gaussian elimination algorithm. (s/f). Umanitoba.Ca. Recuperado el 20 de marzo de 2025, de https://linearalgebra.math.umanitoba.ca/math1220/section-12.html

LUDA UAM-Azc. (s/f). Org.mx. Recuperado el 20 de marzo de 2025, de https://aniei.org.mx/paginas/uam/CursoMN/curso_mn_12.html

Javier, C. R. J. (s/f). *Métodos iterativos de Jacobi y Gauss-Seidel*. Unam.mx. Recuperado el 20 de marzo de 2025, de https://www.ingenieria.unam.mx/pinilla/PE105117/pdfs/tema3/3-3_metodos_jacobi_gauss-seidel.pdf

Extras

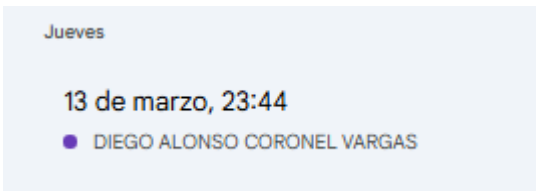
Distribución del trabajo

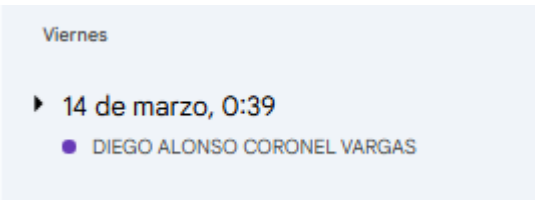
Gráfico Gantt

	Jueves 13: 11 pm	Viernes 14: 12 am	Domingo 16: 4 pm	Domingo 16: 5 pm	Domingo 16: 8 pm	Domingo 16: 11 pm
Eliminación Gaussiana	Diego					
Método de Gauss-Jordan		Diego				
Método de Gauss-Seidel					Oscar	
Método de Jacobi			Brandon	Brandon		Brandon

Tabla Scrum

Diego Alonso Coronel Vargas	Brandon García Ordaz	Oscar Aaron Delgadillo Fernandez
- Eliminación Gaussiana - Método de Gauss-Jordan	- Ejercicio extra de redondeo - Ejercicio extra de truncación	- Método de Gauss-Seidel






16 de marzo, 20:14
OSCAR AARON DELGADILLO FERNANDEZ
16 de marzo, 17:26
BRANDON GARCIA ORDAZ
16 de marzo, 16:20
BRANDON GARCIA ORDAZ


16 de marzo, 23:14
BRANDON GARCIA ORDAZ
16 de marzo, 20:35
OSCAR AARON DELGADILLO FERNANDEZ